通信

进程间的内部通信是分布式系统的核心。如果不自信的研究不同机器之间的进程是如何通信的，那么学习分布式系统就没有意义了。这种通信都是基于底层的网络的。通过消息传递来表达通信比使用基于共享内存的原语要困难。现代分布式系统通常由分散在网络(如Internet)上的数千个甚至数百万个进程组成，然而这些进程之间的通信并不可靠。除非计算机网络的原始通信设施被其他东西取代，否则大规模分布式应用程序的开发是极其困难的。

在本章中，我们首先讨论通信过程必须遵循的规则(称为协议)，并着重于以层的形式构造这些协议。之后我们将着眼于两个广泛使用的模型：Remote Procedure Call（RPC）和 Message-Oriented Middleware（MOM）。同样的，我们将讨论多播问题。

我们在分布式系统中的第一个通信模型是远程过程调用(RPC)。RPC旨在隐藏大多数复杂的消息传递，非常适合客户机-服务器应用程序。然而，以透明的方式实现rpc说起来容易做起来难。我们将查看一些不可忽略的重要细节，同时深入研究实际代码，以说明在性能可以接受的情况下，能将透明性实现到什么程度。

在许多分布式应用程序中，通信并不遵循相当严格的客户机-服务器交互模式。在这些情况下，事实证明用信息来思考更合适。计算机网络的底层通信设施在很多方面都是不合适的，同样是由于他们缺乏分布式的透明性。另一个替代方法是使用一种高级别的消息队列模型，就像使用邮件系统通信一样。面向消息通信是一个重要的主题。我们研究了许多方面，包括应用级别的路由。

最后，由于我们对组播设施的理解有所提高，出现了新颖而优雅的数据传播解决方案。

我们将在本章的最后一节单独关注这个主题，讨论传统的确定性组播方法，以及泛洪和gossiping中使用的概率方法。由于其优雅和简单，后者在过去几年中受到越来越多的关注。

4.1 基础

在开始讨论分布式系统中的通信之前，我们首先概述一些与通信相关的基本问题。在下一节中，我们将简要讨论网络通信协议，因为这些协议构成了任何分布式系统的基础。然后，我们采用不同的方法，对分布式系统中通常发生的不同类型的通信进行分类。

**层级协议**

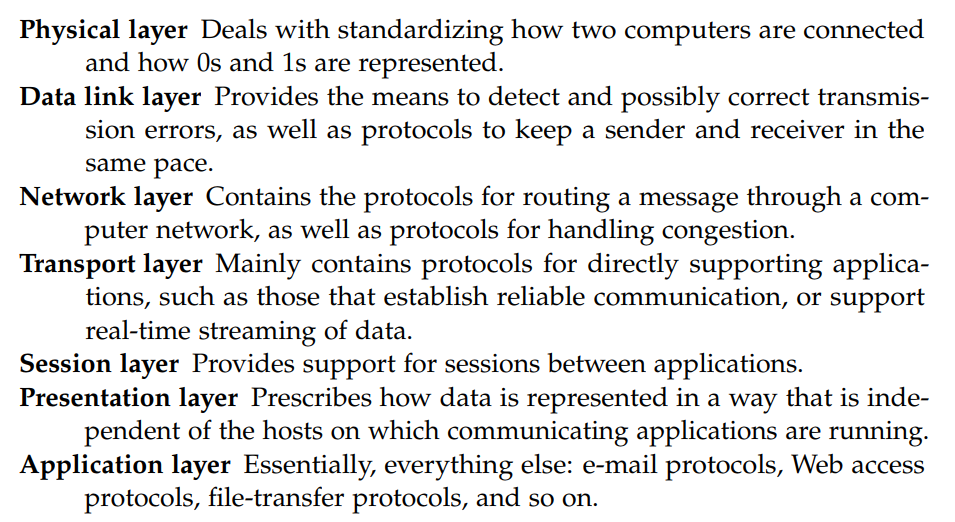
由于缺乏共享内存，分布式系统中的所有通信都基于发送和接收(低层)消息。当进程P希望与进程Q通信时，它首先在自己的地址空间中构建一条消息。然后它执行一个系统调用，使操作系统通过网络向Q发送消息。虽然这个基本的想法听起来很简单，但是为了防止混乱，P和Q必须就发送的比特的意义达成一致。

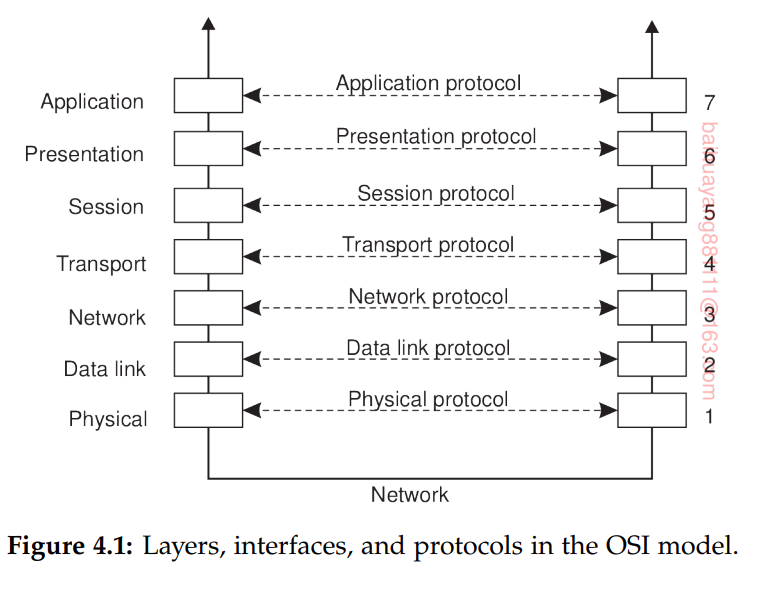
**OSI相关模型**

为了更容易处理通信中涉及的众多级别和问题，国际标准组织(ISO)开发了一个参考模型，该模型清楚地标识了涉及的各个级别，给出了它们的标准名称，并指出哪个级别应该做哪个工作。这个模型被称为开放系统互连参考模型[Day and Zimmerman, 1983]，通常缩写为ISO OSI或者有时只是OSI模型。应该强调的是，作为OSI模型的一部分而开发的协议从来没有被广泛使用，而且基本上已经死了。然而，事实证明，底层模型本身对于理解计算机网络非常有用。虽然我们不打算在这里对这个模型及其所有含义进行全面描述，但是做一个简短的介绍还是会有用的。

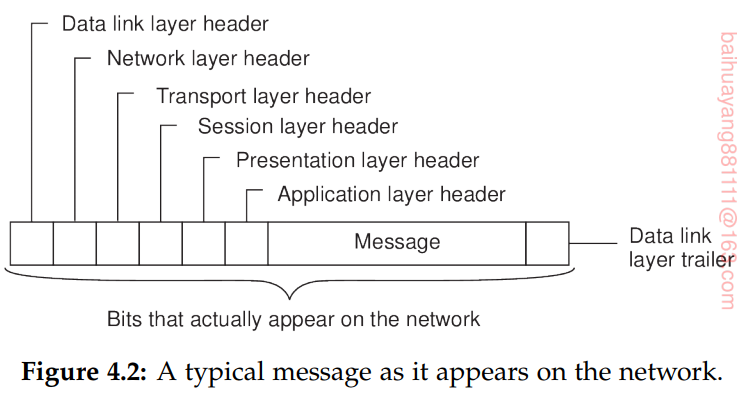
OSI模型是为了让开放系统通信而设计的。开放系统是一种通过使用标准的格式，内容和消息内容的规则，来和其他的系统进行通信的系统。这些规则称之为**通信协议**。为了能够使得计算机可以通过网络进行通信，通信协议是必须要被使用的。一种协议其实就是一种通信服务。这里有两种类型的服务。对于面向连接的服务，在交换数据之前，发送方和接收方首先显式地建立连接，并可能协商它们将使用的协议的特定参数。当它们完成时，它们释放(终止)连接。电话是一种典型的面向连接的通信服务。对于无连接的服务，不需要预先设置。当第一个消息准备好时，发送方就发送它。将信件放入邮箱是使用无连接通信服务的一个例子。在计算机世界中，面向连接和无连接通信都很常见。

在OSI模型中，通信被分为了7层，如图4.1。每一层都为上一层提供了一个或者更多个的特殊的通信服务。通过这种方式，将消息从A发送到B的问题就可以划分成许多可管理的部分，每一部分都可以独立的和其他层来处理。每一层都为上一层提供一个接口。接口由一组操作组成，这些操作一起定义了层准备提供的服务。这七层分别是：





当进程P希望与某个远程进程Q通信时，它构建一条消息，并通过接口将该消息传递给应用程序层。这个接口通常以库过程的形式出现。然后，应用层软件在消息的前面添加一个头，并通过层6/7接口将结果消息传递给表示层。表示层依次添加自己的头并将结果传递到会话层，以此类推。有些图层不仅在前面添加了一个header，而且在末尾添加了一个tail。当它到达底部时，物理层实际上通过将消息放到物理传输介质上来传输消息(到目前为止，看起来可能如图4.2所示)。



当消息到达承载Q的远程机器时，它向上传递，每一层都被剥离并检查它自己的报头。最后，消息到达接收方进程Q，进程Q可能使用相反的路径对消息进行应答（如图4.2，进程p按照从内向外的顺序增加请求头，进程q按照从外向内的顺序解析请求头）。第n层标头中的信息用于第n层协议。

在OSI模型中，一共有7层，如图4.1。在特定系统中使用的协议集合称为协议套件或协议栈。将引用模型与其实际协议区分开来是很重要的。如上所述，OSI协议从来没有流行过，这与为Internet开发的协议(如TCP和IP)形成了鲜明的对比。

**中间件协议**

中间件是一个逻辑上存在于OSI应用层中的应用程序，但它包含许多通用协议，这些协议保证它们自己的层独立于其他更具体的应用程序。我们主要看几个例子。

域名系统(DNS) [Liu and Albitz, 2006]是一种分布式服务，用于查找与名称相关的网络地址，例如www.distributedsystems.net之类的所谓域名的地址。就OSI参考模型而言，DNS是一个应用程序，因此在逻辑上被放置在应用程序层。然而，DNS提供的是一种通用的、独立于应用程序的服务，这一点应该是相当明显的。可以说，它是中间件的一部分。

另一个例子是，有多种方法可以建立身份验证，即提供已声明身份的证明。身份验证协议并不与任何特定的应用程序紧密相连，而是可以作为通用服务集成到中间件系统中。同样，授权协议通常具有与应用程序无关的特性，通过授权协议，经过身份验证的用户和进程只能访问他们具有授权的资源。作为OSI参考模型中的应用程序，这些都是属于中间件的清晰示例。

分布式提交协议规定，在一组进程中(可能分布在许多机器上)，要么所有进程执行特定的操作，要么根本不执行操作。这种现象也称为**原子性**，在事务中得到了广泛的应用。事实证明，提交协议可以独立于特定应用程序呈现接口，从而提供通用事务服务。在这种形式下，它们通常属于中间件，而不属于OSI应用层。

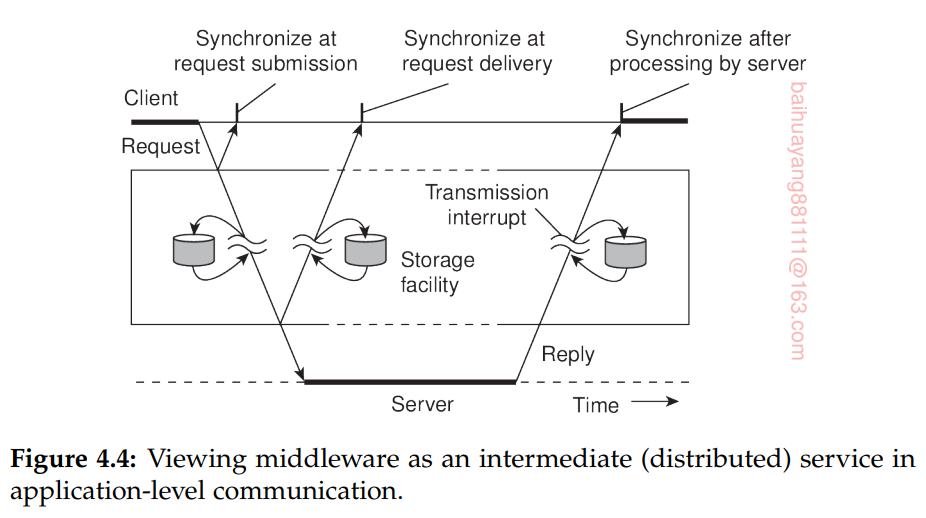
作为最后一个示例，考虑一个分布式锁定协议，通过该协议，可以保护资源不受分布在多台机器上的进程集合的同时访问。不难想象，这样的协议可以以独立于应用程序的方式设计，并且可以通过相对简单的、同样独立于应用程序的接口进行访问。因此，它们通常属于中间件。

这些协议示例并不直接绑定到通信，但是也有许多中间件通信协议。例如，所谓的**远程过程调用**，就是提供了一个工具来在本地调用实现于远程机器的方法过程。该通信服务属于最古老的中间件服务类型之一，用于实现访问透明性。同样的，有用于设置和同步流以传输实时数据的高级通信服务，例如多媒体应用程序所需的流。最后一个例子是，一些中间件系统提供可靠的多播服务，可以扩展到广域网上的数千个接收器。

采用这种方法进行分层，可以得到经过调整和简化的通信参考模型，如图4.3所示。与OSI模型相比，会话和表示层已经被一个包含独立于应用程序协议的中间件层所取代。这些协议不属于我们刚才讨论的底层。网络和传输服务由管理着底层硬件建立通信的操作系统分成了几组通信服务。

**通信类型**

在本章的其余部分中，我们将集中讨论高级中间件通信服务。在此之前，还有其他一些用于区分(中间件)通信的通用标准。为了理解中间件可以为应用程序提供的各种通信替代方案，我们将中间件视为客户机-服务器计算中的附加服务，如图4.4所示。例如电子邮件系统。原则上，邮件传递系统的核心可以看作是中间件通信服务。每个主机运行一个用户代理，允许用户编写、发送和接收电子邮件。发送用户代理将此类邮件传递给邮件传递系统，并期望它最终将邮件传递给目标收件人。同样，接收方的用户代理连接到邮件传递系统，查看是否有邮件进入。如果是，则将消息传输到用户代理，以便用户可以显示和读取它们。



电子邮件系统是一个典型的例子，其中通信是持久的。使用**持久通信**，通信中间件存储已提交用于传输的消息，直到将其交付给接收方为止。在这种情况下，中间件将在图4.4所示的一个或多个存储设施中存储消息。因此，发送应用程序无需在提交消息后继续执行（意思就是提交之后，会有别的中间件帮你进行发送消息）。同样，在提交消息时，接收应用程序不需要执行。

相反，使用**瞬态通信**，通信系统只在发送和接收应用程序执行时才存储消息。更准确地说，从图4.4中可以看出，如果中间件由于传输中断而不能传递消息，或者因为接收方当前没有活动，那么这个消息将被简单地丢弃。通常，所有传输级通信服务只提供瞬时通信。在这种情况下，通信系统由传统的存储转发路由器组成。如果路由器不能向下一个或目标主机发送消息，它将直接丢弃该消息。

通信除了分为持久的和短暂的，还可以分为异步的或同步的。**异步通信**的特征是，发送方在提交消息后可以立即继续执行其他操作（不用等待发送消息响应）。这意味着消息(临时)在提交时立即由中间件存储。使用**同步通信**，发送方将被阻塞，直到知道其请求被接受为止。同步通信有三个特点，第一，发送方会一直阻塞，直到中间件接管了请求。第二，发送方可以同步，直到其请求被发送到预期的接收方为止。第三，同步可以通过让发送方等待，直到其请求被完全处理，也就是说，直到接收方返回响应。

实践中，经常使用的是持久性和同步的各种结合。流行的方法是结合请求提交时的持久性和同步，这是许多消息队列系统的常见方案，我们将在本章后面讨论。同样，在请求被完全处理之后，使用同步的瞬态通信也得到了广泛的应用。这个方案与远程过程调用相对应，我们接下来将讨论这个问题。

4.2 远程过程调用

许多分布式系统都基于进程之间显式的消息交换。然而对于分布式系统的访问透明性来说，隐藏发送和接收这种通信过程是很重要的。这个问题早已为人所知，但直到上世纪80年代，研究人员(Birrell and Nelson, 1984)引入了一种完全不同的沟通方式后，人们才有所行动。尽管这个想法简单得令人耳目一新，但其含义往往是精妙的。在本节中，我们将研究这个概念、它的实现、它的优点和缺点。

简而言之，该提议允许程序调用位于其他机器上的过程。当机器a上的一个进程调用机器B上的一个过程时，a上的调用进程被挂起，被调用的过程在B上执行。参数可以从调用者传输到被调用者，结果会从被调用者传回调用者。所有的信息传递对于编程者都是不可见的。这种方式就成为**远程过程调用**，或者**RPC**。

虽然基本思想听起来简单优雅，但也存在一些微妙的问题。首先，由于调用和被调用过程运行在不同的机器上，所以它们在不同的地址空间中执行，这导致了复杂性。参数和结果也必须传递，这可能很复杂，尤其是在机器不相同的情况下。最后，两台机器中的任何一台都可能崩溃，而且每种可能的故障都会导致不同的问题。尽管如此，其中大部分都可以处理，RPC是一种广泛使用的技术，它是许多分布式系统的基础。

**RPC基础操作**

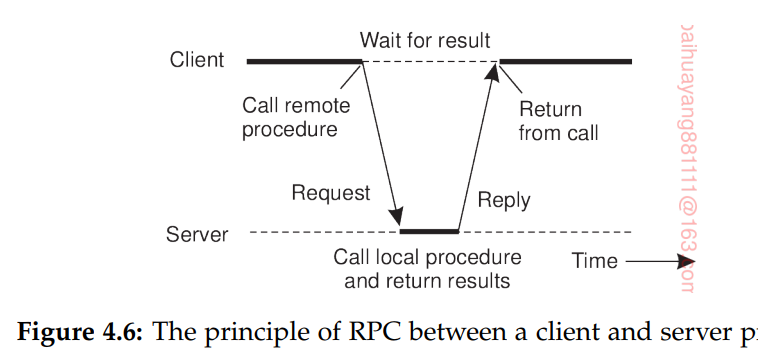
RPC背后的思想是使远程过程调用尽可能看起来像本地过程调用。换句话说，我们希望RPC是透明的——调用过程不应该知道被调用的过程在另一台机器上执行，反之亦然。假设一个程序可以访问一个数据库，该数据库允许它将数据附加到一个存储的列表中，然后返回对修改后列表的引用。该操作如下所示



在传统的(单处理器)系统中，append由链接器从库中提取并插入到目标程序中。原则上，它可以是一个简短的过程，可以通过一些访问数据库的文件操作来实现。

尽管append最终只执行一些基本的文件操作，但它仍然以通常的方式被调用，方法是将其参数推入堆栈。当然，程序员不知道append的实现细节。

RPC以类似的方式实现其透明性。当append实际上是一个远程过程时，会向调用客户机提供另一个版本的append，称为**client stub**。与原来的调用一样，它也是使用普通的调用序列调用的。但是，与原来的方法不同，它不执行append操作。相反，它将参数打包到消息中，并请求将该消息发送到服务器，如图4.6所示。在发送调用后，client stub 调用receive，阻塞自己，知道响应返回。

当消息到达服务器时，服务器的操作系统将其传递给server stub。Server stub 相当于 client stub的服务端：它是一段代码，将来自网络的请求转换为本地过程调用。通常，server stub将调用receive并在等待传入消息时被阻塞。Server stub从消息中解包参数，然后以通常的方式调用服务器过程。从服务器的角度来看，它就好像是由客户机直接调用的——参数和返回地址都位于它们所属的堆栈上，并且没有什么异常。服务器执行其工作，然后以通常的方式将结果返回给调用者(在本例中是server stub)。

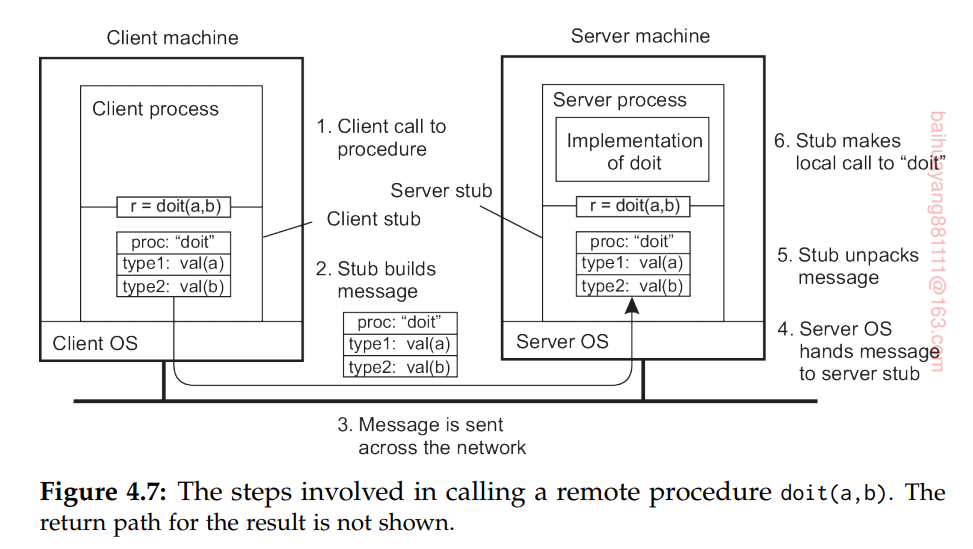
当server stub在调用完成后获得控制权时，它将结果打包到消息中并发送调用将其返回给客户机。之后，server stub通常再次调用receive，等待下一个传入请求。当结果消息到达客户机的机器时，操作系统将其通过之前调用的receive操作传递给client stub，然后解除客户机进程的阻塞。Client stub检查消息，解包结果，将其复制给调用者，然后以通常的方式返回。当调用者在调用append后获得控制权时，它只知道它将一些数据附加到列表中。它不知道这项工作是在另一台机器上远程完成的。

对客户端来说这种无知觉的调用正是整个方案的美妙之处。就其本身而言，远程服务是通过使普通服务本地调用，而不是通过调用send和receive。传递消息的所有细节都隐藏在两个库过程中，就像实际进行系统调用的细节隐藏在传统库中一样。

总而言之，远程过程调用发生在以下步骤中：

1. 客户端调用调用client stub
2. Client stub 在本地操作系统组件消息和发起调用
3. 客户端操作系统向远程操作系统发送消息
4. 远程操作系统将消息传递给server stub
5. Server stub 获取参数并调用服务
6. 服务端完成调用并将结果返回给server stub
7. Server stub将结果打包到消息并调用本地操作系统
8. 服务端操作系统将消息传递给客户端操作系统
9. 客户端操作系统将消息传递给client stub
10. Client stub获取结果返回给客户端。

图4.7显示了一个抽象的双参数过程doit(a,b)的第一步，其中我们假设参数a的类type1，而b的类型为type2。所有这些步骤的最终效果是将客户机过程的本地调用转换为客户机client stub，转换为对服务器过程的本地调用，而客户机或服务器都不知道中间步骤或网络的存在。



**参数传递**

Client stub的作用是获取参数，打包成为消息并发送给server stub。虽然这听起来很简单，但并不像乍一看那么简单。

将参数打包到消息中称为参数封送处理。回到append操作，因此我们需要确保它的两个参数(数据和dbList)通过网络发送，并由服务器正确解释。这里需要注意的是，最终服务器将看到一系列字节，这些字节构成客户机发送的原始消息。然而，这些字节的含义通常不会随消息一起提供额外的信息，更不用说我们将再次面临同样的问题了。元信息应该如何被服务器识别？

除了这个解释问题之外，我们还需要处理字节在内存中的位置在不同机器体系结构之间可能不同的情况。特别是，我们需要考虑这样一个事实，一些机器，如Intel Pentium，从右到左对字节进行编号，而其他许多机器，如较老的ARM处理器，则以另一种方式对字节进行编号(ARM现在同时支持这两种方式)。Intel格式称为**小端**，(旧的)ARM格式称为**大端**。字节排序对于网络也很重要：在这里我们还可以看到，机器在传输(从而接收)位和字节时可能使用不同的顺序。然而，大端字节数通常用于跨网络传输字节。

这个问题的解决方案是转换要发送到独立于机器和网络的格式的数据，然后确保通信双方都希望传输相同的消息数据类型。后者通常可以在编程语言级别上解决。前者是通过使用与机器相关的例程来实现的，这些例程将数据转换为与机器和网络无关的格式。

封送处理和反封送处理都是关于这种到中立格式的转换，并且是远程过程调用的重要组成部分。

我们现在遇到一个难题：指针，或者一般来说，引用是如何传递的？指针仅在使用它的进程的地址空间内有意义。回到我们的append示例，我们声明第二个参数dbList是通过引用存储在数据库中的列表实现的。如果该引用只是指向调用者主内存中某个地方的本地数据结构的指针，我们不能简单地将其传递给服务器。传递的指针值很可能引用完全不同的东西

。

一种解决方案是禁止指针和引用参数。然而，这些都非常重要，因此这种解决方案是非常不可取的。事实上，这通常也没有必要。首先，引用参数通常用于固定大小的数据类型(如静态数组)，或者用于易于在运行时计算其大小的动态数据类型(如字符串或动态数组)。在这种情况下，我们可以简单地复制参数所引用的整个数据结构，用值复制/恢复有效地替换引用复制机制。虽然这在语义上并不总是相同的，但通常已经足够好了。一个明显的优化是，当client stub知道只读取引用的数据时，就不需要在调用结束后将其复制回去。因此，按值复制就足够了。

更复杂的数据类型通常也可以得到支持，当然前提是编程语言支持这些数据类型。例如，Python或Java之类的语言支持用户定义的类，允许语言系统提供这些数据类型的完全自动化封装和解包。但是，请注意，只要我们处理非常大的、嵌套的或其他复杂的动态数据结构，自动(un)封送处理就可能不可用，甚至是不可取的。

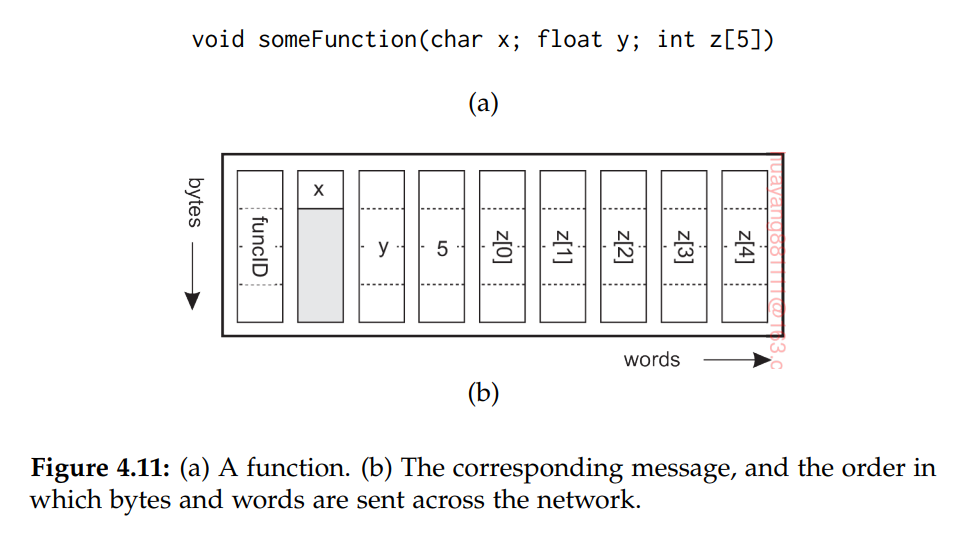
到目前为止所讨论的指针和引用的问题是，它们只在本地有意义:它们引用的内存位置只对调用进程有意义。使用全局引用可以缓解问题：对调用和被调用流程有意义的引用。例如，如果客户机和服务器可以访问相同的文件系统，那么传递一个文件句柄而不是指针就可以了。有一个重要的发现：两个进程都需要确切地知道传递全局引用时应该做什么。换句话说，如果我们考虑的全局引用关联了具体的数据类型，那么我们的调用和被调用的进程的操作应该会有相同的效果。此外，在传递文件句柄时，两个进程应该就具体做什么达成一致。同样，这些问题通常可以通过适当的编程语言支持来解决。

**基于rpc应用程序支持**

根据我们到目前为止所解释的，很明显，隐藏远程过程调用需要调用方和被调用方就它们交换的消息的格式达成一致，并且在传递复杂数据结构时遵循相同的步骤。换句话说，RPC中的双方应该遵循相同的协议，否则RPC将不能正确工作。至少有两种方法可以支持基于rpc的应用程序开发。第一种方法是让开发人员精确地指定需要远程调用什么，从而生成完整的客户端和服务器端stub。第二种方法是将远程过程调用嵌入到编程语言环境中。

**Stub 生成**

考虑图4.11中的someFunction函数。它有三个参数，它有三个参数、一个字符、一个浮点数和一个由五个整数组成的数组。假设一个word是4个字节，RPC协议可能会让一个字符排在一个word的最右边（空出3个空字节），float就是一个字节，数组根据数组的长度代表一组words，如图4.11（b）所示。因此，给定这些规则，someFunction的client stub知道它必须使用图4.11(b)的格式，而server stub知道someFunction的传入消息将具有图4.11(b)的格式。



定义消息格式是RPC协议的一个方面，但这还不够。我们还需要客户机和服务器就简单数据结构的表示达成一致，比如整数、字符、布尔值等等。例如，该协议可以规定整数以2的补码表示，字符以16位Unicode表示，浮点数以IEEE标准#754格式表示，所有内容都以小端字节存储。有了这些附加信息，就可以清楚地解释消息。

现在编码规则固定到最后一位，剩下要做的惟一一件事就是调用者和被调用者就实际的消息交换达成一致。例如，可以决定使用面向连接的传输服务，如TCP/IP。另一种方法是使用不可靠的数据报服务，让客户机和服务器实现一部分的错误控制方案作为RPC协议。实际上，最终还是由开发人员指定首选的底层通信服务。

当RPC协议被定义后，client stub和serverstub需要被实现。幸运的是，相同协议但不同过程的存根通常只在它们与应用程序的接口上不同。接口由一组过程组成，这些过程可以由客户机调用，也可以由服务器实现。接口通常可用与编写客户机或服务器所用的编程语言相同的编程语言来实现(尽管严格地说，这是不必要的)。为了简化问题，通常使用**接口定义语言**(**IDL**)指定接口。然后，在这样的IDL中指定的接口被编译成client stub和server stub，以及适当的编译时或运行时接口。

实践表明，使用接口定义语言大大简化了基于rpc的客户机-服务器应用程序。因为很容易完全生成client和server stub，所以所有基于rpc的中间件系统都提供一个IDL来支持应用程序开发。在某些情况下，使用IDL甚至是强制性的。

**基于语言支持**

到目前为止所描述的方法在很大程度上独立于特定的编程语言。作为一种替代方法，我们还可以将远程过程调用嵌入到语言本身中去。主要的好处是应用程序开发常常变得简单得多。此外，达到高程度的访问透明性通常更简单，因为与参数传递相关的许多问题都可以完全绕过。

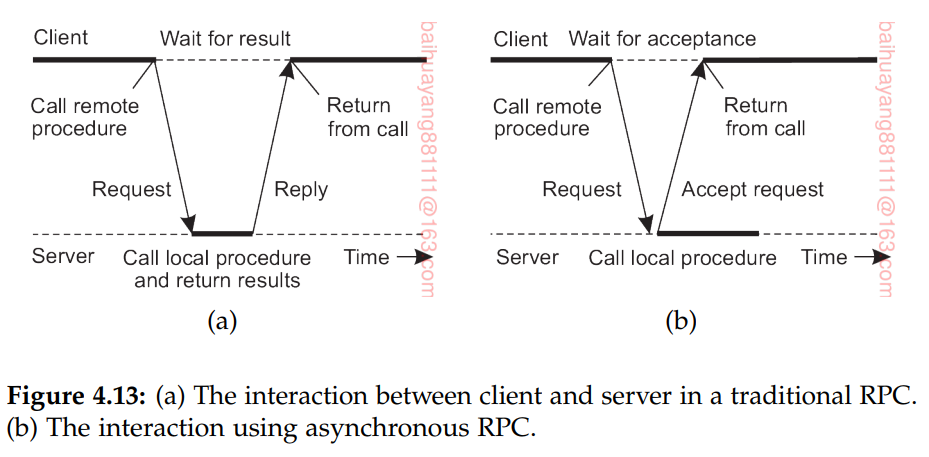
完全嵌入远程过程调用的一个著名示例是Java，其中RPC被称为**远程方法调用(RMI)**。本质上，由自己的(Java)虚拟机执行的客户机可以调用由另一个虚拟机管理的对象的方法。通过简单地读取应用程序的源代码，可能很难甚至不可能看到调用方法的是对本地对象还是对远程对象。

**RPC的变化**

与传统过程调用一样，当客户机调用远程过程时，客户机将阻塞，直到返回响应。当没有返回结果时，这种严格的请求-应答行为是不必要的，或者当需要执行多个rpc时，可能会阻碍效率。在下面，我们将研究到目前为止讨论过的RPC方案的两种变体。

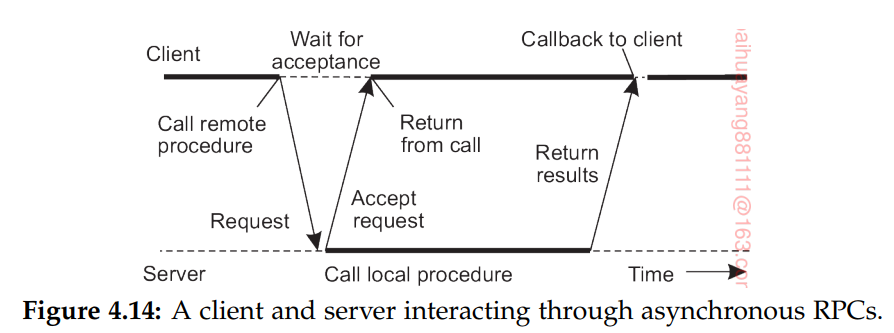
**异步RPC**

为了支持没有结果返回给客户机的情况，RPC系统可以为所谓的**异步RPC**提供一些工具。对于异步RPC，服务器原则上在收到RPC请求时立即向客户机发送一个响应，然后在本地调用所请求的过程。应答作为对客户机的确认，而服务器将继续处理RPC调用。一旦收到服务器的确认，客户机将继续运行，而不会进一步阻塞。图4.13(b)显示了在异步rpc情况下客户机和服务器如何交互。为了进行比较，图4.13(a)显示了正常的请求-应答行为。



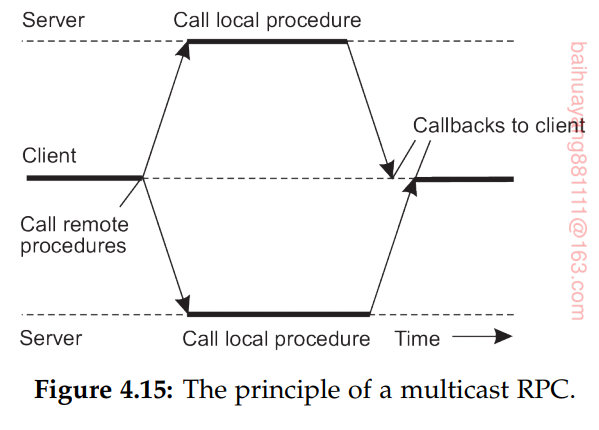
当客户端并不打算等待响应以及在等待期间做些什么工作的时候，异步RPC还是很有用的。典型的情况是，客户机需要独立地与多个服务器联系。在这种情况下，它可以发送一个接一个的调用请求，从而有效地确定服务器或多或少是并行运行的。在发送了所有调用请求之后，客户机可以开始等待返回各种结果。在这种情况下，通过结合回调的异步RPC组织客户机和服务器之间的通信是有意义的，如图4.14所示。在此方案中，也称为延迟同步RPC，客户机首先调用服务器，等待接收，然后继续其他工作。当结果可用时，服务器发送一条响应消息，该消息将导致客户端回调。回调函数是一个用户定义的函数，当发生特殊事件(例如传入消息)时调用它。一个简单的实现是生成一个单独的线程，使得主进程可以继续执行，而新创建的线程被阻塞。当事件发生时，线程被解除阻塞并调用该函数（就是新的线程用来调用回调函数）。

应该注意异步rpc的变体，其中客户机在向服务器发送请求后立即继续执行。换句话说，客户机并不等待服务器接受请求的确认。我们将这种rpc称为单向rpc。这种方法的问题是，当可靠性得不到保证时，客户端不能确定服务端是否会处理其请求。我们将在第八章继续讨论。同样，在延迟同步RPC的情况下，客户机可能轮询服务器以查看结果是否可用，而不是让服务器回调客户机。



**多播RPC**

异步和延迟同步rpc促进了远程过程调用的另一种替代方法，即同时执行多个rpc。**多播RPC**可以归结为向一组服务器发送RPC请求。这个原理如图4.15所示。在本例中，客户机向两台服务器发送一个请求，两台服务器随后独立并行地处理该请求。当完成时，结果返回到发生回调的客户机。



我们需要考虑几个问题。首先，与前面一样，客户机应用程序可能不知道RPC实际上被转发到多个服务器。例如，为了提高容错性，我们可能决定让备份服务器执行所有操作，当主服务器发生故障时，备份服务器可以接管这些操作。通过适当的stub，可以对客户机应用程序完全隐藏已复制的服务器。但是，即使stub也不需要知道服务器是复制的，例如，因为我们使用的是传输层的多播地址。

其次，我们需要考虑如何处理这些响应。特别是，客户机是在收到所有响应之后继续执行，还是只等待一个响应。要看情况而定。当为容错而复制服务器时，我们可能决定只等待第一个响应，或者可能等到大多数服务器返回相同的结果。另一方面，如果服务器被复制来做相同的工作，但是在输入的不同部分上，它们的结果可能需要在客户机继续之前进行合并。同样，这些问题可以隐藏在客户端stub中，但是应用程序开发人员至少必须指定多播RPC的用途。

**例子：DCE RPC**

远程过程调用作为中间件和分布式系统的基础已被广泛采用。在本节中，我们将进一步研究**分布式计算环境(DCE)**，它是由开放软件基金会(OSF)开发的，现在称为开放组。它构成了微软分布式计算环境DCOM [Eddon and Eddon, 1998]的基础，并在Samba中使用，Samba是一个文件服务器和附带的协议套件，允许通过非Windows系统的远程过程调用访问Windows文件系统。

尽管DCE RPC可能不是管理RPC的最现代的方法，但是值得讨论它的一些细节，特别是因为它是大多数传统RPC系统的代表，这些系统使用接口的组合和各种各样的编程语言进行了绑定。我们首先简要介绍DCE，然后考虑它的主要工作原理。关于如何开发基于rpc的应用程序的详细信息可以在[Stevens, 1999]中找到。

**DCE的介绍**

DCE是一个真正的中间件系统，因为它被设计为作为现有(网络)操作系统和分布式应用程序之间的抽象层执行。最初是为Unix设计的，现在已经移植到所有主要的操作系统。其思想是，客户可以获取现有机器的集合，添加DCE软件，然后能够运行分布式应用程序，而不影响现有(非分布式)应用程序。虽然大多数DCE包在用户空间中运行，但是在某些配置中，必须将一部分(分布式文件系统的一部分)添加到底层操作系统的内核中。

底层DCE的编程模型是客户机-服务器模型。用户进程充当客户机来访问服务器进程提供的远程服务。其中一些服务是DCE本身的一部分，但其他服务属于应用程序，由应用程序程序员编写。客户机和服务器之间的所有通信都是通过rpc进行的。

**DCE RPC的目标**

DCE RPC系统的目标是相对传统的。首先也是最重要的，RPC系统使客户机能够通过简单地调用本地过程来访问远程服务。该接口使客户端(即(应用程序)以简单的方式编写的程序，大多数程序员都熟悉。它还使大量现有代码在分布式环境中运行变得很容易，只需很少的更改(如果有的话)。

RPC系统可以对客户机隐藏所有细节，在某种程度上也可以对服务器隐藏所有细节。首先，RPC系统可以自动定位正确的服务器，然后在客户机和服务器软件之间建立通信(通常称为绑定)。它还可以处理两个方向的消息传输，根据需要对它们进行分段和重新组装(例如，如果其中一个参数是一个大数组)。最后，RPC系统可以自动处理客户机和服务器之间的数据类型转换，即使它们运行在不同的体系结构上，并且具有不同的字节顺序。

由于RPC系统能够隐藏细节，客户机和服务器彼此高度独立。客户机可以用Java编写，服务器可以用C编写，反之亦然。客户机和服务器可以运行在不同的硬件平台上，并使用不同的操作系统。还支持各种网络协议和数据表示，所有这些都不需要客户机或服务器的任何干预。

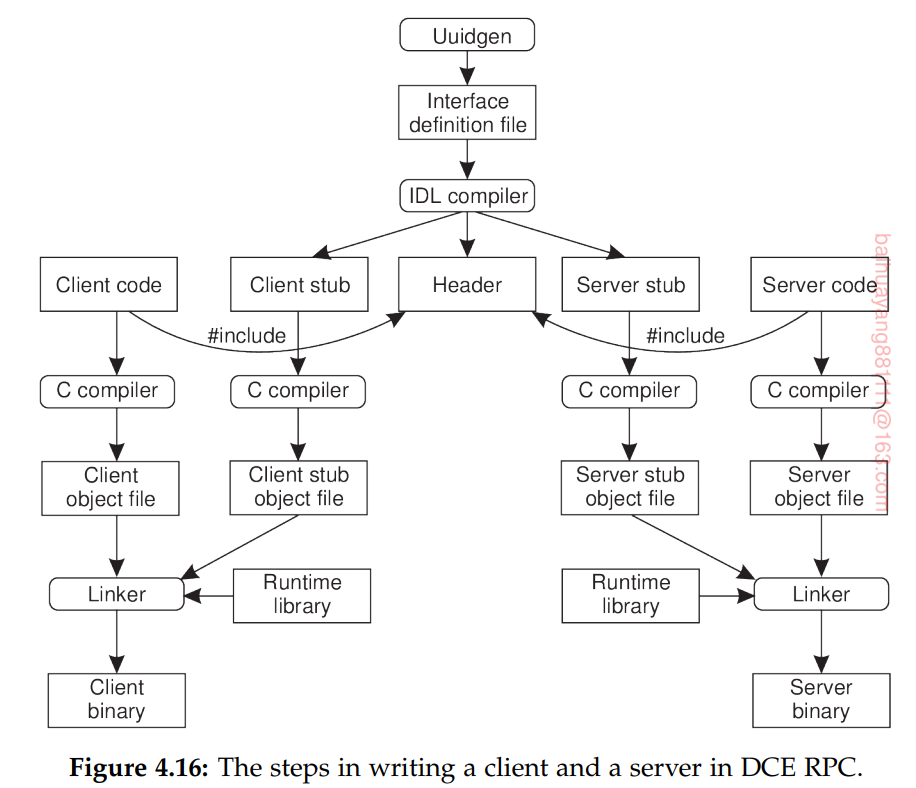
**写一个客户端和一个服务端**

DCE RPC系统由许多组件组成，其中包括语言、库、守护进程和实用程序。这使得编写客户机和服务器成为可能。在本节中，我们将描述各个部分以及它们是如何组合在一起的。图4.16总结了编写和使用RPC客户机和服务器的整个过程。

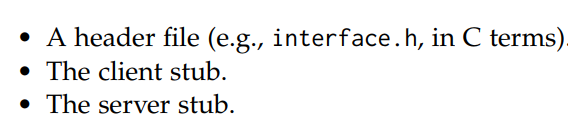
在客户机-服务器系统中，将所有内容结合在一起的粘合剂是接口定义，如接口定义语言(IDL)中所指定的那样。它允许以与ANSI C中的函数原型非常相似的形式声明过程。IDL文件还可以包含类型定义、常量声明和正确封送参数和解封送结果所需的其他信息。理想情况下，接口定义还应该包含过程功能的正式定义，但是这样的定义超出了当前的技术水平，所以接口定义只定义调用的语法，而不是它们的语义。作者最多可以添加一些注释来描述过程的功能。

每个IDL文件中的一个关键元素是指定接口的全局惟一标识符。客户机在第一个RPC消息中发送这个标识符，服务器验证它是否正确。这样，如果客户机无意中试图绑定到错误的服务器，甚至绑定到正确服务器的旧版本，服务器将检测到错误，并且不会进行绑定。

接口定义和惟一标识符在DCE中密切相关。如图4.16所示，编写客户机/服务器应用程序的第一步通常是调用uuidgen程序，要求它生成一个原型IDL文件，其中包含一个接口标识符，保证不会在uuidgen生成的任何地方的任何接口中再次使用。通过在其中编码创建的位置和时间，可以确保惟一性。它由一个128位的二进制数字组成，在IDL文件中以十六进制的ASCII字符串表示。



下一步是编辑IDL文件，填写远程过程及其参数的名称。值得注意的是RPC不是完全透明的。例如，客户机和服务器不能共享全局变量。IDL的规则使得不支持的构造无法解析。当IDL文件完成时，调用IDL编译器来处理它。IDL编译器的输出由三个文件组成：



头文件包含唯一标识符、类型定义、常量定义和函数原型。它应该包含在客户机和服务器代码中(使用#include)。Client stub包含客户机程序将调用的实际过程。这些过程负责收集和打包参数到传出消息中，然后调用运行时系统发送它。Client stub 还负责将响应解包，并将解包的值返回给客户端。Server stub 负责在消息到达时通过服务器上的运行时系统调用过程。然后，这些调用执行该工作的实际服务过程。

下一步是应用程序编写器编写客户机和服务器代码。然后编译这两个过程，以及两个stub过程。然后将生成的客户机代码和client stub对象文件链接到运行时库，为客户机生成可执行的二进制文件。类似地，服务器代码和server stub被编译和链接，以生成服务器的二进制文件。在运行时，启动客户机和服务器，以便实际执行应用程序。

**绑定客户端到服务端**

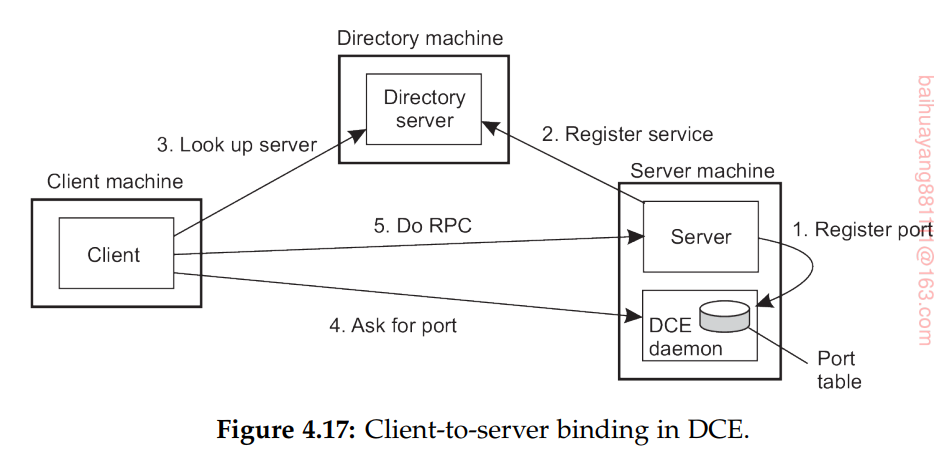
要允许客户机调用服务器，服务器必须已注册并准备接受传入的调用。服务器的注册使客户机能够定位服务器并绑定到它。查找服务器的位置分两步完成：

1. 定位服务器的机器。

2. 在那台机器上找到正确额服务。

第二步有点微妙。基本上，归根结底，要与服务器通信，客户机需要知道服务器机器上的一个端口，它可以将消息发送到该端口。服务器的操作系统使用端口来区分不同进程的传入消息。在DCE中，由称为**DCE守护进程**的进程在每台服务器机器上维护一个(服务、端口)对的表。在传入请求可用之前，服务器会去向操作系统索要端口。然后，它向DCE守护进程注册此端口。DCE守护进程将此信息(包括服务器使用的协议)记录在端口表中，供将来使用。

服务器还向目录服务注册，方法是提供服务器机器的网络地址和可以查找服务器的名称。将客户机绑定到服务器的过程如图4.17所示。



让我们假设客户机希望绑定到一个名为/local/multimedia/video/movies的本地视频服务器。它将这个名称传递给目录服务器，目录服务器返回运行视频服务器的机器的网络地址。然后客户机转到该机器上的DCE守护进程(它有一个众所周知的端口)，并要求它在其端口表中查找视频服务器的端口。有了这些信息，RPC现在就可以执行了。在后续的rpc上不需要这种查找。DCE还允许客户在需要时对合适的服务器进行更复杂的搜索。在机密性或数据完整性至关重要的情况下，安全RPC也是一种选择。

**执行RPC**

实际RPC是透明地以通常的方式执行的。Client stub将参数封送到运行时库，以便使用绑定时选择的协议进行传输。当消息到达服务器端时，根据传入消息中包含的端口将其路由到正确的服务器。运行时库将消息传递给server stub，server stub解组参数并调用服务器。响应则按相反的路线返回。

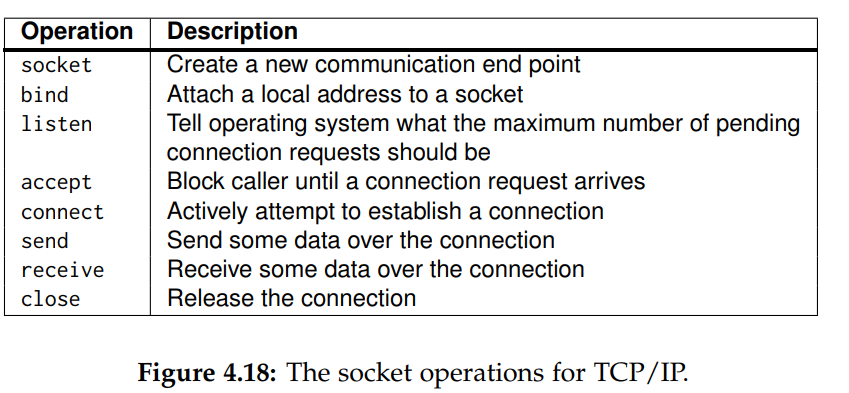
DCE提供了几个语义选项。默认是最多一次操作，在这种情况下，即使存在系统崩溃，也不会执行超过一次的调用。在实践中，这意味着如果服务器在RPC期间崩溃，然后迅速恢复，客户端不会重复该操作，因为担心它可能已经执行过一次。或者，也可以将远程过程标记为幂等过程(在IDL文件中)，在这种情况下，可以多次重复该过程而不会造成影响。例如，可以反复尝试从文件中读取指定的块，直到成功为止。当幂等RPC由于服务器崩溃而失败时，客户机可以等待服务器重新引导，然后重试。其他语义也可用(但很少使用)，包括将RPC广播到本地网络上的所有计算机。在第8.3节中，当讨论存在故障的RPC时，我们将回到RPC语义。

4.3 面向消息通信

许多分布式系统和应用程序都直接构建在传输层提供的简单面向消息的模型之上。为了更好地理解和欣赏作为中间件解决方案一部分的面向消息的系统，我们首先讨论通过传输层套接字进行消息传递。

特别注意标准化传输层的接口，使程序员可以通过一组简单的操作使用它的整个(消息传递)协议套件。此外，标准接口使应用程序移植到另一台机器变得更容易。作为一个例子，我们简要讨论了上世纪70年代在Berkeley Unix中引入的套接字接口，该接口已被用作POSIX标准(只有很少的修改)。

从概念上讲，套接字是一个通信端点，应用程序可以将要通过基础网络发送的数据写入该端点，并从该端点读取传入的数据。套接字就是在本地操作系统通过传输协议基于端口传输数据的抽象。在下面的文本中，我们主要关注TCP的套接字操作，如图4.18所示。



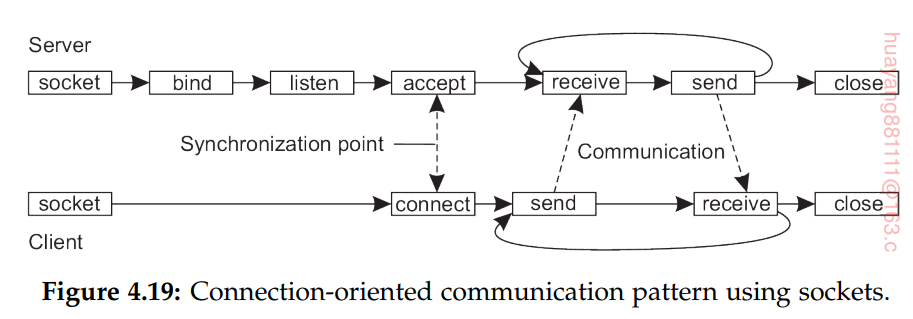
服务端通常按顺序执行前四个操作。当调用socket操作时，调用者为特定的传输协议创建一个新的通信端点。创建通信端点意味着本地操作系统为发送和接收指定协议的消息保留信息。

bind操作将本地地址与新创建的套接字关联起来。例如，服务器应该将其机器的IP地址与(可能已知的)端口号绑定到套接字。绑定告诉操作系统，服务器只想接收指定地址和端口上的消息。在面向连接的通信中，地址用于接收传入的连接请求。

listen操作只在面向连接的通信的情况下调用。它是一个非阻塞调用，它允许操作系统为一个特定数量的请求保留一定量的缓冲区。

Accept将阻塞调用者，直到连接请求到达。当一个请求到达时，本地操作系统创建一个与原始套接字具有相同属性的新套接字，并将其返回给调用者。例如，这种方法将允许服务器复制一个进程，该进程随后将通过新连接处理实际通信。服务器可以返回并等待原始套接字上的另一个连接请求。

现在让我们看看客户端。这里，也必须首先使用套接字操作创建套接字，但是不需要显式地将套接字绑定到本地地址，因为操作系统可以在建立连接时动态分配端口。connect操作要求调用方指定要将连接请求发送到传输层地址。最后，使用套接字时关闭连接是对称的，通过让客户机和服务器调用关闭操作来断开连接。尽管该规则有许多例外，但是使用套接字进行面向连接的通信的客户机和服务器遵循的一般模式如图4.19所示。关于在Unix中使用套接字和其他接口进行网络编程的详细信息可以在[Stevens, 1998]中找到。



**Advanced transient messaging**

基于套接字的瞬态消息传递标准方法非常基础，因此非常脆弱：很容易犯错误。实际上，我们常常需要更高级的面向消息的通信方法来简化网络编程，扩展现有网络协议提供的功能，更好地利用本地资源，等等。

**使用消息传递模式:ZeroMQ**

使网络编程更容易的一种方法是基于以下观察:许多消息传递的应用程序或它们的组件可以根据几个简单的通信模式有效地组织起来。通过随后为每个模式提供增强的套接字，可以更容易地开发网络化、分布式的应用程序。

与Berkeley方法类似，ZeroMQ也提供了套接字，所有通信都通过套接字进行。实际的消息传输通常通过TCP连接进行，所有通信本质上都是面向连接的，这意味着在消息传输发生之前，首先要在发送方和接收方之间建立连接。然而，建立和维护连接主要是在底层进行的:应用程序程序员不需要为这些问题操心。为了进一步简化问题，套接字可以绑定到多个地址，从而有效地允许服务器通过一个接口处理来自不同来源的消息。例如，服务器可以使用一个阻塞接收操作侦听多个端口。因此，ZeroMQ套接字可以支持多对一通信，而不是像标准Berkeley套接字那样只支持一对一通信。ZeroMQ套接字还支持一对多通信，即多播。ZeroMQ的关键之处在于通信是异步的:发送方通常在向底层通信子系统提交消息后继续发送。这种异步的面相消息的通信有一个有意思的副作用，发送方可以在接收方未启动或者未接受传入请求时，向接收方请求连接并发送消息。当然，发生的情况是连接请求和后续消息在发送方一侧排队，而ZeroMQ库中的一个单独线程将负责最终建立连接并将消息传输给接收方。

为了简化问题，ZeroMQ在基于套接字的通信中通过配对套接字建立了更高层次的抽象:用于发送消息的特定套接字类型与用于接收消息的对应套接字类型配对。每对套接字类型对应一个通信模式。ZeroMQ支持的三个最重要的通信模式是请求-应答、发布-订阅和管道。

请求-应答模式用于传统的客户机-服务器通信，就像通常用于远程过程调用一样。客户机应用程序使用请求套接字(REQ类型)向服务器发送请求消息，并期望服务器以适当的响应进行响应。这里假定服务器使用应答套接字(REP类型)。通过避免调用listen操作和accept操作，request-reply模式简化了开发人员的工作。此外，当服务器接收到消息时，要发送的后续调用将自动针对原始发送方（就是等着发送发的下一次发送）。同样，当客户机在发送消息之后调用recv操作(用于接收消息)时，ZeroMQ假定客户机正在等待原始接收者（就是服务端）的响应。意，这种方法在图4.21的本地sendrecv操作中得到了有效的编码，图4.21在注释4.8中讨论过。

对于**发布-订阅**模式，客户机订阅由服务器**发布**的特定消息。在第1章讨论基于事件的协调时，我们简要地遇到了这种模式。实际上，只传输客户机订阅的消息。如果服务端推送了没有订阅的消息，则此消息将被丢弃。这种模式的最简单形式是建立从服务器到多个客户机的多播消息。假定服务器使用类型为PUB的套接字，而每个客户机必须使用SUB类型的套接字。每个客户端套接字都连接到服务器的套接字。默认情况下，客户端不订阅任何消息。这意味着，只要不提供显式订阅，客户机就不会收到服务器发布的消息。

最后，是管道模式，它的特点是有的进程会以其他进程的输出结果作为输入。管道模式的本质是，输出进程并不真正关心哪个进程会将其结果作为输入：即只要输入进程拿到第一个可用的就可以了。同样的，输入进程只关心所有输出进程中，第一个使得输入结果可用的那一个。因此，管道模式的目的是保持尽可能多的流程工作，尽可能快地将结果通过流程管道。

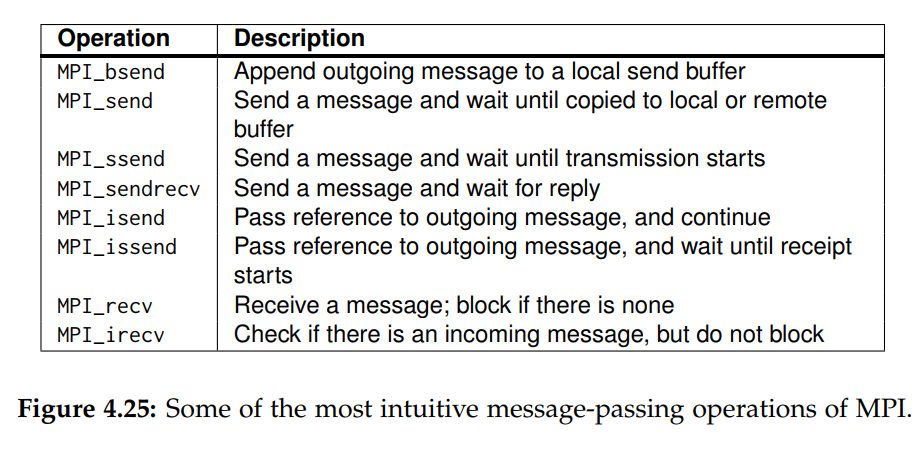
**消息传递接口(MPI)**

随着高性能多计算机的出现，开发人员一直在寻找面向消息的操作，以便轻松编写高效的应用程序。这意味着一些抽象的操作更方便，更利于开发，而其实现只会产生很小的开销。套接字在两个方面有所不足。首先，他们支持的简单的发送和接收操作，处在了错误的抽象级别。其次，套接字被设计成使用诸如TCP/IP之类的通用协议栈在网络之间进行通信。而这些协议不适用于为高速互连网络开发的专有协议，例如用于高性能服务器集群的协议。这些协议需要一个接口来处理更高级的特性，比如不同形式的缓冲和同步。其结果是，大多数互连网络和高性能多核计算机都附带了专有的通信库。这些库提供了丰富的高级的和通常有效的通信操作。然而，所有库都是相互不兼容的，所以应用程序开发人员现在有一个可移植性问题。

与硬件和平台无关的需求最终导致了消息传递标准的定义，即简单地称为**messagepging接口**或**MPI**。MPI是为并行应用程序而设计的，因此适合于瞬态通信（瞬态通信，即通信双方无缓冲区，消息发送和接收方都必须在运行）。它直接使用底层网络。此外，它假定严重的故障(如进程崩溃或网络分区)是致命的，不需要自动恢复。

MPI假设通信发生在一个已知的进程组中。每个组分配一个标识符。组中的每个进程还被分配一个(本地)标识符。因此，一个(groupID, processID)对惟一地标识消息的源或目的地，并用于代替传输级别的地址。计算中可能涉及多个进程组，这些进程组可能相互重叠，并且都在同时执行。

MPI的核心是支持瞬时通信的消息传递操作，其中最直观的操作总结在图4.25中。瞬态异步通信通过一些列MPI\_bsend操作支撑。发送方提交要传输的消息，该消息通常首先复制到MPI运行时系统中的本地缓冲区。复制消息后，发送方继续其他操作。当接受者调用接收操作时，本地MPI运行系统会处理传输，并将本地消息从缓冲区删除。



还有一个名为MPI\_send的阻塞发送操作，其语义依赖于实现。MPI\_send操作可以阻塞调用方，直到指定的消息被复制到发送方一侧的MPI运行时系统，或者直到接收方启动接收操作。通过MPI\_ssend操作，发送方可以通过同步通信进行阻塞，直到其请求被接受以进行进一步处理。最后，还支持最强形式的同步通信：当发送方调用MPI\_sendrecv时，它向接收方发送请求并阻塞，直到接收方返回应答。基本上，这个操作对应于一个普通RPC。

MPI\_send和MPI\_ssend都有一些变体，可以避免将消息从用户缓冲区复制到本地MPI运行时系统内部的缓冲区。这些变体本质上对应于异步通信的一种形式。使用MPI\_isend，发送者传递一个指向消息的指针，然后MPI运行时系统负责通信。发送者立即继续其他操作。为了防止在通信完成之前覆盖消息，MPI提供了检查是否完成的操作，甚至在需要时进行阻塞。与MPI\_send一样，MPI\_ssend未指定消息是实际传输到接收方，还是仅仅由本地MPI运行时系统复制到内部缓冲区。

同样，对于MPI\_issend，发送方也只传递一个指向MPI运行时系统的指针。当运行时系统指示它已经处理了消息时，则发送方将获得一个保证，保证接收方已经接受了消息并正在处理它。

调用操作MPI\_recv接收消息;它阻塞调用者，直到消息到达。还有一种异步变体，称为MPI\_irecv，接收方通过它表示准备接受消息。接收者可以检查消息是否确实到达，或者在到达之前阻塞。

MPI通信操作的语义并不总是直接的，有时可以在不影响程序正确性的情况下交换不同的操作。之所以支持这么多不同形式的通信，主要原因是它为MPI系统的实现人员提供了足够的可能性来优化性能。愤世嫉俗者可能会说，提交者无法做出集体决定，所以它把一切都投了进去。到目前为止，MPI已经是第三个版本，有超过440个操作可用。由于是为高性能并行应用程序设计的，因此可能更容易理解其多样性。

**面向消息的持久化通信**

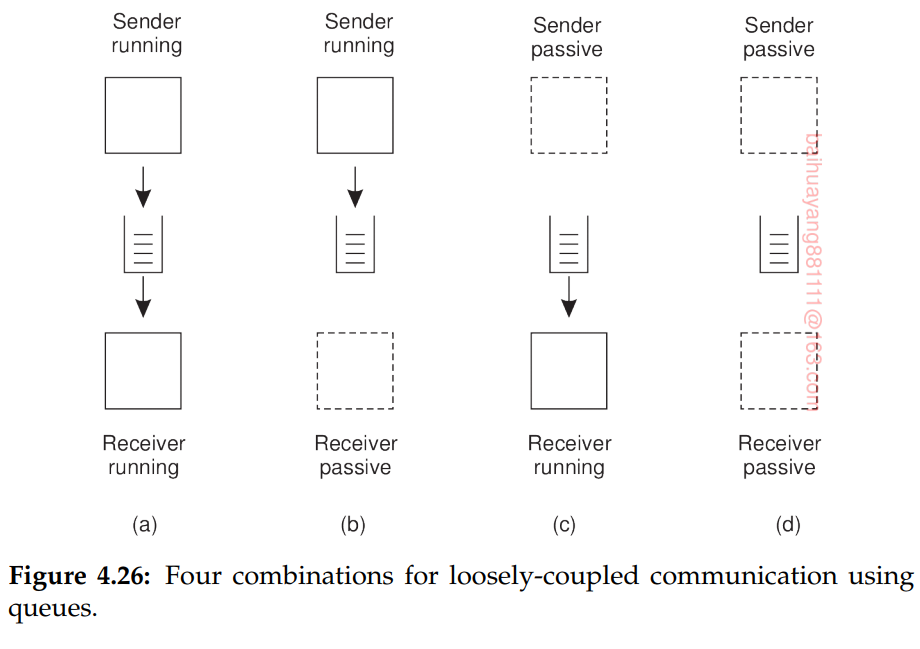
现在我们来到一个重要的面向消息的中间件服务类，通常称为**消息队列系统**，或者仅仅是面向**消息的中间件(MOM)。**消息队列系统为持久异步通信提供了广泛的支持。这些系统的本质是为消息提供中期存储能力，而不需要发送方或接收方在消息传输期间处于活动状态。与套接字和MPI的一个重要区别是，消息队列系统的目标通常是支持消息传输，允许花费几分钟而不是几秒或几毫秒。

**消息队列模型**

消息队列系统的基本思想是应用程序通过在特定队列中插入消息进行通信。这些消息通过一系列通信服务器转发，并最终交付到目的地，即使在发送消息时它处于关闭状态。实际上，大多数通信服务器都是彼此直接连接的。换句话说，消息通常直接传输到目标服务器。原则上，每个应用程序都有自己的私有队列，其他应用程序可以将消息发送到该队列。队列只能由其关联的应用程序读取，但是多个应用程序也可以共享一个队列。

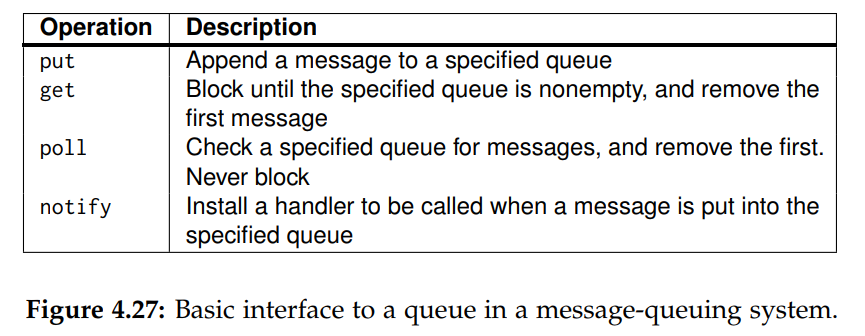
消息队列系统的一个重要方面是，通常只向发送方保证其消息最终将插入到接收方的队列中。 对于何时、甚至是否真正读取消息(完全由接收方的行为决定)，都没有给出任何保证。

这些语义允许通信在时间上松散耦合。因此，当消息被发送到其队列时，接收器不需要执行。同样，发送方不需要在接收方接收其消息时执行。发送方和接收方可以完全独立地执行。事实上，一旦消息被存储到队列中，它就会一直保留在队列中，直到被删除，而不管它的发送方还是接收方正在执行。这给出了关于发送方和接收方执行模式的四种组合，如图4.26所示。



在图4.26(a)中，发送方和接收方都在消息的整个传输过程中执行。在图4.26(b)中，只有发送方在执行，而接收方是被动的，即处于无法进行消息传递的状态。尽管如此，发送方仍然可以发送消息。发送方是被动的和接收方正在执行的组合如图4.26(c)所示（这里的被动 应该指的是机器不运转）。在这种情况下，接收方可以读取发送给它的消息，无论他们的发送方是否在运行。最后，在图4.26(d)中，我们看到了这样一种情况，即即使发送方和接收方是被动的（停止运行），系统也在存储(并可能传输)消息。有人可能会说，只有支持最后一个配置，消息队列系统才能真正提供持久消息传递。

原则上，消息可以包含任何数据。从中间件的角度来看，惟一重要的方面是消息得到了正确的处理（从正确的发送方 到达 正确的接收方）。实际上，寻址是通过系统范围的目标队列的惟一名称来完成的。在某些情况下，消息大小可能是有限的，尽管底层系统也可能以对应用程序完全透明的方式负责对大型消息进行分段和组装。这种方法的一个效果是，提供给应用程序的基本接口可以非常简单，如图4.27所示。



发送方调用put操作，将消息传递给要附加到指定队列的底层系统。这是一个非阻塞调用。get操作是一个阻塞调用，通过该调用，授权的进程可以删除指定队列中最长的挂起消息。只有当队列为空时，进程才被阻塞。此调用的变体允许搜索队列中的特定消息，例如，使用优先级或匹配模式。非阻塞的操作由poll代替。如果队列是空的，或者无法找到特定的消息，则调用过程将继续。

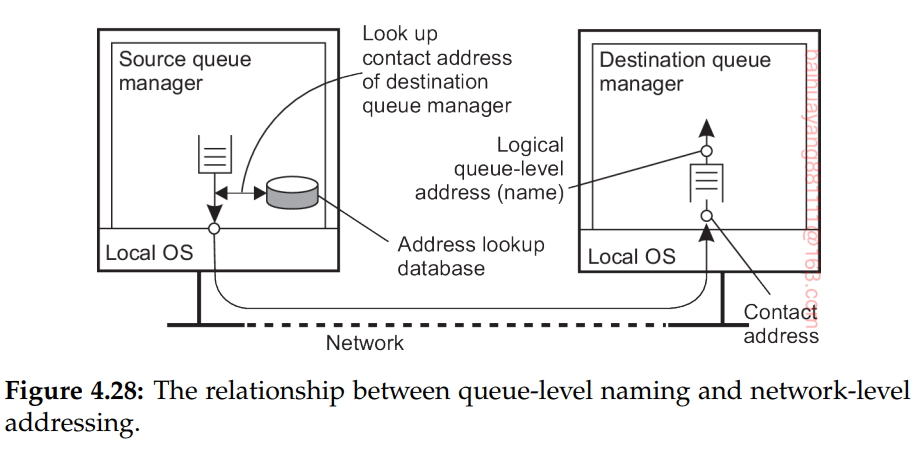
最后，大多数队列系统还允许为处理程序安装回调函数，每当将消息放入队列时，都会自动调用回调函数。回调还可以用于自动启动一个进程，如果当前没有进程执行，该进程将从队列中获取消息。这种方法通常是通过接收方的守护进程来实现的，守护进程不断地监视队列中传入的消息并相应地进行处理。

**消息队列系统的一般架构**

现在让我们仔细看看一般的消息队列系统是什么样子的。首先，队列由队列管理器管理。其次，根据经验，应用程序只能将消息放入本地队列。同样，只从本地队列中获得消息。因此，如果处理应用程序a的队列的队列管理器QMA作为一个单独的进程运行，那么QMA和a进程通常都将放在同一台机器上，或者在最坏的情况下放在同一LAN上。还要注意，如果所有队列管理器都链接到各自的应用程序中，我们就不能再谈论持久异步消息传递系统了。

如果应用程序只能将消息放入本地队列，那么显然每个消息都必须携带有关其目的地的信息。队列管理器的任务是确保消息到达目的地。这给我们带来了一些问题。

首先，我们需要考虑目标队列的地址。显然，为了增强位置透明性，队列最好具有与位置无关的逻辑名称。假设队列管理器（应该是接受队列管理器）是一个使用逻辑名称的单独的进程实现的，意味着每个名称都应该与一个地址相关联，例如(主机、端口)对，并且名称到地址的映射很容易被队列管理器（应该是发送队列管理器）使用，如图4.28所示。实际上，联系地址包含更多信息，尤其是要使用的协议，如TCP或UDP。我们在高级套接字的例子中遇到过这样的联系地址，例如，Note 4.9。



我们需要考虑的第二个问题是，名称到地址的映射实际上如何对队列管理器可用。一种常见的方法是简单地将映射实现为查找表，并将该表复制到所有管理器。显然，这将导致维护问题，每次添加或命名新队列时，都需要更新许多表(如果不是所有表)。处理这些问题的方法有很多，我们将在第5章中讨论。

这就引出了第三个问题，与有效维护名称到地址的映射有关。我们含蓄地假设，如果队列管理器QMA知道管理器QMB上的目标队列，那么QMA可以直接联系QMB来传输消息。实际上，这意味着每个队列管理器的联系地址应该为所有其他队列管理器所知。显然，在处理非常大的消息队列系统时，我们将遇到可伸缩性问题。实际上，通常有一些特殊的队列管理器作为路由器来操作:它们将传入的消息转发给其他队列管理器。通过这种方式，消息队列系统可以逐渐成长为一个完整的、应用程序级的覆盖网络（覆盖网络是一种创建在另一网络之上的计算机网络）。

如果只有少数路由器需要知道网络拓扑，那么源队列管理器只需要知道向哪个相邻路由器转发消息。比如R，给定一个目标队列，路由器R可能只需要跟踪它的相邻路由器，就可以知道将消息转发到哪里。当然，我们仍然需要为所有队列管理器(包括路由器)提供名称到地址的映射，但是不难想象，这样的表可以更小，更容易维护。

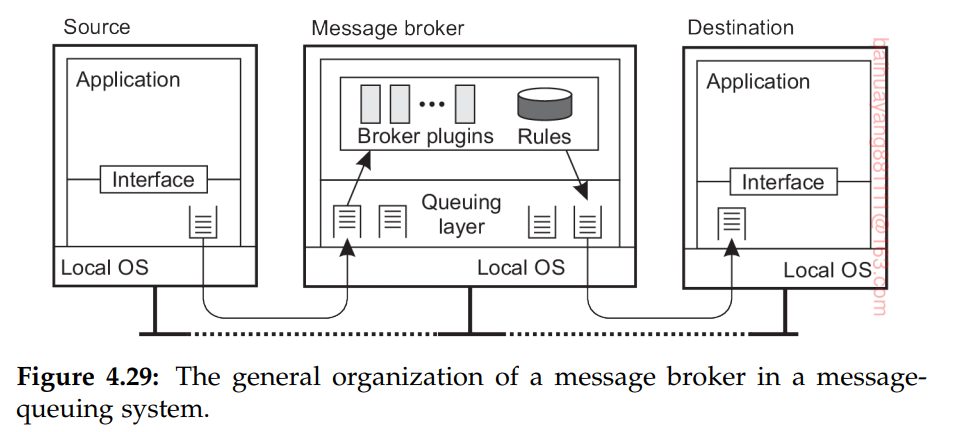
**Message brokers**

消息队列系统的一个重要应用领域是将现有的和新的应用程序集成到一个统一的分布式信息系统中。如果我们假设与应用程序的通信是通过消息进行的，那么集成要求应用程序能够理解它们接收到的消息。实际上，这要求发送方以与接收方相同的格式发送出去的消息，而且它的消息遵循与接收方所期望的相同的语义。发送方和接收方本质上需要使用相同的语言，即遵循相同的消息传递协议。这种方法的问题是，当每次将一个应用程序A添加到具有自己消息传递协议的系统中时，对于其他的与A通信的应用程序B，我们将需要提供转换各自消息的方法。在一个有N个应用程序的系统中，我们需要N×N个消息协议转换器。

另一种选择是就公共消息协议达成一致，就像对传统网络协议所做的那样。不幸的是，这种方法通常不适用于消息队列系统。问题主要是由于这些系统的抽象级别。只有当使用该协议的进程集合具有足够的共同点时，公共消息传递协议才有意义。如果组成分布式信息系统的应用程序集合高度多样化(通常是这样)，那么发明一种一刀切的解决方案是行不通的。

如果我们只关注消息的格式和含义，那么可以通过提高抽象级别来实现共同性，就像XML消息一样。在这种情况下，消息在它们自己的组织中携带信息，并且将他们的描述内容的方式标准化。因此，应用程序可以提供关于可以自动处理的消息组织的信息。当然，这些信息通常是不够的:我们还需要确保消息的语义被很好地理解。

考虑到这些问题，一般的方法是学会接受差异，并尝试提供使转换尽可能简单的方法。在消息队列系统中，转换由队列网络中的特殊节点(称为**消息代理**)处理。消息代理充当消息队列系统中的应用程序级网关。它的主要目的是转换传入消息，以便目标应用程序能够理解它们。注意，对于消息队列系统，message broker只是另一个应用程序，如图4.29所示。换句话说，消息代理通常不被认为是队列系统的组成部分。



消息代理可以像消息的重新格式化程序一样简单。例如，假设传入消息包含数据库中的表，其中记录由特殊的记录结束分隔符分隔，并且记录中的字段具有已知的固定长度。如果目标应用程序希望记录之间有不同的分隔符，并且希望字段的长度可变，则可以使用message broker将消息转换为目标所期望的格式。在更高级的设置中，message broker可以充当应用程序级网关，其中对多个应用程序的消息传递协议信息进行了编码。通常，对于每一对应用程序，我们将有一个单独的子程序，能够在两个应用程序之间转换消息。在图4.29中，这个子程序被画成一个插件，以强调这些部分可以动态地插入或从代理中删除。

最后，请注意，在许多情况下，message broker用于**高级企业应用程序集成(EAI)**，正如我们在第1.3节中讨论的那样。在这种情况下，代理负责根据正在交换的消息匹配应用程序，而不是(仅)转换消息。在这种发布-订阅模型中，应用程序以发布的形式发送消息。特别是，它们可以发布关于主题X的消息，然后将其发送给代理。声明了对主题X上的消息感兴趣的应用程序(即订阅了这些消息的应用程序)将从代理接收这些消息。更高级的中介形式也是可能的。message broker的核心是依赖于将一种类型的消息转换为另一种类型的规则的存储库。问题是定义规则和开发插件。大多数message broker产品都带有复杂的开发工具，但底线仍然是存储库需要由专家填充。这里我们将看到一个完美的例子就是，商业产品经常会误导说能提供“智慧”，而实际上，这些智慧都是在那些专家的脑袋里的。

**示例:IBM的WebSphere消息队列系统**

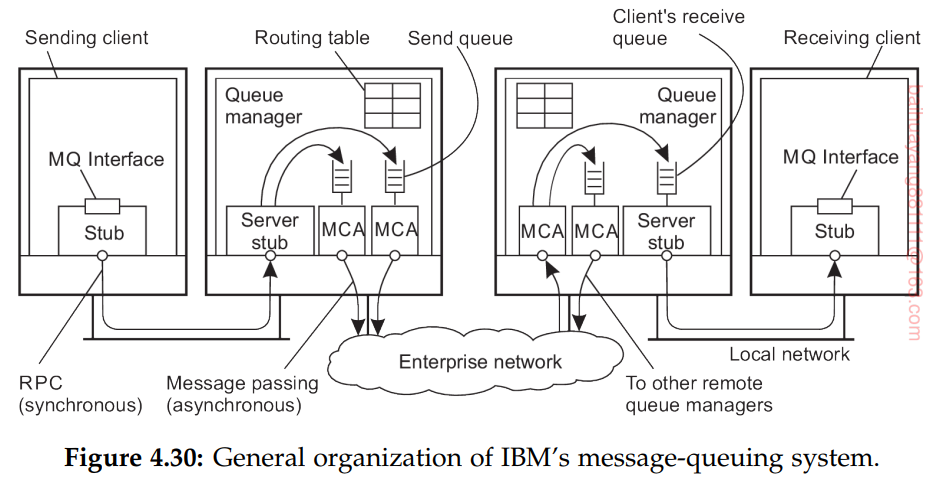
为了帮助理解消息队列系统在实践中是如何工作的，让我们看一下IBM WebSphere中的消息队列系统。以前称为MQSeries，现在称为WebSphere MQ。关于WebSphere MQ有大量的文档，在下面我们只能求助于基本原则。

**概述**

MQ队列网络的基本架构非常简单，如图4.30所示。所有队列都由队列管理器管理。队列管理器负责从其发送队列中删除消息，并将这些消息转发给其他队列管理器。同样，队列管理器负责处理传入的消息，方法是从底层网络中提取它们，然后将每个消息存储在适当的输入队列中。为了让您了解消息传递的含义:消息的最大默认大小为4 MB，但是这可以增加到100 MB。队列通常被限制为2 GB的数据，但是根据底层操作系统的不同，这个最大值可以很容易地设置得更高。

队列管理器通过**消息通道**成对连接，消息通道是传输级连接的抽象。消息通道是发送队列管理器和接收队列管理器之间的单向可靠连接，队列消息通过它传输。例如，基于internet的消息通道被实现为TCP连接。消息通道的两端都由消息通道代理(MCA)管理。发送端MCA基本上只是检查消息的发送队列，将其包装到传输级别的数据包中，然后沿着连接将其发送到相关的接收端MCA。同样，接收端MCA的基本任务是侦听传入包，解包，然后将解包的消息存储到适当的队列中。

队列管理器可以链接到与其管理队列的应用程序相同的进程中。在这种情况下，队列对应用程序隐藏在标准接口后面，但是应用程序可以直接有效地操纵队列。另一种组织是在不同的机器上运行队列管理器和应用程序。在这种情况下，当队列管理器位于同一台机器上时，应用程序将获得相同的接口。但是，该接口被实现为使用传统的基于rpc的同步通信与队列管理器通信的代理。通过这种方式，MQ基本上保留了只能访问应用程序本地队列的模型（这段不理解）。



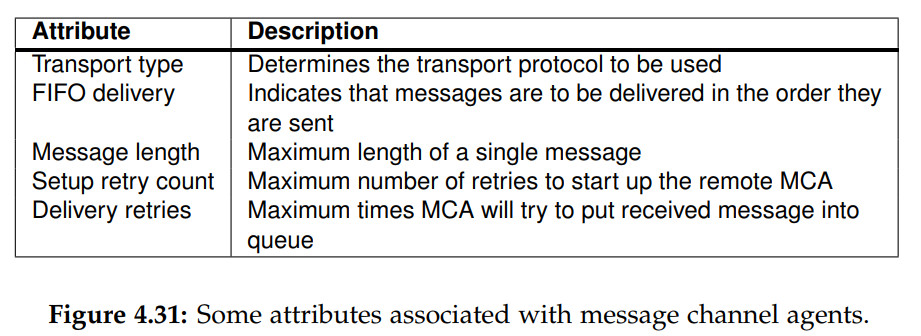
**通道**

MQ的一个重要组件由消息通道组成。每个消息通道都有一个关联的发送队列，从该队列获取应该传输到另一端的消息。只有当发送和接收MCA都启动并运行时，才能沿着通道进行传输。除了手动启动两个MCAs，有几种启动通道的替代方法，下面将讨论其中的一些。

另一种方法是让应用程序通过激活发送或接收MCA直接启动其通道的末端。然而，从透明度的角度来看，这不是一个很有吸引力的选择。启动发送MCA的更好方法是配置通道的发送队列，以便在消息首次放入队列时触发触发器。该触发器与启动发送MCA的处理程序相关联，以便可以从发送队列中删除消息。

另一种选择是通过网络启动MCA。特别是，如果通道的一端已经处于活动状态，它可以发送一条控制消息，请求启动另一个MCA。这样的控制消息被发送到一个守护进程，该守护进程监听要启动的另一个MCA的相同机器上的一个已知地址。如果在一段时间内不再将任何消息放入发送队列，通道在这段时间过后将自动停止。

每个MCA都有一组相关属性，这些属性决定了通道的总体行为。图4.31中列出了一些属性。发送和接收MCA的属性值应该是兼容的，在设置通道之前可能要先协商。例如，两个MCAs显然应该支持相同的传输协议。不可协商的属性的一个例子是，消息是否按照与放入发送队列相同的顺序交付。如果一个MCA想要FIFO交付，另一个必须遵守。可协商属性值的一个例子是最大消息长度，它将被简单地选择为最小值。



**消息传输**

要将消息从一个队列管理器传输到另一个队列管理器(可能是远程队列管理器)，需要每个消息携带其目标地址，为此使用传输头。MQ中的地址由两部分组成。第一部分要向其传递消息的队列管理器的名称组成。第二部分是将消息附加到的管理器下的目标队列的名称。第二部分是将消息附加到的管理器下的目标队列的名称。

除了目标地址之外，还需要指定消息应该遵循的路由。路由规范是通过提供要向其添加消息的本地发送队列的名称来完成的。因此，没有必要在消息中提供完整的路由。回想一下，每个消息通道只有一个发送队列。通过告诉要将消息附加到哪个发送队列，我们可以有效地指定要将消息转发到哪个队列管理器。

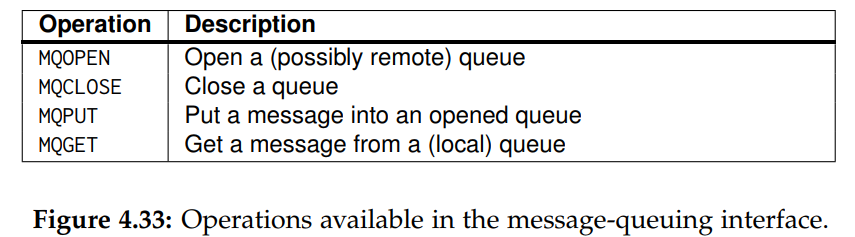
在大多数情况下，路由显式地存储在路由表中的队列管理器中。路由表中的条目是一对(destQM,sendQ)，其中destQM是目标队列管理器的名称，sendQ是本地发送队列的名称，应该将该队列管理器的消息附加到该队列中。

在到达目的地之前，可能需要跨多个队列管理器传输消息。每当这样的中间队列管理器接收到消息时，它只从消息头提取目标队列管理器的名称，并执行routingtable查找，以找到应该将消息附加到其中的本地发送队列。

重要的是要认识到，每个队列管理器都有一个系统范围内惟一的名称，可以有效地用作该队列管理器的标识符。使用这些名称的问题是，替换队列管理器或更改其名称将影响所有向其发送消息的应用程序。使用队列管理器名称的本地别名可以缓解问题。在队列管理器QM1中定义的别名在队列管理器QM2中是另一个名称，但它只对连接到QM1的应用程序可用（就是QM1的专用别名）。别名允许对队列使用相同的(逻辑)名称，即使该队列的队列管理器发生了更改。更改队列管理器的名称需要在所有队列管理器中更改其别名。但是，应用程序可以不受影响。

使用路由表和别名的原理如图4.32所示。例如，链接到队列管理器QMA的应用程序可以使用本地别名LA1引用远程队列管理器。队列管理器将首先在alias表中查找实际目的地，以找到它是队列管理器QMC。到QMC的路由在routing表中找到，该表声明QMC的消息应该附加到传出队列SQ1，后者用于将消息传输到队列管理器QMB。后者将使用其路由表将消息转发给QMC（上面这几段不大懂啊）。

遵循这种路由和别名的方法，可以得到一个相对简单的编程接口，称为**消息队列接口(MQI)**。图4.33总结了MQI最重要的操作。



要将消息放入队列，应用程序调用MQOPEN，在特定的队列管理器中指定目标队列。可以使用本地可用的别名来命名队列管理器。目标队列实际上是否远程对应用程序来说是完全透明的。如果应用程序希望从其本地队列获取消息，还应该调用MQOPEN。只有本地队列能够打开来读取传入的消息。当应用程序访问队列完成时，它应该通过调用MQCLOSE关闭队列。

可以分别使用MQPUT和MQGET向队列写入或从队列读取消息。原则上，消息按优先级从队列中删除。具有相同优先级的消息将以先入先出的方式删除，即先删除最长的挂起消息。也可以请求特定的消息。此外，MQ提供了在消息到达时向应用程序发出信号的工具，从而避免了应用程序必须连续轮询消息队列以获取传入消息。一个有趣的观察结果(也是现代messagequeuing系统中常见的一个观察结果)是，持久性消息传递不是通过简单地使队列保持持久性来获得的。相反，消息被标记为持久的，队列管理器的工作是确保消息能够在崩溃中存活。因此，队列可以同时存储持久性和非持久性消息。

**管理覆盖网络**

从目前的描述可以清楚地看出，管理MQ系统的一个重要部分是将各个队列管理器连接到一致的覆盖网络中。此外，这个网络需要长期维护。对于小型网络，这种维护只需要普通的管理工作，但是当使用消息队列来集成和分解大型现有系统时，事情就变得复杂了。

MQ的一个主要问题是需要手动管理覆盖网络。这种管理不仅涉及在队列管理器之间创建通道，还包括填充路由表。显然，这可能会变成一场噩梦。不幸的是，对MQ系统的管理支持之所以先进，只是因为管理员可以设置几乎所有可能的属性，并调整任何可考虑的配置。然而，底线是通道和路由表需要手动维护。

覆盖管理的核心是**通道控制功能**组件，它逻辑上位于消息通道代理之间。该组件允许操作人员精确地监视通道两端的情况。此外，它用于创建通道和路由表，还用于管理承载消息通道代理的队列管理器。在某种程度上，这种覆盖管理方法与使用单个管理服务器的集群服务器的管理非常相似。在后一种情况下，服务器本质上只向集群中的每台机器提供一个远程shell，以及一些处理机器组的集体操作。分布式系统管理的好消息是，如果您正在寻找解决严重问题的新解决方案，那么它提供了许多机会。

**示例:高级消息队列协议(AMQP)**

关于消息队列系统的一个有趣的观察结果是，它们的开发在一定程度上允许遗留应用程序进行互操作，但与此同时，当涉及到不同消息队列系统之间的操作时，我们经常遇到瓶颈。因此，一旦组织选择使用来自制造商X的消息队列系统，他们可能不得不接受仅由X提供的解决方案。因此，消息队列解决方案在很大程度上是专有的解决方案。

2006年，为了改变这种情况，成立了一个工作组，这导致了**高级消息队列协议**(简称**AMQP**)的规范。AMQP有不同的版本，最新的版本是1.0。AMQP也有多种实现，尤其是1.0之前的版本，1.0版本建立时，AMQP已经非常流行。因为1.0之前的版本与1.0版本非常不同，但是也有一个稳定的用户基础，所以我们可能会看到1.0之前的各种AMQP服务器在1.0服务器旁边(它们不可否认是不兼容的)。

在本节中，我们将描述AMQP，但是会或多或少地故意混合1.0之前的版本和1.0版本，坚持AMQP的本质和精神。

**基础知识**

AMQP围绕应用程序、队列管理器和队列展开。作为网络环境中的常见方法，我们将分别将AMQP看做是消息传递服务，消息传递协议和应用的消息接口。为此，最容易考虑的情况是只有一个队列管理器，作为一个单独的服务器运行，形成AMQP作为服务的实现。应用程序通过本地接口与此队列管理器通信。应用程序和队列管理器之间的通信根据AMQP协议进行。

这种情况如图4.34所示，应该很熟悉。AMQP stub通常使应用程序(以及队列管理器)免受有关消息传输和通信的详细信息的影响。同时，它实现了一个消息队列接口，允许应用程序使用AMQP作为消息队列服务。虽然对于队列管理器来说，AMQP stub和消息管理器的区别很明显，然而这种严格的区分还是在实现的时候才能分辨。然而，如果不严格的话，在概念上处理队列和处理相关通信之间是有区别的，我们将在稍后说明这一点。

**AMQP 通信**

AMQP允许应用程序设置到队列管理器的连接；连接是许多单向**通道**的容器。通道的生命周期可以是高度动态的，而连接则假定是相对稳定的。连接和通道之间的这种差异允许有效的实现，特别是通过使用一个传输层TCP连接来在应用程序和队列管理器之间复用许多不同的通道。在实践中，AMQP假定TCP用于建立AMQP连接。

通过会话建立双向通信：两个通道的逻辑分组。一个连接可能有多个会话，但请注意通道不一定是会话的一部分。

最后，要实际传输消息，需要一个链接。从概念上讲，链接，或者更确切地说是它的端点，会跟踪正在传输的消息的状态。因此，它提供了应用程序和队列管理器之间的细粒度流控制，实际上，可以同时为通过同一连接会话传输的不同消息设置不同的控制策略。流控制是通过credits建立的:接收方可以告诉发送方允许通过特定链接发送多少消息。当要传输消息时，应用程序将其传递到其本地AMQP stub。如前所述，每个消息传输都与一个特定的连接相关联。消息传输通常分为三个步骤：

1. 在发送方，消息被分配一个唯一的标识符，并在本地记录为处于未解决状态。Stub随后将消息传输到服务器，AMQP stub还将其记录为处于未解决状态。此时，服务器端stub将其传递给队列管理器。
2. 接收应用程序(在本例中是队列管理器)，假定处理消息，并通常向其stub报告消息已完成。stub将此信息传递给原始发送方，此时原始发送方的AMQP stub处的消息进入稳定状态
3. 原始发送方的AMQPstub现在告诉原始接收方的stub消息传输已经完成(这意味着从现在开始原始发送方将忘记消息)。接收者的stub现在也可以丢弃关于消息的任何内容，并将其正式记录为已解决。

注意，由于接收应用程序可以向底层AMQP通信层表明它是使用消息完成的，所以AMQP支持真正的可靠的端到端通信。特别是，应用程序，无论是客户机应用程序还是实际的队列管理器，都可以指示AMQP通信层保存消息(即，则消息处于未确定状态)。

**AMQP消息**

AMQP中的消息传递逻辑上发生在处理通信的层之上。在这里，应用程序可以指示需要对消息执行什么操作，还可以查看到到目前为止发生了什么。消息传递正式发生在两个节点之间，其中有三种类型：生产者、消费者或队列。通常，生产者节点和消费者节点表示常规应用程序，而队列用于存储和转发消息。实际上，队列管理器通常由多个队列节点组成。为了进行消息传输，两个节点必须在它们之间建立链接。

接收方可以向发送方表明是否接受了其消息(意味着成功处理了该消息)，或者拒绝了该消息。注意，如果拒绝，这意味着通知将返回给原始发送方。AMQP还支持发送附加通知的大型消息的分段和组装。

当然，AMQP的一个重要方面是它对持久消息传递的支持。实现持久性可以通过几种机制来处理。首先也是最重要的一点是，消息可以被标记为持久的，这表明源希望任何中间节点(比如队列)在发生故障时能够恢复。不能保证这种持久性的中间节点将不得不拒绝消息。其次，源节点或目标节点还可以指示其持久性:如果是持久性的，它将维护其状态，或者维护持久消息的(未解决的)状态。将后者与持久消息有效地结合起来，可以建立可靠的消息传输和持久消息传递。

AMQP实际上是一个消息传递协议，因为它本身不支持发布-订阅原语。它希望这些问题由更高级的专有队列管理器处理，类似于第4.3节中讨论的消息代理。最后，没有理由不能将队列管理器连接到另一个队列管理器。事实上，将队列管理器组织到覆盖网络中是很常见的，在这个网络中消息从生产者路由到消费者。AMQP没有指定如何构建和管理覆盖网络，实际上，基于AMQP系统的不同供应商提供了不同的解决方案。特别重要的是指定消息应该如何通过网络路由。最重要的是，管理员将需要手动执行大量此规范。只有在覆盖具有规则结构(如循环或树)的情况下，才更容易提供必要的路由细节。

4.4 多播通信

分布式系统通信中的一个重要课题是支持向多个接收器发送数据，也称为组播通信。多年来，这个主题一直属于网络协议领域，在这个领域中，已经实现和评估了许多关于网络级和传输级解决方案的建议[Janic, 2005;Obraczka, 1998)。所有解决办法中的一个主要问题是建立传播信息的传播途径。实际上，这涉及到巨大的管理工作，在许多情况下需要人工干预。此外，只要没有统一的提议，isp就不愿意支持多播[Diot et al.， 2000]。

随着peer-to-peer技术的出现，特别是结构化覆盖管理的出现，设置通信路径变得更加容易。由于peer-to-peer解决方案通常部署在应用程序层，因此引入了各种应用程序级多播技术。在本节中，我们将简要介绍这些技术。

除了设置显式通信路径外，还可以通过其他方式实现组播通信。正如我们在本节中所探讨的，基于gossip的信息传播为多播提供了简单(但通常效率较低)的方法。

**应用程序级别的基于树的多播**

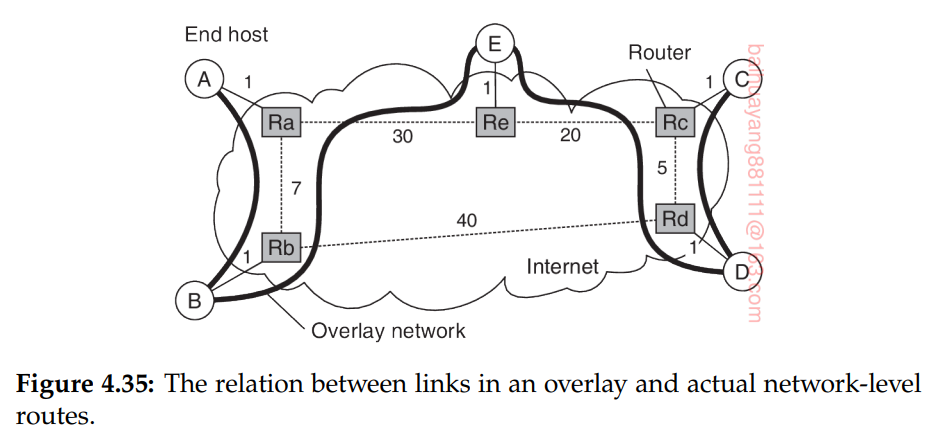
应用程序级多播的基本思想是节点组织成覆盖网络，然后用于向其成员传播信息。一个重要的观察是，网络路由器不涉及组成员关系。因此，覆盖网络中的节点之间的连接可能跨越多个物理链接，因此，与网络级路由所能实现的效果相比，覆盖网络中的路由消息可能不是最优的。

覆盖网络的建设是一个关键的设计问题。实际上，有两种方法。首先，节点可以直接将自己组织成一棵树，这意味着每对节点之间都有一个惟一的(覆盖)路径。另一种方法是将节点组织成网状网络，其中每个节点都有多个邻居，通常，每对节点之间存在多个路径。两者的主要区别在于后者通常提供更高的健壮性：如果连接中断(例如，因为节点失败)，仍然有机会传播信息，而不必立即重新组织整个覆盖网络。

**覆盖网络的性能问题**

从上面给出的高层描述可以清楚地看出，虽然一旦我们将节点组织成覆盖层，构建树本身并不那么困难，但是构建高效的树可能是另一回事。注意，在我们到目前为止的描述中，参与树的节点的选择没有考虑任何性能指标：它完全基于通过覆盖层的(逻辑)路由的消息。

要理解当前的问题，请看图4.35，它显示了一个简单的覆盖网络中由五个节点组成的小集合，其中节点a构成组播树的根。这个图还显示了遍历物理链接的成本。现在，每当向其他节点多播一条消息时，可以看到该消息将两次遍历以下每个链接<B,Rb>,<Ra,Rb>,<E,Re>,<Rc,Rd>,<D, Rd>。如果覆盖网络没有建立<B,E>和<D,E>，而是建立了<A,E>,<C,E>，传播将会更有效。这样的配置可以保存跨物理链接<Ra,Rb>,<Rc,Rd>的双重遍历。



应用程序级多播树的质量通常由三个不同的度量标准来度量:连接压力、伸缩性和树成本。连接压力表示消息通过这条连接的频率。连接压力大于1的原因是，尽管在逻辑级别上，一个数据包可以沿着两个不同的连接转发，但这些连接中的一部分实际上可能对应于相同的物理链路，如图4.35所示。

拉伸或相对延迟惩罚(RDP)度量覆盖层中两个节点之间的延迟，以及这两个节点在底层网络中所经历的延迟的比例。例如 从B到C的消息在覆盖网络上传播会经过以下路径 B -> Rb -> Ra -> Re -> E -> Re -> Rc -> Rd -> D -> Rd -> Rc -> C 总共会消耗73个单位。然而，消息却会从以下网络的路径传播， B –> Rb -> Rd -> Rc -> C 总共消耗47个单位。所以这种延迟比为1.55。很明显的，当构建一个覆盖网络时，我们的目标是（没有蛀牙）最小化延迟比。

最后，树成本是一个全局度量，通常与最小化聚合连接成本相关。例如，如果一个连接的成本被认为是它的两个端点之间的延迟，那么优化树的成本就归结为找到一个最小生成树，其中向所有节点传播信息的总时间是最小的。为了简化问题，假设一个多播组有一个关联的、众所周知的节点，该节点跟踪已经加入树的节点。当一个新节点发出连接请求时，它将与这个**rendezvous 节点**联系，以获得(可能是部分)成员列表。目标是选择能够作为树中新节点的父节点进行操作的最佳成员。有许多替代方案，不同的建议通常遵循非常不同的解决方案。

例如，考虑一个只有一个源的多播组。在这种情况下，最佳节点的选择是显而易见的:它应该是源节点(因为在这种情况下，我们可以确保拉伸将等于1)。但是，这样做时，我们将引入一个星型拓扑结构，其中的源代码位于中间。虽然很简单，但是不难想象源很容易超载。换句话说，节点的选择通常会受到这样一种约束，即只能选择具有k个或更少邻居的节点，而k是一个设计参数。这个约束使树建立算法变得非常复杂，因为一个好的解决方案可能需要对现有树的一部分进行重新配置。

**基于洪泛的多播**

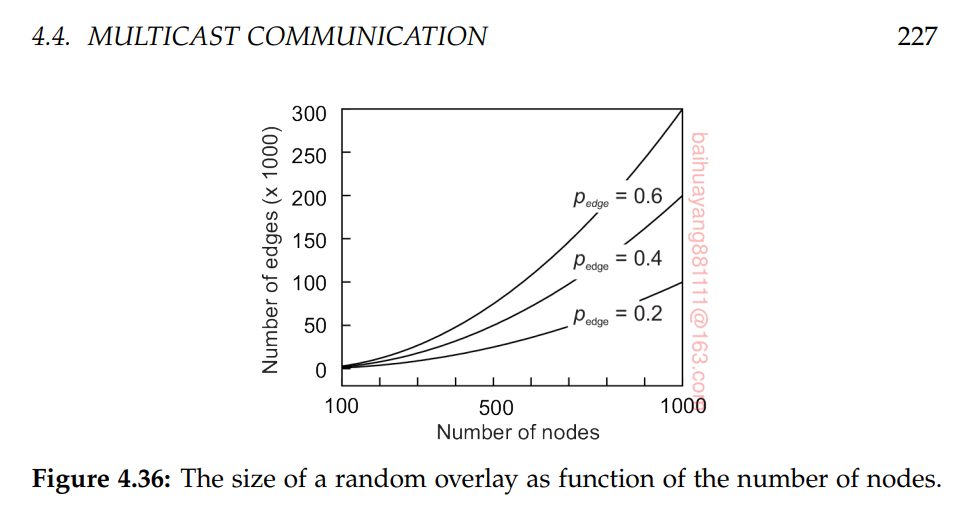
到目前为止，我们假设当一个消息要进行多播时，它将被覆盖网络中的每个节点接收。严格的说，这叫做广播。通常来说，组播是指对一组或者**特定的一组**节点发送消息。当涉及到多播时，一个关键的设计问题是最小化对消息不打算使用的中间节点的使用（就是让不接收消息的节点尽量的少出现）。为了说明这一点，如果覆盖层被组织成多级树，但是只有叶子节点应该接收多播消息，那么很明显，可能有相当多的节点需要存储并随后转发不适合它们的消息。

避免这种低效率的一个简单方法是为每个组播组构造一个覆盖网络。因此，将消息m多播到组G与将消息m广播到组G是一样的。此解决方案的缺点是，属于多个组的节点原则上需要为其所属的每个组维护一个单独的邻居列表。

如果我们假设一个覆盖层对应于一个多播组，而且我们需要广播一条消息，那么一种简单的方法是应用泛洪。在本例中，每个节点只是将一个消息m转发给它的每个邻居，但接收到m的节点除外。如果一个节点跟踪到曾经接受或转发过消息，则它可以简单的将消息丢弃。我们将看到发送的消息大约是覆盖网络中连接数量的两倍，这使得泛洪非常低效。

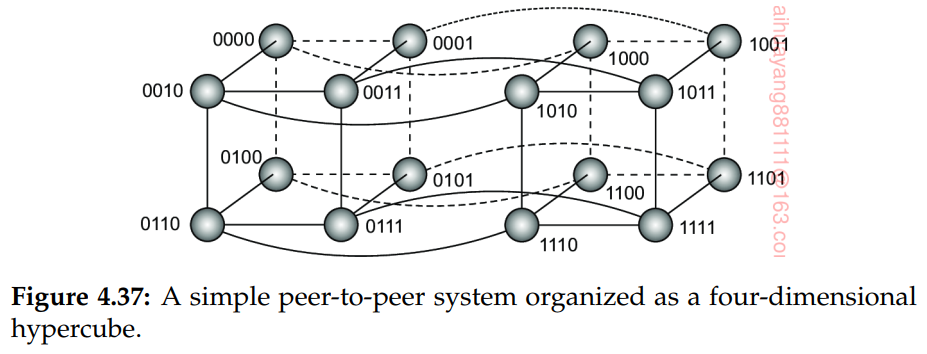
为了理解泛洪的性能，我们将覆盖网络建模为一个有N个节点和M条边的连通图G。请记住，泛洪意味着我们需要发送(至少)M条消息。只有当G是树时，泛洪才是最优的，因为在这种情况下，M = N – 1。在最坏的情况下，当G完全连接时，我们将不得不发送M = (N^2) = 1/2·N·(N - 1)条消息。

假设我们没有覆盖网络结构的信息，并且我们假定最好的结构就是随机网络，即每两个节点的连接概率是P。请注意，我们实际上考虑我们的覆盖网络是一个非结构化的peer-to-peer网络，我们没有任何关于它如何构建的信息。由于连接出现的概率P和1/2\*N(N-1)条总共可能存在的边，不难算出，总的期望变数为两者相乘，即P\*1/2\*N(N-1)。图4.36显示了pedge不同值的节点数量和边缘之间的关系，以便对我们正在处理的内容有一个印象。

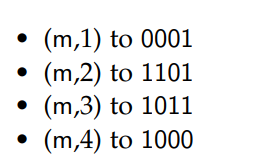


为了减少消息的数量，我们还可以使用Banaei-Kashani和Shahab[2003]引入的概率泛洪，并对其进行正式分析。这个想法非常简单:当一个节点向消息m发送大量消息，并且需要将m转发给一个特定的邻居时，它将以pflood的概率这样做。其效果可能非常显著:在pflood中发送的消息总数将直线下降。然而，也存在一个风险:pflood越低，无法到达网络中所有节点的可能就越大。这个风险是由一个简单的事实引起的，即一个特定节点Q的所有邻居都决定不将m转发给Q。如果Q有n个邻居，那么这种情况发生的概率大约是(1 - pflood)^n。显然，邻居的数量在决定是否转发消息时起着重要的作用，而且，我们确实可以用一个考虑到邻居的度（图的概念，度表示邻居的数量）的概率来代替静态转发的概率。Sereno和Gaeta[2011]进一步发展和分析了这种启发式。为了了解概率广播的效率:在一个由10,000个节点组成的随机网络中，pedge = 0.1，我们只需要设置pflood = 0.01，就可以将发送的消息数量与完全泛洪相比减少50倍以上。

在处理结构化覆盖时，也就是具有或多或少确定性拓扑结构的覆盖时，设计有效的泛洪方案要简单得多。作为一个例子，我们在第2章中讨论过，考虑一个n维超立方体，在n = 4的情况下如图4.37所示。



Schlosser等人[2002]设计了一种简单而有效的广播方案，它依赖于跟踪每个维度的邻居。考虑到n维超立方体中的每个节点都由一个长度为n的位串表示。覆盖层中的每条边都标有它的维数。对于n = 4的情况，节点0000的邻居是集合{0001,0010,0100,1000}。在比较0000和0001时，将0000和0001之间的边缘标记为“4”，表示更改第4位，反之亦然。同样，<0000,0100>的边被标记为“2”，以此类推。一个节点最初向它的所有邻居广播一个消息m，并使用标签的边缘来标记m。在我们的例子中，如果节点1001广播一个消息，则它会发送如下的信息：



当节点接收到广播消息时，它只会沿着维度更高的边缘转发消息。换句话说，在我们的示例中，节点1101只将m转发给节点1111(通过标记为“3”的边连接到1101)和节点1100(通过标记为“4”的边连接到)。通过该方案可以看出，每次广播都需要精确的N - 1条消息，其中N = 2^n，即N维超立方体中的节点数。因此，就发送的消息数量而言，这种广播方案是最优的。（立方体这个没懂）