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**

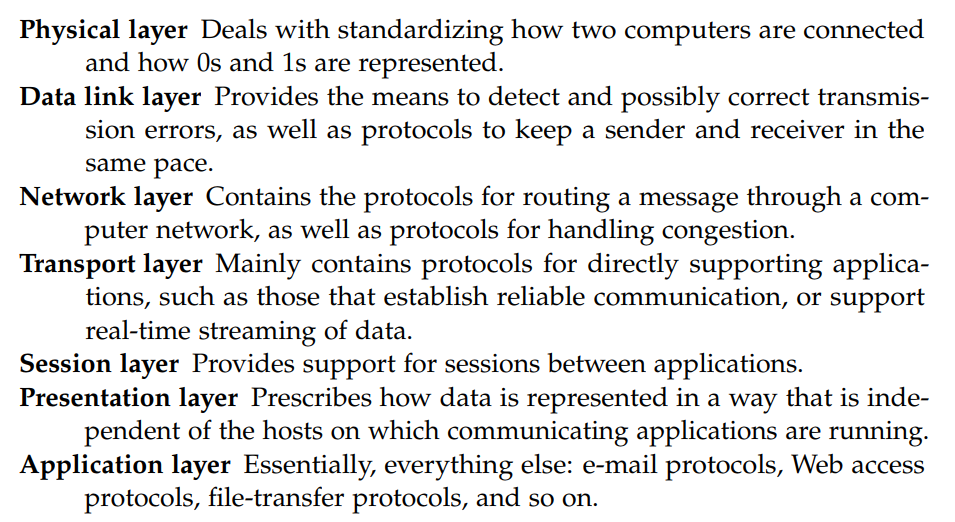
由于缺乏共享内存，分布式系统中的所有通信都基于发送和接收(低层)消息。当进程P希望与进程Q通信时，它首先在自己的地址空间中构建一条消息。然后它执行一个系统调用，使操作系统通过网络向Q发送消息。虽然这个基本的想法听起来很简单，但是为了防止混乱，P和Q必须就发送的比特的意义达成一致。

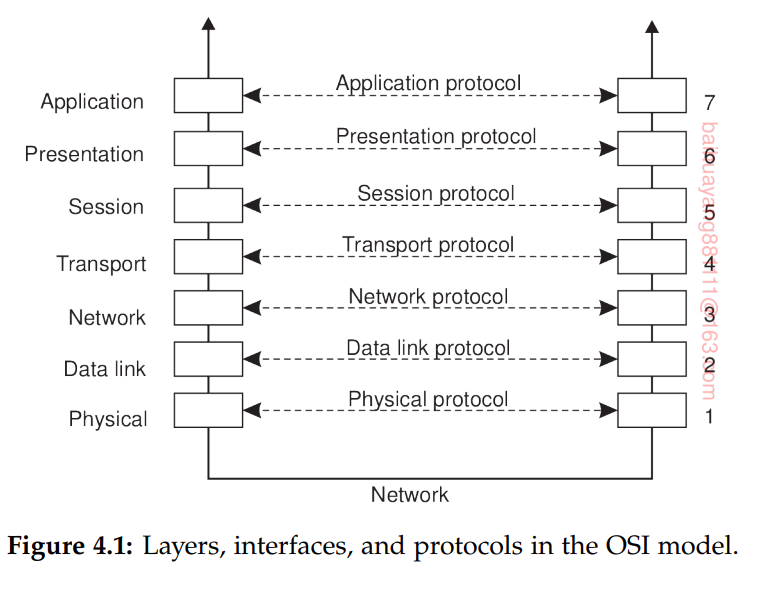
**OSI相关模型**

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。

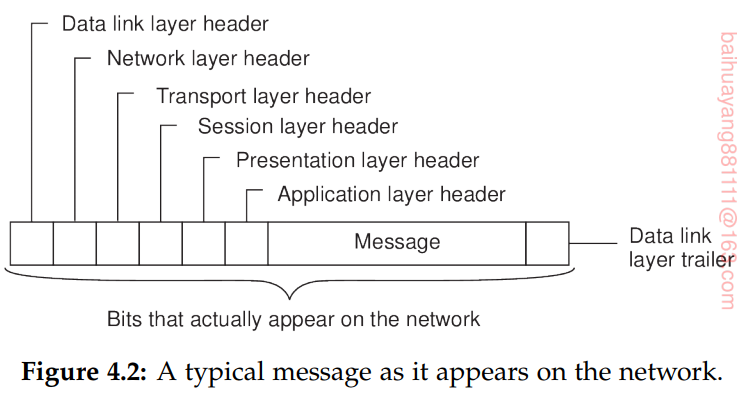
OSI模型是为了让开放系统通信而设计的。开放系统是一种通过使用标准的格式，内容和消息内容的规则，来和其他的系统进行通信的系统。这些规则称之为**通信协议**。为了能够使得计算机可以通过网络进行通信，通信协议是必须要被使用的。一种协议其实就是一种通信服务。这里有两种类型的服务。对于面向连接的服务，在交换数据之前，发送方和接收方首先显式地建立连接，并可能协商它们将使用的协议的特定参数。当它们完成时，它们释放(终止)连接。电话是一种典型的面向连接的通信服务。对于无连接的服务，不需要预先设置。当第一个消息准备好时，发送方就发送它。将信件放入邮箱是使用无连接通信服务的一个例子。在计算机世界中，面向连接和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

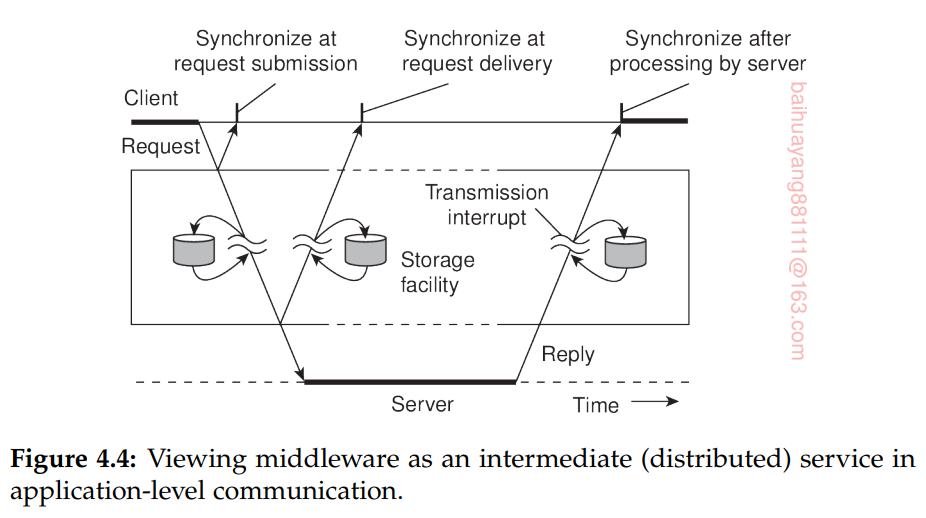
作为最后一个示例，考虑一个分布式锁定协议，通过该协议，可以保护资源不受分布在多台机器上的进程集合的同时访问。不难想象，这样的协议可以以独立于应用程序的方式设计，并且可以通过相对简单的、同样独立于应用程序的接口进行访问。因此，它们通常属于中间件。

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**

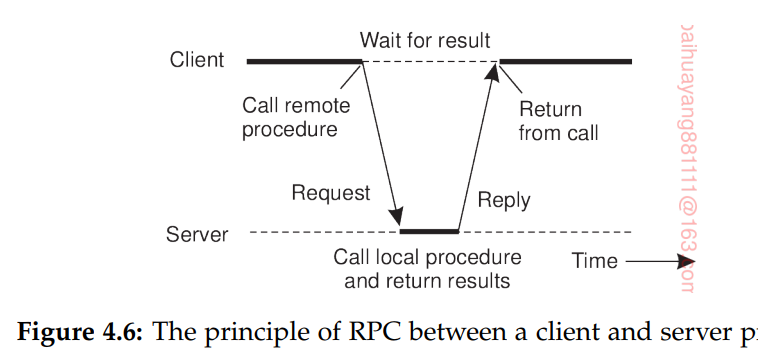
RPC背后的思想是使远程过程调用尽可能看起来像本地过程调用。换句话说，我们希望RPC是透明的——调用过程不应该知道被调用的过程在另一台机器上执行，反之亦然。假设一个程序可以访问一个数据库，该数据库允许它将数据附加到一个存储的列表中，然后返回对修改后列表的引用。该操作如下所示



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)。

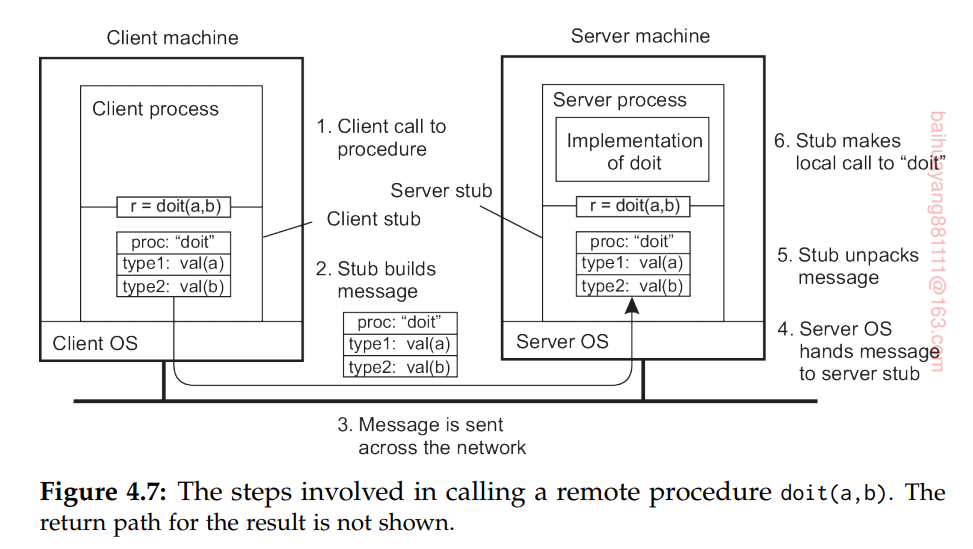
当server stub在调用完成后获得控制权时，它将结果打包到消息中并发送调用将其返回给客户机。之后，server stub通常再次调用receive，等待下一个传入请求。当结果消息到达客户机的机器时，操作系统将其通过之前调用的receive操作传递给client stub，然后解除客户机进程的阻塞。Client stub检查消息，解包结果，将其复制给调用者，然后以通常的方式返回。当调用者在调用append后获得控制权时，它只知道它将一些数据附加到列表中。它不知道这项工作是在另一台机器上远程完成的。

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。

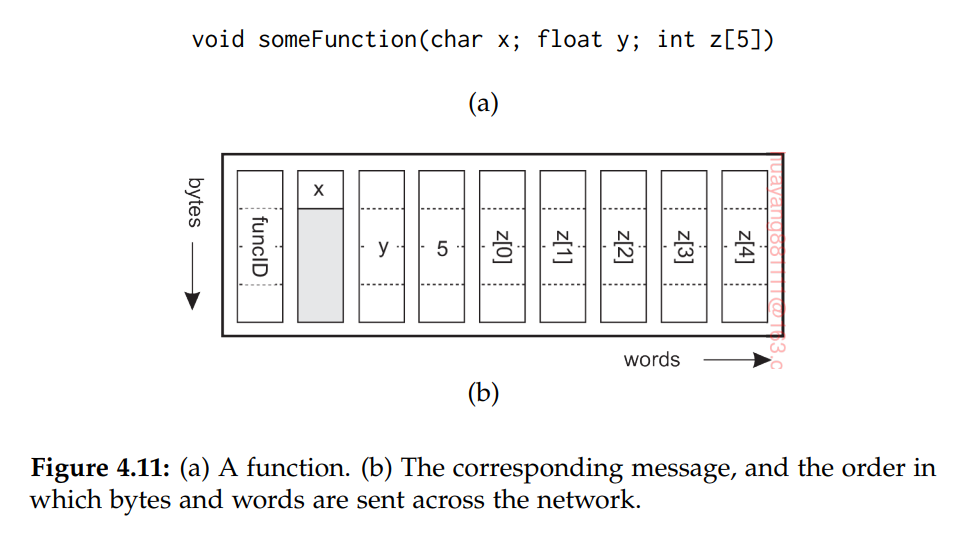
到目前为止所讨论的指针和引用的问题是，它们只在本地有意义:它们引用的内存位置只对调用进程有意义。使用全局引用可以缓解问题：对调用和被调用流程有意义的引用。例如，如果客户机和服务器可以访问相同的文件系统，那么传递一个文件句柄而不是指针就可以了。有一个重要的发现：两个进程都需要确切地知道传递全局引用时应该做什么。换句话说，如果我们考虑的全局引用关联了具体的数据类型，那么我们的调用和被调用的进程的操作应该会有相同的效果。此外，在传递文件句柄时，两个进程应该就具体做什么达成一致。同样，这些问题通常可以通过适当的编程语言支持来解决。

**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**