# 体系结构

分布式系统是由许多复杂的软件组成的。正确的将系统组织好是掌握这种复杂性的关键。有不同的方法可以识别出分布式系统的组织结构，但更显而易见的是，我们可以从软件的逻辑组成和具体硬件实现上来进行区分。

说起分布式系统的组成更多谈起的是软件组成。这些软件架构告知我们不同的软件是如何组织的和如何联系的。这一节我们将先来观察分布式系统的组织架构风格是什么样的。

分布式系统的一个重要目标是通过中间层来分离应用和底层平台。使用中间层是一个重要的架构决定，并且它的主要目标是提供分布式系统的透明程度。

然而，我们需要在分布式系统的透明性，和为了这种透明性需要进行对中间层的技术修改之间做权衡。我们将讨论这些应用在中间层组织结构的技术。

分布式系统的实现需要我们再真机上实例化软件。这里我们有很多种选择。最终的软件架构和系统架构是相关的。在这一节中，我们将深入讨论传统的中心式架构，即大部分的软件都在一个服务器上部署，同时远程客户可以通过简单的通信手段来访问这个服务器。另外，我们也将讨论去中心化的peer-to-peer架构，即所有节点都担当同样的角色。许多现实的分布式系统都采用了混合式的策略，即中心和非中心的混合。

我们将在这节结尾讨论两种应用广泛的分布式系统，NFS文件分享系统和WEB系统。

## 2.1 架构风格

关于分布式架构，我们首先讨论软件成分的逻辑组织，也即软件架构。软件架构的研究已经成熟了，并且通常认为一个好的架构设计是一个软件系统发展成功的关键。

对我们来讲，软件架构风格的讨论更为重要。这种风格是由各个组件之间的联系，组件之间的数据交换以及各个元素是如何加入并配置到这个系统中来决定的。一个组件是一个定义良好的模块化单元，并提供了在其环境中可被替代的接口。系统中的每个组件都是可被替换的。这是因为关闭系统并不是一个好的选择。最好的情况是，只有部分设备暂时出现了故障。只有当被替代的组件的接口已经不能访问了，这种替代才可以发生。一个比较难以理解的概念是连接器，它通常被描述为组件之间的沟通，协调和合作的中间机制。例如，有的连接器是由远程调用，消息传递和流数据组成的。换句话说，一个连接器可以让连接和数据在组件之间进行流式传输。

使用组件和连接器，我们会遇到多种不同的配置，这都将归类于架构风格。不同的风格都已经指出了，下面列几种对分布式系统最为重要的：

Layered architectures

Object-based architectures

Resource-centered architectures

Event-based architectures

在接下来，我们将以此讲解这些架构。在现实的分布式架构中，以上这些架构是被联合使用的。对系统进行细分的方法，已经成为了一个普遍的使用多种架构风格的原则。

**Layered architectures**

层次架构的风格很简单：高层的组件可以对低层的组件进行访问，并等待一个回应。只有在极其例外的情况下才会要低层的去调用高层的情况。这三种情况如下图所示。

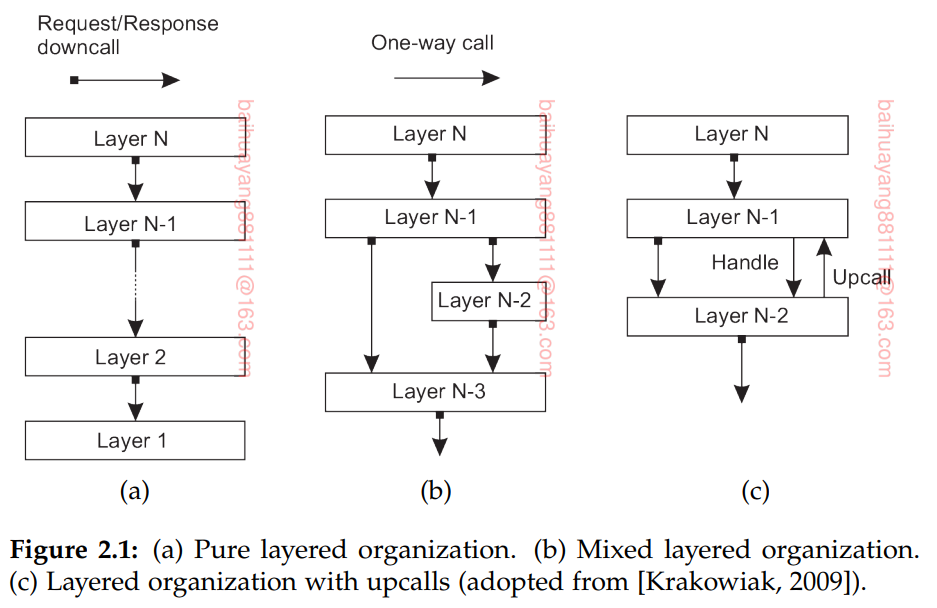


图2.1(a)展示了一个标准的组织结构，即只存在向下调用的情况。这种组织结构通常部署网络通信的情况下。

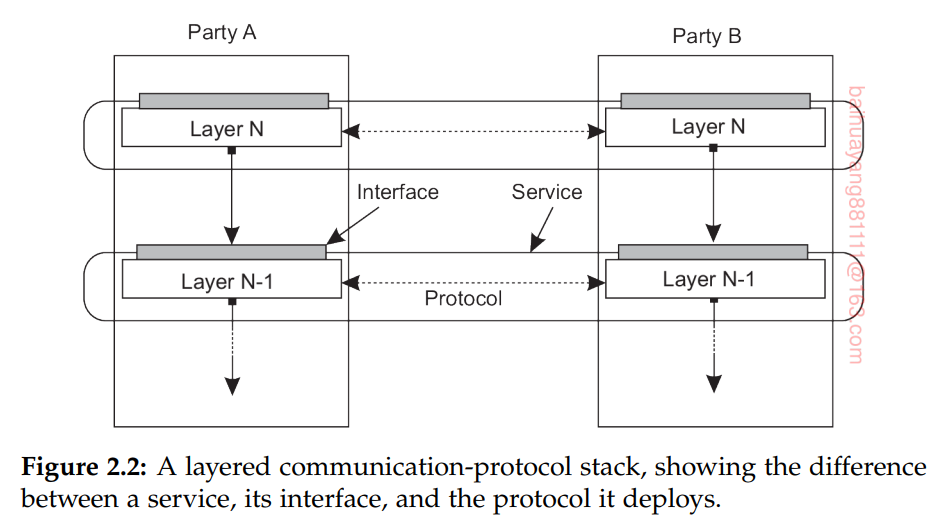
在很多情况下，我们也会遇到如图2.1(b)的情况。比如应用A需要调用操作系统底层接口，并且也需要调用实现了操作系统底层接口的另外一个接口，比如math接口，这样就如2b图所示的结构了。

最后是2.1c这种情况。有些情况下，会出现下层调用上层接口的情况。比如回调，应用发生时，传递给操作系统一个引用，当应用结束时，要求操作系统执行这个引用。

**层级通信协议**

一个比较有名的层级通信架构是通信协议栈。这里讲主要描述它的轮廓，细节放在4.1节讨论。

在通信协议栈里，每一层通过对通信服务的实现，可以将数据从一个地方传输到另一个地方。为此，每一次提供一个接口供其他层进行调用。原则上，这些接口的实现应当完全隐藏。另一个重要的通信协议的功能是，为各种信息交换提供遵循的规则。我们最好能区别出服务，接口和协议这三种的区别。如图2.2所示



为了更清楚的区分，我们考虑一个许多通信系统都提供的一个可靠地，面向连接的服务。在这种情况下，发送数据之前两边需要先建立连接。可靠性保证了数据一定能从一段传送到另一端。这些服务通常能保证了他们发送数据的和接收数据的顺序一致性。

这种网络服务有一种叫做TCP的协议实现了。这种协议保证了数据交换于一个连接上，它能够保存数据的传输顺序，并发现和纠正在数据传输中的错误。这种服务被封装成一种简单的可使用的编程接口。事实上，在不同的操作系统上有不同的可使用的接口。类似这种协议和接口的实现还有很多。

**应用层**

接下来讨论应用逻辑层。考虑到很大一部分的分布式应用都会支持数据库的访问，人们在层次架构上又区分了三种逻辑层，分别是

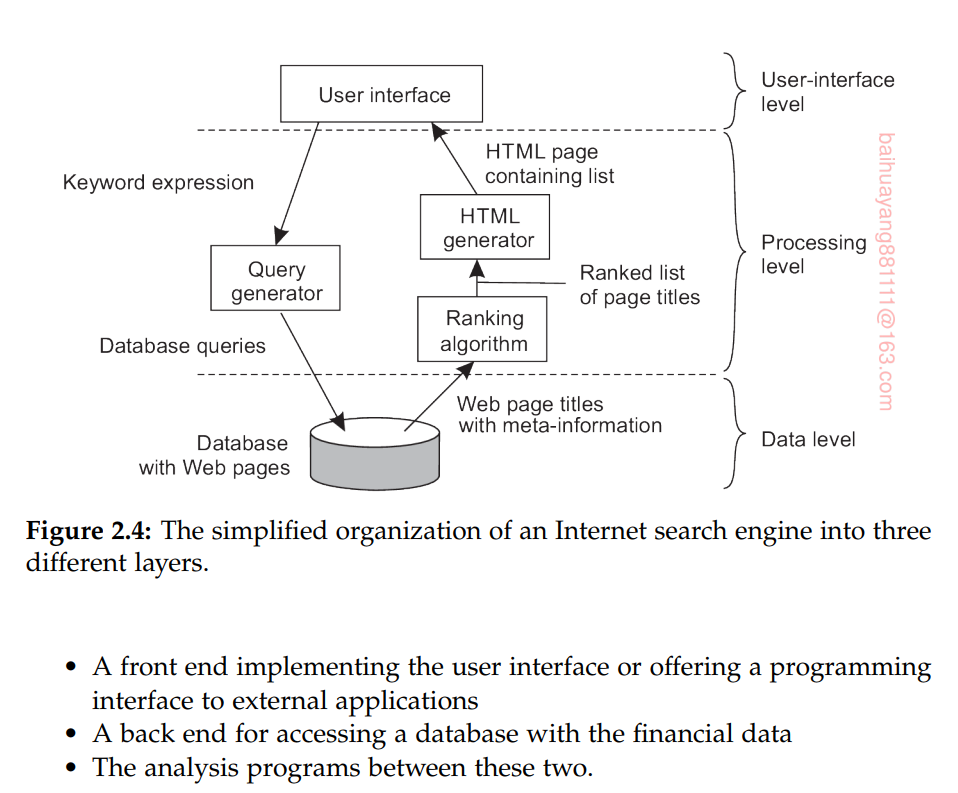
应用接口层

执行层

数据层

通过这么划分，我们可以了解应用程序通常可以有三种不同的方面组成：一方面处理与外部用户或系统的交互，一部分操作数据库和文件系统，中间那部分包括应用的核心方法。中间这部分就是处于执行层。不同于用户接口和数据库层，中间层的实现并没有太多类似的东西。我们将会用例子来说明。

第一个例子，网络搜索引擎。不考虑图片，banners 等一系列花哨复杂的东西，搜索引擎的用户接口非常简单：一个用户键入一个关键词，网页就返回一系列标题。后端系统是一个巨大的已经预处理的并建好索引的网页数据库。这个搜索引擎的核心是将用户的关键词传输到多个数据库中用于查询。然后将结果进行排名，并输出到html网页中。信息检索的部分位于执行层，如图2.4



金融数据分许则需要统计学和人工智能中的复杂方法和技术。在一些情况下，金融分析支持系统需要在高性能的机器上执行，用来满足用户需要的吞吐量和响应。

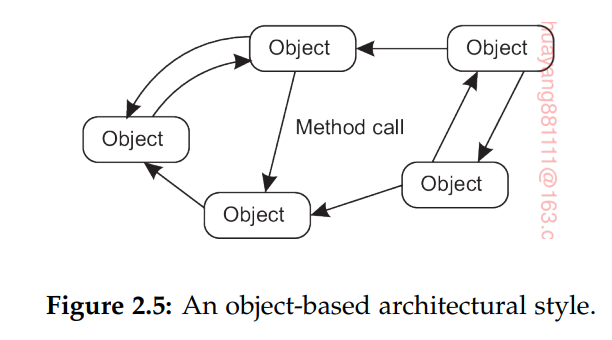
最后一个例子是，考虑一个台式机，由文字处理器，电子表格应用，通信工具组成。这样的办公套装通过支持文件管理和用户主目录操作的公共用户接口来集成。（在办公环境下，用户主目录通常在一个远程机器上。）在这个例子中，处理层会由相对大的程序集合组成，并且每一个都有简单的处理能力。

数据层则存储着应用需要访问和操作的数据。数据层一个重要的性质是，里面的数据都是持久化的，即便没有应用在运行，数据也是一样存储起来。在最简单的行驶中，数据层包括文件系统，但它也通常使用一个完整的数据库。

除了存储数据，数据层通常需要保持各个应用之间的数据一致性。当数据库被使用时，像一些数据元数据如表描述，建约束和应用特殊的元数据都被储存起来。例如，在银行中，我们想要生成一个用户的信用卡债务到达一个特定值的时间的备注。这种类型的信息可以通过数据库的触发器在特定是时候激活一个句柄来实现。

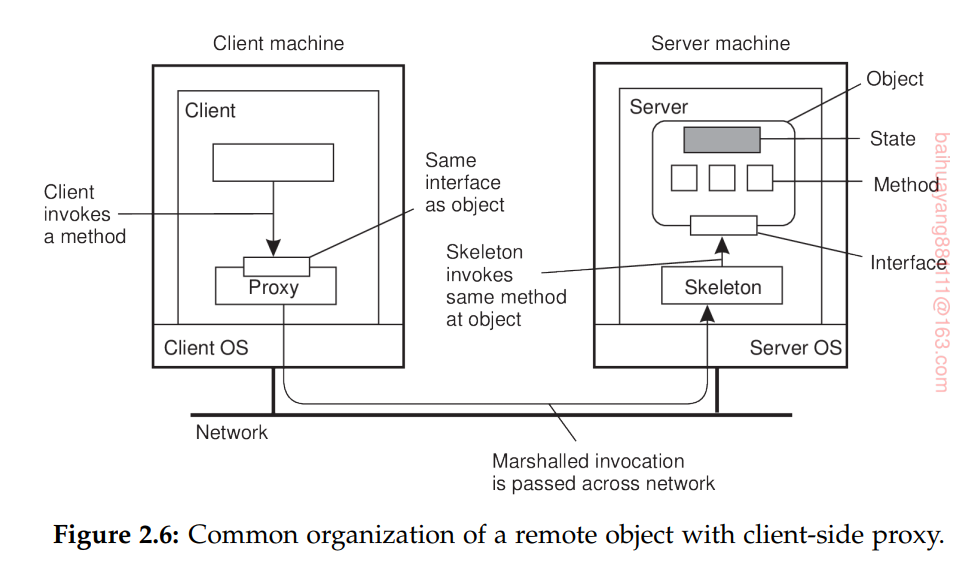
**Object-based and service-oriented architectures （面向对象架构）**

一个更加松散的组织结构是 object-based architectures，如下图所示。本质上，每一个object对应于一个组件，并且每一个组件通过过程调用机制相互连接。，在分布式系统中，一个过程调用通常发生在网络中，所以过程调用通常不是发生在本机器上。



Object-based 架构是很吸引人的，因为它用一种很自然的方式对数据进行了包装，并且这些包装的数据可以很容易的操作。通过Object提供的接口，它的实现细节会被隐藏，原则上我们可以认为 object可以完全独立于系统而存在。或者说，如果接口被清晰的定义然后被废弃使得其他应用无法调用，那么这个object就可以被替换。

这种接口和实现分离的方式允许我们将接口定义在一台机器上，而object依赖在另一台机器上（自我理解，就是类似RPC，一台机器负责所有接口定义，其余机器实现）。这种组织结构如下图2.6所示。



当一个哭护短和一个分布式object绑定时，object接口的实现称之为代理，会被加载到客户端的地址空间。这个代理和RPC系统的客户端stub是一个意思。它的唯一作用是封装方法调用和解封返回消息。实际的object驻留在服务器上，它和客户端机器提供同样的接口。首先进入服务器stub的是调用请求，stub将请求解封并执行方法调用。服务器stub同样负责封装回应并将相应数据返回给客户端的代理。

服务端的stub同城被称为股价，因为它提供了允许服务器中间件访问的裸方法。实际上，他常常包含不完整的代码，这些代码以特定的语言的形式出现，需要开发人员进一步特殊化。

大部分分布式系统的object的一个特性是，他们的状态不是分布式的：他们都取决于一个单一的机器。只有实现了接口的object才能够对其他机器使用。这种objects通常称作remote objects。通常的分布式object中，他们的状态都是在物理上分布在不同机器上的，但这种分布式也隐藏在object的接口中。

可以认为object为基础的架构是将服务封装到独立的单元中，在这里封装是关键词：服务作为一个整体被时限为一个自包装的实体，虽然它也会使用其他的服务。将各种各样服务分离使得他们可以独立操作，我们可以使用 **service-oriented architectures**（面向服务架构） 即，SOAs

在面向服务的架构中，一个分布式应用和系统本质上是有许多不同的服务组成的。并非所有的服务都属于同一个管理组织。我们讨论云计算的时候已经遇到了这个现象：很有可能一个结构使用云端提供的存储服务来运行业务应用。这些存储服务逻辑上被包装成一个单一的单元，并对用户开放接口。

当然，存储是一个很基础的服务，但更复杂的场景会浮现出来。考虑网上商场卖电子书的场景。一个简单的处理方式是应用分层，一个应用用来处理订单，一个应用用来操作存储电子书的数据库。订单处理通常包括商品选择，注册和检测派送渠道并确保交易发生。后者可以使用一个单独的服务来处理，让用户跳转到一个支付界面，当完成后发出用户完成此次交易的通知。

在这种方式下，我们发现将各个服务协调的组合好是一个难题。实际上，这个问题在1.3节已经讨论过了。让每一个服务提供一个定义好的接口。实际上，这意味着每一个服务提供各自的接口，但服务的组合远不只如此。

**资源为基础的架构**

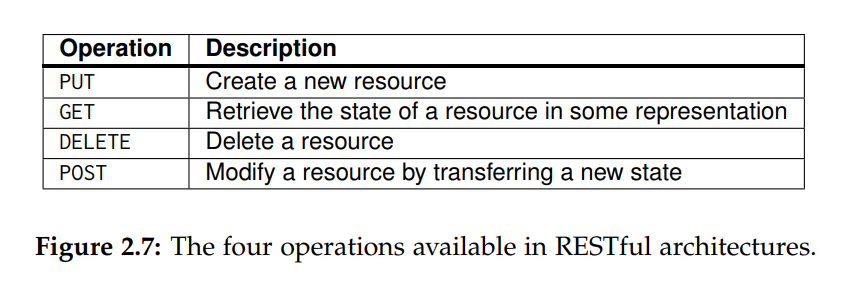
现在互联网上可用的服务越来越多，并且由服务组成的分布式系统的发展也越来越重要，研究者们已经开始重新考虑分布式系统的架构了。服务组合的一个问题是，连接各个组件很容易成为集成服务的噩梦（意思就是连太多了，太复杂了）。作为替代，我们可以将分布式系统想象由组件独自管理的一个资源的集合。资源可以被应用添加和删除，索引或修改。对于WEB来说有个著名的方法 REST。对于RESTful架构有4个特性：

1.资源被唯一的名字确定

2.所有的接口都提供最多四种相同的操作，如下图所示

3.消息完全是字描述的

4.当完成了一次调用后，组件丢弃所有调用者信息



最后一条也即无状态执行，3.4节在讨论。

为了说明Restful是如何工作的，考虑一个云存储服务，比如亚马逊S3。亚马逊S3，支持两种资源：objects，也即文件；buckets 也即文件夹。这里没有buckets套buckets的概念。一个在BucketName中的叫做ObjectName的object相当于如下的URI

<http://BucketName.s3.amazonaws.com/ObjectName>

为了创建一个bucket或者object，应用需要发送一个PUT求情。原则上，使用HTTP协议。换句话说，这是由S3正确解释的另一个Http请求。如果一个bucket和object已经存在，HTTP的错误消息将被返回。

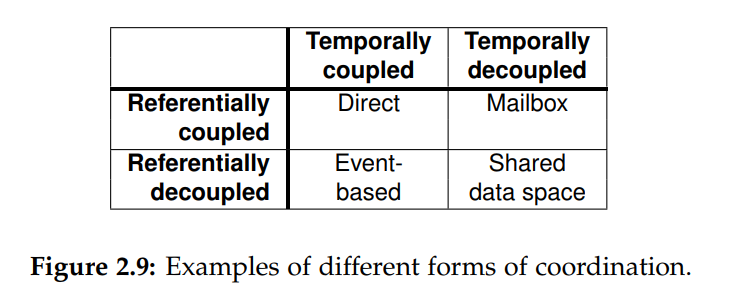
在一个类似的情况下，为了知道一个bucket有哪些object，一个应用需要发送一个get请求。S3将返回一个列表当做一个Http响应。

RESTful架构因为他的简单性得以推广。然而，关于RESTful是否会比特定的服务提供特定的服务接口进行着激烈的讨论。Pautasso比较了两者，发现啊各有千秋。特别是，RESTful架构可以提供复杂通信的解决办法。一个栗子是分布式事务的出现，也即需要所有的服务保持同一个状态。另一方面，这里也有许多RESTful架构完美匹配一些服务集成的列子，然而大量的服务接口却将问题复杂化。

**发布订阅架构**

当系统越来越大并且进程很容易的加入和离开的时候，就需要一种使得进程和进程之间的依赖变得松散的架构。大部分的分布式系统采用了处理和协调之间具有很强隔离性的架构。其思想是将系统系统当做自助操作进程的集合。在这种模型中，协调包括过程之间的合作。它形成粘合剂，将流程执行的活动绑定为一个整体

Cabri提供了一种协作模型，适用于多种分布式系统。我们在时间和引用两个维度上进行了区分，如下图2.9所示：



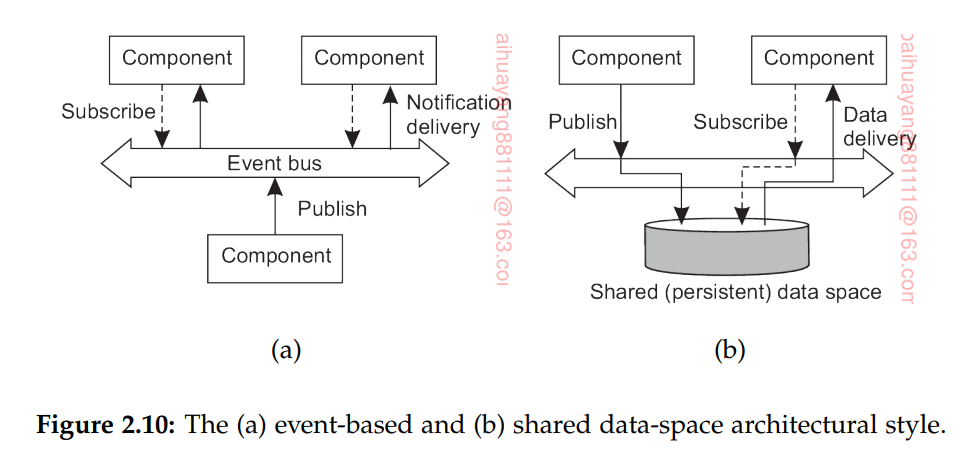
当进程在时间和引用上都欧赫是，协作是在一种比较直接的方式上发生，比如**direct coordination**。在通信中，引用耦合通常以显式引用的形式出现。例如，一个进程只能够和它已知名字或者标识的其他进程进行通信。时间耦合意味着正在通信的进程必须同时启动。在现实中，手机就是一种直接通信。

另一种协调是时间非耦合，引用耦合，**mailbox coordination**（就是发邮件吧）。这种情况下，通信双方在时间上的同步时没有意义的。取而代之的是，通信者可以将信息发送到有相中。由于必须知道通信地址，所以这还是一种引用耦合。

接下来是时间耦合，引用非耦合，**event-base coordination**。在这种引用非耦合系统中，进程之间是互相不知道彼此的。进程唯一能做的是发送一个通知用来描述一个时间的发生。假设这个通知是多种多样的，进程需要**订阅**其中一种通知。在一个完美的event-based 协调模型中，一个被发布的通知将被精确的传送给那些订阅它的进程。然而，这需要订阅者在发布者发布信息时保持运行状态。

一个很有名的在时间和引用上都非耦合的模型是**shard data space**。思想史进程的通信通过元组来实现，这个元组类似数据库中表的一行记录，记录了很多个字段。进程可以将多重元组放入数据分享空间。如果想要获取元组，进程需要提供一种搜索模型匹配这个元组。然后元组就被返回了。

数据分享空间也是和event-based协调相联系的：一个进程定义一种特定的元组，当一个元组向数据共享空间发送元组时，匹配的订阅者将被通知。在这两种情况下，我们都使用了发布订阅架构，而实际上的关键要素是，进程间没有特定的引用。在纯event-based架构和数据共享空间的架构区别如图2.10所示。我们同样将展示一种抽象的发布和订阅者匹配模式，即event bus

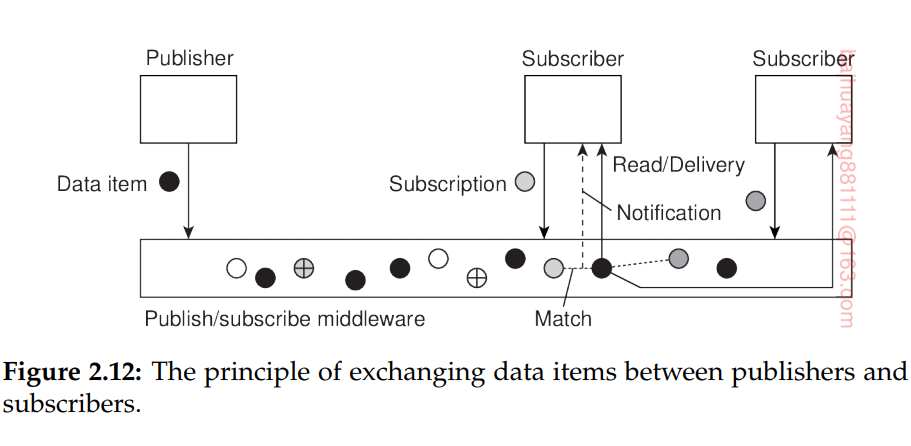


订阅发布系统一个重要的性知识，只有订阅者感兴趣的事件发生时，通信才会发生。所以，命名很重要。我们之后讨论命名，现在重要的是，在许多情况下，数据在发送和接受者之间没有很好的区分。

我们首先假定事件是用来描述一系列的属性的。当一个描述时间的通知可以被其他的进程读取时，事件将会被发送。为实现这一目标，订阅者需要需要将自己感兴趣的时间发送给中间件。这样的描述包括（属性，值）对，这种一般被称为 **topic-based publish-subscribe systems**.

作为一种替代，**content-based publish-subscribe systems，**订阅者由（属性，范围）对组成。在这种情况下，订阅者希望获取属性值落在范围内的属性信息。属性的描述同样可以使用一些谓词来代替，就像SQL。明显的，越是复杂的描述，匹配起来越是困难。

这里我们遇到了一种订阅者需要订阅的情况，如图2.12。在这些场景中，事件最终都是可用的。当匹配成功时，会出现两种场景。第一，中间件会将发送通知连同数据发送给订阅了这组数据的订阅者们。第二，中间件将会给订阅者发送通知，然后订阅者便可以自己去获取数据。



在那些需要事件立刻传送到订阅者那里的场景，中间件通常不会提供存储数据。存储通常是由另一个分离的服务操作的，或者是一些订阅者操作的。换句话说，我们这个是引用非耦合，时间耦合的系统。

当订阅者需要连同通知一起读取数据时，情况发生了改变。中间件系统需要保存数据。在这些情况下，需要额外的数据管理操作。还可以将超时时间附加到数据项，以便在到期时自动删除数据项。

事件可以容易的是订阅者的处理复杂化。为了说明，考虑一个订阅者如”notify when room ZI.1060 is unoccupied and the door is unlocked.”。通常，一个分布式系统可以通过实现一个监视门锁状态的独立场景来实现。跟随之前描述的方法架构，我们需要将这些原事件组合成一个发布数据发送给订阅者。事件组合是一个很难的工作，尤其是当原始事件是从分散在分布式系统中的资源获得的。

订阅发布系统就是这样，关键问题是有效的和可伸缩的实现订阅者和通知的匹配。从外在看，由于非耦合的特性，发布订阅架构提供了构建大型的可伸缩分布式架构的潜力。另一方面，在基于内容的发布-订阅系统中， 且不丧失这种独立性的情况下，设计可伸缩的实现不是一项简单的工作。

## 2.2 中间件组织

在之前的章节，我们讨论了一些构建组织分布式系统的架构风格。现在我们聚焦在中间件的实际组织上，它是独立于整个分布式系统的。特别是，两种设计模式经常被使用在中间件架构中：包装和拦截。每一种都会产生新的问题，但是都会向新的目标前进：开放性（1.2节讨论过）。当我们可以在运行时将中间件组织好时，最终的开放性也会达成。

**包装**

当我们使用现有的组件构建分布式系统时，我们会遇到一个问题：那些遗留的组件的接口对某些应用并不适用。在1.3节我们讨论了中间件可以作为企业应用程序集成的设施，但我们仍然认为组件可以通过本地接口访问。（就是不通过中间件也能访问呗）

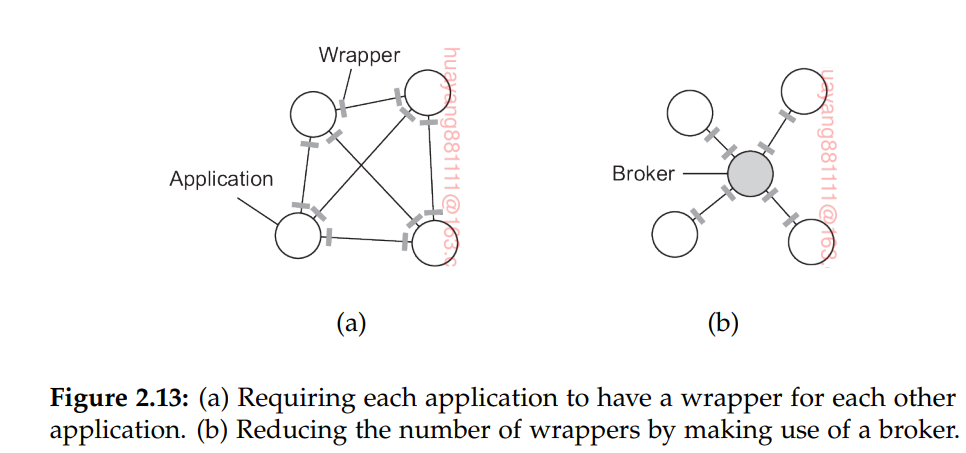
一个包装器或者适配器是一种特殊的组件，它们可以将原来的方法转换成对客户端应用可用的接口。本质上，它解决了接口不兼容的问题。

包装是在面向对象中提出的，在分布式系统中，包装器不仅仅是简单的接口转换。例如，一个对象适配器是一个组件允许应用执行远程的对象，即便这些对象是关系数据库表上操作的或者是库函数操作的。

另一个例子是，亚马逊S3存储服务。这里有两个接口可用，一个是RESTful架构，另一个是传统的架构方法。对于RESTful接口，客户必须使用HTTP协议，本质是与传统的Web服务通信，传统Web服务器充当实际存储服务的适配器，剖析传入请求，传入给S3内部的专门服务器。

包装器对于可扩展系统是十分重要的。对于获取开放性很重要。另一方面，如果一个应用A负责管理应用B所需要的数据，一个方法是为B开发一个特殊的包装器使得其可以访问A的数据。所以如果分布式系统有N个节点，则需要N方个数量级的包装器

所以，组件包装器的减少需要通过中间件来完成。一种方式是实现一个所谓的**broker，**用来处理所有不同应用之间的访问。一个经常用到的类型是message broker 我们将在4.3节讨论。在message broker中，应用简简单单的发送给message broker 他们所需要的请求。这个broker，代理所有相关的应用，已经他们之间的联系，并将相应组合并转换返回给最初的应用。原则上，由于一个broker对每个应用提供了一个接口，我们最多需要N的数量级个包装器，如图2.13所示。



**拦截器**

概念上，一个拦截器就是一个软件组织，用来对输入进行控制并执行一些特殊代码的东东。拦截器最初是中间件来应对应用特殊需求的手段。它对于中间件的开放起到了很重要的作用。要使拦截器通用，需要做大量的工作，并且还不清楚的是，在这种通用性下，简单性和适用性能否被满足。在某些情况下，一些受限的拦截器也会增加软件和操作系统的管理。

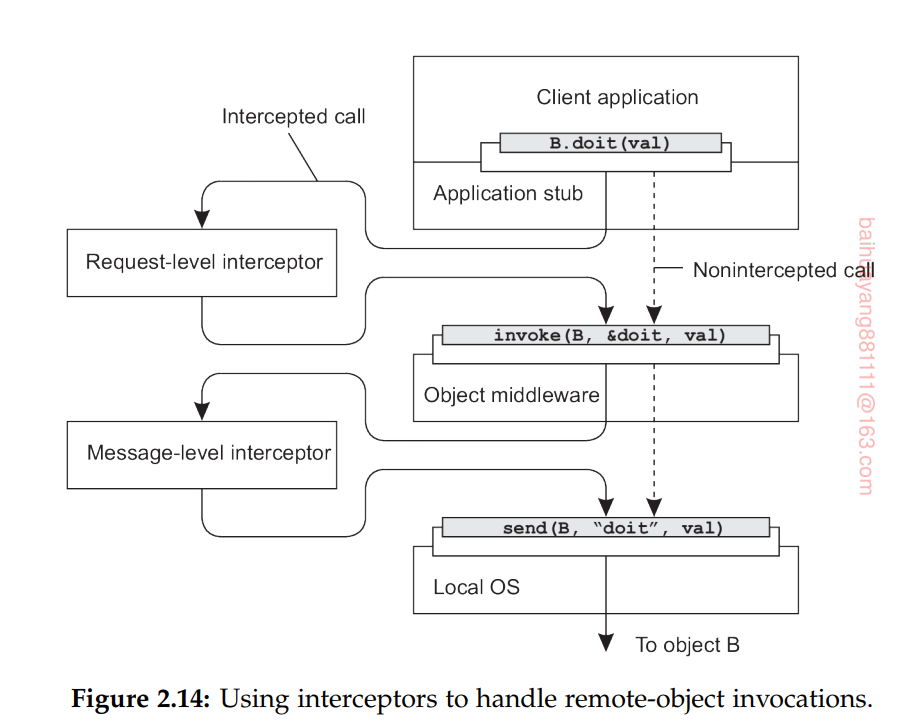
为了具体化，考虑object-based的分布式系统中的拦截器。思路很简单：object A 可以调用 object B的方法，然而后者却和A不在一个机器上。我们将在本书后面讨论这些细节，这种远程 调用可以分三步：

1.A和B都在本地提供同样的接口。

2.A的调用被穿换成通常的object调用，可以通过中间件来实现

3.object 调用被转换成消息在A的本地操作系统的网络的传输层发送。

如图2.14所示，在第一步之后，B.doit(val)的调用被转换成一个通常的调用invoke(B, &doit, val)，里面包含了B的方法名和参数。现在想想object B被复制。在这种情况下，每个副本需要被执行。这里就是拦截器发生作用的地方。在请求层的拦截器，可以将invoke(B, &doit,val)这个调用在每一个副本上发生。优雅的地方在于A并不知道B会产生副本，同样的中间件不需要拥有特殊的组件来处理这些副本的调用。只有这些请求层的被设置在中间件之前的拦器器知道B的副本的存在。



最后，一个远程object调用需要在网络中传输。实际上意味着，本地提供的消息接口需要被调用。在这一层，一个消息层的拦截器会帮助将调用传递给目标object。例如，想象参数val是一组巨大的数组数据。在这种情况下，聪明的做法是通过拦截器将数据碎片化成很小的部分后，再将其在目标处组装起来。这种碎片化会提高性能和可靠性。当然，中间件不用感知到这种碎片化，低层的拦截器会通过本地操作系统透明地处理余下的通信过程。

**可变的中间件**

包装器和拦截器是用来扩展和适配中间件的。适配的需求来源于分布式应用的环境的持续变化。变化包括可移动性，网络服务质量的不同，硬件故障以及电池。这个任务应该放在中间件，而不是让应用去负责应对这些修改。此外，随着分布式系统的增大，很少能够通过临时关闭来更改其部件。我们需要的是能够随时进行更改。

这种环境的强烈影响导致了中间件的设计者们开始考虑适配软件的构建。根据Parlavantzas和Coulson的说法，可修改的中间件应当不仅仅满足可适配，同样应该能够满足有目的的修改来替代让其宕机。在这种情况下，可以认为拦截器提供了一种方法来适应标准的控制流。在运行期间替换软件组件也是一种修改系统的方法。实际上，最有影响力的方法是动态的从各个组建中构建中间件。

基于组件的设计侧重于通过组合支持可修改性。一个系统可以在设计时进行静态的配置，或者在运行时动态的配置。后者需要支持延迟绑定，这是一个在编程环境中成功应用的技术，同样也适用于随意加载和卸载模块的操作系统。关于在运行期自动选择最好组件实现方法已经在研究了，但是，对于分布式系统，这个过程仍然很复杂，尤其是考虑到替换一个组件需要确切的知道对其他组件的影响时。在很多情况下，很多组件都不是那么独立的。

如果我们不能动态的改变这些组成中间件的软件，至少我们应该能够在运行期安装和卸载这些软件。另外，对于每个接口都需要提供显示的规范。如果想要在组件之间保持状态，则需要进一步的措施。总的来说，组织可修改的中间件需要特别注意。

## 2.3系统架构

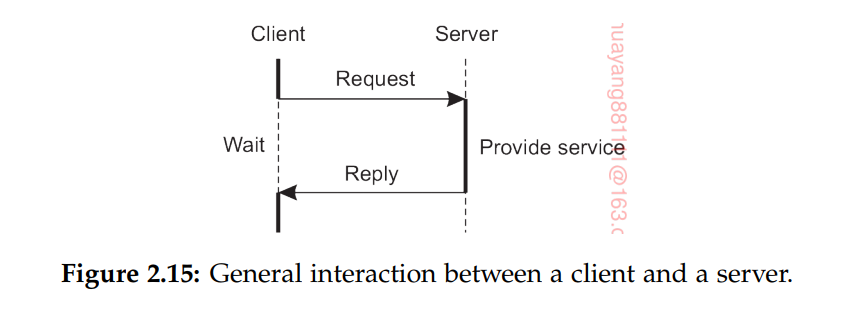
现在我们已经特别讨论了一些已经应用了的架构风格，接下来我们来看看有多少分布式系统是根据软件组件的位置来组织的。软件的组件，它们之间的交互和他们的位置决定了软件架构的实例，也即系统架构。我们将研究中心化和去中心化组织，以及一些混合结构。

**中心化组织**

尽管我们对分布式系统有很多地方意见不一致，有一个地方我们持有相同的观点：从服务器请求服务的角度考虑有助于管理分布式系统的复杂性。接下来，我们先讨论单层组织，之后讨论多层组织。

**简单的客户端-服务端架构**

在最基础的客户端服务端模型中，一个分布式系统的进程被划分为两部分。一个服务器就是一个实现了特殊服务的进程，比如一个文件系统或者数据库服务。一个客户端就是一个请求服务的服务器，发送请求然后等待回应。在客户端-服务端中，有request-reply behavior 如图2.15所示。



当网络是可靠的时候，客户端和服务端之间的通信可以通过一些简单的通信协议来连接。在这种情况下，当客户端发送一个服务请求是，它会简单的将有用的信息打包传送给服务端。而服务端则等待一个请求的进入，然后执行，并将结果打包返回给发送端。

使用一个无连接的协议会有明显的效率优势。只要消息不丢失或损坏，请求/应答协议就可以正常工作。不幸的是，网络中传输故障的发生是没办法组织的。我们能做的只是当没有产生响应时，客户端重新发送信息。问题是，客户端无法判断究竟是请求失效了还是响应丢失了。如果是响应丢失，可以重新发送一次请求。如果操作是像“从银行取10000美元”，最好的方式还是报告这个错误。另一方面，如果操作是“告诉我账户余额是多少”，可能重新发送请求是更好的选择。当一个操作可以重复多次而没有副作用时，我们称之为**幂等**的。有些操作是幂等的而有些不是。我们应当清楚没有简单的方法来处理那些丢失的消息。我们将在8.3节讨论佮处理传输错误。

作为替代，很多客户-服务器系统使用可靠的传输协议。即便在本地网络中，由于这种性能解决方案的性能较低而并不合适，它却在广域的不可靠通信系统中运行的很好。例如，几乎所有的网络应用协议都是基于TCP/IP连接。在这种情况下，当一个客户端发送一个请求时，他首先会先向服务端建立一个连接。服务端也会使用相同的连接来发送数据，之后连接中断。问题就在于建立和撤销连接是相对好资源的，尤其是当请求和回应的数据很小时。

多年来，客户-服务器 模型一直饱受争议。一个重要的原因是无法清晰的区分客户端和服务端。例如，一个服务端由于会向数据库服务器请求数据，也可当做为一个客户端。

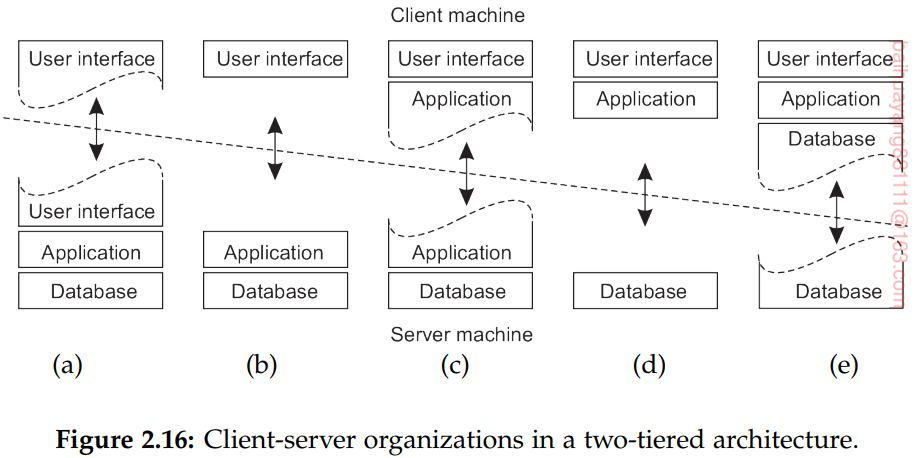
**多重架构**

到目前为止三种逻辑层的有效区分（用户，处理，数据），使得一些构建在物理上分布的客户-服务应用变得可能。最简单的组织会有两种机器：

1.客户端机器只去实现用户层的程序

2.服务端机器实现剩余的 处理层和数据层

在这种组织中，所有任务都是由服务端处理，而客户机实际上只是一个简单的终端，可能只有一个方便的图形界面。然而，这还有一些其他可能。如2.1节所介绍的，很多分布式应用可以分为三层，用户接口层，处理层，数据层。组织客户机和服务器的一种方法是将这些层分布到不同的机器上，如图2.16所示。

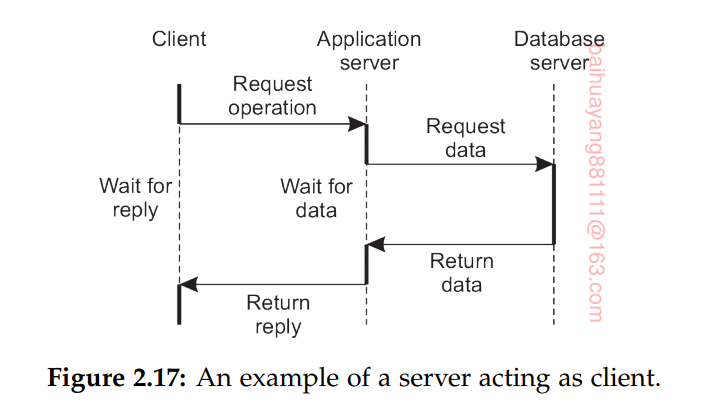


第一步，我们首先区分两种机器，客户端机器和服务端机器，也即我们讨论的**两层架构**。.一个可能的组织是将和终端相关的用户接口放置到客户端机器上，并让应用远程控制来展示其数据，如2.16(a)。一个替代的方式是放置所有的用户接口软件到客户端，如2.16(b)。在这些例子中，我们最终将应用划分为用来和应用交互的图形化的前端和应用本身。在这种模型下，前端不进行任何数据操作，只是展示应用的数据。

继续沿着这条推理线，我们还可以将应用程序的一部分移到前端，如图2.16(c)所示。这个例子是有意义的，比如应用程序在使用表单处理之前需要完全填写。前端会校验表单的正确性，以及和用户的相关性。另外一个例子是，需要在前端检查单词拼写的文字处理应用。

在很多客户端-服务端环境中，2.16(d)和2.16(e)是特别流行的。当客户端是一个连接在分布式系统或分布式数据库的PC机或者工作站时，这种组织将被使用。特别是，很多应用都是跑在客户端的，不过所有的文件操作和数据库操作都是在服务端。例如，很多银行app都是跑在用户自己的机器，用户在自己的机器上完成事务。一旦完成，应用将连接银行服务器的数据库并上传这些事务来惊醒处理。图2.16(e)表示部分客户端也将包婚一部分数据。例如：在WEB系统中，客户端浏览器会缓存一些数据。

到目前为止我们仅仅区分了客户端和服务端，但我们忽略了有时服务端也会扮演客户端的角色，如图2.17.就是一个**三层架构**。



在这种体系结构中，传统上构成处理层一部分的程序由单独的服务器执行，但也可以在客户机和服务器机器之间进行部分分发，一个典型的例子是，什么时候这种三层架构会被使用。一个单独的进程，称为事务处理监视器，协调可能不同数据服务器之间的所有事务。

另一个比较难以理解的例子是Web站点。在这个例子中，一个Web应用去给服务器发送请求，这个服务器又去请求数据库。例如，应用程序服务器可能负责运行代码来检查电子书店提供的某些商品的可用库存。为此，它可能需要与包含原始库存数据的数据库进行交互。

**去中心化组织结构，peer-to-peer 系统**

多层客户端与服务端的架构都是直接将分布式应用分成用户接口层，处理层和数据管理层。不同的层与应用的逻辑组织直接相关。在许多业务环境中，分布式处理相当于将客户机-服务器应用程序组织为多层体系结构。我们将这种分布成为**垂直分布（user->process->database）**垂直分布的特性石，不同的逻辑组件分布在不同的机器上。这个术语与分布式关系数据库中使用的垂直碎片的概念有关。

同样的，从系统管理的角度来看，垂直分布的好处是，功能在逻辑和物理上被分不到不同的机器上，每一个机器都是为一组特定的功能而定制的。然而，垂直分布只是客户端和服务端的一种组织方式。在现代架构中，很多都是**水平分布**。在这种分布上，一个客户端或者服务器都在逻辑上分为了等效的一部分，每一个部分都会对自己的数据进行操作，很好的均衡了负载。这节我们将看到现代架构水平分布的例子，**peer-to-peer 系统**。

从高层次的角度来看，构成peer-to-peer系统的过程都是等价的。这意味着分布式系统的执行功能由系统的每个进程来表示。最为一个结果，进程之间的连接都是对称的，一方是客户端，同时也是服务端。

当出现了这种对称行为后，peer-to-peer架构开始解决如何在覆盖网络组织进程。覆盖网络是指网络节点代表着进程，网络连接代表着通信。一个节点可能不能够直接去和任意节点通信，但是却可以间接的通过网络来实现。存在两种覆盖网络：结构化的和非结构化的。这两种类型都在Lua中被提及。Buford和Yu也提供了很多peer-to-peer系统。Aberer提供了一个参考体系结构，允许不同类型的peer-to-peer系统。AndroutsellisTheotokis和Spinelli提供了一项从内容分发角度进行的调查。最后，Buford等人[2009]、Tarkoma等人[2010]和Vu等人[2010]超越了调查的层面，形成了足够的教材供初学或继续学习。

**结构化的peer-to-peer系统**

就像名字一样，一个结构化的peer-to-peer系统会形成一个特定的拓扑结构：环状，二叉树，或者网格等。这种拓扑结构被用来方便查询数据。结构化对等系统的特点是，它们通常基于使用所谓的无语义索引。这表示每一个数据都会关联一个关键词，这个关键词被用来当做索引。通常使用hash函数：



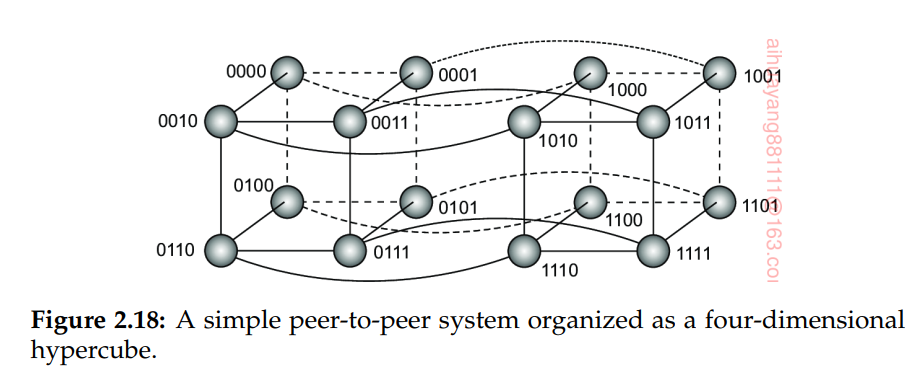
这个peer-to-peer系统整体上可以负责存储键值对。每个节点都从相同的哈希值集合中分配一个标识符，每个节点负责存储与特定键子集关联的数据。本质上，系统可以看做是实现了一个**分布式的哈希表，DHT。**

通过这种方法，相当于将结构化的peer-to-peer系统当做一个数据查找系统。 也就是这个系统提供了一个有效的查找节点的方法：



这就是结构化的peer-to-peer系统的重要作用。任何节点可以通过关键字查找到，这就归结为寻找有效的路由来通过给定的关键字查找节点。

为了说明这些，我们考虑一个简单的peer-to-peer系统，这个系统由一些固定数目的节点组成一个超立方体。这个超立方体是一个n维的立方体。在图2.18中，从四个维度展示了这个超立方体。它可以看作是两个普通的立方体，每个立方体有8个顶点和12条边。为了将超立方体扩展到五个维度，我们将向图中添加另一组两个相互连接的立方体，将两半中相应的边连接起来，以此类推。



对于这个系统，每个数据都关联了十六个节点。可以将key hash到如下的范围(0-2^4-1)。现在假设标识符为0111的节点被请求查找密钥为14的数据，该数据对应于二进制值1110。在这个例子中，我们假设1110这个节点复杂存储key为14的数据。节点0111可以做的是，将请求转发给更接近节点1110的邻居。在这里就是0110和1111。如果选择了0110，将直达数据所在的1110节点。

**非结构化peer-to-peer系统**

结构化系统保持了一种特殊的，确定的覆盖网络。相反，一个非结构化的peer-to-peer系统每个节点维持一组临时的邻居节点。得到所谓的随即图：即u节点和v节点之间存在一条概率还P(u,v)的变。理想情况下，所有节点之间存在边的概率都是相同的，但实际上这种概率是一组范围。

当一个节点加入非结构化的peer-to-peer系统中时，会得到系统的其他节点列表。这个列表可以帮助找到更多的peers等。实际上，节点几乎连续的更改其本地列表。例如，当发现一个邻居节点不再响应时，就应该将它替代。这里也会有其他情景，我们将简短的讨论。

不同于结构化的peer-to-peer系统，不能再通过提前决定好的线路来查找数据，因为邻居节点都是随机构成的。相反，一个非结构化的peer-to-peer系统我们需要重新排序来查找节点。我们来看两个极端例子并考虑如何在其中查找节点。

**Flooding（泛洪）：**在flooding这种情形种，一个节点u对它的所有节点获取数据的请求。当某个节点遇到过这个请求，就会忽视它。否则，v将在本地求情数据。如果v获得了数据，v将向u直接发出响应，并将数据发送给直接前驱节点，前驱节点再发送给前驱节点，以此类推。如果v没有获取请求数据，它将把数据发送给它的邻居。

明显的，flooding的代价是非常昂贵的，因为请求通常会有一个TTL（Time-to-Live）值，即允许转发请求的最大跳数。选择合适的TTL很重要：太小的话，请求可能达不到数据。太大则导致网络通信资源浪费。

作为一种选择，可以先将TTL设置为1，即只访问邻居节点，如果差找不到数据，则增大TTL的值。

**Random walks（随机游走）：**节点u随机找一个邻居节点v获取数据。如果v没有数据，则将请求转发到它的一个随机节点上。这就是随机游走。明显的，一个随机游走会减少网络流量，但是会花更多的时间来寻找到请求的数据。为了减少等待时间，一个节点可以同时发出n个邻居节点进行随机游走。实际研究表明，这样做的时间花费，大约减少了一个因子n。Lv 报到处，n设置为16或者64，会比较有效率。

随机游走同样需要设置TTL来终止其继续转发，同样的，也需要判断这个节点之前是否接受过同样的请求，如果有，则抛弃。

注意，这两种方法都不依赖于特定的比较技术来确定何时找到了请求的数据。对于结构化的peer-to-peer 系统，我们使用key来比较，对于上述两种方法，任何比较技术都可以。

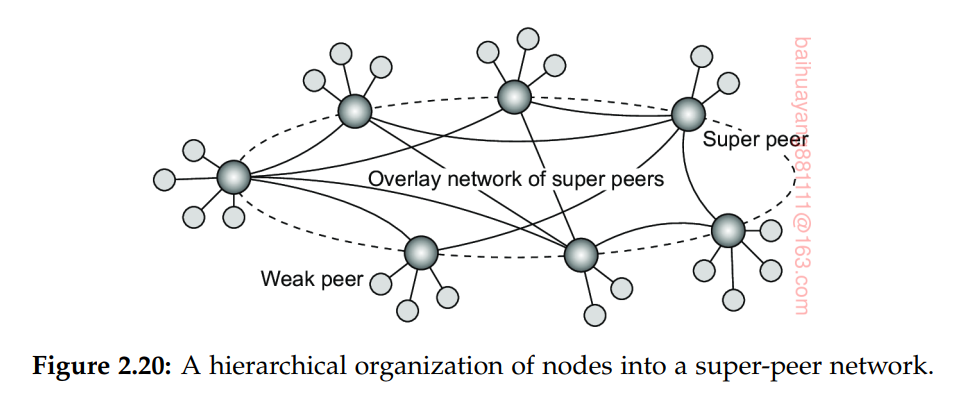
在flooding和随机游走之间存在着基于策略的搜索方法。节点可以根据后续节点的响应反馈，来决定下次是否将其作为后续节点的首选。同样的，我们也可以限制flooding到更少的邻居，但在任何情况下，都应先考虑有很多邻居的邻居。

**Peer-to-peer网络的混合组织**

值得注意的是，在非结构化的peer-to-peer系统中，当网络增长时，定位相关的数据会成为问题。这个伸缩性问题的原因是很简单的：这里没有特定的路由去寻找特定的数据，本质上，节点唯一可以使用的技术就是通过泛洪或随机遍历网络来搜索请求。作为一种替代，peer-to-peer系统都是用索引来维护特殊的节点。在其他情况下，放弃peer-to-peer系统的对称性是明智的。考虑节点间提供资源的节点协作问题。例如，在一个协作的CDN（内容分发），节点可以提供存储空间来承载Web文档的副本，从而允许Web客户机访问附近的页面，从而快速访问它们。我们需要考虑的是找到哪里是存储副本最好的位置。

那些维护索引或者充当代理的节点都称之为**super peers。**像名字一样，super peers 的建立使得peer-to-peer网络向层次化组织靠拢。一个简单的例子如图2.20。在这种组织中，每一个普通的peer，在这称之为**weak peer**，会被当做一个client与super peer 相连。所有weak-peer的通信，都要通过super peer。

在许多情况下，weak peer 和 super peer 之间的联系是固定的：每当一个weak peer 加入网络时，它都会与一个 super peers 进行连接，直到它离开了网络。明显的，super peers 将会长时间留在网络中。为了弥补super peer 节点的潜在不稳定行为，可以部署备份方案，比如讲每个super peer 节点与另一个 super peer 节点配对，要求 weak peer节点同时连接到两个super peer。



和一个super peer 有一个固定的连接并不总是一个最好的选择。例如，在文件共享系统上，weak peer最后能够和那些保持了week peer相关文件的索引的super peer建立联系。在这种情况下，当weak peer 需要找寻一个特殊的文件时，super peer可以很容易的找到他。Garbacki等人[2010]描述了一种相对简单的方案，在该方案中，weak-peer和super-peer之间的关联可以随着weak-peer发现更好的super-peer而改变。特别是，返回查找操作结果的super peer比其他super peer具有优先级。

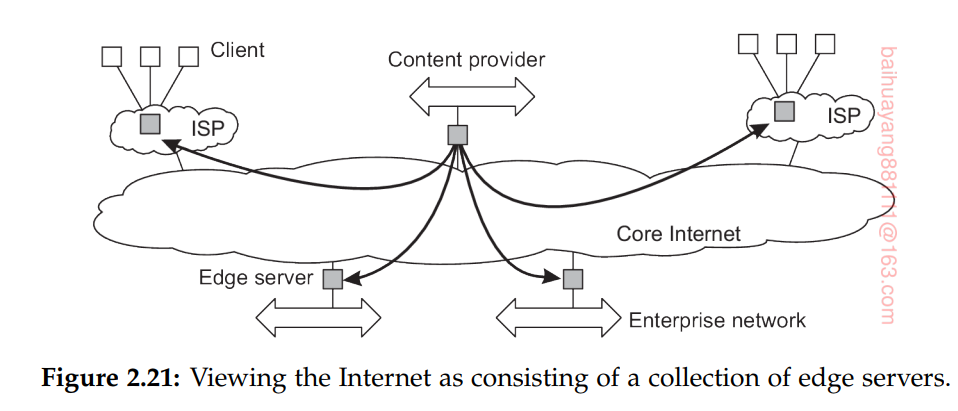
正如我们所看见的，peer-to-peer 网络提供了一系列弹性的方法来让节点加入和离开网络。然而，当使用super peer网络时，一个问题出现了，如何选择这种符合条件的super peer节点呢？这个问题类似于**领导节点选取问题**，我们将在6.4节讨论。

**混合架构**

到目前为止，我们聚焦在客户服务器和一些peer-to-peer架构上。许多分布式系统结合了许多架构的特性，正如我们在peer-to-peer网络中遇到的一样。这一节，我们将塔伦一些特殊的分布式系统，在这些系统客户-服务端会和去中心化架构相关。

**Edge-server systems**

一类重要的混合架构的分布式系统是 **edge-server systems**。这类系统的服务端被放置在系统的“边缘”。这个边界是由企业网络和实际Internet之间的边界形成的，例如，由Internet服务提供者**Internet Service Provider(ISP)**提供的边界。同样，如果家中的终端用户通过ISP连接到Internet，则可以认为ISP位于Internet的边缘。这样就构成了图2.21的网络组织。



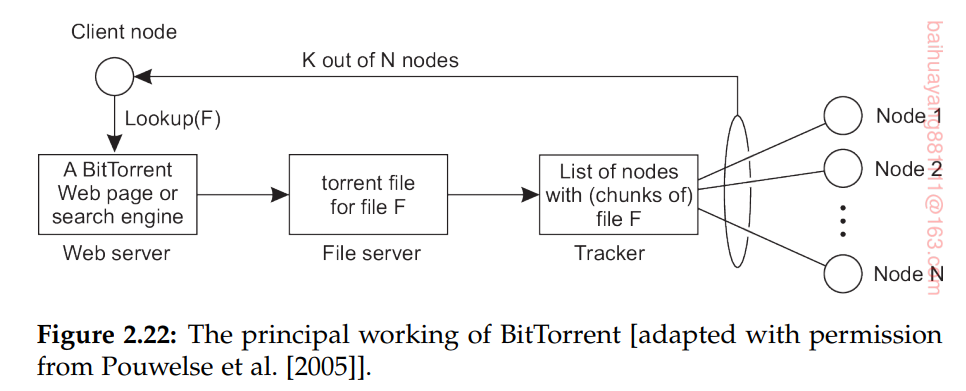
终端用户或者客户端通常都是通过边缘服务器来连接到互联网的。边缘服务器的目的是提供内容，或者是经过处理或者转码的内容。更有趣的是，一组边缘服务器，可以被用来优化内容与应用的分布。基本模型是，对于特定的组织，一个边缘服务器充当所有内容来源的源服务器。该服务器可以使用其他边缘服务器来复制Web页面等。

现在边缘服务系统的概念发展得更近了一步，数据中心以云计算为核心，边缘服务来辅助计算和存储，本质上构成了分布式云系统。在**fog computing**中，甚至终端用户设备也构成系统的一部分，并且(部分)由云服务提供商控制。

**协作的分布式系统**

混合结构在协作的分布式中被使用。这些系统的主要问题是第一步的启动问题，为此通常一个传统的客户-服务方案被部署。一旦有节点加入系统，便可使用去中心化的协作方式。

为了具体阐述，我们讨论广为流传的**BitTorrent**的文件分享系统。BitTorrent是一个peer-to-peer文件下载系统。它的工作流如图2.22所示。基础思想史，当一个终端用户查找一个文件时，他会从其他用户那里下载这个文件的各个组成块，直到这些下载的块可以完整的组成一个完整的文件。一个重要的设计目标是确保协作。在很多的文件分享系统中，一个明显的问题是很大一部分人只是单纯的下载了文件，然而并没有其他的贡献，这种现象称作**free riding**。为了阻止这种情况，在BitTorrent中，只有提供了下载内容给其他人的用户，可以下载文件。



为了下载一个文件，用户需要访问一个全局文件夹，也即一个网站。这个文件夹包含了torrent 文件的引用。一个**torrent 文件**包含了需要下载特定文件的必要信息**。**特别是，它包含了一个对**tracker**的连接，这个tracker准确记录了被请求文件被多少个活跃节点保存着。活跃节点也是当前正在下载文件的节点。明显的，这里将会有许多不同的trackers，即使每个文件（或一组文件）通常只有一个tracker。

一旦确定可以从何处下载文件的块，下载节点将被激活。这时，他将会帮助那些提供块下载的节点。这种强制性来自一个非常简单的规则：如果节点P注意到节点Q下载的比上传的多，P可以决定降低向Q发送数据的速率。如果P可以从Q下载一些东西，那么这个方案就可以很好地工作。由于这个原因，节点经常被提供对许多其他节点的引用，使它们能够更好地交易数据。

很明显，BitTorrent结合了中心化和非中心化的解决方法。事实证明，这个系统的瓶颈是tracker。作为BitTorrent的一种可替代的实现，节点被加入到一个分离的结构化的peer-to-peer系统中帮助追踪文件下载。实际上，中央跟踪器的负载现在分布在参与节点上，每个节点充当相对较小的torrent文件集的跟踪器。这保持了跟踪器协调文件协作下载的原始功能。然而，我们注意到，在当今使用的许多BitTorrent系统中，跟踪功能实际上已经被最小化到一次性提供当前参与下载文件的peer点。从那一刻起，新参与的peer点将只与那些peer点通信，而不再与初始跟踪器通信。请求文件的初始跟踪器通过所谓的magnet link在DHT中查找。我们将在5.2节介绍DHT为基础的查找。