协调

在前面的章节，我们研究了进程和进程之间的通信。通信很重要，但这并不是全部。更重要的是进程之间的协调和同步。这种协调是以各种命名为基础的，通过这种方式可以共享资源。

在本章中，我们主要关注进程如何同步和协调它们的操作。例如，重要的是多个进程不能同时访问像文件这种共享资源，而是通过协作授予彼此临时的文件的独占访问权。另一个例子是，多个进程有时可能需要就事件的顺序达成一致，例如来自进程P的消息m1是在来自进程Q的消息m2之前还是之后发送的。

同步和协调是两个密切相关的现象。在**进程同步**中，我们确保一个进程等待另一个进程完成其操作。在处理数据同步时，问题是要确保两组数据是相同的。当涉及到协调时，目标是管理分布式系统中活动之间的交互和依赖关系[Malone and Crowston, 1994]。从这个角度来看，可以认为协调封装了同步。（就是协调包括了同步，因为同步要确保两组数据相同）

事实证明，与单处理器或多处理器系统相比，分布式系统中的协调常常要困难得多。本章所讨论的问题和解决方案本质上是相当普遍的，并且出现在分布式系统的许多不同情况中。 我们首先讨论基于实际时间的同步问题，然后讨论同步，在同步中，只有相对顺序才重要，而不是绝对时间顺序。

重要的是，在许多情况下，一组进程可以指定一个进程作为协调器，这可以通过选择算法来实现。

我们将在单独的一节中讨论各种选举算法。在此之前，我们研究了一些用于协调共享资源的互斥的算法。作为一类特殊的协调问题，我们还深入研究了将进程置于多维平面上的定位系统。当处理非常大的分布式系统时，这样的定位非常方便。

我们已经遇到了发布-订阅系统，但是还没有详细讨论如何将订阅与发布匹配起来。有很多方法可以做到这一点，我们看一下中心式和非中心式的实现。

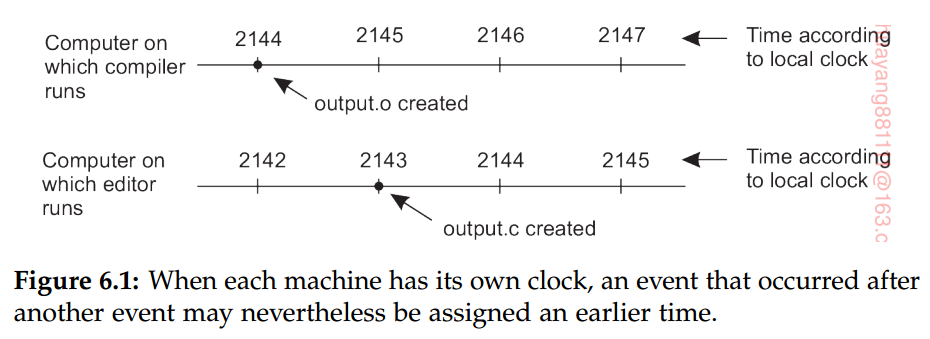
最后，我们考虑三个不同的基于gossip的协调问题:聚合、对等抽样和覆盖构造。分布式算法有各种各样的风格，它们是为非常不同类型的分布式系统开发的。

6.1时钟同步

在一个中心化的系统中，时间是明确的。当进程想知道时间时，它只需调用操作系统。如果进程A询问时间，稍后进程B询问时间，那么B得到的值将大于(或可能等于)A得到的值。它肯定不会更低。然而在分布式系统中，按时间达成协议并非易事。作为一个简单的例子，请考虑一下Unix make程序缺少全局时间的影响。（这么说是因为分布式系统，不容易获取一个准确的统一时间）。通常，在Unix中，大型程序被分割成多个源文件，因此对一个源文件的更改只需要重新编译一个文件，而不是所有文件。如果一个程序由100个文件组成，不必重新编译所有文件，因为一个文件已经被修改，这大大提高了程序员的工作速度。

make正常工作的方法很简单。当程序员修改完所有源文件后，他运行make，它检查所有源文件和目标文件最后修改的时间。如果源文件input.c有时间2151和相应的目标文件输入。o有时间2150,make知道input.c自输入以来已经更改。创建了o，因此必须重新编译input.c。（编译文件时间小于源文件，需要重新编译）。另一方面，如果output.c有时间2144和output.o有时间2145，则不需要编译。因此make遍历所有源文件，找出需要重新编译的文件，并调用编译器重新编译它们。

现在想象一下，在一个没有全球时间协议的分布式系统中会发生什么。假设output.o的时间是上面提到的2144，不久之后output.c被修改，但是被分配了时间2143，因为它机器上的时钟稍微落后，Make不会调用编译器,如图6.1所示。生成的可执行二进制程序将包含来自旧源和新源的目标文件的混合。（就是有些新修改的文件没有编译） 它可能会崩溃，程序员会疯狂地试图理解代码的错误。



还有更多的例子需要精确地计算时间。上面的示例可以很容易地重新定义，以便对时间戳进行归档。此外，想想金融经纪、安全审计和协作感知等应用领域，就会发现准确的时间安排非常重要。由于时间对于人们的思维方式是如此的基本，而且没有让所有的时钟同步的效果可能是如此的戏剧化，所以我们从这个简单的问题开始研究同步是合适的。在一个分布式系统中，有可能同步所有的时钟吗?答案出人意料地复杂。

**物理时钟**

几乎所有的计算机都有一个记录时间的电路。尽管“时钟”一词被广泛用于指代这些设备，但它们实际上并不是通常意义上的时钟。定时器可能是一个更好的词。计算机定时器通常是精密加工的石英晶体。当石英晶体处于张力下时，其振荡频率取决于晶体的种类、切割方式和张力的大小。与每个晶体相关联的是两个寄存器、一个**计数器**和一个**保持寄存器**。晶体的每一次振动都会使计数器减小1。当计数器为零时，将生成一个中断，并从持有寄存器重新加载计数器。通过这种方式，可以为定时器编写程序，以每秒60次或任何其他所需频率生成中断。每个中断称为一个时钟滴答。

当系统启动时，它通常要求用户输入日期和时间，然后将日期和时间转换为某个已知起始日期之后的滴答数，并存储在内存中。大多数计算机都有一个特殊的电池备份CMOS RAM，这样就不需要在以后的引导中输入日期和时间。在每一个时钟滴答声中，中断服务过程都会向存储在内存中的时间增加1。这样，(软件)时钟就能保持最新。

对于一台计算机和一个时钟，如果这个时钟差一点点并不重要。由于机器上的所有进程使用相同的时钟，所以它们在内部仍然是一致的。例如，如果文件input.c有时间2151和文件output.o有时间2150,make会重新编译源文件，即使时钟差了2，即实际时间分别是2153和2152。（对于单一计算机系统，只需要看相对时间）真正重要的是相对时间。

一旦引入多个cpu，每个cpu都有自己的时钟，情况就会发生根本性的变化。尽管晶体振荡器运行的频率通常相当稳定，但不可能保证不同计算机中的晶体都以完全相同的频率运行。实际上，当一个系统有n台计算机时，所有n个晶体的运行速度略有不同，导致(软件)时钟逐渐失去同步，并在读出时给出不同的值。这种时间值上的差异称为**时钟偏移**。由于这种时钟偏差，程序期望与文件、对象、进程或消息关联的时间是正确的，并且独立于生成时间的机器(即)可能会失败，正如我们在上面的make示例中看到的。

在某些系统(如实时系统)中，实际时钟时间很重要。在这种情况下，需要外部物理时钟。由于效率和冗余的原因，通常认为需要多个物理时钟，这就产生了两个问题：(1)我们如何将它们与真实世界中的时钟同步，(2)我们如何将时钟彼此同步?

保持全球时间的基础是所谓的全球协调时间（UTC）。UTC是所有现代民用计时的基础，是一个世界性的标准。为了给需要精确时间的人提供协调世界时，世界上大约有40个短波电台在每一个UTC秒的开始时刻发送一个短脉冲。这些台站的精度在±1 msec左右，但由于大气的随机波动会影响信号路径的长度，实际精度并不好于±10 msec（毫秒）。

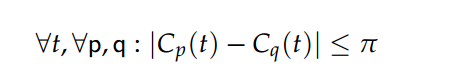
几颗地球卫星也提供UTC服务。 地球同步环境业务卫星可以提供0.5 msec的UTC精度，其他一些卫星甚至做得更好。通过结合来自多个卫星的接收信号，地面时间服务器的精度可以达到50 nsec。UTC接收器是商用的，许多计算机都配备了一个。

**时钟同步算法**

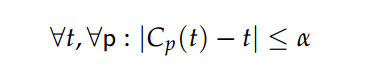
如果一台机器有一个UTC接收器，目标就是保持所有其他机器与它同步。如果没有一台机器有UTC接收器，那么每台机器都会记录自己的时间，目标是尽可能地让所有机器保持在一起。已经提出了许多算法来实现这种同步。

所有的时钟都是基于一些谐振子：一种以一定频率共振的物体，我们由此可以推导出时间。原子钟是以铯133原子的跃迁为基础的，铯133原子的跃迁不仅很高，而且很稳定。大多数计算机中的硬件时钟使用基于石英的晶体振荡器，石英振荡器也能产生非常高、稳定的频率，尽管不如原子钟稳定。计算机中的软件时钟是由计算机的硬件时钟派生而来的。特别地，硬件时钟假定每秒导致中断f次。当这个计时器停止时，中断处理程序将向计数器添加1，该计数器记录自过去某个约定时间以来的滴答(中断)次数。这个计数器就像一个软件时钟C，在频率F处共振。

当UTC时间为t时，用Cp(t)表示p机上软件时钟的值。时钟同步算法的目标是保持分布式系统的两台机器的软件时钟差在**π**精度內。



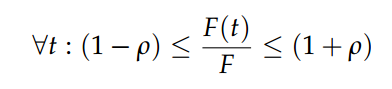
请注意，精度仅指分布式系统中机器之间时钟的偏差。当涉及到UTC时间时，我们称之为准确度。



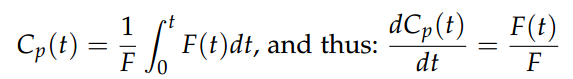
时钟同步的整个概念是我们保持时钟精确，称为内部同步，而准确，称为外部同步。一组准确的时钟在α,内精确绑定π= 2α。然而，精确并不能让我们对时钟的准确性得出任何结论。

在理想状态下，对于所有的p和t，我们将有Cp(t) = t。不幸的是，硬件时钟以及软件时钟都受到**时钟漂移**的影响：由于时钟的频率不理想，受温度等外部因素的影响，不同机器上的时钟会逐渐显示出不同的时间值。这就是所谓的时钟漂移率：即每单位时间与参考时钟的差值。一个典型的基于石英的硬件时钟的时钟漂移率约为每秒1微秒，或大约每年31.5秒。计算机硬件时钟的漂移率要低得多。

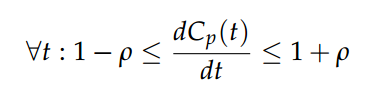
硬件时钟的规范包括其**最大时钟漂移率ρ**。如果F(t)表示硬件时钟在t时刻的实际振荡频率，F表示其理想(恒定)频率，则硬件时钟符合其规范。



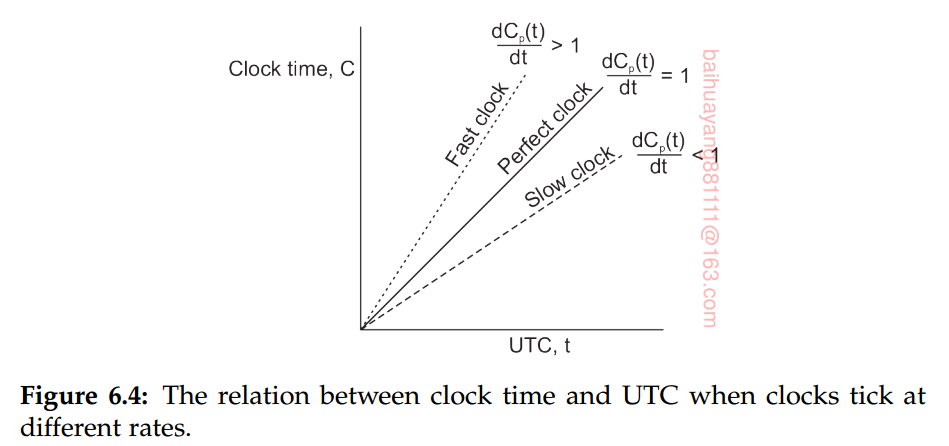
通过使用硬件中断，我们将软件时钟直接耦合到硬件时钟，从而也耦合到硬件时钟的时钟漂移率。（就是这种漂移率也会体现在软件时钟当中）。



特别是,我们带来了我们的终极目标,即保持软件时钟漂移率ρ也有限。



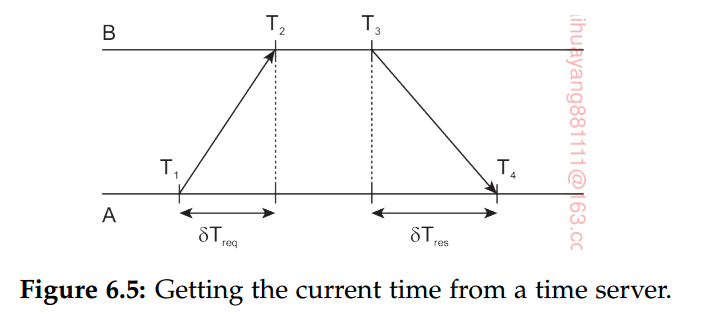
慢时钟、完美时钟和快时钟如图6.4所示



如果两个时钟漂移分别从UTC相反的方向进行,那一次∆t同步后,他们的差异可能会高达2ρ·∆t。如果系统设计师想要保证精密π,也就是说,没有两个相差超过π秒时钟,那么时钟必须至少每π/(2ρ)秒重新同步一次。（这里指软件时钟）不同的算法在如何实现这种重新同步上有很大的差异。

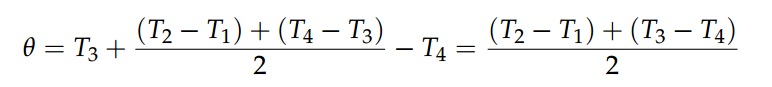
**网络时间协议**

Cristian[1989]最初提出的许多协议中的一种常见方法是让客户机与时间服务器联系。后者可以准确地提供当前的时间，例如，因为它配备了UTC接收器或准确的时钟。然而问题是在与服务器联系时，消息延迟将会报告过期。诀窍是为这些延迟找到一个好的估算方法，考虑图6.5中的情况：



在本例中，A将向B发送一个请求，时间戳为T1。反过来，B将记录接收T2的时间(从它自己的本地时钟中提取)，并返回一个带有值T3的响应时间，并携带以前记录的值T2。最后，A记录响应到达的时间T4。我们假设传播从A到B的传播延迟和从B到A的传播延迟是近似的，即*δTreq* = *T*2 *− T*1 *≈ T*4 *− T*3 = *δTres*。在这种情况下，A可以估计它相对于B的偏移量

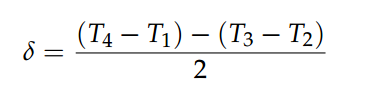
为：



当然，时间是不允许倒流的。如果的时钟快,θ< 0,这意味着,在原则上,向回设置它的时钟。然而，这是不允许的，因为它可能会导致严重的问题，比如在时钟更改之后编译的对象文件比在时钟更改之前修改的源文件早一段时间。

这种改变必须逐步实行。一种方法如下。假设计时器被设置为每秒生成100个中断。正常情况下，每个中断会增加10毫秒的时间。当需要减速时，中断例程每次只增加9 msec，直到纠正为止。类似地，时钟可以通过在每个中断处增加11毫秒而不是一次向前跳转来逐步前进。（就是对齐时钟通过改变时钟中断每次的时间调整，而不是直接调整时间）

在**网络时间协议(NTP)**的情况下，该协议是在服务器之间成对设置的。换句话说，B也会探测到A的当前时间。上面的计算给出了θ，估算δ的延迟为:



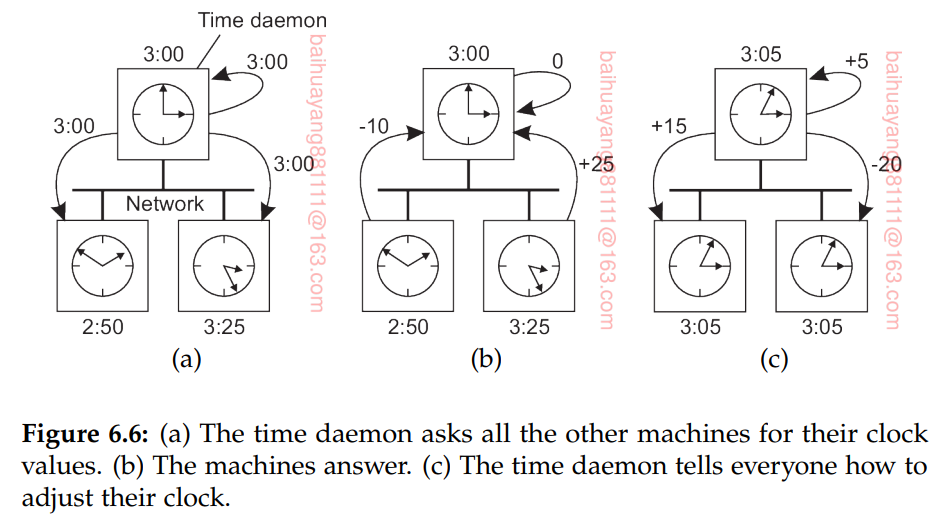
缓存8对(δ,θ)值，然后取出最好的。

对称地应用NTP，原则上也应该让B将其时钟调整到A的时钟。然而，如果已知B的时钟更准确，那么这样的调整将是愚蠢的。为了解决这个问题，NTP将服务器划分为层。具有参考时钟(如UTC接收器或原子钟)的服务器称为层1服务器(时钟本身据说在0层运行)。当A与B接触时，只有当A所处的地层水平高于B时，才会调整时间。而且，在同步之后，A的阶层水平会比B的高。换句话说，如果B是层k服务器，那么如果a的原始层级别已经大于k，那么a将成为层(k + 1)服务器。由于NTP的对称性，如果A的地层水平低于B, B就会调整到A。

关于NTP有许多重要的特性，其中许多与识别和屏蔽错误以及安全攻击有关。

**伯克利算法**

在许多时钟同步算法中，时间服务器是被动的。其他机器会定期询问时间。它所做的就是响应他们的查询。在伯克利的Unix中，采用了完全相反的方法。这里的时间服务器(实际上是一个时间守护进程)是活动的，它会不时轮询每台机器，询问它的时间。基于这些答案，它计算出一个平均时间，并告诉所有其他机器将时钟调快到新的时间，或者将时钟调慢，直到达到某个特定的时间。这种方法适用于没有UTC接收器的系统。时间守护进程的时间必须由操作符定期手动设置。该方法如图6.6所示。



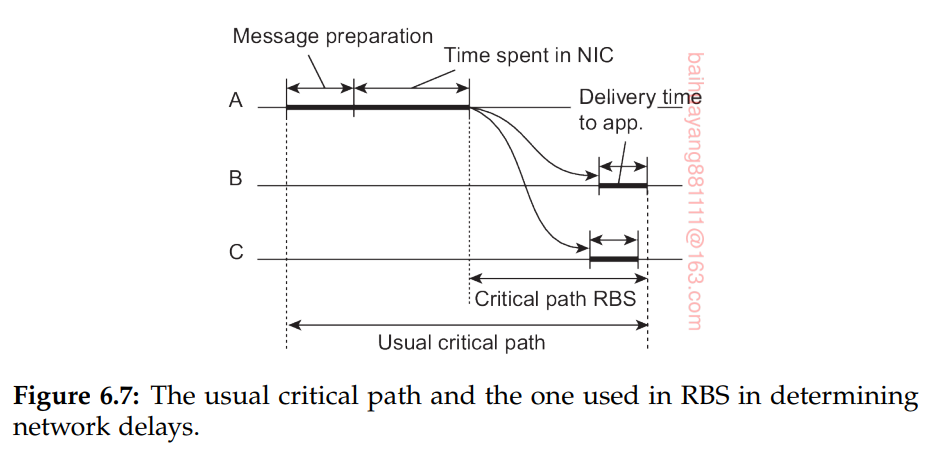
在图6.6(a)中，时间守护进程在3:00时告诉其他机器它的时间并询问它们的时间。在图6.6(b)中，它们响应时间守护进程的进度。有了这些数字，时间守护进程计算平均值，并告诉每台机器如何调整它的时钟(参见图6.6(c)))。

注意出于某种目的，让所有机器达到同一个时间是最有效的方式（而不是都达到最准确的时刻）。这个时间也不一定要与电台每小时宣布的实时时间一致。果在图6.6的示例中，时间守护进程的时钟永远不会被手动校准，那么只要其他节点没有与外部计算机通信，就不会造成任何危害。如果能让时间准确且一致当然是最好的，然而这并不现实。因此，Berkeley算法通常是一个内部时钟同步算法。

**无线网络中的时钟同步**

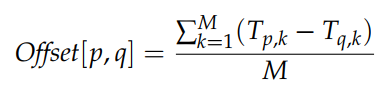
更传统的分布式系统的一个重要优势是，我们可以轻松高效地部署时间服务器。此外，大多数机器可以相互联系，从而实现相对简单的信息传播。这些假设在许多无线网络中不再有效，尤其是传感器网络。这种网络的节点资源受限，多跳路由开销较大。此外，优化算法的能源消耗往往是更重要的。这些和其他的一些观察导致了无线网络时钟同步算法的不同设计。在下面，我们考虑一个特定的解决方案。**Reference broadcast synchronization (RBS)**是一种时钟同步协议，它与其他提议有很大的不同[Elson et al.， 2002]。首先，协议不假定有一个节点能够准确地计算实际可用时间。它的目标不是提供所有节点的UTC时间，而是仅仅在内部同步时钟，就像Berkeley算法所做的那样。其次，我们目前讨论的解决方案旨在使发送方和接收方同步，本质上遵循双向协议。RBS偏离了这种模式，只允许接收方同步，而将发送方排除在循环之外。

在RBS中，发送方广播参考消息，使其接收方能够调整时钟。一个关键的观察是，在一个传感器网络中，如果没有多跳路由，向其他节点传播信号的时间大致是恒定的。在本例中，传播时间从消息离开发送者的网络接口开始度量。因此，消息传输中的两个重要变化源不再在估计延迟方面发挥作用：这两个方面是用于构造消息的时间，以及用于访问网络的时间。如图6.7



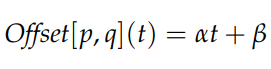
注意，在诸如NTP之类的协议中，在消息传递到网络接口之前，会将时间戳添加到消息中。此外，由于无线网络是基于争用协议的，所以通常不知道消息实际传输需要多长时间。这些非确定性已经在RBS中进行了估算。剩下的是接收端的交付时间，但是这个时间的变化要比网络访问时间小得多。

RBS的基本理念很简单：当一个节点广播一个引用消息m时，每个节点p只记录它接收到的时间Tp,m。注意，T p,m是从p的本地时钟读取的。忽略时钟偏移，两个节点p和q可以交换彼此的传输时间，以估计它们的相互、相对偏移量。



M是发送消息的总数量。这个信息很重要:节点p将知道q的时钟相对于它自己的时钟的值。此外，如果它只是简单地存储这些偏移量，就不需要调整自己的时钟，从而节省能源。

不幸的是，时钟可能会漂移。其结果是，像上面那样简单地计算平均偏移量是行不通的：最后发送的值比第一个值更不准确。此外，随着时间的推移，偏移量可能会增加。Elson等人[2002]使用一种非常简单的算法来补偿这一点：他们不是计算平均值，而是使用标准线性回归来计算偏移量作为函数:



常数a和b由(Tp,k, Tq,k)计算得到。这种新形式将允许通过节点p更精确地计算q的当前时钟值，反之亦然。

6.2逻辑时钟

时钟同步自然与时间有关，尽管可能没有必要准确地描述实时：分布式系统中的每个节点在当前时间上达成一致可能就足够了。我们可以更进一步。对于运行make，只要保证两个节点同意重新将input.c编译为input.o就足够了。在本例中，重要的是跟踪彼此的事件(例如生成input.c的新版本)。对于这些算法，通常将时钟称为逻辑时钟。

Lamport[1978]在一篇开创性的论文中指出，尽管时钟同步是可能的，但它不一定是绝对的。如果两个进程不交互，就没有必要对它们的时钟进行同步，因为缺乏同步是无法观察到的，因此不会造成问题。此外，他指出，通常重要的不是所有流程都同意确切的时间，而是它们对事件发生的顺序表示一致。在make示例中，重要的是input.c是比input.o更旧还是更新。不是他们各自的绝对创造时间。

**Lamport的逻辑时钟**

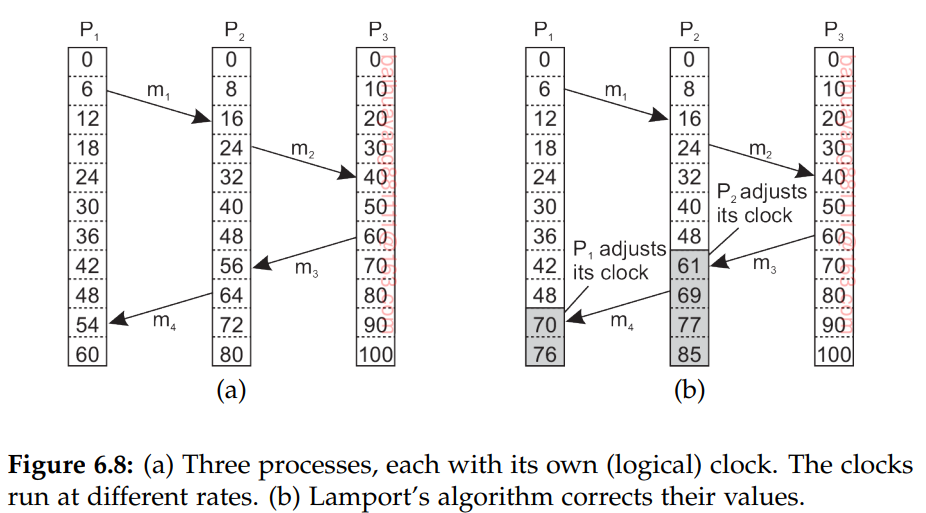
为了同步逻辑时钟，Lamport定义了一个名为happens before的关系。a !b读作“事件a发生在事件b之前”，意思是所有进程都同意首先发生事件a，然后发生事件b。在两种情况下可以直接观察到事件之间的关系：

1. 如果a和b是同一进程中的事件，且a发生在b之前，则a !b是正确的。
2. 如果a是一个进程发送消息的事件，b是另一个进程接收消息的事件，那么a !b也是正确的。不能在发送消息之前接收消息，甚至不能在发送消息的同时接收消息，因为到达消息需要有限的、非零的时间。

Happens-before是一个传递关系，所以如果a !b和b !c，然后a !c。如果两个事件，x和y，发生在不交换消息的不同进程中(甚至不通过第三方进行间接交换)，那么x !y不是真的，y!x也不是真的。这些事件被称为并发的，这仅仅意味着对于事件何时发生或哪个事件首先发生，没有什么可说的(或需要说的)。

我们需要的是一种测量时间概念的方法，这样对于每个事件a，我们都可以为它分配一个时间值C(a)，所有进程都同意这个时间值。这些时间值必须具有这样的属性:如果a !b，则C(a) < C(b)。换句话说，如果a和b是同一进程中的两个事件，并且a发生在b之前，那么C(a) < C(b)。类似地，如果a是一个进程发送的消息，而b是另一个进程接收该消息，那么C(a)和C(b)的分配必须使每个节点都同意C(a)和C(b)的值，并且C(a) < C(b)。此外，时钟时间C必须始终向前(增加)，而不是向后(减少)。对时间的修正可以通过增加一个正值来实现，而不是减去一个正值。

现在让我们看看Lamport提出的为事件分配时间的算法。考虑图6.8中描述的三个过程。这些进程运行在不同的机器上，每台机器都有自己的时钟。为了便于讨论，我们假设时钟被实现为一个软件计数器：计数器每T个时间单位增加一个特定的值。但是，每个进程的时钟增量值是不同的。进程P1中的时钟分别增加了6个单位、进程P2中的8个单位和进程P3中的10个单位。（下面，我们解释Lamport时钟实际上是事件计数器，这解释了为什么它们的值在进程之间可能不同）。

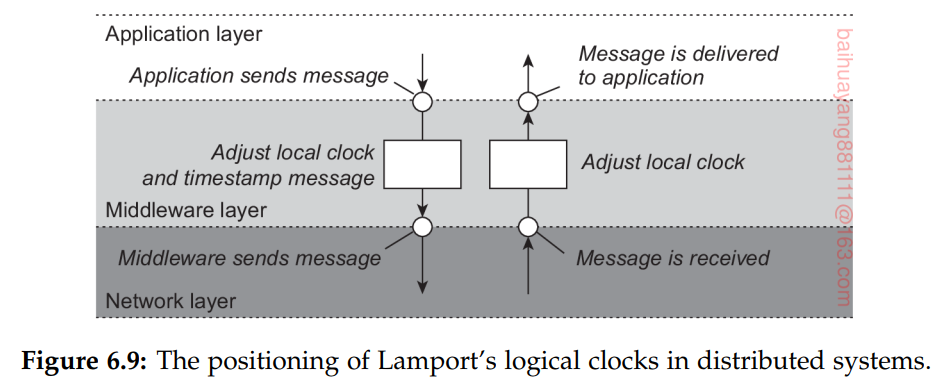


在第6时刻，进程P1向进程P2发送消息m1。这条信息需要多长时间到达取决于你相信谁的时钟。无论如何，进程P2中的时钟到达时读取16。如果消息中包含开始时间6，那么进程P2将得出这样的结论，即该过程花费了10滴答的时间。当然，这只是个可能的值。根据这个推理，消息m2从P2到P3需要16个节拍，这也是一个合理的值。

现在考虑消息m3。它在60离开P3，在56到达P2。类似地，从P2到P1的消息m4在64处离开，在54处到达。这些值显然是不可能的（不符合happen before，发送比接受时间要早）。必须防止这种情况的发生。

Lamport的解决方案直接遵循happens before关系。由于m3在60离开，它必须在61或更晚到达。因此，每条消息根据发送者的时钟携带发送时间。当消息到达并且接收方的时钟显示消息发送前的值时，接收方将其时钟快进到比发送时间多一个的值。在图6.8中，我们看到m3现在达到61。类似地，m4到达70。

让我们更精确地阐明这个过程。此时，区分三个不同的软件层是很重要的，正如我们在第1章中已经遇到的:网络层、中间件层和应用层，如图6.9所示。下面的内容通常是中间件层的一部分。



为了实现Lamport的逻辑时钟，每个进程Pi维护一个本地计数器Ci。这些计数器按照以下步骤更新[Raynal and Singhal, 1996]

1. 在执行事件(即，通过网络发送消息，向应用程序或其他一些内部事件发送消息)，Pi递增Ci: Ci=Ci + 1。

2. 当进程Pi向进程Pj发送一条消息m时，它在执行上一步之后将m的时间戳ts(m)设置为Ci。

3.接收到消息m后，进程Pj将自己的本地计数器调整为Cj=max（Cj, ts(m)），然后执行第一步并将消息传递给应用程序

在某些情况下，需要额外的要求:没有两个事件完全同时发生。为了实现这个目标，我们还使用惟一的进程标识符来断开连接，并使用元组而不是仅使用计数器的值。如果有个进程Pi在时间40，则标记为<40,i>。如果我们有一个<40,j>的事件，并且i<j，则<40,i> < <40,j>。（这里是用元组表示，当相同时间发生两个相同事件时加以区分先后）

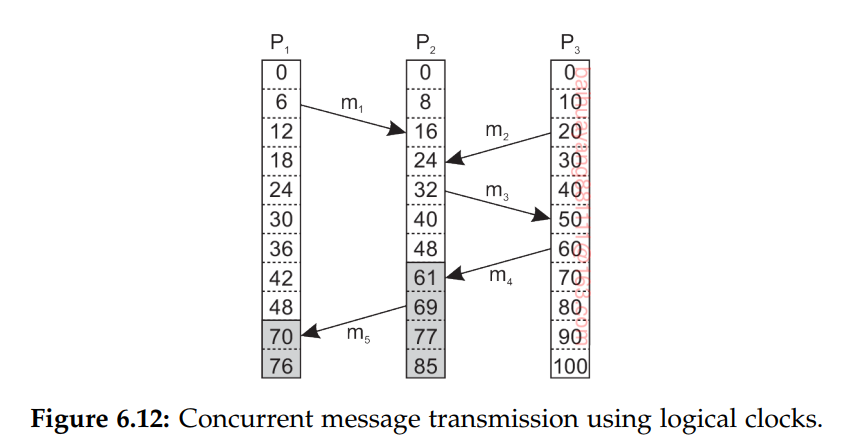
注意，通过分配事件时间C(a) Ci(a)，如果a发生在进程Pi (a)的时间Ci(a)，我们就得到了我们最初寻求的全局时间值的分布式实现;我们就这样构造了一个逻辑时钟。

**向量时钟**

Lamport的逻辑时钟导致了这样一种情况，分布式系统中的所有事件都是完全有序的，如果事件a发生在事件b之前，那么a也将按照事件b之前的顺序排列，即C(a) < C(b)。

但是，对于Lamport时钟，仅仅比较两个事件a和b的时间值C(a)和C(b)，并不能说明它们之间的关系。换句话说，如果C(a) < C(b)，那么这并不一定意味着a确实发生在b之前。

要解释这一点，请考虑图6.12中所示的三个流程所发送的消息。用Tsnd(mi)表示发送消息mi的逻辑时间，同样，用Trcv(mi)表示接收消息的时间。通过构造，我们知道对于每个消息Tsnd(mi) < Trcv(mi)。但是我们能够通过Trcv(mi) < Tsnd(mj)得到关于 mi和mj的什么结论呢？



在mi = m1和mj = m3的情况下，我们知道这些值对应于进程P2发生的事件，这意味着m3确实是在收到消息m1之后发送的。这可能表明消息m3的发送取决于通过消息m1接收到的内容。同时，我们还知道Trcv(m1) < Tsnd(m2)。但是，从图6.12中我们可以看出，发送m2与接收m1无关。

这里的问题是Lamport时钟不能捕捉**因果关系**。在实践中，因果关系是通过**矢量时钟**来捕捉的。为了更好地理解这些来自何处，我们遵循Baquero和Preguica[2016]给出的解释。事实上，如果我们为每个事件分配一个惟一的名称，例如进程ID和本地递增计数器的组合，那么跟踪因果关系就很简单：pk是进程P发生的第k个事件。然后，问题归结为记录**因果历史**。例如，如果两个局部事件发生在进程P，则事件p2的因果历史H(p2)为{p1,p2}

现在假设进程P向进程Q发送一条消息,则Q的最近的因果历史是{p}。为了跟踪因果关系，P还发送它最近的因果历史（假设是{p1,p2}并扩展到p3）。到达时，Q记录事件(q2)，并将两个因果历史合并成一个新的：{p1,p2,p3,q1,q2}。

判断一个时间p是否是q的前因，可以通过检查H(p)是否是H(q)的子集的方式（如果是前因，就是子集）。通过我们的符号，在假定q是H(q)的最后一个事件时，我们可以判断p是否属于H（q）。因果历史的问题在于，它们的表述不是很有效。但是，没有必要跟踪来自同一进程的所有后续事件：只关注最后一个就行了。如果我们随后为每个流程分配一个索引，我们可以将一个因果历史表示为一个向量，其中第j个条目表示在流程Pj中发生的事件的数量。然后，可以通过向量时钟捕捉因果关系，通过让每个进程Pi维护一个具有以下两个属性的向量VCi来构建向量时钟：

1. VCi[i]是到目前为止在Pi处发生的事件数。换句话说，VCi[i]是进程Pi的本地逻辑时钟
2. 如果VCi[j] = k，则Pi知道在Pj发生了k个事件。因此，Pi知道Pj的本地时间。

第一个属性是通过在流程Pi处发生的每个新事件发生时递增VCi[i]来维护的。第二个属性由承载向量和发送的消息来维护。具体来说，执行以下步骤：

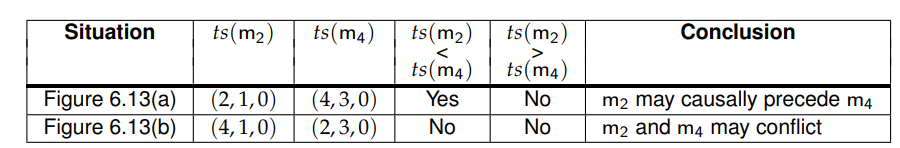
1. 在执行事件的时候，VCi[i] <- VCi[i] +1。这相当于记录一个发生在Pi处的新实践。
2. 当进程Pi向Pj发送一条消息m时，它在执行上一步之后将m的(向量)时间戳ts(m)设置为VCi
3. 接收到消息m后，再执行第一步之后，进程pj通过为每个k设置 VCj[k] <- max{VCj[k], ts(m)[k]}，然后将消息发送到应用层。

注意，如果事件a具有时间戳ts(a)，则ts(a)[i] - 1表示在该事件在事件a之前，Pi处处理的事件数。因此，当Pj从Pi接收带有时间戳ts(m)的消息m时，它知道在发送m之前在Pi处发生的事件的数量。然而，更重要的是，在Pi发送消息m之前，Pj还被告知Pi知道在其他进程中发生了多少事件。换句话说，时间戳ts(m)告诉接收方在发送m之前其他进程中有多少事件，以及m可能依赖于哪些事件。

要了解这意味着什么，请考虑图6.13，其中显示了三个进程。

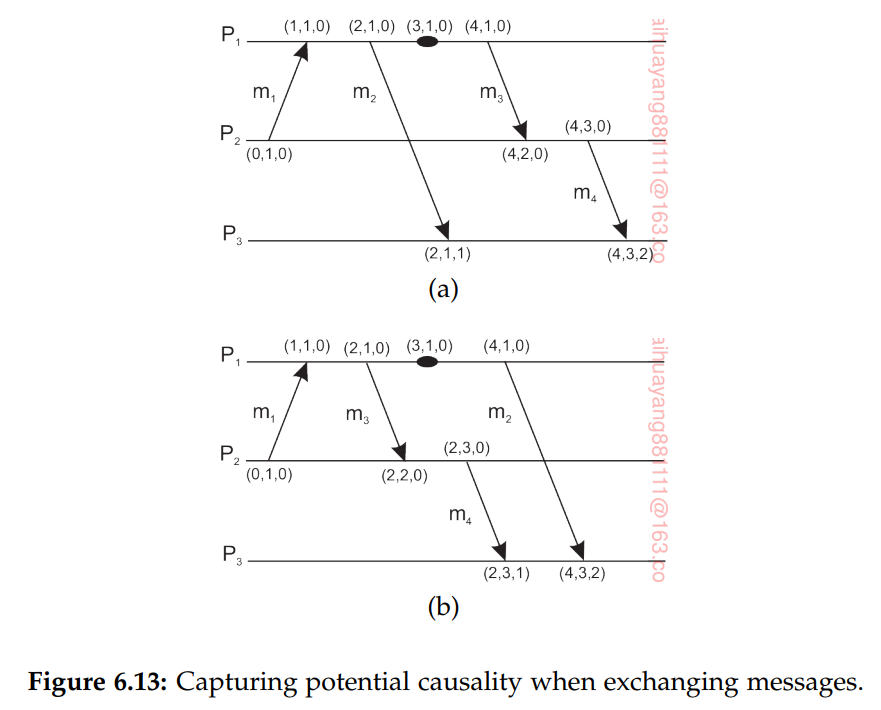
要了解这意味着什么，请考虑图6.13，其中显示了三个流程。在图6.13(a)中，P2在逻辑时间VC2 =(0,1,0)发送一条消息m1来处理P1。因此消息m1接收时间戳ts(m1) = (0,1,0)， P1收到消息后将其逻辑时间调整为VC1(1,1,0)并将其发送。发送的消息m2 P1与时间戳P3 ts (m2) =(2 1 0)。在P1发送另一个消息之前,m3,事件发生在P1,最终导致时间戳m3(4 1 0)值。在收到m3,过程P2发送消息m4 P3,时间戳ts (m4) = (4 3 0)。

现在考虑图6.13(b)所示的情况。在这里，我们将发送消息m2延迟到消息m3发送之后，以及事件发生之后。不难看出，ts(m2) = (4,1,0)， ts(m4) =(2,3,0)，与图6.13(a)相比，情况如下：



我们使用 ts(a) < ts(b) 当且仅当 对于所有k ts(a)[k] <= ts(b)[k]。因此，通过使用矢量时钟，P3进程可以检测m4是否与m2有因果关系，或者是否存在潜在的冲突。顺便提一下，如果不知道消息中包含的实际信息，就不可能肯定地说确实存在因果关系，或者可能存在冲突。

（上面这段没懂）（又看了一遍，大概意思是，如果要判断两个事件发生的前后关系，则必须保证向量上的每个点的值都大于等于或者小于等于另一个向量。如a (2,1,0)都小于(4,3,0)而b。 4大于2 而 1小于3 则无法判断m2和m4的先后关系，导致冲突。 由于之前可以通过判断每个事件的历史前置事件来判断两个事件的先后， 现在只记录各个事件发生的总数，至于why，还需要看一下）



6.3互斥

分布式系统的基础是多个进程之间的并发性和协作。在许多情况下，这也意味着进程需要同时访问相同的资源。为了防止这种并发访问破坏资源，或者使资源不一致，需要解决方案来授予进程互斥访问权。在本节中，我们将介绍一些重要的和有代表性的分布式算法。

**概述**

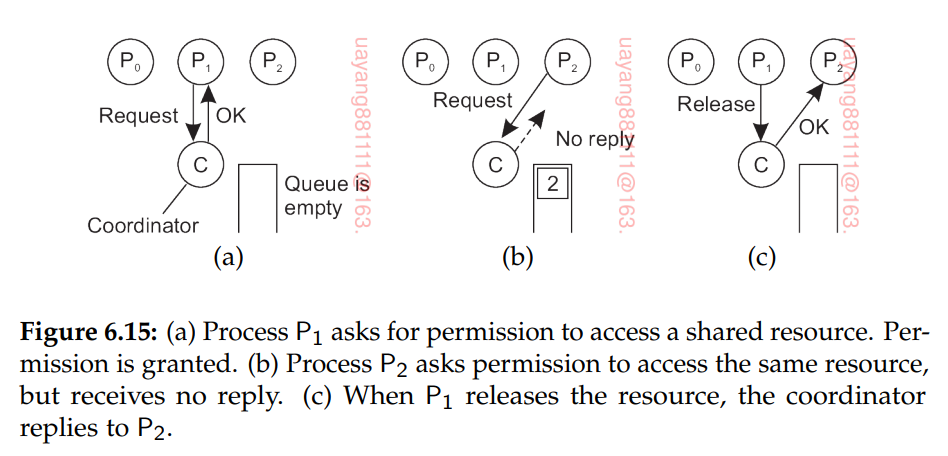
分布式互斥算法可以分为两大类。在基于**令牌**的解决方案中，互斥是通过在进程之间传递称为**令牌**的特殊消息来实现的。即只有一个令牌可用，并且允许任何拥有该令牌的人访问共享资源。完成后，令牌将传递给下一个进程。如果具有令牌的进程对访问资源不感兴趣，则将其传递。

基于令牌的解决方案有几个重要的特性。首先，根据进程的组织方式，可以相当容易地确保每个进程都有机会访问资源。也即他会防止**饥饿进程**的出现。其次，可以很容易地避免几个进程无限期地等待彼此继续的死锁，这有助于简化它们。基于令牌的解决方案的主要缺点是相当严重的：当令牌丢失时(例如，由于持有它的进程崩溃)，需要启动一个复杂的分布式过程，以确保创建了一个新令牌，并且也是惟一的令牌。

作为一种替代方案，许多分布式互斥算法采用基于许可(permission)的方法。在这种情况下，希望首先访问资源的进程需要获得其他进程的许可。有许多不同的方式来授予这样的许可，在接下来的部分中，我们将考虑其中的一些。

**一个中心化算法**

在分布式系统中实现互斥的一个简单方法是模拟它在单处理器系统中是如何完成的。一个进程被选为协调员。每当进程想访问共享资源时，它就向协调器发送一条请求消息，说明它想访问哪个资源并请求许可。如果当前没有其他进程访问该资源，协调器将返回一个授予权限的应答，如图6.15(a)所示。当响应到达时，请求者可以继续。



现在假设另一个进程，图6.15(b)中的P2请求访问资源的权限。协调器知道已经有一个不同的进程在访问该资源，因此不能授予权限。拒绝权限的确切方法依赖于系统。在图6.15(b)中，协调器只是避免响应，从而阻塞了等待响应的进程P2。或者，它可以发送一个回复说“权限被拒绝”。无论哪种方式，它都会暂时对来自P2的请求进行排队，并等待更多的消息。

当进程P1处理完资源后，它向协调器发送一条消息，释放它的独占访问，如图6.15(c)所示。协调器从延迟请求队列中取出第一项，并向该进程发送一条grant消息。如果进程还在阻塞状态，则它会解开阻塞并让其访问。如果已经发送了拒绝权限的显式消息，则进程必须轮询传入的流量或稍后阻塞（就是排队吧）。无论哪种方式，当它看到grant时，它也可以继续。

显而易见，该算法保证了互斥性：协调器一次只允许一个进程访问资源。这也是公平的，因为请求是按照收到请求的顺序被批准的。没有进程永远等待(没有饥饿)。该方案也易于实现，并且每次使用资源(请求、授予、发布)只需要三条消息。它的简单性使得它对于许多情况都是一个有吸引力的解决方案。

集中式方法也有缺点。协调器是一个单点故障，因此如果它崩溃，整个系统可能会崩溃。如果一个进程在发出请求后阻塞，它就不能分辨是否是这个已经故障的协调器发出的“拒绝权限”。此外，在大型系统中，单个协调器可能成为性能瓶颈。尽管如此，在许多情况下，其简单性带来的好处超过了潜在的缺点。此外，正如我们在6.3节中所说明的，分布式解决方案不一定更好。

**一个分布式算法**

使用Lamport的逻辑时钟，并受Lamport最初的分布式互斥解决方案(我们在注释6.3中讨论过)的启发，Ricart和Agrawala[1981]提供了以下算法。他们的解决方案需要系统中所有事件的总顺序。也就是说，对于任何一对事件(比如消息)，必须明确到底是哪个事件首先发生。

算法如下。当一个进程想要访问一个共享资源时，它构建一个包含资源名称、进程号和当前(逻辑)时间的消息。然后它将消息发送给所有其他进程，也包括它自己。假定发送的消息是可靠的;也就是说，没有信息丢失。

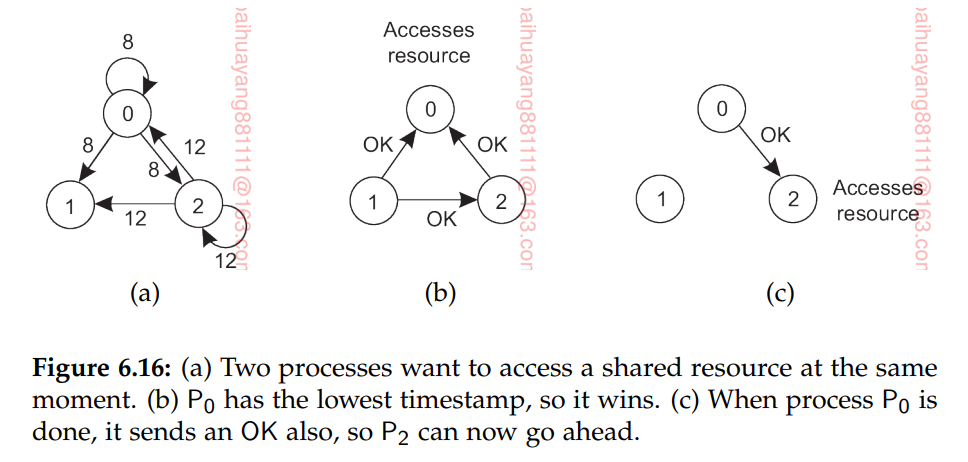
当一个进程从另一个进程接收到请求消息时，它所采取的操作取决于它自己对消息中命名的资源的状态。必须明确区分三种不同的情况:

1.如果接收方没有访问资源，也不想访问它，它将向发送方发送OK消息。

2.如果接收方已经访问了该资源，那么它将不响应。相反，它对请求进行排队

3.如果接收方也想访问资源，但还没有访问资源，那么它会将传入消息的时间戳与已发送给所有人的消息中包含的时间戳进行比较。最早的时间戳将获胜。如果传入消息的时间戳较低，则接收方发回OK消息。如果它自己的消息的时间戳较低，则接收方对传入的请求进行排队，不发送任何消息。

在发出请求请求权限之后，流程将停止工作，等待其他所有人都给出了权限。当得到了其他所有进程的允许，就可以继续访问资源了。当它完成时，它向队列中的所有进程发送OK消息，并从队列中删除它们。如果没有冲突，它显然是有效的。但是，假设两个进程试图同时访问资源，如图6.16(a)所示



P0在8时刻像所有人发送请求，同时，P2在12时刻发送请求。P1则对资源不感兴趣。P0和p2对资源的访问发生了冲突，并且比较了各自的时间戳。P2发现他输了（时间戳靠后），所以他将把权限让给P0并发送OK。进程P0现在将来自P2的请求排队等待后续处理，然后访问资源，如图6.16(b)所示。当它完成时，它从队列中删除P2中的请求，并向P2发送OK消息，允许后者继续执行，如图6.16(c)所示。该算法之所以有效，是因为在冲突的情况下，最低的时间戳获胜，并且每个人都同意时间戳的顺序。

该算法保证了互斥不存在死锁或饥饿。如果流程的总数是N，那么进程需要发送和接收的数量就是2（N-1）：N-1个发送消息，N-1个接收消息。

不幸的是，这个算法会产生N个故障点。如果任何进程崩溃，它将无法响应请求。这种故障将被截石位拒绝许可，之后将会阻塞进程进入各自的许可区域。该算法可以修补如下

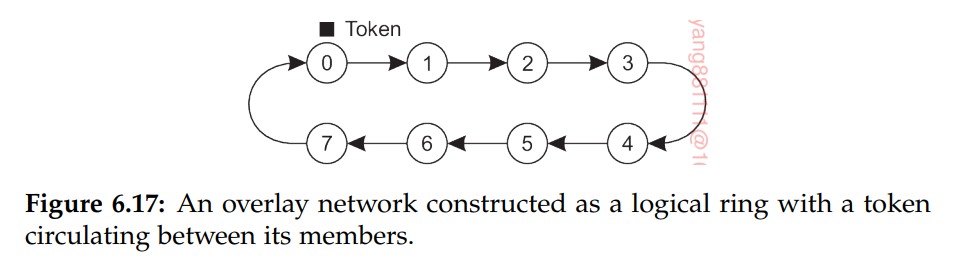
，当请求传入时，接收方总是发送一个应答，要么授予许可，要么拒绝许可。无论何时丢失请求或应答，发送方都会超时并继续尝试，直到应答返回或发送方得出目的地已死的结论。在请求被拒绝之后，发送方应该阻止等待后续的OK消息。

算法的另一个问题是，要么必须使用组播通信原语，要么每个进程必须维护组成员列表本身，包括进入组、离开组和崩溃的进程。该方法最适合于从不更改其组成员关系的小组进程。最后，请注意，所有进程都参与了有关访问共享资源的所有决策，这可能会给运行在资源受限机器上的进程带来负担。

该算法可以进行各种小的改进。例如，获得每个人的许可是多余的。所需要的只是一个方法来防止两个进程同时访问资源。该算法可以修改为当它从大多数其他进程(而不是所有进程)收集到权限时授予权限。

**环形令牌算法**

图6.17展示了在分布式系统中完全不同的实现互斥的方法。在软件中，我们以逻辑环的形式构造一个覆盖网络，其中每个进程在环中分配一个位置。重要的是，每个进程都知道自己之后的下一个是谁。



初始化环时，进程P0将被赋予一个**令牌**。这个令牌在圆环上流通。假设有N个进程，令牌在点对点消息中从进程Pk传递到进程P(k+1) mod N中。当进程从其邻居处获得令牌时，它将检查是否需要访问共享资源。如果是这样，进程将继续进行，完成所需的所有工作，并释放资源。当它完成后，它沿着圆环传递令牌。不允许使用相同的令牌立即再次进入资源。

如果一个进程由它的邻居传递令牌，并且对资源不感兴趣，那么它只传递令牌。因此，当没有进程需要资源时，令牌只是在环上循环。

该算法的正确性是显而易见的。在任何时刻，只有一个进程拥有令牌，因此只有一个进程能够真正访问资源。由于令牌在进程之间以定义良好的顺序循环，因此不会发生饥饿。一旦一个进程决定它想要访问该资源，最坏情况下它将不得不等待其他进程使用该资源。

（所以说是一个令牌环对应一个资源了？？？多个资源就有多个环？？？）

这个算法有它自己的问题。例如，如果令牌曾经丢失，因为它的持有者崩溃，或者由于包含令牌的消息丢失，则必须重新生成令牌。事实上，检测它是否丢失可能很困难，因为在网络上令牌连续出现之间的时间是无界的（就是无法用一个时间来界定多久没收到令牌，就表明了该令牌丢失）。令牌一小时未被发现并不意味着它已经丢失;有人可能还在使用它。

如果进程崩溃（这里应该是非持有令牌者），该算法也会遇到麻烦，但是恢复相对容易。如果我们需要一个接收令牌的进程来确认接收，当它的邻居试图给它发送令牌并失败时，将检测到一个死进程。此时，死进程可以从组中删除，令牌持有者可以将该令牌从死进程的头部抛向行下一个成员，如果需要，还可以抛向死进程之后的成员。当然，这样做需要每个人都维护当前的环配置。

**去中心化算法**

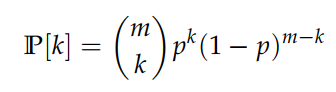
让我们看看完全分散的解决方案。Lin等[2004]提出使用投票算法。假设每个资源被复制N次。每个副本都有自己的协调器来控制并发进程的访问。（为什么资源要被复制N份？？）

然而，每当一个进程想要访问资源时，它只需要获得m > N/2协调员的多数票即可。我们假设，当协调器不授予访问资源的权限时(当它向另一个进程授予权限时，它将这样做)，它将告诉请求者。

假设当一个协调器崩溃时，它会快速恢复，但会忘记崩溃前进行的任何投票。一种方法是协调器在任意时刻重置自己。这么做面临的风险是，重置将使协调器忘记它之前已授予某些进程访问资源的权限。因此，在恢复之后，它可能错误地再次将此权限授予另一个进程。

（就是崩溃之后给同一个进程投了两次票）

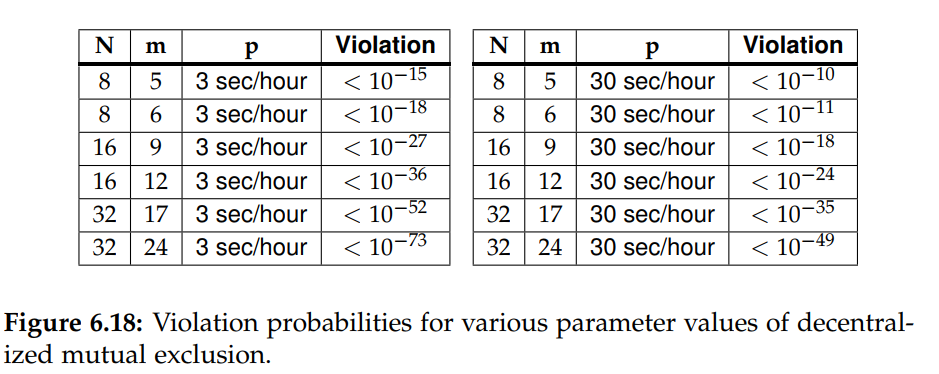
设p = Dt/T为协调器在时间间隔Dt内重置的概率，而生存周期为T。所以概率P[K]表示m中的k个协调器将在一个时间间隔内重置的概率，这个结果是



如果f个协调器重置，那么当只有少数非故障坐标时，即m−f≤N/2时，或者f≥m−N/2时，投票机制的正确性就会受到破坏。发生这种情况的概率是



图6.18显示了不同的N、m和p值违反正确性的概率，可以对这一概念有直观的印象。注意，我们通过考虑协调器重置的每小时秒数来计算p，并将此值作为访问资源所需的平均时间。p的值被认为是(非常)保守的。结论是，一般来说，与其他类型的失败相比，违反正确性的概率非常低，可以忽略不计。



为了实现这个方案，我们可以使用一个资源被复制N次的系统。假设资源叫做rename。然后，我们可以假设第i个副本名为renamei，然后使用已知的散列函数计算惟一的键。因此，每个进程都可以生成给定资源名称的N个键，然后使用一些常用的命名系统查找负责副本(以及控制对副本的访问)的每个节点。如果申请访问资源的权限被拒绝，那它将隔一段时间再来申请。这个方案的问题是，如果许多节点想访问相同的资源，那么利用率会迅速下降。在这种情况下，有如此多的节点竞争访问权限，最终没有人能够获得足够的选票而不使用资源。这个问题的解决方案可以在[Lin et al.， 2004]中找到。

**6.4选举算法**

许多分布式算法需要一个进程充当协调器、启动器或其他特殊角色。一般来说，哪个进程承担这一特殊责任并不重要，但必须由其中一个进程来承担。在本节中，我们将研究用于选择协调器的算法(将其用作特殊进程的通用名称)。

如果所有的进程都完全相同，没有区别的特征，就没有办法选择其中一个是特殊的。因此，我们假设每个进程P都有一个惟一的标识符id(P)。一般来说，选择算法试图定位具有最高标识符的进程，并将其指定为协调器。这些算法在定位协调器的方式上有所不同。此外，我们还假设每个进程都知道其他进程的标识符。换句话说，每个流程都完全了解流程组，其中必须选出一个协调员。进程不知道哪些进程当前处于上升状态，哪些进程当前处于下降状态。选举算法的目标是确保当选举开始时，所有进程都同意新的协调员是谁，从而结束选举。有许多算法和变体，其中几个重要的算法在Tel[2000]和Lynch[1996]的教科书中进行了讨论。

**The bully algorithm**

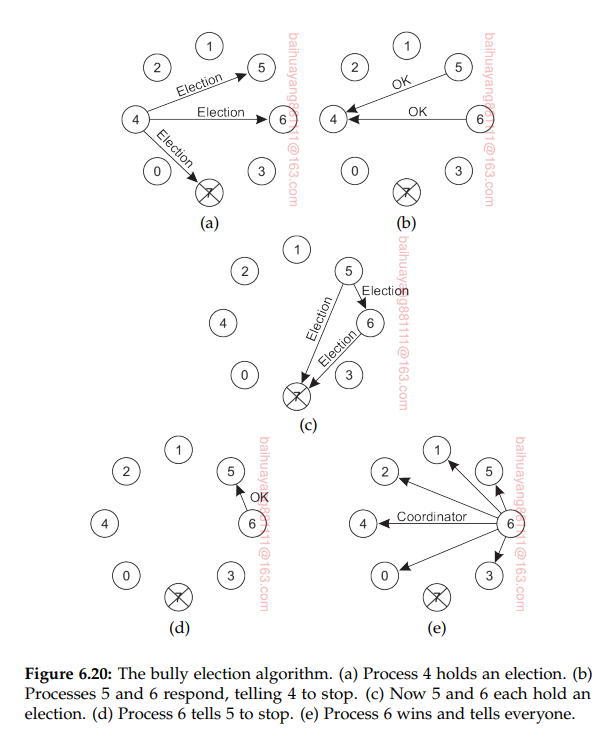
选择协调器的一个著名解决方案是Garcia-Molina[1982]设计的bully算法。在下面，我们考虑N个进程{P0，…， PN – 1}，令id(Pk) = k。当任何进程注意到协调员不再响应请求时，它将发起一次选举。一个进程Pk进行如下选举:

1. Pk向所有标识符更高的进程发送一个选举消息:Pk+1, Pk+2，…,PN−1。
2. 如果没有人回应，Pk将赢得选举并成为协调员。
3. 如果其中一个上级回答了，它就会接管Pk的工作

任何时候，一个进程都可以从其更低标识符的进程那里得到选举信息。当这样的消息到达时，接收方向发送方发送OK消息，表明他还活着，并将接管。然后接管人举行选举，除非它已经在举行选举。最终，除了一个进程之外，所有进程都放弃了，这个进程就是新的协调器。它通过向所有进程发送一条消息，告诉它们立即启动它是新的协调器，从而宣告了它的胜利。

如果之前被否决的程序重新启动，它将举行选举。如果它恰好是目前正在运行的编号最高的进程，它将赢得选举并接管协调员的工作。因此，城里最大的家伙总是赢家，因此得名“霸凌算法”。

在图6.20中，我们看到了一个欺凌算法如何工作的例子。该组由8个进程组成，标识符编号从0到7。以前进程P7是协调器，但是它已经崩溃了。进程P4是第一个注意到这一点的进程，因此它向所有高于它的进程(P5、P6和P7)发送选举消息，如图6.20(a)所示。进程P5和P6都响应OK，如图6.20(b)所示。在收到第一个响应后，P4知道它的工作已经结束，知道P5或P6中的任何一个将接管并成为协调器。进程P4只是坐下来，等待看谁将成为赢家(尽管此时它可以做出很好的猜测)。

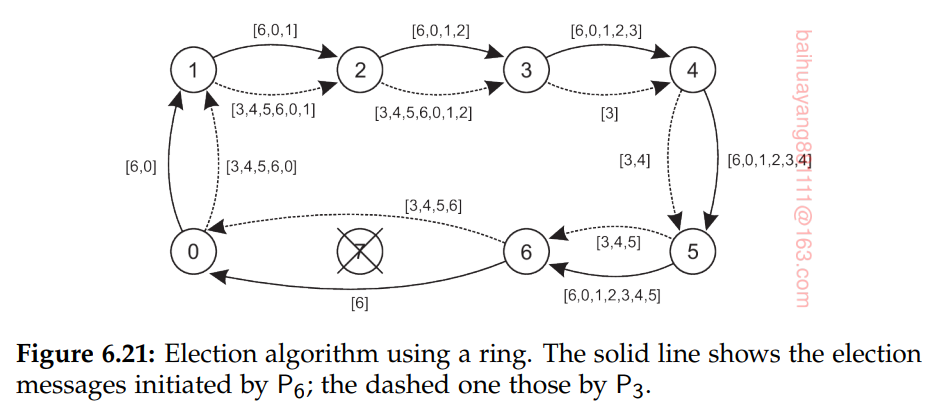


在图6.20(c)中，P5和P6都举行了选举，每个进程只向标识符高于自身的进程发送消息。在图6.20(d)中，P6告诉P5它将接管。此时，P6知道P7已经挂了，所以它(P6)是赢家。如果要从磁盘或其他地方收集状态信息，以便从旧协调器停止的地方继续，则p6现在就需要做这些事情。当它准备接管时，它通过向所有正在运行的进程发送协调器消息来宣布接管。当P4收到此消息时，它现在可以继续执行在发现P7已挂时尝试执行的操作，但这次使用P6作为协调器。这样就可以处理P7的故障并继续工作。如果进程P7重新启动，它将向所有其他进程发送协调器消息并强制它们提交。

**环形算法**

考虑以下基于使用(逻辑)环的选择算法。与一些环形算法不同，这个算法不使用令牌。我们假定每个进程都知道谁是它的继承者。当任何进程注意到协调器不起作用时，它构建一个包含自己的进程标识符的选举消息，并将该消息发送给它的继任者。如果后继进程关闭，发送方将跳过后继进程，并沿着环传递到下一个成员，或之后的成员，直到找到正在运行的进程为止。在此过程的每个步骤中，发送方都将自己的标识符添加到消息中的列表中，从而有效地使自己成为被选为协调器的候选人。

最终，消息返回到启动它的进程。当流程接收到包含其自身标识符的传入消息时，将识别此事件。时，消息类型将更改为COORDINATOR并再次分发，这一次通知所有其他人协调器是谁(具有最高标识符的列表成员)以及新环的成员是谁。当这条消息传播一次后，它将被删除，所有人将重新开始工作。



在图6.21中，我们看到如果P3和P6两个进程同时发现前面的协调器进程P7崩溃了，会发生什么。每一个都建立了一个选举信息，并开始传播它的信息，独立于另一个。最终，这两条消息都会一直传递下去，P3和P6都会将它们转换为协调器消息，具有完全相同的成员和相同的顺序。当它们都再次出现时，它们都将被移除。传播额外的信息没有害处;在最坏的情况下，它会消耗一些带宽，但这并不被认为是浪费。

**无线环境中的选举**

基于某些假设的传统的选举算法，在无线算法中是不现实的。例如，他们假设消息传递是可靠的，并且网络拓扑结构不会改变。这些假设在大多数无线环境中都是错误的，特别是在移动自组网络环境中（就是移动网络）。

目前只制定了少数适用于特设网络的选举协议。Vasudevan等人[2004]提出了一种能够处理故障节点和分区网络的解决方案。他们的解决方案的一个重要特性是，最好的协调节点是可以选举出来的，而不是像前面讨论的解决方案中或多或少是随机选出来的。这种协议是如下这么工作的。为了简化我们的讨论，我们只关注于ad hoc网络，而忽略了节点可以移动。

考虑特设无线网络。为了选举领导人，网络中的任何节点(称为源节点)都可以通过发送选举消息给它的近邻(即，其范围内的节点)。当节点第一次接收到一个选举时，它将发送方指定为其父节点，然后向除父节点外的所有近邻发送一个选举消息。当一个节点从父节点以外的节点接收到选举消息时，它会发出响应它收到了。

当节点R指定节点Q为其父节点时，它将选举消息转发给它的近邻(不包括Q)，并等待确认信息到达，然后再确认来自Q的选举消息。这种等待有一个重要的后果。首先，请注意，已经选择父节点的邻居将立即响应R。更具体地说，如果所有邻居都已经有一个父节点，那么R就是一个叶子节点，并且能够快速地向Q报告。在此过程中，它还将报告诸如电池寿命和其他资源容量等信息。（个人理解，R为Q的父节点，R发消息给Q，Q会将其转发给其余邻居，若1、这些邻居会将Q作为父节点，则继续传递下去，否则2、Q为叶子节点，即不是任何节点父节点，将迅速返回消息给R，像一种递归过程）。

这一信息将允许Q稍后将R的容量与其他下游节点的容量进行比较，并选择最合适的节点进行领导。（个人理解，容量是指子节点的个数，然而事实似乎是一个人为设定的值。。。）。当然，Q之所以发出选举信息，只是因为它自己的母公司P也这么做了。反过来，当Q最终确认P之前发送的选举消息时，它也会将最合适的节点传递给P。通过这种方式，源最终将知道选择哪个节点为leader最好，然后它将该信息广播给所有其他节点。

这个过程如图6.22所示。每个节点都有一个容量。网络广播从节点b和j发起，如图6.22(b)，到6.22(e)时，选举的消息传播到了所有节点，这里省略了f和i的传播。然后每个节点都向其父节点报告容量最大的节点，如图6.22(f)。例如，当节点g收到其子节点e和h的确认时，它会注意到h是最好的节点，将[h, 8]传播到它自己的父节点b。最后，源节点将注意到h是最好的leader，并将此信息广播给所有其他节点。

当发起多个选举时，每个节点将决定只加入一个选举。为此，每个源都用唯一的标识符标记其选择消息。节点将只参与标识符最高的选举，停止在其他选举中运行的任何参与。通过一些小的调整，这个协议也可适用于出现网络分区以及网络中新增或减少节点的情况。

**大规模系统的选举**

许多leader选举算法只适用于相对较小的分布式系统。此外，算法通常只关注单个节点的选择。在某些情况下，实际上应该选择多个节点，例如在peer-to-peer网络中的**super peer**点，我们在第2.3节中讨论过。在本节中，我们将特别关注选择**super peer点**的问题

**super peer**点选择需要满足以下要求(参见[Lo et al.， 2005]):

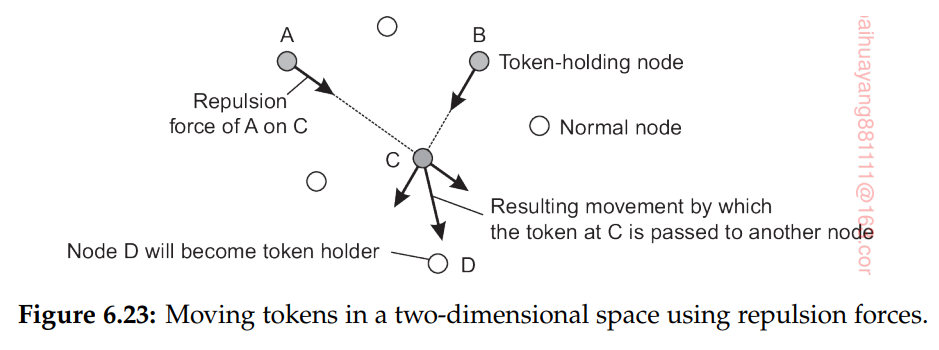
1. 普通节点应该具有对**super peer**节点的低延迟访问。、
2. **super peer**应该均匀地分布在覆盖网络中。
3. 相对于覆盖网络中的节点总数，应该有一个预定义的**super peer**点部分。
4. 每个超级对等点的服务不应该超过固定数量的普通节点

在基于dhb的系统中，基本思想是为**super peer**点保留一小部分标识符空间。在基于dhg的系统中，每个节点接收一个随机的、统一分配的m位标识符。现在我们将前k个bit进行翻转来表示**super peer**点。例如，如果我们需要N个**super peer**点，那么前 log2(N)[向上取整] bit可以用来表示这些**super peer点。**

为了解释，我们假定了一个Chord system 系统，设置m=8，k=3. 当查找负责特定密钥K的节点时，我们可以首先决定将查找请求路由到负责模式K^11100000的节点，然后将该节点视为**super peer**。注意，每个具有标识符ID的节点都可以通过查找ID ^ 11100000来检查它是否是超级对等节点，以查看这个请求是否路由到它自己。如果将节点标识符统一分配给节点，可以看到，在总共有N个节点时，超级对等点的数量平均为2^(k – m) \*N。

另一种完全不同的方法是基于多维几何空间中的定位节点。在本例中，假设我们需要在覆盖层中均匀地放置N个**super peer**点。基本思想很简单:总共有N个令牌分布在随机选择的N个节点上。任何节点都不能包含一个以上的令牌。两个令牌会相互排斥。最终的结果是，如果所有的符号都施加相同的斥力，它们就会相互远离，并在几何空间中均匀地分布。

这种方法要求持有令牌的节点了解其他令牌。为此，我们可以使用一种gossiping协议，通过该协议令牌可以在整个网络中传播。如果一个节点发现作用于它的总力超过了一个阈值，它将使令牌朝着合力的方向移动，如图6.23所示。当节点持有令牌一段时间后，该节点将把自己提升到**super peer**节点。



6.5位置系统（location system）

当查看分布在广域网上的非常大的分布式系统时，通常需要考虑**渐进性**。假设一个分布式系统组织为一个覆盖网络，其中两个进程是覆盖网络中的邻居，但实际上它们在底层网络中相隔很远。如果这两个进程进行了大量的通信，那么最好确保它们在物理上彼此靠近。在本节中，我们将研究基于位置的技术，以协调进程的放置及其通信

**全球定位系统**

让我们考虑如何确定你在地球上任何地方的地理位置开始。这个定位问题本身是通过一个高度具体、专用的分布式系统，即GPS来解决的，GPS是全球定位系统的缩写。GPS是1978年推出的一种基于卫星的分布式系统。虽然它最初主要用于军事应用，但到目前为止，它已被用于许多民用应用，特别是交通导航。然而，这存在更多的应用程序域。例如，现在的智能手机允许用户跟踪彼此的位置。这个原则也可以很容易地应用到跟踪其他东西，包括宠物、孩子、汽车、船等等。

全球定位系统使用多达72颗卫星，每颗卫星在高度约20,000公里的轨道上运行。每颗卫星最多有四个原子钟，这些原子钟由地球上的特殊站定期校准。一颗卫星不断地广播它的位置，并且用它的本地时间来标记每条消息的时间。这种广播允许地球上的每一个接收器精确地计算自己的位置，原则上，只使用四颗卫星。为了解释，让我们首先假设所有的时钟，包括接收器的时钟，都是同步的。

为了计算位置，首先考虑二维情况，如图6.24所示，其中绘制了三颗卫星，以及表示与每颗卫星距离相同的点的圆圈。我们看到这三个圆的交点是一个唯一的点。（确定一个点的原理，由于卫星只能发送探测点到卫星的距离，所以用同心圆来表示所有的可能，而使用三个卫星就能锁定一个点）

这个相交圆的原理可以扩展到三维，这意味着我们需要知道到四个卫星的距离来确定地球上接收器的经度、纬度和高度。这种定位相当简单，但是当我们从理论转向实践时，确定到卫星的距离就变得复杂了。我们至少需要考虑两个重要的现实事实。

1. 卫星位置的数据到达接收器需要一段时间。
2. 接收机的时钟通常与卫星的时钟不同步。

假设来自卫星的时间戳是完全准确的。dertaR表示接收者时钟和实际时间的差值。当接收到带有时间戳Ti的卫星Si的消息时，接收机测得的时延Di由两部分组成: 实际的时延和时钟差值：



这里Tnow是实际时间。则卫星和我们的真是距离为（c为光速）：



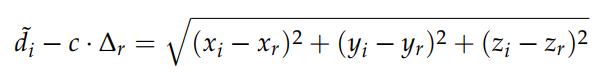
而测量距离为：



被重写为：



所以，算上时钟误差，实际距离为：



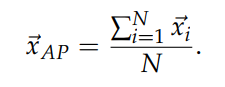
其中xi、yi、zi表示卫星Si的坐标。我们现在看到的是一个含有四个未知数(xr, yr, zr，还有Dr)的二次方程系统。所以我们需要4个卫星参考点和 dertaR，因此，GPS测量也会给出实际时间。

到目前为止，我们假设测量是完全准确的。当然，实际上并不是。有许多错误的来源,从卫星原子钟的这一事实并不总是完美的同步,卫星不知道准确的位置,接收机的时钟精度有限,信号传播速度并不是常数(似乎慢下来当输入信号,例如,电离层),等等。平均而言，这将导致5-10米的误差。需要特殊的调制技术以及特殊的接收器来提高精度。使用所谓的差分GPS，通过广域链路发送校正信息，可以进一步提高精度。

**当GPS无法使用的时候**

GPS的一个主要缺点是它通常不能在室内使用。为此，还需要其他技术。一种日益流行的技术是利用大量可用的WiFi接入点。基本思想很简单：如果我们有一个已知访问点及其坐标的数据库，并且我们可以估计到一个访问点的距离，那么只要有三个检测到的访问点，我们应该能够计算出我们的位置。当然，实际并没有这么简单。

一个主要的问题是确定接入点的坐标。一个流行的方法是通过**war driving**来实现这一点：使用支持无线网络的设备和GPS接收器，有人驾驶或步行穿过一个区域，并记录观察到的接入点。访问点可以通过其SSID或mac级网络地址来标识。在估计接入点AP的坐标之前，应该在几个位置检测接入点AP。计算质心是一种简单的方法。假设我们在N个不同的位置检测到了AP,x1,x2…xn其中每个位置xi由GPS接收机提供的一个(经纬度)对组成。然后我们简单地估计AP的位置xap为



将观测到的信号强度考虑在内，并给予观测到的信号强度相对较高的位置比只检测到微弱信号的位置更大的权重，可以提高精度。最后，我们得到了访问点坐标的估计值。这种估计的准确性受到以下很大的影响：

每个GPS测点的精度

接入点具有非均匀传输范围的事实

采样检测点个数N

研究表明，一个接入点的坐标估计可能离实际位置几十米远。此外，接入点来来去去的速度相对较高。然而，定位和定位接入点是广泛流行的，例如开放访问Wigle数据库，它是通过人群资源填充的。

**节点逻辑定位**

与其试图在分布式系统中找到节点的绝对位置，另一种方法是使用逻辑的、基于邻近的位置。在**几何覆盖网络**中，每个节点在一个m维几何空间中都有一个位置，这样该空间中两个节点之间的距离就反映了一个真实世界的性能指标。计算这样的位置是**网络坐标系统**(Network coordinate System，简称NCS)的核心业务，Donnet等[2010]对此进行了调查。最简单也是最实用的例子是距离对应于节点间的延迟。换句话说,给定两个节点P和Q,那么距离dˆ(P, Q)反映需要多长时间将一个消息从P,发送到Q,反之亦然。我们使用符号dˆ表示距离。

几何叠加网络有许多应用。考虑服务器O上的Web站点被复制到多个服务器S1……， SN的情况。当客户机C从O请求一个页面时，后者可能会决定将该请求重定向到最接近C的服务器，即提供最佳响应时间的服务器。如果每个服务器副本的位置都已知，则很容易做出选择。注意，这样的选择只需要在O处进行本地处理。换句话说，例如，不需要对C和每个副本服务器之间的所有延迟进行采样。

另一个例子是最优复制位置。再次考虑一个收集了客户位置的Web站点。如果站点要将其内容复制到N个服务器，那么它可以计算出放置副本的N个最佳位置，从而使客户机到副本的平均响应时间最小。如果客户机和服务器具有反映节点间延迟的几何位置，那么执行这样的计算几乎是完全可行的。

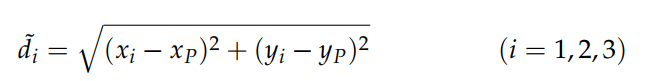
最后一个例子是**基于位置的路由**。在这种方案中，仅使用定位信息将消息转发到其目的地。例如，一个简单的路由算法，让每个节点将消息转发给最接近目标的邻居。虽然可以很容易地看出这个特定的算法不需要收敛，但是它说明了只使用局部信息来做决策。不需要像传统路由算法那样，将链路信息或诸如此类的信息传播到网络中的所有节点。

（这里讲的就是，用一些逻辑上的指标来构建逻辑地理位置。逻辑指标可以用点到点之间的传送时间来标量，这个可以表示为点点距离，然后得知这些信息，其他的一些功能就可以实现了）。

**集中式定位**

在m维几何空间中定位节点需要测量m+1个距离，如之前讲的2维需要三个测量点。假设节点P想计算它自己的位置，它与其他三个具有已知位置的节点接触，并测量它到每个节点的距离。

节点P通过求解含有两个未知数的三个二次方程，计算出自己的坐标(xP, yP)



在这里,我们使用d˜表示测量或估计距离。如之前所说，d˜也可以表示P和测量点(**xi, yi**)的延迟。这个延迟可以估计为往返延迟的一半，但是应该清楚，它的值会随着时间的推移而不同。如果P想要重新计算位置的话，结果就是对P重新定位了。另外，如果其他节点使用P的当前位置来计算自己的坐标，那么很明显，定位P的误差会影响其他节点的定位精度。

还应该清楚的是，一组节点之间的测量距离通常并不一致。例如，假设我们在一维空间中计算距离，如图6.25所示。在这个例子中，我们发现RQ距离为2.0，PQ距离为1.0，而PR距离为2.8（不是3.0），这表明一组测量结果的不一致性。

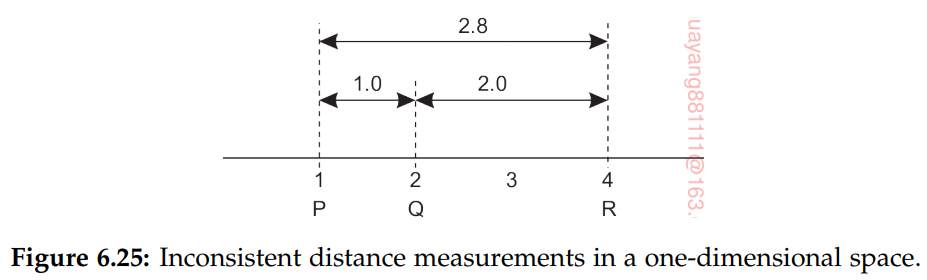
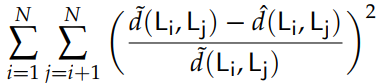


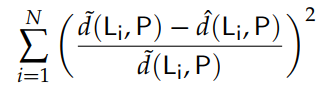
图6.25还显示了如何改进这种情况。在我们的简单例子中，我们可以通过计算二维空间中的位置来解决不一致性。然而，在处理许多度量时，这本身并不是一个通用的解决方案。事实上，考虑到Internet延迟测量可能违反三角形不等式，通常不可能完全解决不一致。三角形不等式指出，在几何空间中，对于任意三个节点P Q R，它总是成立的：



有很多方法可以解决这些问题。Ng和Zhang[2002]提出的一种常用方法是使用N个特殊节点L1，…， LN，被称为Landmarks。（后面简短介绍了这个方法，没看明白鸭）Landmarks首先计算这些个特殊L1。。。LN节点之间的距离，然后寻找一个中心节点，并利用中心节点计算这些landmark节点的坐标。这个中心节点的选择应使下面公式计算出的误差最小化：



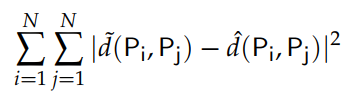
这里d^(Li,Lj)表示实际定位后 Li和Lj的距离。（~表示测量值，^表示选了中心点后的计算值）。其中有个隐藏的参数，即维度m。明显的N>m，但没有人会阻止我们尽量选择一个小的m。在这种情况下，节点P 通过测量和N个点的距离，并通过最小化以下这个公式来计算坐标：



事实证明，通过精心的挑选，m=6或者7，然后误差控制在因数为2的数量级。

**分散的定位**

解决此问题的另一种方法是将节点集合视为一个巨大的系统，其中节点通过spring彼此连接。。|d~(P,Q)-d^(P,Q)| 表示相对于springs系统静止的情况，节点P和节点Q位移的程度。通过让每个节点(稍微)改变其位置，可以看出系统最终会收敛到一个聚合误差最小的最优组织。在一个有N个节点的系统中，P1…， PN, Vivaldi的目标是最小化以下汇总误差:

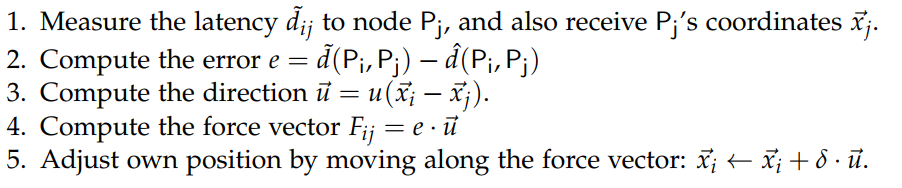


d~表示测量值，d^表示实际值。X表示节点P的坐标。在这种情况下，每个节点都被放置在一个集合区域，然后Pi对Pj施加的力为：



F>0表示Pi推着Pj，反之则拉。现在节点P反复执行以下操作：

（由于这段没懂，直接截图以下内容）



关键因素是d的选择:太大，系统会振荡;如果规模太小，并且收敛到稳定的状态，将需要很长时间。关键是要有一个自适应值，当误差较大时，这个值也很大，但当只需要进行少量调整时，这个值就很小。

6.6分布式事件匹配

作为过程间协调的最后一个主题，我们对分布式事件匹配进行讨论。事件匹配，或者更准确地说，通知过滤，是发布-订阅系统的核心。问题归结为以下几点：

进程通过订阅指定它感兴趣的事件S。

当流程发布事件发生的通知N时，系统需要查看S是否与N匹配。

在匹配的情况下，系统应该向订阅者发送通知N，可能包括与发生的事件相关的数据。

因此，我们至少需要满足两件事: (1)根据事件匹配订阅， (2) 并在匹配的情况下通知订阅方。两者可以分开，但不一定总是这样。在下面的代码中，我们假设存在一个函数match(S, N)，当订阅S与通知N匹配时，该函数返回true，否则返回false。

**中心化的实现**

一种非常简单的实现是提供一个中心服务器来处理所有的订阅和通知。在这种模式中，订阅者只需提交订阅，订阅随后被存储。当发布者提交通知时，将针对每个订阅检查该通知，当找到匹配项时，将复制该通知并将其转发给关联的订阅方。

显然，这不是一个可伸缩的解决方案。但是，如果能够有效地进行匹配，并且服务器本身具有足够的处理能力，那么这个解决方案在很多情况下都是可行的。例如，使用集中式服务器是实现Linda元组空间或Java空间规范的解决方案。同样，许多在单个部门或组织（单个系统，或者系统中的单一组织）中运行的发布-订阅系统也可以通过中央服务器实现。在这些情况下，重要的是能够有效地实现匹配功能。在实践中，在处理基于主题的筛选时经常会遇到这种情况:然后通过检查属性值是否相等来进行匹配。

请注意，扩展集中实现的一个简单方法是在多个服务器上确定地划分工作。一个标准的方法是使用两个函数，Baldoni等人[2009]解释了这一点：

函数sub2node(S)，通过订阅S，映射到一组非空的主机

函数not2node(N)，通过通知N，映射到一组非空的主机

Sub2node(S)映射的主机叫做S的**rendezvous nodes**。同样sub2Node(N)是N的**rendezvous nodes**。唯一的约束条件是sub2node(S) 和 sub2node(N)不是空。也就是必须有一台机器能够处理订阅和通知。实际上，基于主题的发布订阅系统通过对主题名称使用哈希函数来满足这个约束。

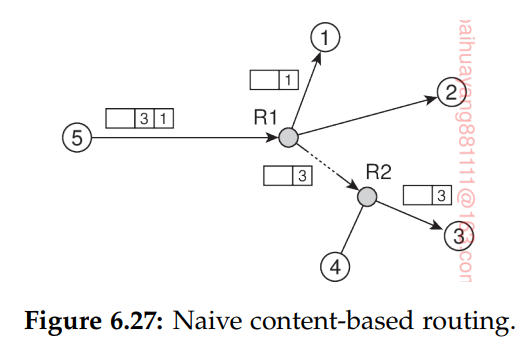
通过在多个服务器之间分配匹配并划分工作，可以扩展这种中心化的服务器。这些服务器通常称之为**brokers**，都是在覆盖网络上组成的。然后，问题就变成了如何将通知路由到适当的订阅者集。在Baldoni等人[2009]之后，我们区分了三个不同的类别:(1)flooding，(2)选择性路由 (3)基于gossip的传播。Kermarrec和Triantafillou[2013]提供了一项关于结合对等网络和发布-订阅系统的广泛调查。

确保通知到达其订阅者的直接方法是广播。有两种方式。第一，我们再每个**broker**上部署订阅，而只在一个broker上进行发布通知。后者将处理标识匹配订阅并随后复制和转发通知。另一种方法是只在一个broker上存储订阅，同时向所有代理广播通知。在这种情况下，匹配分布在broker之间，这可能导致代理之间的工作负载更加平衡。（以上的方式 应该是flooding）

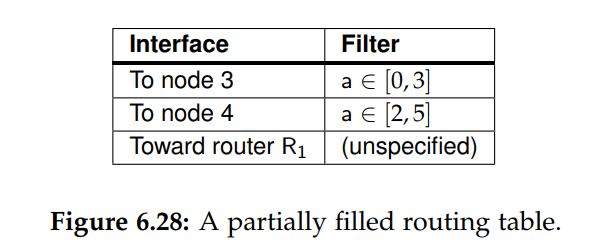
当系统很大时，flooding并不是最好的方式。相反，在覆盖网络上进行跨越代理的路由是必须的。这是典型的**以信息为中心的网络**，它利用基于**名称的路由**[Ahlgren et al.， 2012;Xylomenos等，2014]。基于名称的路由是选择性通知路由的一种特殊情况。在此设置中至关重要的是，代理可以通过考虑通知消息的内容来进行路由决策。更准确地说，假设每个通知携带足够的信息，可以用来切断已知不指向其订阅者的路由。

Carzaniga等人[2004]提出了一种实用的选择路由方法。考虑一个由N个代理组成的发布-订阅系统，其中客户机也可以发送订阅和检索通知。Carzaniga等人提出了一种两层路由方案，其中最低层由连接N个代理的共享广播树组成。有多种方法可以设置这样的树，从网络级组播支持到应用程序级组播树，如我们在第4章中讨论的。在这里，我们还假设已经用N个代理作为结束节点建立了这样一个树，以及一组组成路由器的中间节点。注意，服务器和路由器之间的区别只是逻辑上的:一台机器可以同时承载这两种进程。

每个代理向其他代理广播它的订阅。因此，每个代理将能够编译(subject, destination)对的列表。然后，每当进程发布通知N时，其关联的代理将目标代理预先添加到该消息中。当消息到达路由器时，路由器可以使用列表决定消息应该遵循的路径，如图6.27所示。



我们可以改进路由器的功能，以决定将通知转发到何处。为此，每个代理通过网络广播其订阅，以便路由器可以组成路由过滤器。例如，假设图6.27中的节点3订阅的通知的属性a位于[0,3]范围内，但是节点4希望消息的属性位于 [2,5]。在本例中，路由器R2将创建一个路由过滤器，作为一个表，其中为它的每个传出链接(在本例中是三个:一个到节点3，一个到节点4，一个到路由器R1)提供条目，如图6.28所示



更有趣的是路由器R1发生了什么。在本例中，来自节点3和节点4的订阅规定，间隔为[0,3]和[2,5]=[0,5]的任何通知都应该沿着路径转发到路由器R2，这正是R1将存储在其表中的信息。不难想象，可以支持更复杂的订阅组合。

这个简单的例子还说明，当节点离开系统时，或者当它不再对特定的通知感兴趣时，它应该取消订阅，并将此信息广播给所有路由器。这种取消订阅，可能导致调整各种路由过滤器。延迟调整在最坏的情况下将导致不必要的流量，因为通知可能沿着不再有订阅者的路径转发。所以，需要及时作出调整，使性能保持在可接受的水平。

最后一种分布式事件是基于gossiping。其基本思想是，对相同通知感兴趣的订阅者从他们自己的覆盖网络(通过闲话构建)中形成，因此一旦发布了通知，只需将其路由到适当的覆盖中。（就是相同的订阅者抱个团）。对于后者，可以使用随机游走。这种方法在TERA [Baldoni et al.， 2007]中得到了应用。作为一种地台，在PolderCast [Setty]中发布者首先加入订阅用户的覆盖层，然后才会大量发送通知。订阅者覆盖层为每个主题构建，并构成一个环形结构，其中包含快捷方式（shortcut），以方便有效地传播通知。

6.7 Gossip-based coordination

作为协作的最后一个主题，我们来看几个部署Gossip的重要示例。下面，我们将分别研究聚合、大规模peer抽样和覆盖构造。

**聚合**

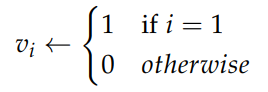
让我们来看看流行病协议的一些有趣的应用。我们已经提到了扩展更新，这可能是部署得最广泛的应用程序。同样，Gossip可以用来发现具有一些传出链接的节点，从而继续应用Gossip。

另一个有趣的应用领域是简单地收集并在实际上聚合信息[Jelasity et al.， 2005]。考虑以下信息交换。每个节点最初选择一个任意的数，比如vi。当节点Pi与节点Pj接触时，每个节点更新其值为:



明显的，在这次更新之后，Pi和Pj将拥有相同的值。事实上，不难看出最终所有节点的值都是相同的，即所有初始值的平均值。传播速度是指数的。

计算平均值有什么用？考虑下这种情况，所有的节点Pi都将Vi设置为0，除了Pj，它的Vj=1：



如果有N个节点，那么最终每个节点将计算平均值，即1/N。因此，每个节点Pi都可以估计系统的大小为1/vi。

当节点定期加入和离开系统时，计算平均值可能会很困难。解决这个问题的一个切实可行的办法是引入epochs。假设P1是稳定的，它会简单的开始一个epoch。当节点Pi第一次看到一个新的epochs时，它将自己的变量vi重置为零，并再次开始计算平均值。（黑人问号？？？）

当然，其他结果也可以计算出来。例如，我们不需要像P1这样的固定节点来开始计算平均值，我们可以很容易地选择一个随机节点，如下所示。每个节点Pi最初将vi设置为相同区间(0,1)的随机数，并将其永久存储为mi。在节点Pi和Pj之间进行交换时，每个节点将其值更改为：

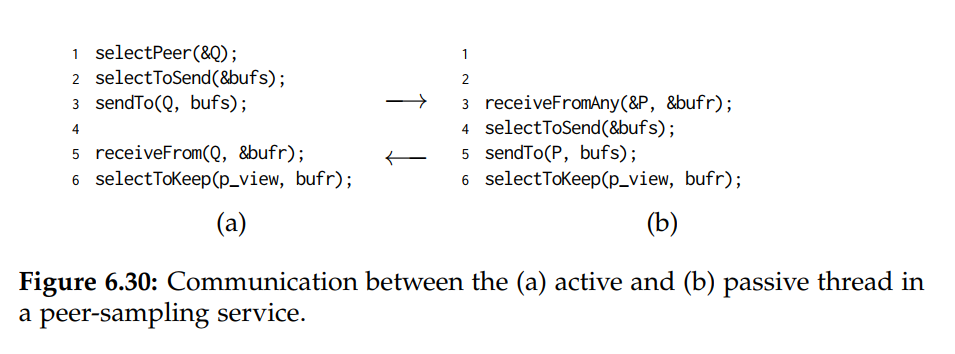


对于mi < vi的每个节点Pi，在开始计算平均值时，将失去作为发起者的竞争。最终，只会有一个赢家。当然，尽管很容易得出一个节点已经失败的结论，但要判断它是否已经成功则要困难得多，因为它仍然不确定是否所有结果都已得到。解决这个问题的办法是很乐观的：一个节点总是假定它是赢家，除非证明其他人是赢家。此时，它只需将用于计算平均值的变量重置为零。注意，到目前为止，可能同时执行了几个不同的计算(在我们的示例中，计算最大值和平均值)。

A peer-sampling service

在传染病协议中一个重要的方面是节点P从网络中的所有可能节点中随机选择节点Q的能力。当我们仔细考虑这个问题时，我们可能会遇到一个严重的问题：如果网络由成千上万个节点组成，P如何能够在不全面了解网络的情况下随机选择其中一个节点呢？对于较小的网络，通常可以求助于注册了每个参与节点的中央服务。显然，这种方法永远无法扩展到大型网络。

解决方案是构建一个完全分散的**对等抽样服务**，简称**PSS**。事实证明，使用流行病协议可以构建PSS，这有点违反直觉。Jelasity等人[2007]研究过，每个节点维护一个c邻居列表，理想情况下，其中每个邻居表示当前节点集中随机选择的活动节点。这个邻居列表也称为**局部视图**。构建这种局部视图的方法有很多。在它们的解决方案中，会假定节点定期交换来自其局部视图的条目。每个条目标识网络中的另一个节点，并具有关联的age，该age指示改节点的实际年龄。这里使用两个线程，如图6.30所示。



不同的选择操作如下:

•selectPeer:从局部视图中随机选择一个邻居

•selectToSend:从partial视图中选择一些其他条目，并添加到所选邻居的列表中。

•selectToKeep:将接收到的条目添加到部分视图中，删除重复项，并将视图缩小到c项

活动线程主动与另一个节点通信。它从它的局部视图中选择通信点。它继续构造一个包含c/2 + 1条目的列表，其中包括一个标识自身的条目。其他的条目从当前的局部视图中获取。在将局部视图列表发送给邻居节点后，它将等待一个响应。

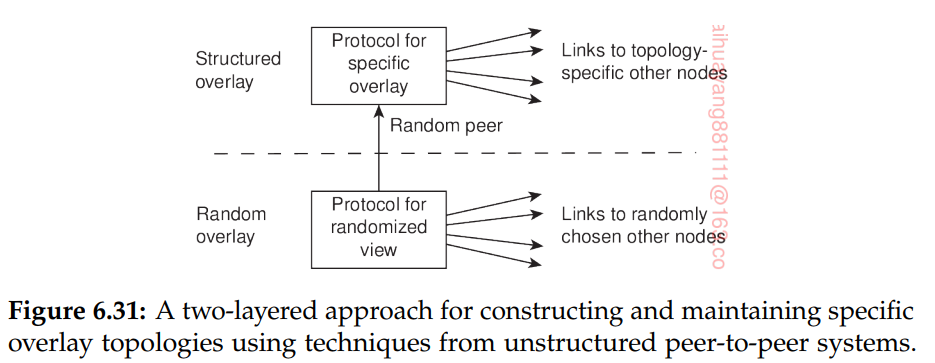
与此同时，该邻居还将通过图6.30(b)所示的被动线程构造一个列表，该被动线程的活动与主动线程非常相似。

关键是构建新的局部视图。这个视图，从接受的列表获取，包含了c个对等点的条目。本质上，这里有两种方式来构建一个新的视图。首先，这两个节点可能决定丢弃它们彼此发送的条目。实际上，这意味着他们将交换部分原始视图。第二种方法是丢弃尽可能多的旧条目(在实践中，这意味着在每一轮谣言传播之后，每个部分视图中的每个条目的年龄都会增加1)。

事实证明，只要对等点定期运行刚才描述的交换算法，从动态变化的部分视图中随机选择一个对等点，就与从整个网络中随机选择一个对等点没有什么区别。当然，从局部视图中选择对等点的频率应该与刷新局部视图的频率大致相同。因此，我们构建了一个完全分散的基于谣言的对等抽样服务。同行抽样服务的一个简单且经常使用的实现是Cyclon [Voulgaris et al.， 2005]。

**Gossip-based overlay construction （没懂）**

虽然结构化和非结构化对等系统似乎形成严格的独立类，但实际上并不需要这样(参见Castro等人[2005])。一个关键的观察是，通过仔细地交换和选择部分视图中的条目，可以构建和维护覆盖网络的特定拓扑。这种拓扑管理通过采用两层方法实现，如图6.31所示。



最低层构成一个非结构化的对等系统，其中节点定期交换其部分视图的条目，目的是提供对等抽样服务。在本例中，准确性指的是部分视图应该包含引用随机选择的活动节点的条目。最低层将其部分视图传递给更高的层，在那里将进行额外的条目选择。这将导致与所需拓扑对应的第二个邻居列表。Jelasity和Kermarrec[2006]提出使用一个排序函数，根据相对于给定节点的某种标准对节点进行排序。一个简单的排序函数是通过增加到给定节点P的距离来对一组节点排序。在这种情况下，如果最低层继续传递随机选择的节点，节点P将逐步构建其最近邻居的列表。

例如，考虑一个大小为N×N的逻辑网格，网格的每个点上都有一个节点。每个节点都需要维护一个c近邻列表，其中(a1, a2)到(b1, b2)的节点距离定义为d1 + d2, di = min(N - |ai – bi|, |ai – bi|)。如果最低层定期执行图6.30所示的协议，则将演化的拓扑是一个环面，如图6.31所示。

6.8总结

与进程之间的通信密切相关的是分布式系统中的进程如何同步的问题。同步就是在正确的时间做正确的事情。在分布式系统和计算机网络中，普遍存在的一个问题是没有全局共享时钟的概念。换句话说，不同机器上的进程对时间有自己的概念。

在分布式系统中有多种同步时钟的方法，但是所有的方法本质上都基于交换时钟值，同时考虑到发送和接收消息所需的时间。通信延迟的变化及其处理方法在很大程度上决定了时钟同步算法的准确性。

在很多情况下，知道绝对时间是不必要的。重要的是不同进程中的相关事件以正确的顺序发生。Lamport指出，通过引入逻辑时钟的概念，可以使一组进程就事件的正确顺序达成全球协议。从本质上讲,每一个事件, 如发送或接收消息, s分配一个全局唯一合乎逻辑的时间戳C (e),这样当事件发生在b前的话则有, C (a) < C (b)。Lamport时间戳可以扩展到向量时间戳:如果C(a) < C(b)，我们甚至知道事件a在b之前。

一类重要的同步算法是分布式互斥算法。这些算法确保在一个分布式进程集合中，一次最多只能访问一个共享资源。如果我们使用一个协调器，就可以很容易地实现分布式互斥。完全分布式算法也存在，但它们的缺点是通常更容易出现通信和进程故障。

进程之间的同步通常需要一个进程充当协调器。在协调器不是固定的情况下，有必要让分布式计算中的进程决定谁将成为协调器。这样的决定是通过选举算法做出的。选择算法主要用于协调器可能崩溃的情况。但是，它们也可以应用于peer-to-peer系统中超级对等点的选择。

与这些同步问题相关的是在几何叠加中定位节点。其基本思想是在m维空间中分配每个节点的坐标，使几何距离可以作为两个节点之间延迟的精确度量。坐标分配方法与GPS中确定位置和时间的方法非常相似。

尤其具有挑战性的是分布式事件匹配，它位于发布-订阅系统的核心。相对简单的情况是，当我们有一个中心实现时，基本上可以通过一对一的比较来实现订阅与通知的匹配。然而，一旦我们的目标是分配负载，我们就会面临这样的问题:在不知道需要哪种通知的情况下，预先决定哪个节点负责订阅的哪个部分。这对于基于内容的匹配尤其有问题，因为最终需要高级过滤技术将通知路由到适当的订阅者。

最后，基于gossip的协调中最重要的方面是能够从整个覆盖层中随机选择另一个对等点。事实证明，通过确保以随机的方式定期刷新部分视图，我们可以使用闲话实现这样一个同行抽样服务。将对等抽样与局部视图中条目的选择性替换相结合，可以有效地构建结构化覆盖网络。