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。

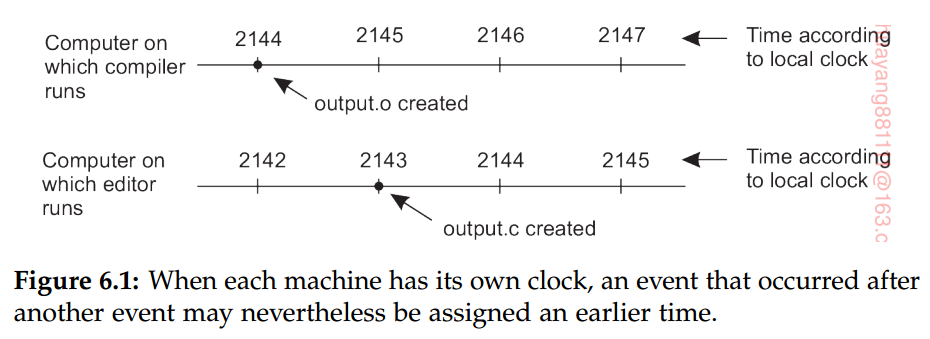
最后，我们考虑三个不同的基于gossip的协调问题:聚合、对等抽样和覆盖构造。分布式算法有各种各样的风格，它们是为非常不同类型的分布式系统开发的。

**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）。

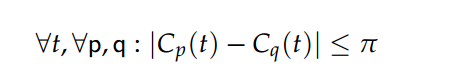
几颗地球卫星也提供UTC服务。 地球同步环境业务卫星可以提供0.5 msec的UTC精度，其他一些卫星甚至做得更好。通过结合来自多个卫星的接收信号，地面时间服务器的精度可以达到50 nsec。UTC接收器是商用的，许多计算机都配备了一个。

**时钟同步算法**

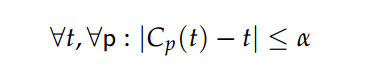
如果一台机器有一个UTC接收器，目标就是保持所有其他机器与它同步。如果没有一台机器有UTC接收器，那么每台机器都会记录自己的时间，目标是尽可能地让所有机器保持在一起。已经提出了许多算法来实现这种同步。

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。



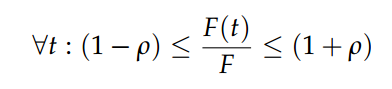
请注意，精度仅指分布式系统中机器之间时钟的偏差。当涉及到UTC时间时，我们称之为准确度。



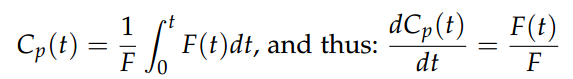
时钟同步的整个概念是我们保持时钟精确，称为内部同步，而准确，称为外部同步。一组准确的时钟在α,内精确绑定π= 2α。然而，精确并不能让我们对时钟的准确性得出任何结论。

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。

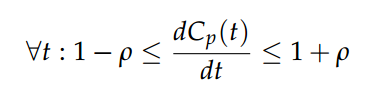
硬件时钟的规范包括其**最大时钟漂移率ρ**。如果F(t)表示硬件时钟在t时刻的实际振荡频率，F表示其理想(恒定)频率，则硬件时钟符合其规范。



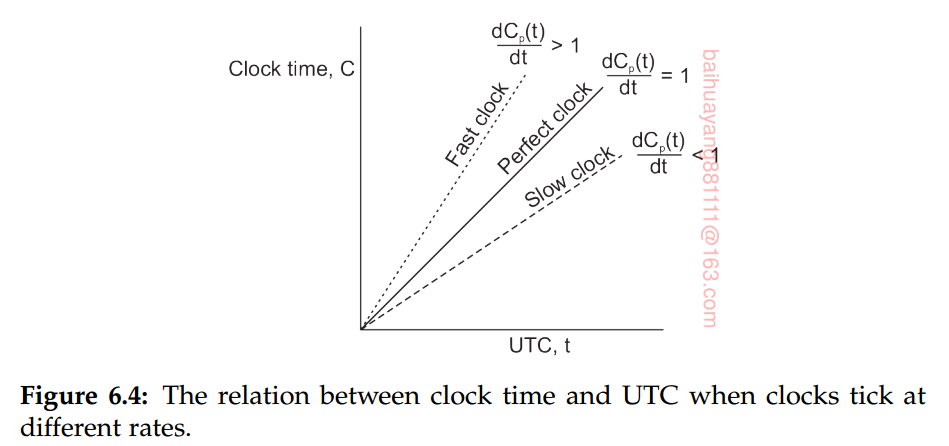
通过使用硬件中断，我们将软件时钟直接耦合到硬件时钟，从而也耦合到硬件时钟的时钟漂移率。（就是这种漂移率也会体现在软件时钟当中）。



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。



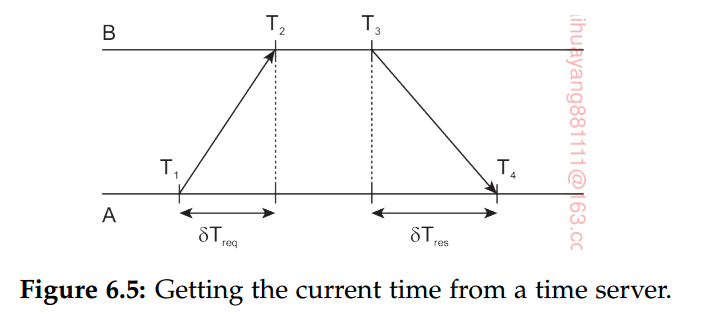
慢时钟、完美时钟和快时钟如图6.4所示



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。

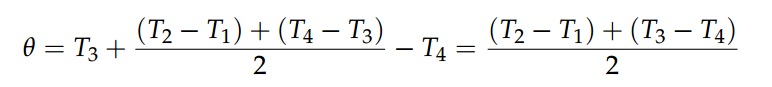
**网络时间协议**

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

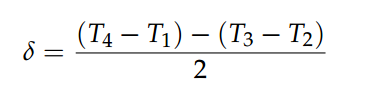
为：



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。