时间，时钟，及分布系统的事件排序

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport, Massachusetts Computer Associates, Inc.

Concurrency: the Works of Leslie Lamport. 179–196. Oct. 2019. DOI:10.1145/3335772.3335934
Originally published in *Comm. of the ACM*. 21(7), 558–565. July 1978. DOI:10.1145/359545.359563

译者：Ying ZHANG. 2020-06, 11; 2021-04, 08, 10
https://ying-zhang.github.io/time/1978-Lamport-cn.pdf

研究了分布系统中一个事件先于另一事件发生的概念，基于此概念定义了事件的偏序。给出了用于同步逻辑时钟的分布式算法，该逻辑时钟可以对事件全局排序。通过一个解决互斥问题的方法展示了全序的用途。然后，改进算法以同步物理时钟，并得出物理时钟同步偏差的上界。

引言

时间是我们思维中的基础概念。它源自事件发生的顺序这一**更基本**的概念。我们说某件事发生在 3:15，是指它发生在时钟读数 3:15 **之后**，且在 3:16 之前。事件按时间排序这一概念普遍存在于我们的思维中。例如，在航空订票系统中，我们规定，航班满员**之前**的订票请求应当成交。但是，我们将看到，在考虑分布系统中的事件时，必须重新审视这一概念。

分布系统由一组在空间上分离的不同进程组成。这些进程通过交换消息相互通信。互连计算机的网络（例如 ARPA 网络）就是一个分布系统。单个计算机也可以看作是分布系统，其中 CPU、存储单元和输入输出通道是独立的进程。如果消息传输的延迟比单个进程中事件的间隔还显著，那么就可以认为该系统是分布的。

我们将主要关注空间分离的计算机系统。但是，我们的许多结论是普遍适用的。特别的，由于某些事件的发生顺序不可预测，因此单机中的多进程系统会遇到与分布系统类似的问题。

在分布系统中，有时无法区分两个事件哪个先发生。因此，“先于发生（happened before）”关系只是系统中事件的偏序〔译注：Partial Ordering，直译为“部分序”，即只有部分事件存在明确的先后顺序，不是所有事件都存在明确的先后顺序〕。由于人们没有完全意识到这一事实及其含义，从而经常导致问题。

我们在本文中讨论了由“先于发生”关系定义的偏序，并给出了一个分布式算法，将偏序扩展为所有事件的一致全序。该算法可以为实现分布系统提供有用的机制。我们用一个解决互斥问题的简洁方法来展示上述算法的应用。若通过此算法获得的顺序与用户感知的顺序不同，则可能发生非预期的异常行为。可以通过引入实际的物理时钟来避免这种情况。我们描述了一种用于同步物理时钟的简单方法，并推导了物理时钟失步漂移的上限。

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

偏序

如果事件 a 发生的时刻比事件 b 发生的时刻早，人们常说事件 a 先于事件 b 发生。人们可能会依据时间的物理理论来论证这一定义是正确的。但是，在讨论系统是否正确实现规约之前，首先必须用系统内可观察的事件表示出规约。若规约是基于物理时间的，则系统必须包含真实的时钟。**即使系统确实包含真实的时钟，也仍然存在问题：即时钟并不是十分准确，不能保持精确的物理时间。因此，我们将在不使用物理时钟的情况下定义“先于发生”的关系**。

我们首先来更准确地定义系统。假设系统由一组进程组成。每个进程都包含一系列事件。取决于不同的应用场景，一个事件可能是在计算机上执行一段子程序，或者仅仅是执行一条机器指令。我们假设一个进程的所有事件组成一个序列，如果 a 先于 b 发生，那么在这个序列中，a 排在 b 之前。换言之，一个进程可以定义为已经事先全部排序的一组事件。这也是进程（process）的字面含义。[1] 容易扩展定义，以允许将进程拆分为不同的子进程，此处不再赘述。

假设发送或接收消息是进程中的事件。我们可以定义“先于发生（happened before）”关系如下（用“$\rightarrow$”表示）。

定义：系统中事件集合上的关系“$\rightarrow$”是满足以下三个条件的最小关系：

（1）若 a 和 b 是同一进程中的事件，且 a 早于 b 发生，则 $a \rightarrow b$。

（2）若 a 是一个进程的消息发送事件，b 是另一进程接收同一消息的事件，则 $a \rightarrow b$。

（3）若 $a \rightarrow b$ 且 $b \rightarrow c$，则 $a \rightarrow c$。

对两个不同的事件 a 和 b，若 $a \nrightarrow b$ 且 $b \nrightarrow a$，则称 a 和 b 是**并发的（concurrent）**〔译注：与其称为“并发的”，不如称为“未知的”或“不关心的”〕。

我们假设对任意事件 a，$a \nrightarrow a$ （事件先于自身发生似乎没有实际意义）。这意味着 $\rightarrow$ 是系统中所有事件的集合上的**反自反**（irreflexive）的偏序关系。

可以用如图 1 的“时空图”来帮助展示偏序的定义。水平方向表示空间，垂直方向表示时间——较早的时间在下方。圆点表示事件，竖线表示进程，波浪线表示消息。[2] 易知，$a \rightarrow b$ 表示可以沿着进程线和消息线，按时间流逝方向从图中的 a 到达 b。例如，图 1 中有 $p_1 \rightarrow r_4$ 〔译注：经 q_2，q_4，r_3〕。

该定义的另一种理解是 $a \rightarrow b$ 意味着事件 a **可能因果地**影响事件 b。若两个事件不能彼此影响，则它们是并发的。例如，图 1 中事件 p_3 和 q_3 是并发的。尽管图中暗示 q_3 发生的物理时间比 p_3 更早，但在 p_4 接收到消息之前，进程 P 无法得知进程 Q 在 q_3 做了什么（在事件 p_4 之前，P 最多可能知道 Q **计划**在 q_3 做些什么）〔译注：p_4 事件不会影响“p_3 和 q_3 是并发的”这一关系〕。

对于熟悉狭义相对论的时空不变量的读者来说，这个定义是很自然的，如 [1] 的例子或 [2] 的第一章中所述。在相对论中，事件的顺序定义为消息**可能**发送的顺序。然而，我们采取了更实际的方法，只考虑**确实已经**发出的消息。要想确定系统是否正常，仅能依据那些**确实已经**发生的事件，而非那些**可能**发生的事件。

逻辑时钟

现在，我们将时钟引入系统。我们从一个抽象的观点开始：即时钟只是将数值分配给事件的一种方式，这个数值被认为是事件发生的时间。更准确地说，我们为**每个**进程 P_i 分别定义一个时钟 C_i，它为进程中的任一事件 a 分配一个值 $C_i\langle a\rangle$。**整个系统**的时钟由函数 C 表示，它为任一事件 b 分配数值 $C\langle b\rangle$，其中，若 b 是进程 P_j 中的一个事件，则 $C\langle b\rangle = C_j\langle b\rangle$。到目前为止，我们并没有对数值 $C_i\langle a\rangle$ 与物理时钟的关系做出任何假设，因此可以认为时钟 C_i 是逻辑的，而非物理时钟。它们可以由无实际计时装置的计数器来实

[1] 对事件粒度的选择会影响进程中事件的顺序。例如，消息接收事件可能是指设置计算机的中断位，也可能是指执行中断处理子程序。由于中断处理的顺序不同于中断发生的顺序，因此不同的〔粒度〕选择将影响进程中消息接收事件的顺序。

[2] 注意，接收到的消息可能是乱序的。我们允许将发送多个消息作为单个事件，但为方便起见，我们假定单个消息的接收不与其它消息的发送或接收同时发生。

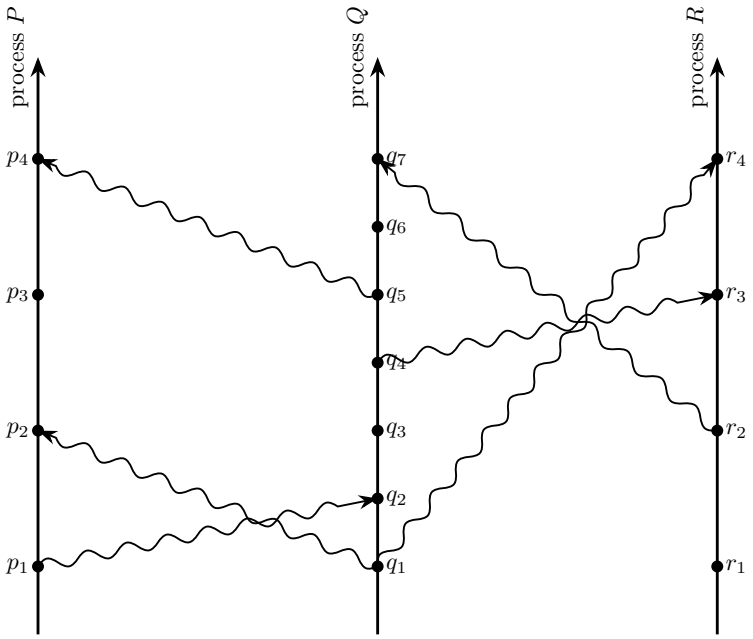

图 1

现〔译注：物理时钟主要由计数器和稳定的频率信号源两部分组成。逻辑时钟是由不断发生的事件推动的。逻辑时钟和物理时钟都是单调的，但逻辑时钟不是均匀的〕。

我们现在考虑这样的时钟系统中正确性的含义。我们不能根据物理时间来定义正确性，因为这需要引入保持物理时间的时钟。我们的定义必须基于事件发生的顺序。最合理条件是，如果一个事件 a 先于另一事件 b 发生，那么 a 发生的时刻应该比 b 的更早。我们将此条件形式化描述如下。

时钟条件：对于任意事件 a，b，若 $a \to b$，则 $C\langle a\rangle < C\langle b\rangle$。

注意，**否命题**并不成立〔译注：即不能由 $a \not\to b$ 得出 $C\langle a\rangle \geq C\langle b\rangle$。此外，逆命题也不成立，即不能由 $C\langle a\rangle < C\langle b\rangle$ 得出 $a \to b$：当然，对同一进程中的事件，逆命题显然是成立的，但对跨进程的事件不总是成立〕，否则将推导出两个并发的事件必定同时发生的错误结论〔译注：考虑到并发关系是对称的〕。在图 1 中，p_2 和 p_3 均与 q_3 是并发的，若否命题成立，则意味着它们都必须与 q_3 发生在相同的时刻，这将违反**时钟条件**，因为已知 $p_2 \to p_3$。

从关系“$\to$”的定义易知，若满足以下两个条件，则会满足**时钟条件**。

C1 若 a 和 b 都是进程 P_i 中的事件，且 a 先于 b 发生，则 $C_i\langle a\rangle < C_i\langle b\rangle$。

C2 若 a 是进程 P_i 发送消息的事件，b 是进程 P_j 接收到该消息的事件，则 $C_i\langle a\rangle < C_j\langle b\rangle$。

让我们在时空图中考虑时钟。我们想象，每个数值代表了一个进程的时钟的一次“滴答”，滴答发生在进程的事件**之间**。例如，若 a 和 b 是进程 P_i 的两个连续的事件，且 $C_i\langle a\rangle = 4$，$C_i\langle b\rangle = 7$，则时钟滴答 5、6 和 7 均发生在这两个事件之间。我们可以画一条虚线的“滴答线”，将不同进程所有相同数值的滴答连起来。从图 1 的时空图**可能**得到图 2。条件 **C1** 意味着各进程线上的任意两个事件之间必有一条滴答线，条件 **C2** 意味着每个消息线必然与某个滴答线相交。从 $\to$ 的图形含义可以很容易地看出为什么这两个条件意味着**时钟条件**。

我们可以将滴答线视为某些时空中直角坐标系的时间坐标线。重画图 2，将这些时间坐标线拉直，就得到了图 3。图 3 是对图 2 中事件的等效表示。如果不在系统中引入物理时间的概念（这需要物理时钟），那么无法区分图 2 和图 3 哪个更好。

读者可能会发现将二维空间的进程网络可视化很有帮助，该网络会产生三维时空图。进程和消息仍然由线表示，但滴答线变成了二维曲面。

现在让我们假设进程是算法，事件表示其执行期间的某些动作。我们将展示如何将时钟引入满足“**时钟条件**”的进程中。进程 P_i 的时钟 C_i 用一个变量（register）来表示，$C_i\langle a\rangle$ 是时钟 C_i 在事件 a 期间的

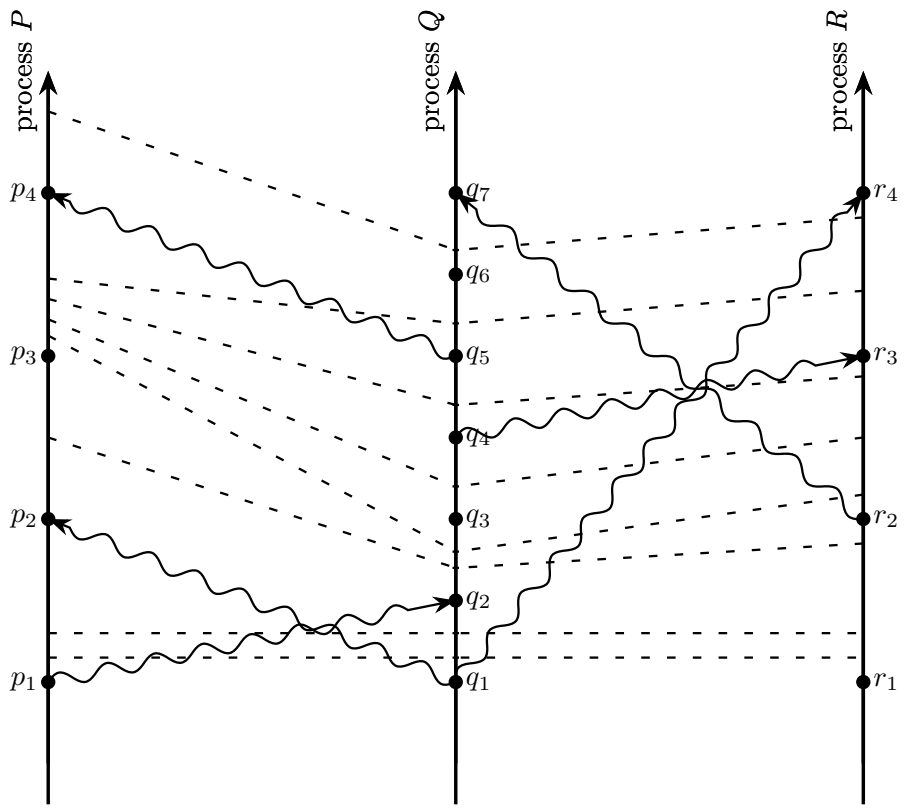
process P
process Q
process R
p_4
p_3
p_2
p_1
q_7
q_6
q_5
q_4
q_3
q_2
q_1
r_4
r_3
r_2
r_1

图 2

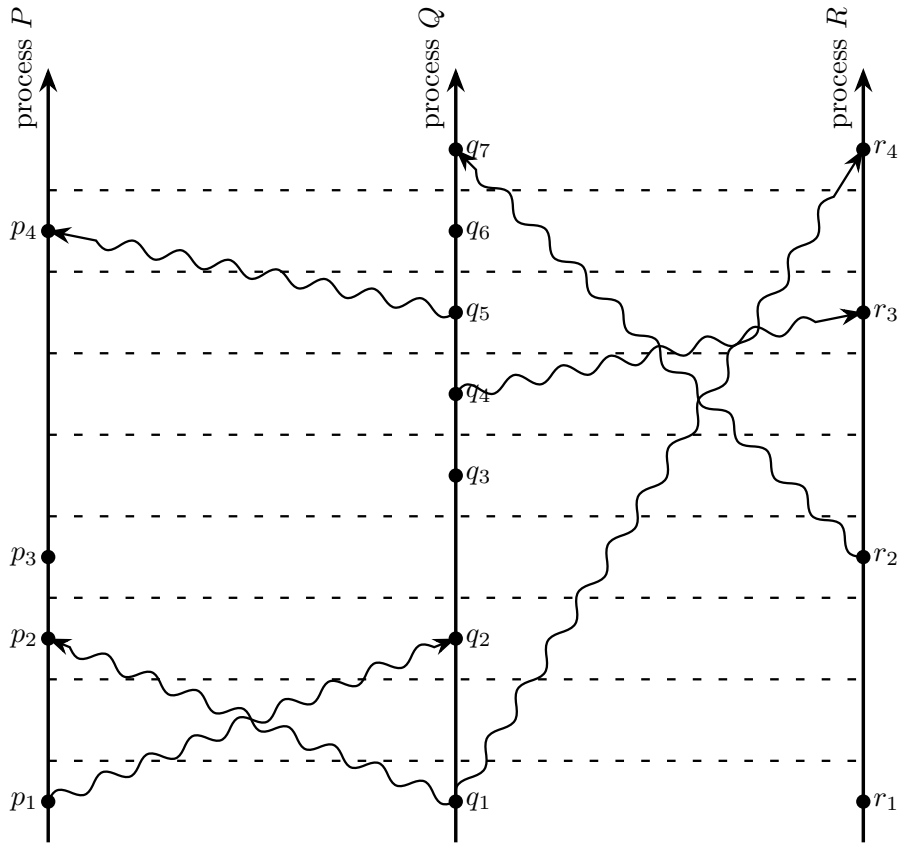
process P
process Q
process R
p_4
p_3
p_2
p_1
q_7
q_6
q_5
q_4
q_3
q_2
q_1
r_4
r_3
r_2
r_1

图 3

值。我们在两个事件之间改变时钟 C_i 的值，从而防止 C_i 值的改变与某个事件同时发生。

为了保证时钟系统满足**时钟条件**，我们需确保其满足条件 **C1** 和 **C2**。条件 **C1** 很简单；进程仅需遵循以下的实现规则：

IR1 每个进程 P_i 在任意两个相继的事件之间增加 C_i。

为了满足条件 **C2**，我们要求每个消息 m 包含一个**时间戳** T_m，该时间戳等于发送消息的时刻。收到时间戳为 T_m 消息后，进程必须推动其时钟，使其晚于 T_m。更确切地说，我们有以下规则。

IR2 (a) 若 a 是进程 P_i 发送消息 m 这一事件，则消息 m 需包含时间戳 $T_m = C_i\langle a\rangle$；

IR2 (b) 一旦收到消息 m，进程 P_j 就将其时钟 C_j 设置为不小于当前值，且大于 T_m 的值。

在 **IR2(b)** 中，我们假定接收消息时先设置时钟 C_j，然后才认为表示接收消息 m 的事件发生了（这只是记号的细节，实际实现中无关紧要）。显然，**IR2** 确保能够满足 **C2**。因此，**IR1** 和 **IR2** 这两条简单的实现规则能满足**时钟条件**，从而它们保证了逻辑时钟系统的正确性〔译注：由上述规则可知，接收消息的事件通常是时钟跳跃点，当然单调性总是保持的〕。

对事件全局排序

我们可以使用满足“**时钟条件**”的时钟系统对系统所有的事件全局排序。我们只需按事件发生的逻辑时刻对事件排序。对时刻相同的情况，进程可以采取任意的全局顺序 $\prec$。

更准确地说，我们定义关系 $\Rightarrow$ 如下：

若 a 是进程 P_i 的事件，b 是进程 P_j 的事件，则 $a \Rightarrow b$ **当且仅当**

(i) $C_i\langle a\rangle < C_j\langle b\rangle$ ； 或

(ii) $C_i\langle a\rangle = C_j\langle b\rangle$ 且 $P_i \prec P_j$。

易知，这定义了全序，且**时钟条件**意味着，若 $a \to b$，则 $a \Rightarrow b$〔译注：反之则不然〕。换句话说，关系 $\Rightarrow$ 是将“先于发生”这一偏序关系补充为全序的一种方式〔译注：注意定义中的“当且仅当”。由上述条件 (i)，不一定能得出 $a \to b$，但一定能得出 $a \Rightarrow b$，因为这是由定义规定的；由上述条件 (ii)，在全序的定义下，两个事件总是存在“先后”关系，这也是“全”的本意，或者 $a \Rightarrow b$，或者 $b \Rightarrow a$，不再有“并发”〕。[3]

$\Rightarrow$ 的顺序取决于时钟系统 C_i，不是唯一的。在满足**时钟条件**的情况下，选择不同的时钟，会产生不同的 $\Rightarrow$ 关系。对扩展了偏序 $\to$ 的全序关系 $\Rightarrow$，存在一个满足**时钟条件**的，产生该全序关系的时钟系统。只有偏序 $\to$ 是由系统事件唯一确定的。

能将事件全局排序对实现分布系统是非常有用的。实际上，实现正确的逻辑时钟系统就是为了获得全序。我们将通过解决以下的互斥问题来说明事件全序的用途。考虑一个由固定数量的进程组成的系统，这些进程共享单个资源。一次只能有一个进程使用该资源，因此，这些进程必须同步以避免冲突。我们希望找到一种将资源授予进程的算法，满足以下三个条件：(I) 已获得资源的进程必须先释放资源，然后才能将资源授予另一进程。(II) 必须按发出请求的顺序来满足资源请求。(III) 若获得资源的进程最终会释放资源，则每个请求最终都能得到满足〔译注：一般的互斥算法中，进程获得资源的顺序是可以任意的，不必按请求顺序，即不必考虑条件 II〕。

我们假定，初始状态下资源已授予了某个进程。

上面这些要求是非常自然的。它们准确地指出了解决方案正确性的涵义。[4] 注意这些条件与事件顺序的关系。条件 II 没有提及对并发的两个请求应先满足哪一个。

重要的是，要认识到这不是一个微不足道的问题。使用一个中心调度进程按接收的顺序来满足请求这

[3] 顺序 $\prec$ 建立了进程的优先级。若需要一种“更公平”的方法，则可以使 $\prec$ 为时钟值的函数。例如，若 $C_i\langle a\rangle = C_j\langle b\rangle$ 且 $j < i$，则可以在 $j < C_i\langle a\rangle \mod N \le i$ 时令 $a \Rightarrow b$，否则令 $b \Rightarrow a$；其中 N 是进程总数〔译注：这种方式使逻辑时钟相同时，进程 i 和 j 的事件顺序不是固定的，但是确定的，即总是可以通过当时的逻辑时钟值计算出来〕。

[4] “最终（eventually）”一词应明确，但需较多篇幅，且偏离了本文主题。

种方案是无法实现的，除非有额外的假设。要理解这一点，令 P_0 为调度进程。假设进程 P_1 先向 P_0 发送共享资源请求，然后又向进程 P_2 发送其它消息。收到消息后，P_2 也向 P_0 发送共享资源请求。有可能 P_2 的请求比 P_1 的先到达 P_0。如果首先满足了 P_2 的请求，那么就违反了条件 II〔译注：即使分别保证了 P_1 向 P_0，P_2 向 P_0 发送消息的顺序，也不能保证 P_1 发出的消息与 P_2 发出的消息之间的顺序。使用中心调度进程的算法与下面的分布式算法相比，如果允许 P_0 等待所有其它节点的消息，按全序对消息排序，然后再做出决策，那么就可以符合请求的顺序了。由于其它进程几乎同时向 P_0 发送消息，所以很可能这些消息在 P_0 看来是并发的，但总是可以对这些消息排出全序〕。

为了解决该问题，我们使用规则 IR1 和 IR2 实现一个时钟系统，用以定义所有事件的全序 $\Rightarrow$。这也给出了所有请求和释放操作的全序。通过此顺序，找到解决方案变得很简单。**它仅需要确保每个进程都了解所有其它进程的操作**。

为了简化问题，我们做一些假设。这些假设不是必需的，引入它们是为了避免陷入实现细节。我们首先假设，对于任意两个进程 P_i 和 P_j，从 P_i 向 P_j 发送的消息按发送的顺序被收到。此外，我们假设消息最终都会被收到（可以通过引入消息序号和消息确认协议来避免这些假设）。我们还假设一个进程可以直接向其它任意进程发送消息。

每个进程都维护私有的**请求队列**，对其它进程都不可见。我们假设请求队列初始仅有一条消息 "T_0 : P_0 **请求资源**"，这里，P_0 是初始获得资源的进程，而时间戳 T_0 小于任何时钟的初始值。

算法有以下五个规则。为了方便起见，假定每个规则定义的动作是单个事件。

1. 为了请求资源，进程 P_i 向其它每个进程发送消息 "T_m : P_i 请求资源"，并将该消息放入自己的请求队列中，其中 T_m 是消息的时间戳。

2. 当进程 P_j 收到消息 "T_m : P_i 请求资源" 时，将其加入自己的请求队列中，并向 P_i 发送（带时间戳的）确认消息。[5]

3. 为了释放资源，进程 P_i 删除其请求队列中的 "T_m : P_i 请求资源" 消息，并向其它每个进程发送（带时间戳的）"P_i 释放资源" 消息。

4. 当进程 P_j 收到 "P_i 释放资源" 的消息时，它也删除自己的请求队列中的 "T_m : P_i 请求资源" 消息。

5. 当满足以下两个条件时，进程 P_i 将获得资源：

   (i) 其请求队列中有一条 "T_m : P_i 请求资源" 的消息，该消息在请求队列中的其它所有请求之前，消息按全序关系 $\Rightarrow$ 排序（为了定义**消息的**全序关系 "$\Rightarrow$"，我们用**发送消息的事件**来标识该消息）。

   (ii) P_i 从其它每个进程都收到了消息，且其时间戳都晚于 T_m。[6]

   请注意，规则 5 的条件 (i) 和 (ii) 由各进程 P_i 在本地自行检查。

容易验证这些规则定义的算法满足条件 I–III。首先，注意到**规则 5** 的条件 (ii)，结合消息顺序接收的假设，保证了 P_i 了解到比它更早的所有请求。由于**规则 3 和 4** 是仅有的从请求队列中删除消息的规则，因此易知**条件 (I)** 成立。**条件 (II)** 源自全序 $\Rightarrow$ 扩展了偏序 $\rightarrow$ 的事实。**规则 2** 保证在 P_i 请求资源之后，**规则 5** 的条件 (ii) 最终会成立。**规则 3 和 4** 意味着，若获得资源的进程最终会释放资源，则**规则 5** 的条件 (i) 最终会成立，从而证明**条件 (III)**。

这是一种分布式算法〔译注：所谓分布式算法，可以理解为每个进程都运行相同的算法，进程可以交换信息，但都在本地独立地做出决策，且所有进程都可以独立地做出全局一致的决策结果〕。每个进程都独立地遵循这些规则，没有中心同步进程或集中存储。这个算法可以推广，为类似的分布式多进程系统实现任意的互斥。互斥由**状态机 (State Machine)** 表示。该状态机由可行命令集合 **C**，可行状态集合 **S** 和函

[5]若 P_j 已经向 P_i 发送了时间戳比 T_m 更晚的消息，则无需发送该确认消息。

[6]若 $P_i \prec P_j$，则 P_i 仅需收到来自 P_j 的时间戳 $\geq T_m$ 的消息。

数 $\mathbf{e} : \mathbf{C} \times \mathbf{S} \to \mathbf{S}$ 组成。关系 $\mathbf{e}(C, S) = S'$ 表示状态机在状态 S 执行命令 C，使机器的状态变为 S'。在我们的例子中，命令集合 C 由所有“P_i 请求资源”命令（消息）和“P_i 释放资源”组成，状态是等待中的请求命令组成的队列，队首的请求是获得资源的请求。执行**请求**命令会将请求添加到队列的末尾，执行**释放**命令会从队列中删除该命令。[7]

每个进程使用所有进程发出的命令**独立地**模拟状态机的执行。由于所有进程都根据时间戳（使用全序关系 $\Rightarrow$）对命令排序，因此可以实现同步，且每个进程都使用相同的命令序列。**当一个进程收到所有其它进程发出的时间戳都小于或等于 T 的命令后，就可以执行时间戳为 T 的命令了**。具体的算法是显而易见的，我们不再赘述。

这种方法允许在分布系统中实现任意所需的多进程同步。但是，算法需要所有进程都参与。一个进程必须知道其它进程发出的所有命令，因此单个进程的失效（failure）将使任何其它进程都无法执行状态机命令，从而使系统停止运行。

失效问题很棘手，详细地讨论超出了本文的范围。我们可以观察到，失效的概念仅在物理时间的背景下才有意义。没有物理时间，就无法区分失效的进程和暂停的进程。只有等待响应的（物理）时间太久了，用户才能知道系统“崩溃”了〔译注：逻辑时间的值不能反映物理的时间长短，只反映事件发生的数量。**仅使用逻辑时钟是无法实现心跳机制的**。物理时间是必不可少的：时钟系统不必是精确同步的，但至少要有超时机制〕。在 [3] 中描述了一种在一个进程失效或通信失效的情况下仍然可行的方法。

异常行为

我们的资源共享算法根据全序 $\Rightarrow$ 对请求排序。这允许以下类型的“异常行为”。考虑一个全国范围的互连计算机系统。假设某人在计算机 **A** 上发出请求 A，然后打电话给另一个城市的朋友，让他在另一台计算机 **B** 上发出请求 B。这很可能使请求 B 的时间戳较早〔译注：逻辑时间戳〕，排在了请求 A 之前。发生这种情况是因为系统无法知道 A 实际在 B 之前，因为该“提前”信息基于系统之外的消息。

让我们更仔细地研究这个问题的根源。令 $\mathcal{S}$ 为所有系统事件的集合。我们考虑另一个事件集合 $\underline{\mathcal{S}}$，其中包含 $\mathcal{S}$ 中的事件，以及所有相关的外部事件，例如本例中的打电话事件。令 $\to$ 表示 $\underline{\mathcal{S}}$ 中的“先于发生”关系。在这个例子中，有 A $\to$ B，但 A $\not\to$ B。显然，若仅基于 $\mathcal{S}$ 中的事件，不以任何方式将这些事件与 $\underline{\mathcal{S}}$ 中的其它事件相关联，则没有算法能保证请求 A 排在 B 之前。

有两种方法可以避免这种异常行为。第一种方法是在系统中显式引入有关顺序 $\to$ 的必要信息。在我们的示例中，发出请求 A 的人可以从系统接收该请求的时间戳 T_a。在发出请求 B 时，他的朋友要给 B 指定晚于 T_a 的时间戳。这让用户来负责避免异常行为。

第二种方法是构建满足以下条件的时钟系统。

强时钟条件：对于 $\mathcal{S}$ 中的任意事件 a，b：若 A $\to$ B，则 $C\langle a\rangle < C\langle b\rangle$。

这比普通的“**时钟条件**”强，是因为 $\to$ 比 $\to$ 更强。我们的逻辑时钟通常不能满足这个条件。

令 $\underline{\mathcal{S}}$ 表示物理时空中的一组“真实”事件，并令 $\to$ 表示由**狭义相对论**定义的事件偏序。**宇宙的一个奇妙之处，是可以构造一个物理时钟系统，该系统内的时钟彼此完全独立地运行，但仍能满足“强时钟条件”**。因此，我们可以使用物理时钟来消除上述异常行为。现在我们将注意力转向此类时钟。

物理时钟

我们在时空图中引入物理时间坐标，令 $C_i(t)$ 表示时钟 C_i 在物理时刻 t 的读数。[8] 为了数学上的方便，我们假设时钟连续运行，而不是离散的“滴答”。（离散时钟可认为是最多有 1/2 个“滴答”读数误差的连

[7] 如果进程不是严格地交替执行请求和释放命令，那么执行**释放**命令可能会从队列中删除零个、一个或多个请求。

[8] 我们假设为牛顿时空。若时钟的相对运动或重力作用不可忽略，则必须通过时间坐标系变换，才能从物理的时钟得到 $C_i(t)$。

续时钟）。更确切地说，我们假定是 $C_i(t)$ 是关于 t 的连续可微函数，仅在时钟**对时（reset）**的点有孤立的不连续跳变。$\mathrm{d}C_i(t)/\mathrm{d}t$ 表示时钟在时刻 t 的运行速率。

为了使时钟 $C_i(t)$ 成为真正的物理时钟，它必须以接近正确的速率运行。也就是说，对于所有 t，我们都必须具有 $\mathrm{d}C_i(t)/\mathrm{d}t \approx 1$。更准确地说，我们将假定满足以下条件：

PC1 存在一个常数 $\kappa \ll 1$，对于所有 i：$|\mathrm{d}C_i(t)/\mathrm{d}t - 1| < \kappa$。

对于典型的晶振时钟，$\kappa \leq 10^{-6}$〔译注：PC1 讨论的是时钟“频率的相对偏差”。10^{-6} 也记为 ppm（part per million），即百万分之一。通常计算机、手机和网络设备中晶振的频率相对偏差为几个至几十个 ppm，与此处 $\kappa \leq 10^{-6}$ 有出入，但不影响下文〕。

仅靠时钟各自以正确的速率运行还不够。它们必须保持同步，使对任意 i，j 和 t，都有 $C_i(t) \approx C_j(t)$。更准确地说，必须有一个足够小的常数 ϵ，满足以下条件：

PC2 对于所有 i，j：$|C_i(t) - C_j(t)| < \epsilon$。

如果我们认为图 2 中的垂直距离表示物理时间，那么 **PC2** 要求每个滴答线段高度的变化量小于 ϵ。

由于两个不同的时钟永远不会以完全相同的速率运行，导致它们之间的偏差会越来越大。因此，我们必须设计一种算法来确保 **PC2** 始终成立。但是，首先让我们看看为了防止异常行为，κ 和 ϵ 需要小到何种程度。我们要确保相关物理事件所在的系统 $\mathcal{S}$ 满足**强时钟条件**。假设时钟已经符合了常规的**时钟条件**，那么我们只需考虑 $\mathcal{S}$ 中的两个事件 a 和 b，当 $a \nrightarrow b$ 这一情况，**强时钟条件**也应成立。因此，我们只需要考虑不同进程中发生的事件。

引入值 μ：若事件 a 在物理时刻 t 发生，事件 b 在另一个进程，且满足 $a \rightarrow b$，则 b 发生在物理时刻 $t+\mu$ 之后。换句话说，μ 小于进程间消息传递的最短时间。我们总是可以令 μ 等于进程之间的最短距离除以光速。但是，取决于 $\mathcal{S}$ 中消息传递的方式，可能采用更大的 μ。

为避免异常行为，我们必须确保对于任意 i，j 和 t：$C_i(t+\mu) - C_j(t) > 0$。结合 **PC1** 和 **PC2**，可以将所需的 κ 和 ϵ 的上限与 μ 的值联系如下。我们假设时钟**对时（reset）**操作始终将其向前拨，绝不向后拨（向后拨可能导致违反 **C1**）。由 **PC1** 可知，$C_i(t+\mu) - C_i(t) > (1-\kappa)\mu$。由 **PC2**，易知要使得 $C_i(t+\mu) - C_j(t) > 0$，则要求下面的不等式成立：

$$\epsilon/(1-\kappa) \leq \mu$$

该不等式结合 **PC1** 和 **PC2**，意味着不再有异常行为。

现在我们描述保证 **PC2** 成立的算法。令 m 是物理时刻 t 发出，并在时刻 t' 接收到的消息。我们定义 $\nu_m = t' - t$，是消息 m 的**总延迟**。当然，该延迟对于接收 m 的进程是未知的。但是，我们假定接收进程知道**最小延迟** $\mu_m \geq 0$，其中 $\mu_m \leq \nu_m$。我们称 $\xi_m = \nu_m - \mu_m$ 为消息的**不定（unpredictable）延迟**。

现在，我们为物理时钟修改规则 **IR1** 和 **IR2** 如下：

IR1′ 对于每个 i，若 P_i 没有在物理时刻 t 接收消息，则 C_i 在 t 可微，且 $\mathrm{d}C_i(t)/\mathrm{d}t > 0$。

IR2′ (a) 若 P_i 在物理时刻 t 发送消息 m，则 m 的时间戳 $T_m = C_i(t)$。

IR2′ (b) 在时刻 t' 接收到消息 m 后，进程 P_j 将 $C_j(t')$ 设置为 $\max(C_j(t'-0),\ T_m + \mu_m)$。[9]

尽管规则是用物理时间形式化描述的，但是进程仅需知道其自身的时钟读数以及所接收消息的时间戳。为了数学上的方便，我们假设每个事件都在准确的物理时刻瞬时发生，并且同一进程中的不同事件在不同时刻发生。这两条规则是规则 **IR1** 和 **IR2** 的特化，从而使我们的时钟系统满足**时钟条件**。现实事件持续有限时间这一事实不会给实现算法带来任何困难。在实现中唯一真正关心的是确保离散时钟滴答得足够频繁，以使 **C1** 成立。

现在我们证明该**时钟同步算法**可满足条件 **PC2**。我们假设进程系统由有向图描述，其中从进程 P_i 到进程 P_j 的边（arc）代表一条通信线路，P_i 经由该边直接向 P_j 发送消息。若对于任意 t，P_i 在物理时刻

[9] $C_j(t'-0) = \lim_{\delta \to 0} C_j(t' - |\delta|)$

t 和 $t+\tau$ 之间至少向 P_j 发送一条消息，则称每 τ 秒经该边发送一条消息。对于任意两个不同的进程 P_j，P_k，若从 P_j 到 P_k 的一条路径最多有 d 条边，则最小的 d 称为有向图的**直径**。

除了使 **PC2** 成立外，以下定理还给出了系统首次启动时，同步时钟所耗时间的上界。

定理：假设一个直径为 d 的进程的强连通图始终符合规则 **IR1′** 和 **IR2′**。并假设对于任意消息 m，存在常数 μ，使 $\mu_m \leq \mu$，且对于所有 $t \geq t_0$：(a) 条件 **PC1** 成立；(b) 若存在常数 τ 和 ξ，使得每 τ 秒内，在每条边上发送的消息的不定延迟都小于 ξ，则条件 **PC2** 也成立。假设 $\mu + \xi \ll \tau$，对所有 $t \gtrsim t_0 + \tau d$，有 $\epsilon \approx d(2\kappa\tau + \xi)$。

该定理的证明相当困难，请见附录。关于同步物理时钟的问题，已经有很多研究工作了。我们请读者参考 [4] 以了解该主题。文献中有用于估计消息延迟 μ_m 和调整时钟频率 $\mathrm{d}C_i/\mathrm{d}t$（对允许调整该值的时钟）的方法。但是，时钟永远不能回拨的要求使我们的情况与先前研究的情况有所不同，我们相信这个定理是一个创新成果。

结论

我们已经看到，“先于发生”的概念定义了分布式多进程系统中事件的不变偏序。我们描述了一种用于将该偏序扩展为某种程度上任意的全序的算法，并说明了如何使用此全序来解决一种简单的互斥问题。未来将展示如何扩展这种方法来解决任意互斥问题。

该算法定义的全序在某种程度上是任意的。若它与系统用户所感知的顺序不一致，则可能会产生异常行为。可以使用正确同步的物理时钟来防止这种异常行为。我们的定理给出了物理时钟同步偏差的上界。

在分布系统中，重要的是要意识到事件发生的顺序只是偏序。我们认为，这种想法对理解任意多进程系统都很有用。它应该有助于人们理解多进程的基本问题，而不受问题的解决机制的影响。

致谢 使用时间戳对操作排序，以及异常行为的概念来自 Paul Johnson 和 Robert Thomas。

1976 年 3 月收稿；1977 年 10 月修订

参考文献

[1] Schwartz, J.T. *Relativity in Illustrations.* New York U. Press, New York, 1962.

[2] Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.

[3] Lamport, L. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks.* 2(1978), 95–114.

[4] Ellingson, C., and Kulpinski, R.J. Dissemination of System-time. *IEEE Trans. Comm.* Com-21, 5 (May 1973), 605–624.

附录

定理的证明

对于任意 i 和 t，定义 C_i^t 为在时刻 t 等于 C_i 并与 C_i 以相同速率运行的时钟，但 C_i^t 永不对时 (reset)。换言之，对所有 $t' \geq t$，

$$C_i^t(t') = C_i(t) + \int_t^{t'} [\mathrm{d}C_i(t)/\mathrm{d}t]\mathrm{d}t \tag{1}$$

注意到，对所有 $t' \geq t$，

$$C_i(t') \geq C_i^t(t') \tag{2}$$

记 t_0 为系统的初始时刻。假设进程 P_1 在时刻 t_1 向进程 P_2 发送一条消息，进程 P_2 在 t_2 时刻收到该消息，且不定延迟 $\leq \xi$，其中 $t_0 \leq t_1 \leq t_2$。对所有 $t \geq t_2$，我们有：

$$\begin{aligned} C_2^{t_2}(t) &\geq C_2^{t_2}(t_2) + (1-\kappa)(t-t_2) && [\text{由(1)和PC1}] \\ &\geq C_1(t_1) + \mu_m + (1-\kappa)(t-t_2) && [\text{由IR2}'\text{(b)}] \\ &= C_1(t_1) + (1-\kappa)(t-t_1) - [(t_2-t_1) - \mu_m] + \kappa(t_2-t_1) \\ &\geq C_1(t_1) + (1-\kappa)(t-t_1) - \xi \end{aligned}$$

因此，基于上述假设，对所有 $t \geq t_2$，我们有：

$$C_2^{t_2}(t) \geq C_1(t_1) + (1-\kappa)(t-t_1) - \xi \tag{3}$$

假设对 $i = 1, ..., n$，我们有 $t_i \leq t_i' \leq t_{i+1}$，$t_0 \leq t_1$，且进程 P_i 在时刻 t_i' 向进程 P_{i+1} 发送一条消息，进程 P_{i+1} 在 t_{i+1} 时刻收到该消息，不定延迟 $\leq \xi$，重复应用不等式 (3)，对 $t \geq t_{n+1}$，得到下面的结果

$$C_{n+1}^{t_{n+1}}(t) \geq C_1(t_1') + (1-\kappa)(t-t_1') - n\xi \tag{4}$$

由 PC1，IR1′ 和 IR2′，可以推导出

$$C_1(t_1') \geq C_1(t_1) + (1-\kappa)(t_1' - t_1)$$

将上式结合 (4)，并应用 (2)，我们得到，对 $t \geq t_{n+1}$

$$C_{n+1}(t) \geq C_1(t_1) + (1-\kappa)(t-t_1) - n\xi \tag{5}$$

对任意两个进程 P 和 P'，我们可以找到一个进程序列，$P = P_0$，...，$P_{n+1} = P'$，其中 $n \leq d$，通信的有向边依次从 P_i 到 P_{i+1}。由定理的假设 (b)，存在时刻 t_i，t_i'，且 $t_i' - t_i \leq \tau$，$t_{i+1} - t_i' \leq \nu$，其中 $\nu = \mu + \xi$。因此，对 $n \leq d$ 和任意 $t \geq t_1 + d(\tau + \nu)$，类似式 (5) 的不等式成立。对任意 i，j，和任意的 t，t_1，其中 $t_1 \geq t_0$ 且 $t \geq t_1 + d(\tau + \nu)$，我们有：

$$C_i(t) \geq C_j(t_1) + (1-\kappa)(t-t_1) - d\xi \tag{6}$$

令 m 为任一消息，其时间戳为 T_m，并假设它在时刻 t 发出，在 t' 时刻被收到。我们假设消息 m 也有自己的时钟 C_m，它的运行速率是恒定的，即 $C_m(t) = t_m$，$C_m(t') = t_m + \mu_m$。那么，$\mu_m \leq t' - t$ 意味着 $\mathrm{d}C_m/\mathrm{d}t \leq 1$。规则 IR2′(b) 将 $C_j(t')$ 设置为 $\max(C_j(t'-0), C_m(t'))$。因此，时钟对时操作仅是将钟的值设置为与其它某个时钟的值相等。

对任意时刻 $t_x \geq t_0 + \mu/(1-\kappa)$，令 C_x 为在时刻 t_x 读数最大的时钟。因为所有时钟的运行速率不超过 $1+\kappa$，我们有，对所有 i 和所有 $t \geq t_x$：

$$C_i(t) \leq C_x(t_x) + (1+\kappa)(t-t_x) \tag{7}$$

考虑下面两种情况：(i) C_x 是进程 P_q 的时钟 C_q；(ii) C_x 是进程 P_q 在时刻 t_1 发出的消息 m 的时钟 C_m。对第 (i) 种情况，(7) 变成了

$$C_i(t) \le C_q(t_x) + (1+\kappa)(t - t_x) \tag{8i}$$

对第 (ii) 种情况，由于 $C_m(t_1) = C_q(t_1)$，且 $\mathrm{d}C_m/\mathrm{d}t \le 1$，我们有

$$C_x(t_x) \le C_q(t_1) + (t_x - t_1)$$

因此，由式 (7) 得到

$$C_i(t) \le C_q(t_1) + (1+\kappa)(t - t_1) \tag{8ii}$$

由于 $t_x \ge t_0 + \mu/(1-\kappa)$，我们得到

$$\begin{aligned}
C_q(t_x - \mu/(1-\kappa)) &\le C_q(t_x) - \mu && [\text{由PC1}] \\
&\le C_m(t_x) - \mu && [\text{由}m\text{的选择}] \\
&\le C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && [\mu_m \le \mu,\ t_x - t_1 \le \nu_m] \\
&= T_m && [\text{由}C_m\text{的定义}] \\
&= C_q(t_1) && [\text{由IR2}'\text{(a)}]
\end{aligned}$$

因此，$C_q(t_x - \mu/(1-\kappa)) \le C_q(t_1)$，则 $t_x - t_1 \le \mu/(1-\kappa)$，且 $t_1 \ge t_0$。

对第 (i) 种情况，令 $t_1 = t_x$，结合 $(8i)$ 和 $(8ii)$，可以推导出，对任意 t, t_x，其中 $t \ge t_x \ge t_0 + \mu/(1-\kappa)$，存在进程 P_q 和时刻 t_1，其中 $t_x - \mu/(1-\kappa) \le t_1 \le t_x$，对所有 i，满足：

$$C_i(t) \le C_q(t_1) + (1+\kappa)(t - t_1) \tag{9}$$

选择 t 和 t_x，使 $t \ge t_x + d(\tau + \nu)$，结合 (6) 和 (9)，可以得出，存在 t_1 和进程 P_q，对所有 i：

$$\begin{aligned}
C_q(t_1) + (1-\kappa)(t - t_1) - d\xi &\le C_i(t) \\
&\le C_q(t_1) + (1+\kappa)(t - t_1)
\end{aligned} \tag{10}$$

令 $t = t_x + d(\tau + \nu)$，有

$$d(\tau+\nu) \le t - t_1 \le d(\tau+\nu) + \mu/(1-\kappa)$$

上式结合 (10)，有

$$\begin{aligned}
C_q(t_1) + (t - t_1) - \kappa d(\tau+\nu) - d\xi &\le C_i(t) \\
\le C_q(t_1) + (t - t_1) + \kappa[d(\tau+\nu) + \mu/(1-\kappa)]&
\end{aligned} \tag{11}$$

利用 $\kappa \ll 1$ 和 $\mu \le \nu \ll \tau$ 的假设，我们可以将 (11) 重写为下面的近似不等式。

$$\begin{aligned}
C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) &\lesssim C_i(t) \\
&\lesssim C_q(t_1) + (t - t_1) + d\kappa\tau
\end{aligned} \tag{12}$$

因为上式对所有的 i 都成立，从而有

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi)$$

上式对所有 $t \gtrsim t_0 + dt$ 都成立。 ■

对假设 $\mu + \xi \ll \tau$ 不成立的情况，证明中的的式 (11) 给出了 $|C_i(t) - C_j(t)|$ 的准确上界。证明还给出了一种简单的方法，用于快速初始化时钟，或任何原因时钟失步后重新同步：每个进程都发送一条消息，该消息由其它所有进程中继转发。该方法可以由任意进程发起。假设每个消息的不定延迟都小于 ξ，则需要不到 $2d(\mu + \xi)$ 秒就能实现同步。

我的文章（节选）

Leslie Lamport

https://lamport.azurewebsites.net/pubs/pubs.html#time-clocks

27. ***Time, Clocks and the Ordering of Events in a Distributed System***

Communications of the ACM, 21, 7 (July 1978), 558–565. 被转载到了一些合辑, 包括 *Distributed Computing: Concepts and Implementations*, McEntire et al., ed. IEEE Press, 1984.

Jim Gray 曾经告诉我，他听过对本文的两种截然不同的观点：微不足道和非常精彩。我不能与前者争论，而我不愿与后者争论。

本文缘起 Paul Johnson 和 Bob Thomas 的文章“维护复制数据库（The Maintenance of Duplicate Databases, http://www.rfc-archive.org/getrfc.php?rfc=677）”。我相信他们的文章介绍了在分布式算法中使用消息时间戳的想法。我恰好对狭义相对论有扎实深入的理解（请参阅 [5]）。这使我能够立即掌握他们所做努力的本质。狭义相对论告诉我们，时空中的事件没有一致的（invariant）全局顺序。不同的观察者可能无法就两个事件发生的先后顺序达成一致。只存在偏序，其中事件 e_1 早于事件 e_2，等价于 e_1 可以因果地影响 e_2。我意识到 Johnson 和 Thomas 算法的本质是使用时间戳来提供与因果顺序一致的事件全序。我的这个认识很巧妙。意识到这一点之后，其它的所有事情就很简单了。由于 Thomas 和 Johnson 没有准确地理解他们自己的工作，他们没有得到完全正确的算法。他们的算法允许违反因果关系的异常行为。我很快写了一条简短的笔记指出这一点，并纠正了算法〔译注：观察者与时钟所在惯性参考系的相对速度会影响观察到的时钟频率。不同惯性参考系中的观察者对发生在异地事件的同时性存在分歧；由于光速不可超越，所以因果关系总是保持的〕。

不久，我就意识到，对全局事件排序的算法可以用来实现任何分布系统。分布系统可以描述为，由多个处理器组成的网络（network of processors）实现的特殊的顺序状态机。对输入请求全局排序的能力立即产生了一种算法，该算法可通过处理器网络实现任意状态机，进而实现任意分布式系统。因此，我写了这篇关于如何实现任意分布式状态机的论文。作为示例，我使用了我能想到的最简单的分布系统示例—分布式互斥算法。

这是我的论文中最常被引用的。许多计算机科学家声称读过它。但是我很少遇到有人知道这篇论文说过关于状态机的任何事情。人们似乎认为这篇文章是关于分布系统中事件的因果关系的，或者与分布式互斥问题有关。有的人一直坚持认为，本文根本没提到状态机。我甚至不得不回头再读一遍，以使自己确信，我确实记得自己写的东西。

本文介绍了逻辑时钟的同步。之后，我决定看看它为物理时钟提供了什么样的同步。因此，我包括了一个有关物理时钟同步的定理。令我感到惊讶的是，证明竟如此困难。这也预示了 [62] 的经历。

该论文获得了 2000 年 PODC 影响力论文奖（后更名为“Edsger W. Dijkstra 分布计算奖”）。它还于 2007 年获得了 ACM SIGOPS 名人堂奖。

5. ***The Geometry of Space and Time***

未出版（1968 年左右），没有电子版本

从 1965 年至 1969 年，我在马尔伯勒学院（Marlboro College）教授数学。学院每周为公众举办讲座。每次讲座是由一名教员或由教员邀请的外部演讲者进行的。我做了一个有关相对论的讲座，后来把它写成了这本简短的著作。我没有在意将其发表。它的篇幅太短了（75 页），不够作为一本“真正的”书。而且在六十年代后期，公众对科学的兴趣也很小。我仍然认为这本专著很好地阐述了主题。不幸的是，关于广义相对论的后半部分已经过时了，因为它没有提到黑洞。虽然广义相对论方程最早的数学解就出现了黑洞，但直到六十年代末，许多物理学家才开始认真考虑黑洞可能存在并研究其性质。

62. ***Synchronizing Clocks in the Presence of Faults***

(与 Michael Melliar-Smith 合作)

Journal of the Association for Computing Machinery 32, 1 (January 1985), 52–78.

实际的拜占庭协议实现需要同步的时钟。可容忍拜占庭错误的实现需要一种可以容忍这类错误的时钟同步算法。当我到 SRI 工作后，人们普遍认为，可以让每个进程用拜占庭协议来广播其时钟值，从而同步时钟。但我从未相信会这么简单。因此，我终于尝试写下准确的时钟同步算法并证明其正确性。[46]（“拜占庭将军问题”）中两种基本的拜占庭协议算法确实推广到了时钟同步算法。此外，Melliar-Smith 设计了交互式收敛算法，该算法也包含在此文中（我记得，该算法是他对论文的主要贡献，而我编写了所有证明）。

编写证明比我预想的要困难得多（请参阅 [27]）。我尽了最大的努力使它们尽可能简短易懂。因此，当一位审稿人说“证明似乎写得很仓促，可以稍作简化”时，我很生气。在对评论的答复中，我说那个审稿人是“白痴”。后来，Nancy Lynch 承认她就是那个审稿人。那时，她已经写出了自己的时钟同步证明，并意识到了难度。

多年后，SRI 的 John Rushby 和他的同事为我的证明编写了机器验证的版本。他们只发现了几个小错误。我对此感到非常自豪。在掌握如何编写可靠的结构化证明之前（请参阅 [101] “如何编写证明”），我已经足够严谨，且受过了训练，使这些证明基本正确。