China-pub.com
下载

第3章　分布式系统设计的形式方法

这一章我们正式讨论分布式系统，介绍了几个概念如：时钟、事件、时间和状态以及两种流行的描述分布式系统的方法：时空视图和交叉视图。

3.1　模型的介绍

这一节给出了描述一个分布式系统属性常用的模型的概述。模型的作用在于精确地定义要建立或分析的系统的属性和特征并提供检验这些属性的基础。不同的模型用于说明不同的属性。以下是三个有代表性的分布式系统的模型：

- 数学函数。它由一个输入域，一个输出域和一个把输入转换为输出的规则组成。一般，数学函数不是一个算法。数学函数可以被分解，也就是说，它可以用逻辑和底层函数的结合加以说明。这个特点使得函数可以有分层结构。分层的好处在于它能够组织大量的数据并检查顶层函数和它分解的许多底层互连函数间的输入和输出的一致性。这种方法的一个限制就是它没有记忆性：给定一个输入就产生一个输出，但它不保存数据。
- 有限状态自动机（FSM）。FSM是一系列输入、一系列输出、一系列状态、一个初始状态和一对函数，这对函数用于指定作为给定输入的结果的输出和状态转换。这个模型对于说明数据处理很理想因为数据处理机的输入、输出可以和 FSM的输入、输出一一对应，存储器中的数据可以和FSM的状态对应，代码可以和状态的转换对应。它的限制在于：首先，FSM固有地串行化所有基础并发；其次，这个模型明确假设一个输入的所有处理在下一个输入到达之前完成。我们将在3.1.1小节详细讨论这个模型。
- 图模型。计算的图模型是一个由顶点（或节点）和边（弧或连接）组成的有向图，它用于说明控制流和数据流。图中的每个节点代表一个处理步骤，它有一个或多个输入弧，一个或多个输出弧。图模型的局限性在于它没有体现“状态”的概念，“状态”是从对一个输入数据集的处理中保存下来，用于处理后来的输入数据。佩特里网（Petri net）是一种特殊类型的图模型，我们将在本章的后面讨论。

3.1.1　状态机模型

正如第1章中定义的一样，分布式系统是一系列的处理单元 $PE=\{PE_1, PE_2, \ldots, PE_n\}$，这些$PE$不共享存储器并且只通过消息传递通信。从逻辑的层次看，分布式系统是一系列协同合作的进程$P=\{P_1, P_2, \ldots, P_n\}$。为了讨论的简单，我们假设每个进程和一个 PE相关联，虽然事实上，几个进程可能在同一个PE上运行。

从一个进程（或一个PE）的观点看，系统只有两部分：时钟和局部状态。时钟可能是物理的或是逻辑的（这将在以后的小节中讨论）。局部状态是进程中变量的集合。局部状态只因事件

China-pub.com 下载

而改变。进程执行两种类型的事件：内部的和外部的。外部事件包括发送事件和接收事件。

- 内部事件只改变本地进程（或 *PE*）的状态。
- 发送事件如send 消息列表 to 目的地，把“消息列表”发送到“目的地”进程。
- 接收事件如receive 消息列表from 信源，从“信源”进程接收“消息列表”。

一个分布式系统也可以用有限状态自动机（FSM）[12]来说明，它是由一个可能命令的集合 C 和一个可能全局状态的集合 S组成的。每个全局状态是局部状态的集合。全局状态的精确定义包括局部状态和通道的状态，这将在以后讨论。函数 e（事件）定义如下：

$$e: C \times S \rightarrow S$$

$s \xrightarrow{c} s'$ 代表C中的命令c在状态s下的执行结果：导致系统状态从 s改变到s'。

一个系统是确定性的当且仅当对任意的 s　S和c　C，最多有一个 s'使得 $s \xrightarrow{c} s'$，否则系统就是非确定性的。图3-1表示了一个确定的系统，其中 $C=\{write, read\}$。$write_i$和$write_j$代表不同的写命令。然而，许多分布式系统是非确定的。状态 s'从状态s可到达当且仅当 $s \xrightarrow{*} s'$，其中$*$是关系“→”的传递闭包。关于分布式系统的状态自动机的系统阐述可以参考[5]。

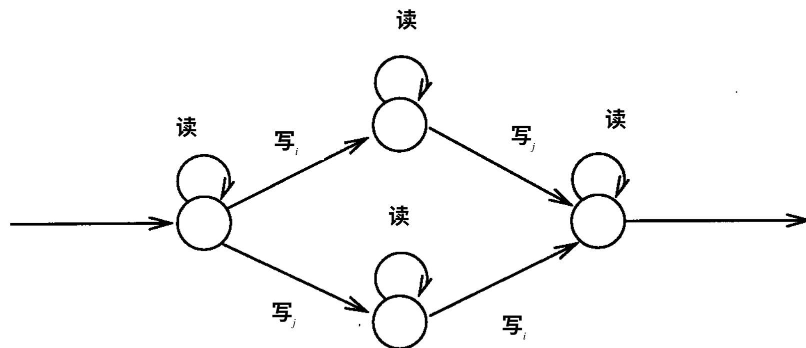


图3-1　一个确定的系统

在这一节中，我们大略地介绍了时钟、事件、状态（局部的和全局的）和时间的概念。这些概念将在以后的小节中详细讨论。

3.1.2　佩特里网

佩特里网[1、16]是研究分布式计算机系统行为的强有力工具。作为一个建模工具，佩特里网可用于对系统的静态属性和动态属性进行建模；作为一个面向图形的规范说明工具，它是加强用户和专家之间的交互以达到相互理解的最好工具之一。

佩特里网是一个5元组$C=(P, T, I, O, u)$，其中$P=\{p_1, p_2, \cdots, p_n\}$是一个有限的位置集，$n>0$，$T=\{t_1, t_2, \ldots, t_m\}$是一个有限的变迁集，$m>0$。$T$和$P$不相交（$P \cap T=\Phi$）。$I: T \rightarrow P^{\infty}$是一个输入函数，把变迁映射到位置集的子集。$O: T \rightarrow P^{\infty}$是输出函数。向量$u=(u_1, u_2, \ldots, u_n)$给出了每个位置的令牌数，称做标志，也就是说，位置 P_i有个u_i令牌。一个佩特里网的例子如下。

实例3.1　大学学期系统[25]。

$$C=(P, T, I, O, u)$$

$$P=\{p_1, p_2, p_3, p_4\}$$

其中，p_1=秋季学期，p_2=春季学期，p_3=夏季学期，p_4=非夏季学期

$$T=\{t_1, t_2, t_3\}$$

其中，t_1=春季学期的开学，t_1=夏季学期的开学，t_1=秋季学期的开学

$$u=\{1, 0, 0, 1\}$$

$$O(t_1)=\{p_2\} \qquad I(t_1)=\{p_1\}$$

$$O(t_2)=\{p_3\} \qquad I(t_2)=\{p_2, p_4\}$$

$$O(t_3)=\{p_1, p_4\} \qquad I(t_3)=\{p_3\}$$

佩特里网可以用图来表示，圆圈代表位置，横线代表变迁。输入和输出函数分别用从位置到变迁的有向弧和从变迁到位置的有向弧表示。每个令牌用一个圆点表示。实例 3.1的佩特里网图在图3-2中表示。在这个例子中，大学学期系统包含三个学期：秋季，春季和夏季，每个学期用一个位置表示。特别地，位置 p_1代表秋季学期，位置 p_2代表春季学期，位置 p_3代表夏季学期，位置p_4代表非夏季学期。变迁 t_1代表春季学期的开学，变迁 t_2代表夏季学期的开学，变迁 t_3代表秋季学期的开学。佩特里网中变迁的执行可以通过令牌的移动来表示。令牌通过变迁的触发移动。当一个变迁有效时，也就是说，它的每个输入位置至少有一个令牌时，该变迁可以触发，即从它的每个输入位置中移走一个令牌，在它的每个输出位置中放入一个令牌。不同的令牌分布模式代表了不同的系统状态。在图 3-3中，大学学期系统的例子是三个状态的重复，初始状态是秋季学期，其中，每个状态用一个 4元组（P_1中的令牌数，P_2中的令牌数，P_3中的令牌数，P_4中的令牌数）表示。

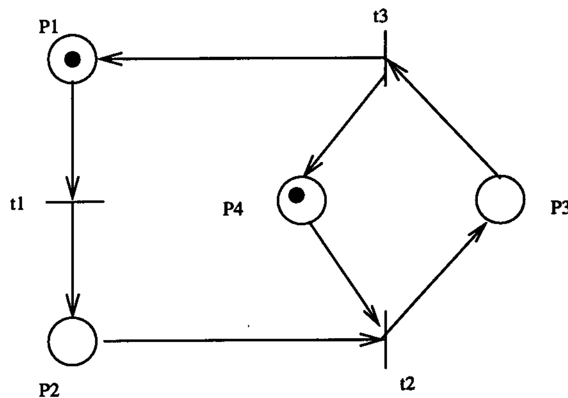


图3-2　大学学期系统的佩特里网图

图3-3中的图也叫做可达性树。它是一棵树，树的节点代表佩特里网的标志，弧线代表由变迁触发的可能的状态改变。注意，当一个节点（状态）重复自身时，它变为一个终止节点。另一种选择就是增加一条弧，指向产生循环的节点。在这种情况下，可达性树就叫做可达性图。在大学学期系统中，删去秋季学期的第二个节点，增加一条从夏季学期节点到第一个秋季学期

China-pub.com 下载

节点的弧。

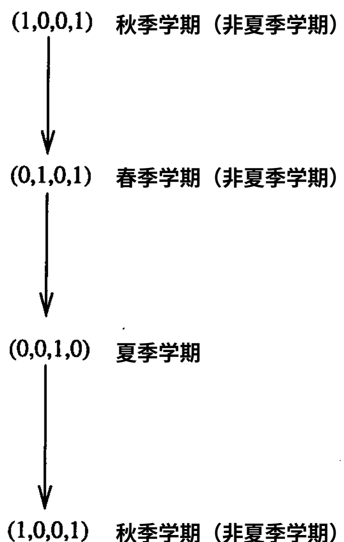


图3-3 大学学期系统的状态图

佩特里网可用于精确地说明顺序、选择、并发和同步的概念。图 3-4使用佩特里网表示了这些机制。图3-4a是一个顺序语句S_1；S_2，图3-4b是一个条件语句：

$$[\ C \rightarrow S_1 \square \neg C \rightarrow S_2\ ]$$

图3-4c是一个重复语句：

$$*[\ C \rightarrow S_1\ ]$$

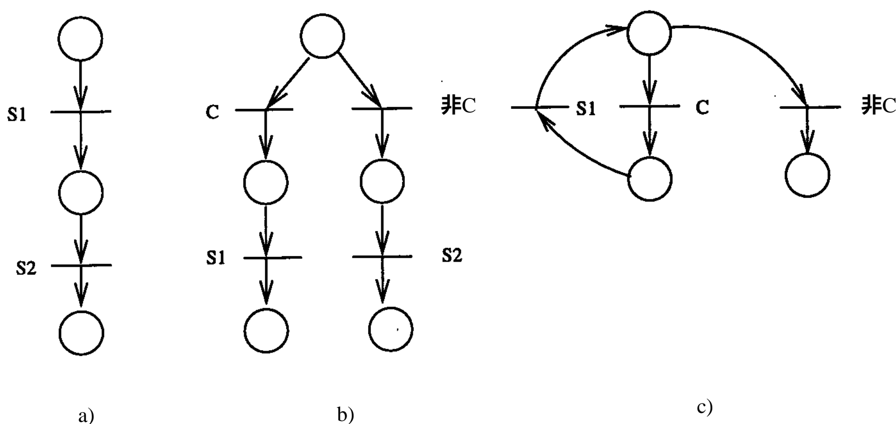


图3-4 佩特里网实例：a）顺序语句，b）条件语句，c）重复语句

注意，顺序、条件和重复语句是三种基本的控制结构。佩特里网也能说明不明确的（非确定的）行为，也就是说，当有一个令牌的位置有多个可触发的出去变迁时。这种情况下，任意选择一个变迁执行。

China-pub.com
下载

通过定义一个佩特里网处理机，佩特里网能自动生成一个分布式系统的模拟。该设备检查所有变迁的状态，选择满足触发条件的其中一个，再根据相应的规则移动令牌。重复该步骤直到不再有前提得到满足。

Wu[26、27]提出了一个高级扩展佩特里网用于对一个有一系列进程的分布式系统建模。在这个模型中，每个进程用一个子网来模拟。进程的每个内部动作直接用一个佩特里网表示。分布式系统的结构被组织成几个层次。最高层只表示了进程间通信（一个外部事件）。所有的内部事件在底层规范中详细说明。底层规范可以通过替换模型中的每个位置（变迁）为一个更详细的佩特里网来表示，这种过程称为细化，即用一个开始和结束于某个位置（变迁）的子网替代该位置（变迁）。

进程间通信的佩特里网模型表示在图 3-5，其中不同进程的局部状态（位置）用虚线分开。粗的横线代表使用不同触发规则的变迁。触发规则取决于不同的实现，有三种可能的实现进程间通信的方法：同步的（例如在 CSP中）、异步的（例如在 DCDL中）和缓冲的。注意，异步通信是一种特殊类型的缓冲通信，它有极大的缓冲区。每种实现方法相应的佩特里网细化在图 3-6 中表示，其中位置里的数字 n表示该位置中的令牌数。

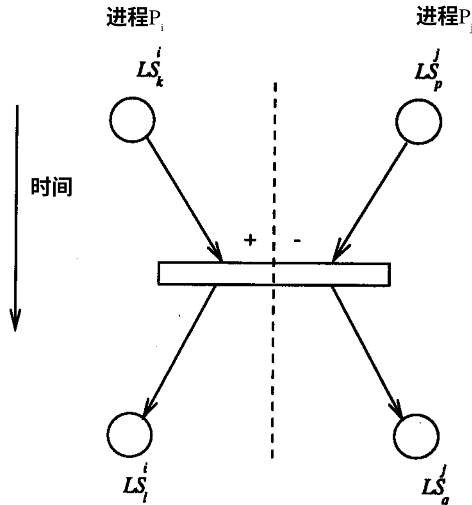


图3-5 进程间通信的佩特里网模型，其中进程P_i是发送方(+sign)，进程P_j是接收方(－sign)

分布式系统的许多有意义的属性都可以用佩特里网来定义和验证。例如，可以通过分析一个佩特里网判定一个分布式系统是否总是正常终止；或是否存在死锁的可能性，比如，佩特里网可能会进入一个没有任何变迁可触发的状态；或是否存在活锁的可能性，比如，佩特里网在一个无限循环中兜圈子。

佩特里网还可以用状态模型表示。在 [27]中，Wu把佩特里网规范说明图中的每个位置定义为进程中的一个状态。整个系统在某个给定时间的状态 GS_x定义为向量（LS_x^1 ，…， LS_x^n ），其中LS_x^y 是进程P_y在全局状态x下的局部状态（$1 \leq y \leq n$），n是进程总数。沿着时间的进程局部状态定义了佩特里网规范说明图中的偏序关系。例如，图 3-5中的局部状态定义的偏序关系如下: $LS_k^i < LS_l^i$， $LS_k^i < LS_q^j$， $LS_p^j < LS_l^i$， $LS_p^j < LS_q^j$, $LS_l^i < LS_q^j$, $LS_k^i < LS_p^j$ ，其中“ = ”表示两个有相

China-pub.com 下载

同序的局部状态。

根据这个定义，系统可以用一系列的全局状态来表示：

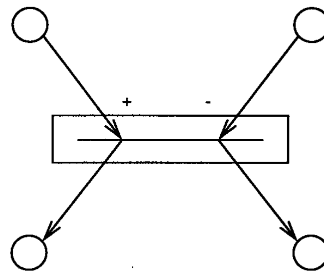
$GS_0 \xrightarrow{t_1} GS_1 \xrightarrow{t_2} GS_2 \ldots$ 其中 $GS_x < GS_{x+1}$

如果 $GS_x = (LS_x^1, LS_x^2, \ldots, LS_x^n)$

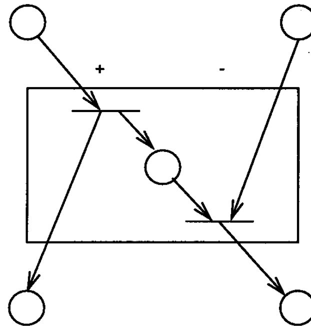
并且 $GS_{x+1} = (LS_{x+1}^1, LS_{x+1}^2, \ldots, LS_{x+1}^n)$,

则 $GS_x < GS_{x+1}$ $\quad {}_y[LS_x^y < LS_{x+1}^y) \quad (LS_x^y = LS_{x+1}^y)] \quad {}_y[LS_x^y < LS_{x+1}^y]$

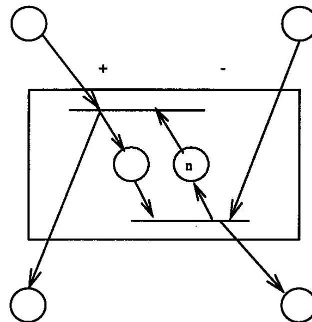
$(1 \leq y \leq n)$



a)



b)



c)

图3-6 不同类型的进程间通信的佩特里网：a）同步的，b）异步的，c）缓冲的

China-pub.com
下载

每个t_i是一个进程间通信变迁，GS_x是进程间通信t_i之后的全局状态。这种称为行为有序树的序列是多进程系统的扩展可达性树。在行为有序树中，一个状态是从它的父状态通过一个或多个变迁达到的。如果是多个变迁的话，新状态是当一次只触发一个变迁时所能到达的可能的状态的集合。

尽管有它的优点，使用佩特里网描述一个分布式系统还是有两个基本限制。首先，就像有限状态自动机模型，佩特里网是平面型的，也就是说，佩特里网本质上是单层次的。但是，有一些利用子网（如以上定义）概念的独立的方法，它们使用子网层次的概念来处理大型而复杂的分布式系统。例如，前面讨论的佩特里网模型就是在进程间的层次上描述一个系统，可以细化每个节点（圆圈）来描述进程内部的细节，即每个进程内的局部事件。另一个重要的限制是佩特里网只说明了控制流，而没有说明数据流。虽然变迁的条件可以根据数据值指定，但改变数据的值的语义并不是佩特里网固有的一部分。现在有一些这方面的尝试试图解决这个问题[10、29]，但一些细节还没有完全解决。

3.2　因果相关事件

分布式系统中事件之间的因果关系，更正式的讲，事件之间的因果优先关系，是一个对分布式计算进行推理、分析并得出结论的强有力概念。

3.2.1　发生在先关系

在集中式系统中，总是可以确定两个不同进程中的两个事件的发生顺序，因为集中式系统有单一的公共存储器和时钟。然而，在分布式系统中，没有公共的存储器和时钟。所以，有时不能确定两个事件中的哪个先发生。发生在先关系[12]是定义在分布式系统中的事件上的一个偏序关系。该定义是基于这样一个事实，即进程是顺序的而且在单个进程中执行的所有事件是全序的。同样，一个消息（或消息列表）只能在它被发送之后才能被接收。

发生在先关系（用符号→表示）的定义如下：

1．如果a和b是同一个进程中的事件并且a在b之前被执行，则$a \rightarrow b$。

2．如果a是某个进程发送消息的事件，b是另外一个进程接收该消息的事件，则$a \rightarrow b$。

3．如果$a \rightarrow b$且$b \rightarrow c$，则$a \rightarrow c$。

一般，$a \not\rightarrow a$对任何事件a都成立。这说明→是一个非自反的偏序。

3.2.2　时空视图

发生在先关系的定义可以通过时空视图最好地说明，水平方向代表空间，垂直方向代表时间，带标志的垂直线代表进程，带标志的点代表事件，带箭头的线代表消息。

实例3.2　图3-7表示有三个进程P_1、P_2、P_3的分布式系统。每个进程有四个带标志的事件。事件之间的发生在先关系如下：

$$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3$$
$$b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3$$
$$c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$$

China-pub.com 下载

$$a_0 \rightarrow b_3$$

$$b_1 \rightarrow a_3,\ b_2 \rightarrow c_1,\ b_0 \rightarrow c_2$$

我们还可以从$b_1 \rightarrow b_2$、$b_2 \rightarrow c_1$、$c_1 \rightarrow c_2$推出$b_1 \rightarrow c_2$。如果$a \rightarrow b$或$b \rightarrow a$，则事件a和b是因果关联的。如果两个不同的事件a和b，$a \not\rightarrow b$并且$b \not\rightarrow a$，则称事件a和b是并发的。对于图3-7所示的例子，事件a_2和c_0是并发的，即$a_2 \parallel c_0$。

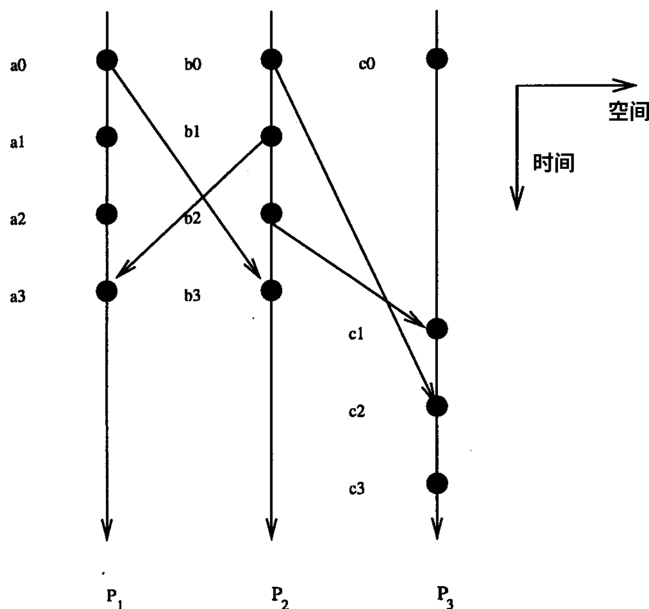


图3-7　一个分布式系统的时空视图

3.2.3　交叉视图

时空视图对于表示发生在先关系是十分有用的。然而，虽然我们可能把一个分布式系统的执行看成是一系列的事件，经验表明我们更希望根据状态来推理一个系统。时空视图偏重于事件。要关于状态进行推理，我们就需要交叉视图。在交叉视图中，我们假设事件是全序的。如果两个事件不互相影响，也就是说，如果它们不满足发生在先关系，我们就可以认为它们以任意一种顺序发生。交叉视图模型基于以下事实：因为有一个全局时间的概念，所以系统中的事件总是按照某种确定的顺序发生。例如，我们可以根据事件的开始时间（或结束时间）进行排序。因为没有绝对精确的时钟，所以我们无法确知这种全序关系，但既然它一定存在，我们就可以假定它的存在。

我们在一系列进程和事件上表示全序关系　　，唯一的限制是　　必须和→保持一致。例如，$a_0\, a_1\, c_0\, b_0\, b_1\, b_2\, b_3\, c_1\, c_2\, a_1\ \ a_2\, c_3\, a_3$是图3-7例子的一个全序。

注意，对应于每个交叉视图，有唯一确定的时空视图；但对应于一个时空视图，则有多个交叉视图。有许多方法可以使偏序完善为全序。交叉视图中的全序包含了额外的信息，比如，

China-pub.com 下载

不同进程的事件之间的序。

3.3　全局状态

全局状态由局部状态集和通信通道的状态集组成。注意，这是全局状态的细化定义，原来全局状态只是定义为局部状态集。仅仅定义为局部状态集可能产生不完全的系统视图。考虑一个银行系统，它有两个支行A和B。假设我们从账户A（原来有$500）转账$100到账户B（原来有$200）。如果两个账户都在它们的局部状态发生变化后才记录下它们的变化，那么当 $100还在传送途中时，也就是说，当$100还在连接账户A和B的通道上时，账户A的余额为$400，账户$B$的余额为$200。注意，以上情况将产生不完全的系统视图，但它仍然是（弱）一致的。然而，如果我们假设账户A在传送之前记录它的状态而账户 B在传送完成后记录它的状态，那么总的余额就是$500＋$300＝$800。显然，这是一个不一致的全局状态。

一般，在一致的全局状态中，通信通道的状态应为在发送方记录状态之前沿着该通道发送的消息序列扣除掉在接收方记录状态之前沿着该通道接收到的消息序列。注意，要记录通道的状态以保证以上规则是很难的。另一种记录全局状态的选择是不使用通道状态。记录下来的状态可能是一致的也可能是不一致的。一种无需通道状态就可获取一致状态的算法将在 3.3.3小节讨论。

基于传送的假定，通道可以分为：FIFO（先进先出）、因果顺序传送和随机顺序传送。如果一个通道保持通过它发送的消息的顺序，就被称做是 FIFO。因果顺序传送[22]的定义如下：假设进程P_1和P_2分别在事件e_1和e_2中给进程P_3发送消息m_1和m_2。如果e_1在e_2之前发生，则进程P_3在接收m_2之前接收m_1。对于随机顺序传送的通道没有限制。在没有明确说明给定通道的类型时，我们都假设它是FIFO通道。类似地，对于每个进程一般有以下几个假设：(a) 正常情况下，每个进程有一个唯一的id，并且每个进程知道它自己id和其他进程的id。(b) 每个进程知道它们的邻居包括它们的id。

3.3.1　时空视图中的全局状态

为了定义时空视图中的全局状态，我们定义了时间片的概念。直观上，时间片反映了某个时刻的系统。然而，由于我们没有假定任何类型的时钟，我们必须根据事件来定义时间片这个概念。基本想法是把时间片看做是事件的一个划分：发生在时间片“之前”（表示为→）的事件和发生在时间片之后的事件。我们正式定义时间片为事件的前集 E：如果b　E并且$a \to b$，则a　E。如果E　E'，则时间片E比时间片E'早。不像真正的时钟时间，时间片不是全序的。

给定一个时间片E，我们如下定义和该时间片相关的全局状态GS（E）：如果E中没有进程P_i的事件，则LS_i就是P_i的初始状态；否则，LS_i定义为P_i在E中最后一个事件的最终状态。对某个通道c，我们定义c在GS（E）中的状态为一个消息序列，这些消息是所有被E中的某个事件发送但被一个不在E中的事件接收的所有消息。

实例3.3　考虑一个由三个支行A、B和C通过单向通道连网组成的分布式银行系统(见图 3-8)。

China-pub.com 下载

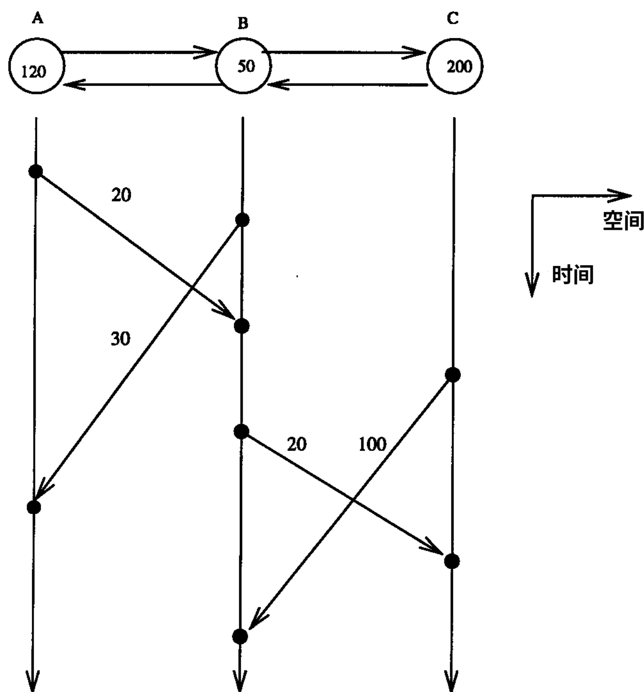


图3-8 银行系统的网络实例

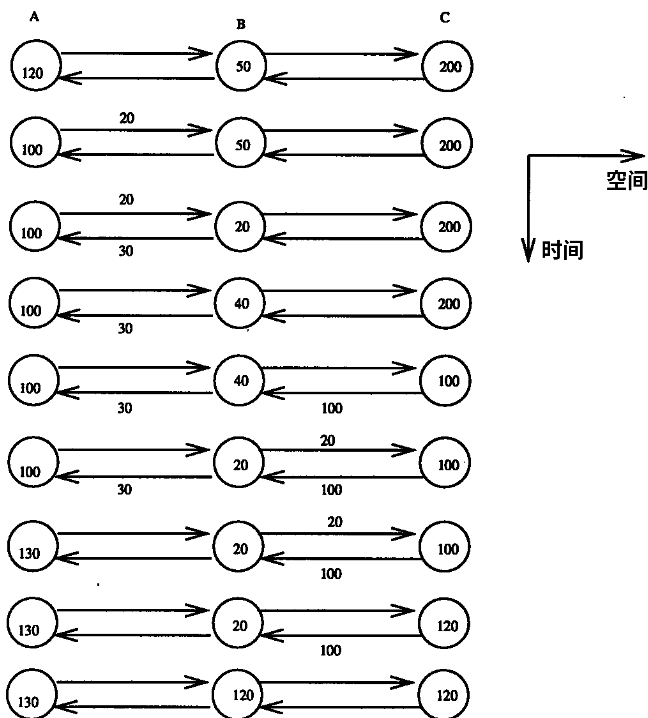


图3-9 图3-8中的系统的全局状态序列

China-pub.com 下载

图3-8表示了银行系统的事件（事务）的时空视图。图 3-8中的系统的全局状态序列在图 3-9中表示。注意，全局状态是基于图3-8中事件的位置（时刻）得到的（每个触发的事件着重表示）。在这个例子中，全局状态的改变是由一个外部事件触发的：发送或接收。在图 3-8的例子中，有四对发送和接收事件，对应于八个外部事件。

3.3.2　全局状态：一个形式定义

令LS_i为进程P_i的局部状态，则全局状态$GS=$（LS_1，LS_2，…，LS_n）。这里的全局状态仅由局部状态定义，也就是说，不包括每个通道的状态。所以状态可能是一致的也可能是不一致的。为了定义一致的状态，我们需要以下两个概念，它们都表示为集合，其中 $s(m)$（和$r(m)$）是消息m的发送（和接收）事件。

传送中：$transit(LS_i, LS_j)=\{ m|s\quad m) \quad (LS_i \wedge r(m) \quad LS_j\}$

不一致的：$inconsistent(LS_i, LS_j)=\{ m|s(m) \quad LS_i \wedge r(m) \quad LS_j \}$

集合$transit$包括了进程P_i和P_j之间的通道上的所有消息。集合 $inconsistent$包括了所有接收事件记录在P_j而发送事件没记录在P_i的消息。

全局状态GS是一致的当且仅当

$$i, \quad j, \ inconsistent(LS_i, LS_j)=\Phi$$

全局状态GS是非传送中的当且仅当

$$i, \quad j, \ transit\ (LS_i, LS_j)=\Phi$$

如果一个全局状态是一致的并且是非传送中的，那么它就是强一致的，也就是说，局部状态集是一致的并且没有消息正在传送中。

时间片的概念还可以通过一种图形化称之为切割（ cut）的表示法来得到。分布式计算的切割是一个集合$C=\{c_1, c_2, \ldots, c_n\}$，其中$c_i$是进程$P_i$中的切割事件，也就是对应于切割的一个局部状态。设$e_i$是$P_i$的一个事件。一个切割$C$是一致的当且仅当

$$P_i, \quad P_j, \ \nexists e_i, \ \nexists e_j \ (e_i \rightarrow e_j) \wedge (e_j \rightarrow c_j) \wedge (e_i \nrightarrow c_i)$$

其中，c_i 和c_j在C中。符号 ∄ 代表“不存在”。

实际上，一个切割 C是一致的当且仅当没有两个切割事件是因果关联的。一个切割可以图形化地表示为由一条点线连接的切割事件集。如图 3-10，点线可能“跨越”也可能不“跨越”通信线。在图 3-10a和3-10b中，切割线都没有跨越通信线，所以对应的状态是一致的。在图3-10c中，虽然切割线跨越了通信线，但两个切割事件不是因果关联的，所以它不会导致切割的不一致。实际上，这种情况对应于正在传送中的消息——一个一致的但非强一致的状态。在图 3-10d中，两个切割事件是因果关联的，所以它们形成了一个不一致的切割。在图3-9中，初始状态和最后状态是强一致的全局状态，其他的是一致的（但不是强一致的）。

China-pub.com 下载

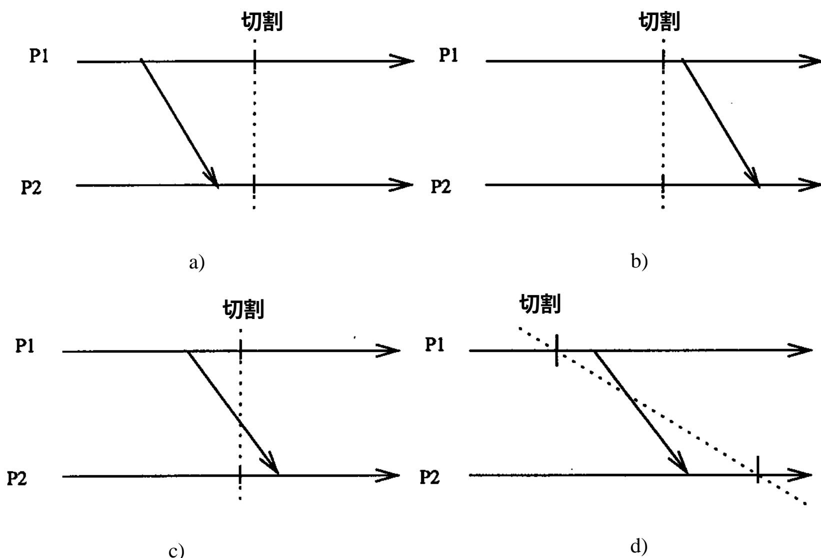


图3-10 四种类型的跨越消息传送线的切割

3.3.3 全局状态的“快照”

Chandy和Lamport[6]提出了一个简单的分布式算法用于捕获一致的全局状态，也叫做全局状态的快照。它假设所有的通道都是 FIFO并且有一套标志沿着这些通道传送。在每个节点上有一个进程在运行。

发送方 *P*的规则：

[*P*记录它的局部状态

|| *P*在所有还没发送过标志的通道上发送一个标志

]

接收方 *Q*的规则：

/*沿着通道 *chan*收到一个标志*/

[*Q*还没有记录它的状态 →

  [记录通道 *chan*的状态为一个空序列并遵循“发送方的规则”

  ]

□ *Q*已经记录了它的状态 →

  [记录通道 *chan*的状态为沿着通道 *chan*接收到的消息序列，这些消息序列是在最后一次记录状态之后、接收到标志之前收到的

  ]

]

标志的作用是“清除”对应的通道，虽然发送一个标志后仍然可以沿着一个通道发送正常的消息。如果一个节点上的进程在它的每个输入通道上已经执行并接收到一个标志消息的话，

就称它已经完成它的那部分算法。很容易说明如果至少有一个进程启动了算法，也就是说，它扮演发送方P的角色，那么每个进程都将最终完成它的那部分算法。而且，同时启动快照算法的进程数无关紧要。

图3-11表示了快照算法，它应用于一个有三个进程（节点）P_i，P_j和P_k的系统。每个进程通过双向通道与其他两个进程相连。$chan_{ij}$代表从进程P_i到进程P_j的通道。假定进程P_i启动了快照算法。它同时执行三个动作：(a）计算局部状态，(b）发送一个标志到通道$chan_{ij}$和$chan_{ik}$，(c）设置一个计数器对来自输入通道$chan_{ji}$和$chank_i$的消息进行计数。一旦进程P_j从通道$chan_{ij}$接收到标志，它也执行三个动作：(a）计算局部状态并记录通道$chan_{ij}$的状态为空，(b）发送一个标志到通道$chan_{ji}$和$chan_{jk}$，(c）对来自输入通道$chank_j$的消息设置一个计数器。类似地，进程P_k也执行三个动作。我们假设从进程P_i来的标志比从进程P_k来的标志早到达进程P_j。否则，在步骤(c）中的输入通道将是$chan_{ij}$。一旦从进程P_k来的标志到达进程P_j，P_j就记录通道$chan_{kj}$的状态为自设置计数器以来沿着这个通道接收到的消息的序列。于是进程P_j终止自己因为它已经从每个输入通道接收到一个标志消息并已经完成局部状态的计算。类似地，进程P_k在接收到从P_i和P_j发来的标志后终止。进程P_i在接收到从P_j和P_k发来的标志后终止。

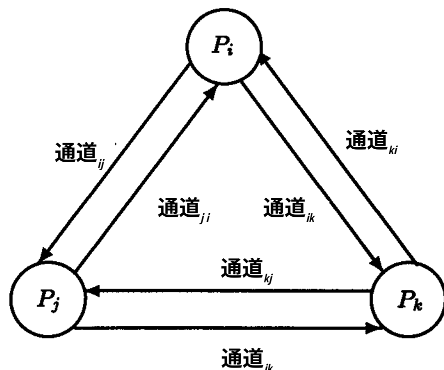


图3-11　有三个进程P_i，P_j和P_k的系统

有一些对Chandy和Lamport快照算法的扩展。Lai和Yang[11]提出了一个简单的在非FIFO通道上的快照算法。Ahuja[2]还提出了一个更优的基于F－通道和flush原语概念的算法。

3.3.4　一致全局状态的充要条件

正如前面所提到的，一致全局状态在许多分布式应用中是很重要的。那么，任意一个检查点（一个局部状态）或检查点集属于一个一致的全局快照的确切条件是什么？Netzer和Xu[15]通过对Lamport的发生在先关系的广义化（称为Z字形路径）确定了这个条件。

一条Z字形路径存在于进程P_i的检查点A到进程P_j的检查点B。P_i和P_j可以是同一个进程当且仅当存在消息m_1，m_2，…，m_n（$n \geq 1$）满足：

- m_1是进程P_i在检查点A之后发送的。
- 如果m_l（$1 \leq l \leq n$）被进程P_k接收，则m_{l+1}在同一个或后面的检查点间隔（checkpoint interval）被P_k发出。注意，m_{l+1}可能在m_l被接收之前或之后发送。

China-pub.com 下载

• m_n被进程P_j在检查点B之前接收。

一个进程的检查点间隔是在两个连续的检查点之间进行的所有计算。检查点 C包括在一个Z字形循环中当且仅当存在一条从C到它自身的Z字形路径。

注意，Z字形路径和基于Lamport的发生在先关系的因果路径（causal path）有点不同。基本上，存在一条从A到B的因果路径当且仅当存在一条A之后开始，B之前结束的消息链，其中每个消息在链中的前一个消息被接收到之后发送。在 Z字形路径中，这样的消息链是允许的。另外，还允许这样的消息链，即链中的任何消息在前一个消息被接收之前发送，只要发送和接收在同一个检查点间隔。在图 3-12中，从C_{11}到C_{31}的由消息m_1和m_2组成的路径是一条Z字形路径。从C_{11}到C_{32}的由消息m_1和m_3组成的路径是一条因果路径（同时也是一条 Z字形路径）。检查点C_{22}形成一个由消息m_4和m_1组成的Z字形循环。

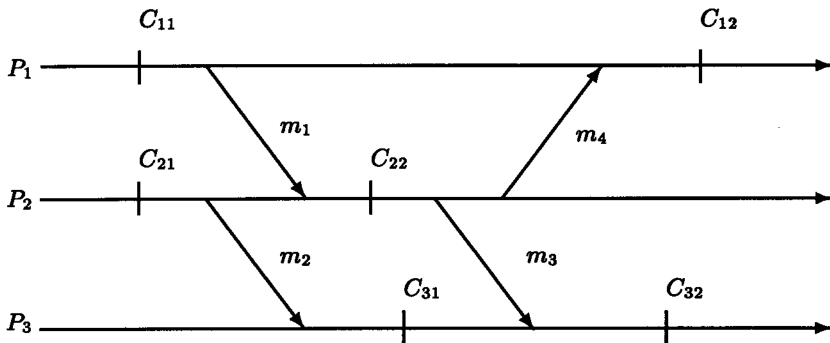


图3-12 有三个进程的分布式系统

在[15]中提出了如下一致性定理：一个检查点集 S，其中每个检查点来自不同的进程，可以属于同一个一致全局状态当且仅当 S中不存在这样的检查点，它有一条到 S中任何其他检查点（包括它自身）的Z字形路径。作为直接的推论，我们有：

• 检查点C可以属于一个一致全局状态当且仅当 C不属于一个Z字形循环。

• 检查点A和B（属于不同的进程）可以属于同一个一致全局状态当且仅当（ a）没有包括A或B的Z字形循环存在，（b）在A和B之间不存在Z字形路径。

一般，每个进程包括两个虚拟的检查点。一个紧接在执行开始之前，另一个紧跟在执行结束之后。在图3-12中，检查点C_{11}和C_{21}可以属于同一个一致全局状态，检查点C_{12}和C_{32}也可以。C_{22}不属于任何一个一致全局状态。

3.4 逻辑时钟

分布式系统没有内在的物理时间，所以只能对它进行近似。即使是在能够精确到几十毫秒的Internet的网络时间协议中，它对于捕获分布式系统中的因果关系仍是不够的。在前面小节中定义的发生在先关系可用于捕获因果关系，而且它可以很容易地通过给定的系统时空视图得到。但是在没有共享全局存储器的系统中，每个进程如何得到并保存这样一种关系呢？ Lamport[12]提出了时戳法则，也叫做逻辑时钟法则。

China-pub.com
下载

3.4.1 标量逻辑时钟

在这个模型中，每个进程 P_i 有一个逻辑时钟 LC_i（它可能是通过一个没有实际计时机制的计数器实现的）。LC_i 被初始化为 $init$（≥0）并且它是一个非减的整数序列。进程 P_i 发出的每个消息 m 都被标上 LC_i 的当前值和进程的标号 i，形成一个3元组（m，LC_i，i）。

LC_i 的更新基于以下两条规则：

• 规则1：在发生一个事件（一个外部发送或内部事件）之前，我们更新 LC_i：

$$LC_i = LC_i + d \qquad (d>0)$$

（d 在规则1的每个应用中可以有不同的值）

• 规则2：当收到一个带时戳的消息（m，LC_j，j）时，P_i 执行更新：

$$LC_i = \max(LC_i, LC_j) + d \qquad (d>0)$$

我们假设每个 LC_i 被初始化为 $init$（≥0），对于不同的进程可以有不同的 $init$ 值。

时戳方法也可以这样实现：只有当外部事件发生时才改变逻辑时钟。这可以通过在以上算法中删去内部事件条件来实现。

利用时戳方法我们可以确定图3-7中每个事件的逻辑时间，见表3-1，假设 d=1，$init$ = 0。

表3-1 实例3.2中的事件和它们的逻辑时间

事件	a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3	c_0	c_1	c_2
逻辑时间	1	2	3	4	1	2	3	4	1	4	5

容易证明时戳方法满足以下条件：对于任意两个事件 a 和 b，如果 $a \rightarrow b$ 则 $LC(a) < LC(b)$，其中 $LC(a)$ 是事件 a 的逻辑时间。但是，一般情况下，条件反过来并不成立，也就是说，当 $LC(a) < LC(b)$ 时不一定有 $a \rightarrow b$。例如，对于图3-7中的事件 c_0 和 a_2，有 $LC(c_0) < LC(a_2)$，但 $c_0 \nrightarrow a_2$。

我们还可以在事件集上定义以下全序关系 ，同时仍满足上述条件，这只是许多种可能的序关系中的一种：

$$a\ (\text{in } P_i) \quad a\ (\text{in } P_j)$$

当且仅当

$$(1)\ LC(a) < LC(b)\ \text{或者}\ (2)\ LC(a) = LC(b)\ \text{且}\ P_i < P_j$$

其中，<是进程集的任意一种全序，例如，<可以定义为 $P_i < P_j$ 当且仅当 $i < j$。根据这个定义，图3-7中的事件在假设 $P_a < P_b < P_c$ 下可以排序为：

$$a_0\ b_0\ c_0\ a_1\ b_1\ a_2\ b_2\ a_3\ b_3\ c_1\ c_2$$

全序关系还提供了分布式系统的交叉视图。虽然和时空视图相比，交叉视图中失去了一些系统信息，但交叉视图对于实现几种类型的分布式算法十分有用 [18]，例如，将要在以后的章节中讨论的分布式互斥算法。

3.4.2 扩展

我们讨论的时戳机制使用线性的时间，也就是说，时间用一个普通的实数来表示。然而，

China-pub.com 下载

线性的时钟系统不能区分由于局部事件引起的时钟前进和由于进程间的消息交换引起的时钟前进。时戳方法的扩展包括向量时间和矩阵时间。向量时间方法[14]使用n维的整数向量表示时间，而矩阵时间方法[28]则使用$n \times n$的矩阵。使用向量时间和矩阵时间的目的在于：

基于规则2（前一节），每个接收进程收到消息的同时还将收到一个逻辑的全局时间，这个时间是发送方在发送该消息时所确定的时间。这一点使得接收方可以更新它的全局时间视图。很自然，当时间中包括信息时，接收方将得到更准确的全局时间视图。

在向量时间方法中，每个进程P_i和一个向量$LC_i[1 \cdots n]$相关联，其中

- $LC_i[i]$ 描述了进程P_i，即自身进程。
- $LC_i[j]$ 表示P_i关于P_j进展的知识。
- $LC_i[1 \cdots n]$组成P_i对于逻辑全局时间的局部视图。

对于每个进程P_i，规则1和规则2修改如下：

- 规则1 在发生一个事件（一个外部发送或内部事件）之前，我们更新 $LC_i[i]$：

$$LC_i[i] := LC_i[i] + d \quad (d > 0)$$

- 规则2 每个消息捎带发送方在发送时的向量时钟。当接收到一个消息（m, LC_j, j）时，P_i执行更新：

$$LC_i[k] := \max(LC_i[k], LC_j[k]), \quad 1 \le k \le n$$

$$LC_i[i] := LC_i[i] + d$$

图3-13表示实例3.2中的向量时钟的进展，增量值d=1，初始值$init$=0。

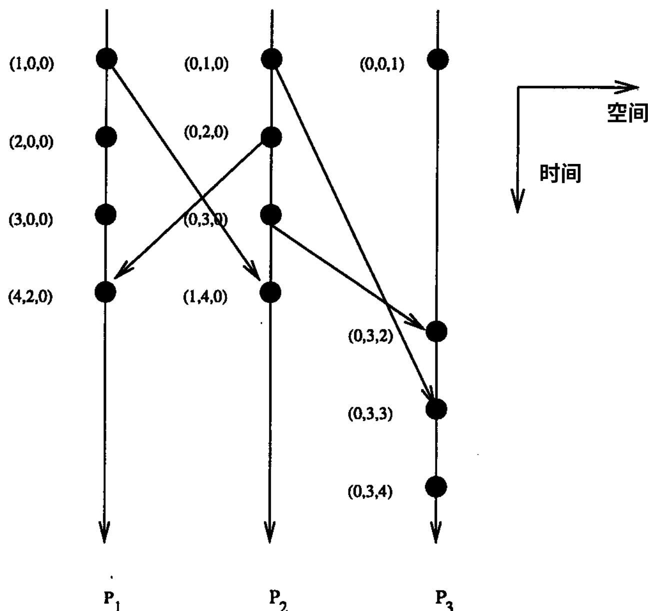


图3-13 向量时钟实例

China-pub.com
下载

当d=1并且$init$ =0，$LC_i[\ i\ ]$对内部事件计数，$LC_i[\ j\ ]$相应于P_j产生的事件数，这些事件在因果关系上处于当前P_i的事件之前。LC_i在P_i读取。$\sum_j Lc_i(j)-1$ 对所有因果关系上处于接收事件之前的事件，包括内部事件进行计数。

设两个事件a和b分别被LC_i和LC_j加以时戳。和一个事件相联系的时戳是紧跟在这个事件之后但在下一个事件之前的时戳。我们有：

$$a \to b \iff LC_i < LC_j$$

和

$$a \parallel b \iff LC_i \parallel LC_j$$

其中

$$LC_i < LC_j \iff \forall_k (LC_i[k] \le LC_j[k]) \wedge \exists_l (LC_i[l] < LC_j[l])$$
$$LC_i \parallel LC_j \iff \neg(LC_i < LC_j) \wedge \neg(LC_j < LC_i)$$

在矩阵时间方法中，每个P_i和一个矩阵$LC_i[1\ldots n，1\ldots n]$相关联，其中

- $LC_i[\ i, i]$是局部逻辑时钟。
- $LC_i[\ k, l]$表示P_i具有的关于P_k对于P_l的局部逻辑时钟所知的观点（或知识）。

实际上，行$LC_i[i,*]$是一个向量时钟，所以这一行继承了向量时钟系统的属性。另外，我们有以下事实。如果

$$\min（LC_i[k, i]）\ge t$$

那么P_i知道每一个其他进程都知道它在局部时间t之前的进展。

矩阵时间的更新可以类似地用规则1和规则2来实现。这个问题的详细讨论见[21]和[28]。逻辑时钟在动态产生和删除进程的系统上的应用见[8]。基本上，动态的方法同时考虑进程的创建和进程的终止。一个计算由一个或多个可能嵌套进程的实例组成。

- 进程的创建。如果事件a和进程实例Q在进程实例P中发生，事件b在Q中发生，并且Q在a之后开始，则$a \to b$。
- 进程的终止。如果事件a和进程实例Q在进程实例P中发生，事件b在Q中发生，并且a在Q终止之后发生，则$b \to a$。

3.4.3　有效实现

当一个给定的分布式系统中有大量的进程并且进程间有大量的通信时，向量时钟和矩阵时钟必须捎带大量的信息来更新逻辑时钟。在[23]中，Singhal和Kshemkalyani提出了一个微分的方法来减少每次通信发送的信息量。这种方法是基于以下事实：在连续两个事件之间，只有几个向量时钟的元素可能改变。当一个进程P_i给P_j发送消息时，P_i只要捎带它的向量时钟中的那些自从上一次给P_j发送消息以来改变过的元素。然而，每个进程需要维护额外的向量以存储上一次和其他进程交互时的时钟值信息。向量时钟的其他有效实现在[20]中讨论。在移动计算环境下的有效实现见[17]。

逻辑时钟已经应用于许多实际领域和理论领域：（a）语言，（b）调试分布式系统，（c）并

China-pub.com 下载

发测量。有关这些应用的详细讨论见 [9]。

3.4.4 物理时钟

物理的时钟连续运行，而不像逻辑时钟那样以离散的“滴答”方式计时。为了让时钟 PC_i 成为真正的物理的时钟并使不同地点的两个时钟同步，Lamport[12]定义了以下两个条件：

- 准确速率条件：

$$i \ / \ dPC_i(t)\ |dt - 1 \ |< \alpha \quad (\alpha << 1)$$

- 时钟同步条件：

$$i \quad j \ / \ PC_i(t) - PC_j(t) \ |< \beta \quad (\beta << 1)$$

Lamport也提出了一个时钟同步的方法如下：

1) 对每个i，如果P_i在物理时间t没收到消息，则PC_i在t可微并且$d\,PC(t)/dt > 0$。

2) 如果P_i在物理时间t发送一个消息m，则m包含了$PC_i(t)$。

3) 当在时间t收到消息（m，PC_j）时，进程P_i设置PC_i为maximum（PC_i（$t-0$），$PC_j+\mu_m$）其中μ_m是预先定义的从一个进程发送消息m到另一个进程的最小延迟。

在以上算法中，$t-0$表示紧接在t之前的时刻。$d\,PC(t)/dt>0$意味着PC总是往前设。

3.5 应用

这一节我们将考虑形式模型的两个应用：一个是全序在分布式互斥中的应用，另一个是逻辑向量时钟在消息排序上的应用。

3.5.1 一个全序应用：分布式互斥

互斥对于分布式系统和操作系统的设计都是一个重要的问题[7]。假设一个分布式系统由一个共享单一资源的固定进程集组成。一次只能有一个进程可以使用资源，所以进程间必须同步以避免冲突。

用于分配资源给进程的互斥算法必须满足以下条件[12]：

- 资源必须先释放再分配。
- 对资源的请求必须按它们提出的顺序得到批准。
- 如果得到资源的进程将最终释放资源，那么对资源的每个请求将最终得到批准。

为了讨论的简单，我们假设消息按它们发送的顺序被接收，而且发出的消息最后一定会被收到。与P_i相关的LC_i是一个线性的时间。每个进程维护一个请求队列。算法是基于以下规则设计的[12]。

1) 为了请求资源，进程P_i发送带时戳的消息给所有的进程（包括它自身）。

2) 当一个进程收到请求资源的消息时，它把该消息放在它的局部请求队列中并发回一个带时戳的应答。

3) 为了释放资源，进程P_i发送一个带时戳的释放资源消息给所有的进程（包括它自身）。

4) 当进程收到来自P_i的释放资源的消息时，它从它的局部请求队列中清除所有来自P_i的请求。

China-pub.com
下载

进程P_j当（a）它的请求r在它的请求队列的顶部，（b）它已经从所有其他进程处收到时戳比r的时戳大的消息时被给予资源。

3.5.2 一个逻辑向量时钟应用：消息的排序

在许多应用中，比如数据库管理系统，消息按照它们发送的顺序被接收是很重要的。例如，如果每个消息是一个更新，每个负责更新复本数据的进程按相同的顺序接收更新消息以维护数据库的一致性。以上要求也称做消息的（因果）排序。

设$s(m)$ $(r(m))$表示从进程P_i到进程P_j的消息m的发送（接收）事件。$C(e)$表示事件e被逻辑时钟（LC）或物理时钟（PC）记录下来的时间。如果以下条件为真，那么消息是有序的：

$$C(s(m)) > C(s(n)) \quad C(r(m)) > C(r(n))$$

图3-14表示两种发生乱序消息的情况。在图3-14a中，两个从同一个进程（P_1）发出的消息在接收方（P_2）发生乱序。这种情况可以用逻辑时钟解决，因为通过赋值d=0，所有初始值为0，逻辑时钟的值对应于进程中的事件数。当接收方收到一个乱序消息时，它将一直等待直到该消息之前的所有消息到达。

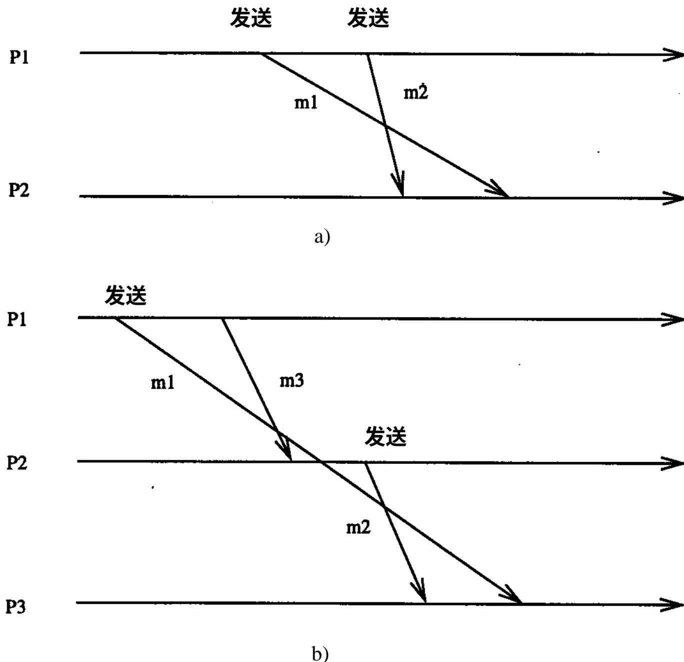


图3-14 两个乱序的消息

在图3-14b中，两个来自不同进程的因果关联的消息到达接收方时乱序。在这种情况下，来自P_1的消息m_1比来自P_2的消息m_2先发出。逻辑向量时钟可以用于这种情况。$LC_2[1]$代表P_2对于P_1的进展的知识。由于$LC_2[1]$在消息m_2中捎带给进程P_3，所以当P_3收到来自P_2的消息m_2时，它知道P_1的情况。P_3在接受来自P_2的消息m_2之前将等待来自P_1的消息m_1。

China-pub.com 下载

所以，为了保证分布式系统中消息的序，基本想法是只有前一个消息递交后才能接受下一个消息。以下是Birman等人[4]提出的协议，它要求进程只能通过广播消息来通信。

/*向量时钟（LC）用于系统中的所有进程 */

1) 在广播一个消息之前，P_i更新它的LC_i。

2) 当P_j收到来自P_i的带LC_i的消息m时，它延迟递交m直到：

(a) $LC_j[i] = LC_i[i] - 1$

(b) $LC_j[k] \geq LC_i[k]$， $k \neq i$

条件a保证进程P_j已经收到所有来自P_i的消息，是P_i发出了消息m。条件b保证P_j已经收到了所有那些在P_i在发出消息m之前收到的消息。以上因果顺序协议要求进程通过广播消息进行通信。一个不要求进程只通过广播消息进行通信的协议[22]也存在，但它非常麻烦和复杂。

3.6 分布式控制算法的分类

一般来说，对分布式算法的严格分类是危险的。然而，对一个分布式算法的质量（特别是对故障的弹性）的评估是十分有用的。有许多不同类型的分布式算法。它们在以下一些属性上存在差异：

- 进程结构。一个分布式算法由一系列进程组成。这些算法在每个进程的正文结构方面存在差异。
- 进程间通信方法。一些常用的方法包括访问共享存储器、发送和接收点到点消息、组通信如广播和执行远程过程调用。
- 定时和传播延迟的知识。对于事件的定时和传播延迟的知识可以做出不同的假设。
- 故障模型。运行分布式算法的系统可能是不可靠的。当一个处理机出现故障时，这样一个系统可能表现出不同的行为：处理机可能停止工作（故障－停止模型），可能暂时失效（暂时－故障模型），或者可能表现出任何行为（拜占庭－故障模型）。
- 应用类型。分布式算法在它们所解决的问题方面存在差异。

进程间通信的方法在第2章详细讨论过，故障模型将在第8章讨论。对于进程结构，Raynal[18]提供了一个基于对称层次的评估：

- 非对称。每个进程执行不同的程序正文，例如，客户服务器算法。
- 正文对称。所有进程执行的正文都是相同的，除了每个正文中的对执行该正文的进程名字的引用不同。
- 强对称。正文相同并且没有对进程名字的引用。根据接收到的消息，仍然可能有相同或不同的行为。
- 完全对称。正文是相同的并且所有进程的表现一样。

定时和传播延迟的知识可以单独讨论。在一个分布式系统中，消息的传输延迟和单个进程中的时间事件相比是不可忽略的。了解传播延迟对于决定一个分布式系统的种类十分重要。Leann[13]提出了一个基于传播延迟的分布式系统的分类：

- 类型1分布式系统。对于任何两个进程，进程间通信的传播延迟是固定的、有限的并且是

China-pub.com 下载

绝对精确可知的，它们对于不同的进程对可能不同。

- 类型2分布式系统。传播延迟是可变的、有限的并且它们的值不是绝对精确可知的。
- 类型3分布式系统。传播延迟是可变的、有限的并且是后验（a posteriori）绝对精确可知的。
- 类型4分布式系统。传播延迟是可变的，但它们的值是有上限的。

在以上分类中，LeLann假设对于每个分布式系统，存在几个时空参考并且每个进程在时间段（$t-\delta$，t）内至少被观察一次，其中，δ是一个有限值，t是局部时间。除非特别指出，否则本书中考虑的分布式系统都属于类型2、类型3或类型4，也就是说，考虑的系统没有关于传播延迟的知识。

关于事件定时的知识有几个不同的假设，这些假设可能用于分布式算法。一种极端是处理机是完全同步的，以一种锁步同步的方式执行通信和计算。另一种极端是处理机可以是完全异步的，以任意速度和顺序工作。任何介于这两种极端之间的称做部分同步。在这样一个模型中，处理机有关于事件定时的部分知识。虽然对大多数情况同步模型是不现实的，但在一个更为现实的模型（如异步的模型）中，它却经常是解决一个复杂问题很有用的一个中间步骤。有时可以通过“模拟”在一个实时系统（它是异步的）上运行同步算法，而对于其他情况，在一个实时系统上直接实现同步算法是不可能的。同步器[3]是一个分布式异步算法，它允许任何同步算法在一个异步系统上运行。这个概念的详细讨论见[19]。

分布式算法领域有许多不同类型的问题，其中典型的问题如下[24]：（a）与容错相关的问题，（b）与通信相关的问题，（c）与同步相关的问题，（d）与控制相关的问题。

3.7 分布式算法的复杂性

在顺序算法中，时间复杂性和空间复杂性是两个常用的衡量标准。然而，分布式系统中一般没有时间的概念，尤其是在异步系统中。分布式系统的一个更流行的衡量标准是消息复杂性，它是算法交换的消息总数。更细化的衡量标准是比特复杂性，它衡量实际交换的消息量（以比特来衡量）。大多数情况下使用消息复杂性除非考虑的系统中包括许多长消息和短消息。

许多分布式算法是不确定的。性能测试可能因不同运行而异。在许多系统中（特别是异步系统），每次运行时的活动进程数都不相同，例如，选择一个领导者的问题，其中一个或多个进程可能启动选择进程。所以，一般需要区分最佳情况、最差情况和平均情况下的复杂性。

参考文献

1 Agerwala, T., “Putting Petri nets to work,” *IEEE Computers*, 12, 12, Dec. 1979, 85-94

2 Ahuja, M., “Flush primitives for asynchronous distributed systems”, *Information Processing Letters*, 34, 1990, 5-12

3 Awerbuch, B., “Complexity of network synchronization”, *Journal of the ACM*, 32, 1985, 804-823

4 Birman, K.P.er al., “Lightweight causal and atomic group multicast”, *ACM Transactions on Computer Systems*, 9, 3 1991, 272-314

China-pub.com 下载

5 Bochmann, G. V., *Concepts for Distributed System Design*, Springer-Verlag, 1983

6 Chandy, K. M. and L. and L. Lamport, "Distributed snapshots: determining global states of distributed systems", *ACM Transactions on Computer Systems*, 3, 1, Feb. 1985, 63-75

7 Deitel, H. M., *An Introduction to Operating Systems*, Second Edition, Addison-Wesley Publishing Company, 1990

8 Fidge, C., "Logical time in distributed computing systems," *IEEE Computers*, 24, 8, August 1991, 28-31

9 Fidge, C., "Logical time in distributed computing systems", in *Readings in Distributed Computing Systems*, T.L. Casavant and M. Singhal, eds., IEEE Computer Society Press, 1994, 73-82

10 France, R., J. Wu, M. M. Larrondo-Petrie, and J. M. Bruel, "A tale of two case studies: Using integrated methods to support rigorous requirements specification", *Proc. of the Int'l Workshop on Formal Methods Integration*, 1996

11 Lai, T. And T. Yang, "On distributed snapshots", *Information Processing Letters*, 25, 1987, 153-158

12 Lamport, L., "Time, clock, and the order of events in a distributed system", *Communications of the ACM*, 21,7, July 1978, 558-568

13 LeLann, G., "Distributed systems-towards a formal approach", *Proc. of the IFIP Congress*, 1977, 155-160

14 Mattern, F., "Virtual time and global states of distributed systems", *Proc. of Parallel and Distributed Algorithms Conf.*, 1988, 215-226

15 Netzer, R. H. And J. Xu, "Necessary and sufficient conditions for consistent global snapshots", *IEEE Transactions on Parallel and Distributed Systems*, 6, 2, Feb. 1995, 165-109

16 Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Inc., 1981.

17 Prakash, R., M. Raynal, and M. Singhal, "An efficient causal ordering algorithm for mobile computing environments", *Proc. of the 16th Int'l Conf. on Distributed Computing Systems*, 1996, 744-751

18 Raynal, M., *Distributed Algorithms and Protocols*, John Wiley&Sons, 1988

19 Raynal, M. and J. M. Helary, *Synchronization and Control of Distributed Systems and Programs*, John Wiley&Sons, 1990

20 Raynal, M. and M. Singhal, "Capturing causality in distributed systems", *IEEE Computers*, 29, 2, Feb. 1996, 49 -56

21 Sarin, S. K. And L. Lynch, "Discarding obsolete information in a replicated database system", *IEEE Transactions on Software Engineering*, 13, 1, Jan. 1987, 39-46

22 Schiper, A. J. er al., "A new algorithm to implement causal ordering", *Proc. of the Int'l Workshop on Distributed Algorithms*, 1989, 219-232

China-pub.com
下载

23 Singhal, M. and A. Kshemkalyani, "An efficient implementation of vector clicks", *Information Processing Letters*, 43, Aug. 1992, 47-52

24 Tel, G., *Topics in Distributed Algorithms*, Cambridge University Press, 1991

25 Wu, J. and E. B. Fernandez, "Petri nets modeling techniques and applocation", in *Modeling and Simulation*, W. G. Vogt and M. H. Mickle, eds., 21, 3, 1989, 1311-1317

26 Wu, J. and E. B. Fernandez, "A simplification of a conversation design scheme using Petri nets", *IEEE Transactions on Software Engineering*, 15, 5, May 1989, 658-660

27 Wu, J. and E. B. Fernandez, "Using Petri nets for determining conversation boundaries in fault-tolerant software", *IEEE Transactions on Parallel and Distributed Systems*, 5, 10, Oct. 1994, 1106-1112

28 Wuu, G. T. J. and A. J. Bernstein, "Efficient solutions to the replicated log and dictionary problems", *Proc. of the 3rd ACM Symp. in PODC*, 1984, 233-242

29 Yau, S. S. And M. U. Caglayan, "Distributed software system design representatin using modified Petri nets", *IEEE Transactions on Software Engineering*, 9, 6, Nov. 1983, 733-744

习题

1．考虑一个可以动态创建和终止进程的系统。一个进程可以生成新的进程。例如，P_1生成P_2和P_3。对于这样一个动态进程集中的事件，修改发生在先关系和线性逻辑时钟方案。

2．对于图3-15所示的分布式系统

（a）给出所有相关联的事件对。

（b）为所有的事件提供逻辑时间，使用

（i）线性时间

（ii）向量时间

假设每个LC_i初始化为0并且d=1。

3．基于与每个事件相关联的向量逻辑时钟定义一个分布式系统中事件的全序（ ）。注意，对于两个事件a和b：如果$a \rightarrow b$，则a b。

4．为问题2中的系统中的所有事件提供线性逻辑时钟。假设所有的LC初始化为0而且P_a、P_b、P_c的d分别为1、2、3。条件$a \rightarrow b$ $LC(a) < LC(b)$是否仍成立？对于d的任何其他设置呢？为什么？

5．用DCDL表示Lamport的互斥算法。

6．扩展Lamport的互斥算法，使之适用于每个进程收到消息的顺序可能和这些消息发送时的顺序不一样的情况。我们假设消息的传输延迟限制为时间δ。

7．使用扩展佩特里网对图3-7中的系统建模。

8．使用一个四进程的例子说明全局状态的快照。假设四个进程（节点）通过双向链路完全连接并且只有一个进程启动快照过程。

China-pub.com
下载

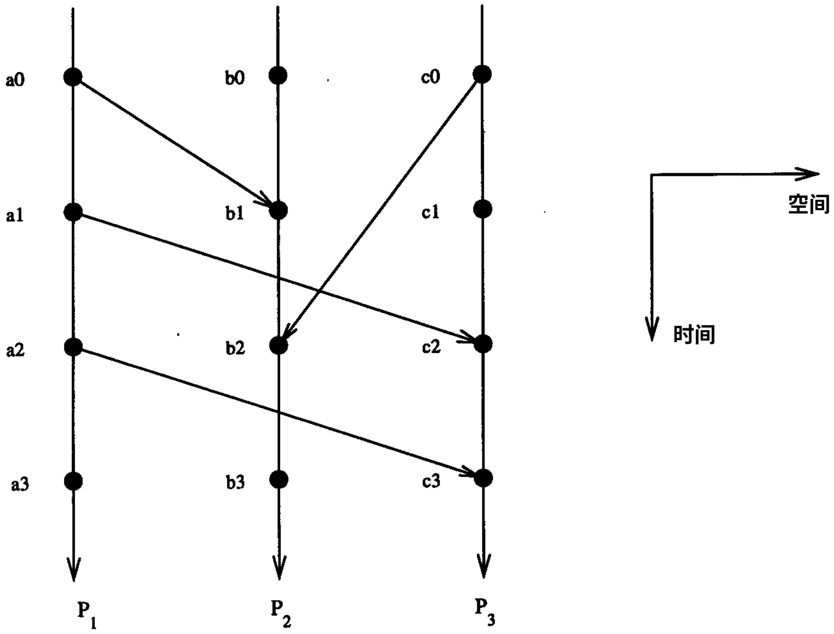


图3-15 习题2的分布式系统