其他知识点：

- 逻辑并发：当两个事件间无因果影响时，两个事件是逻辑并发的。

  物理并发：不同事件在物理时间的同一时刻发生。

- Happens-before 关系是一种偏序关系，通常记为 `→`。三种关系：a->b，b->a，a||b。它定义如下：

  1. **同一进程内的顺序性**：如果在同一个进程中，事件 `a` 在事件 `b` 之前发生，则 `a → b`。 例如，在进程 `P1` 中，发送消息的操作 `a` 必定发生在接收消息的操作 `b` 之前。
  2. **消息传递的因果性**：如果事件 `a` 是一个消息的发送，事件 `b` 是该消息的接收，则 `a → b`。
  3. **传递性**：如果 `a → b` 且 `b → c`，那么 `a → c`。
  4. **并发事件**：如果两个事件 `a` 和 `b` 之间不存在 `a → b` 或 `b → a` 的关系，则称它们为并发事件，记为 `a || b`。

第一章：分布式系统概念

- 分布式系统：
  - 概念：使用互联网络将分布在不同位置的计算机连接起来

    特点：物理分布，逻辑集中；个体独立，整体统一
  - 通信：内存共享、消息传递
  - 通信原语：同步/异步、阻塞/非阻塞、

第二章：全局一致状态

- 信道模型：CO < FIFO < Non-FIFO

  每个信道都运行一个FIFO消息**队列**，消息顺序是由信道维持的；Non-FIFO：每个信道都运行一个**集合**，其中发送进程向集合加入消息，接收进程从中移除消息，**被加入或移除的消息顺序是随机的**；CO因果：基于**happens before**关系，send(m1) -> send(m2) ==> recv(m1) -> recv(m2)

- 全局一致状态：
  - 定义：进程本地状态（寄存器状态、内存、变量等）+ 进程之间通信状态（消息发送和接收）；满足因果性：一个事件被接收，则其一定被发送
  - 判断：画切线，左侧(现在及其过去)包含发送事件则为全局一致；若左侧包含接收事件而不包含发送事件，则不是。意义是保证所有节点或进程看到的值都是一致的

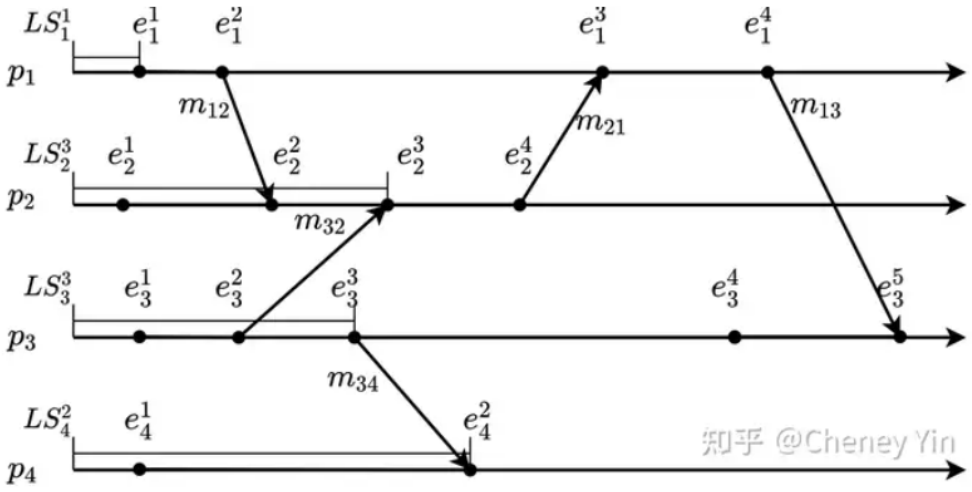


图2 某分布式运行的时空图

在图2中，包含本地状态$\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$的全局状态$GS_1$是不一致的，因为$p_2$的状态$LS_2^3$已经记录了消息$m_{12}$的接收，然而$p_1$的状态$LS_1^1$还未记录该消息的发送，所有不满足一致性条件，即没有发送消息，就没有接收消息。

在图2中，包含本地状态$\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$的全局状态$GS_2$是一致状态，因为除了包含消息$m_{21}$的$C_{21}$外的其余信道都是空的（空信道意味着发送的消息都已被接收）。

- 全序集：集合中所有事件均可以比较发生的顺序（因果和并发）： ei<ej或ej<ei，具有传递性：ei<ej，ej<ek =>ei<ek； 用于全局一致性判断

  反对称性：一个事件不可同时满足：ei<ej且ej<ei

  偏序集：集合中的事件只能部分比较因果性，并发事件不能比较因果性；用于因果一致性判断；因果性具有传递性，并发没有

  偏序->全序转化：添加额外时钟

特性	偏序（Partial Order）	全序（Total Order）
顺序性	并发事件无法比较	任意两个事件都可以比较其先后顺序
场景	因果关系跟踪（如矢量时钟）	全局一致性（如分布式事务、日志）
性能	高效，避免不必要的顺序化处理	开销较高，需要额外的同步机制
使用场景	因果一致性模型	强一致性模型

- 通信模型：

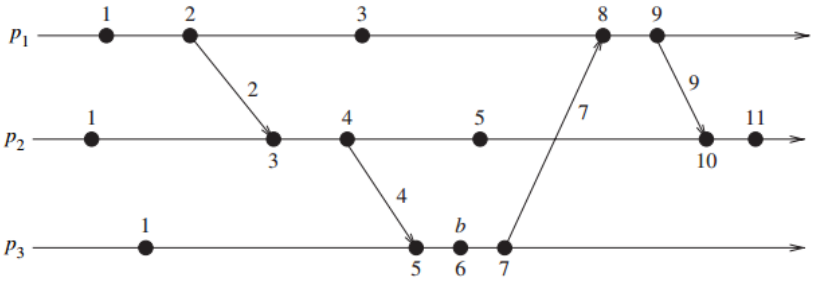
  同步通信：消息发送结束后，发送进程阻塞等待接收进程接受到消息才继续下一步发送；简单，但性能不好

  异步通信：消息发送后，发送进程可以继续发送消息，消息会在适当时候被接收进程接收；高并行度；buffer管理问题，因为发送进程可能突然发送大量消息

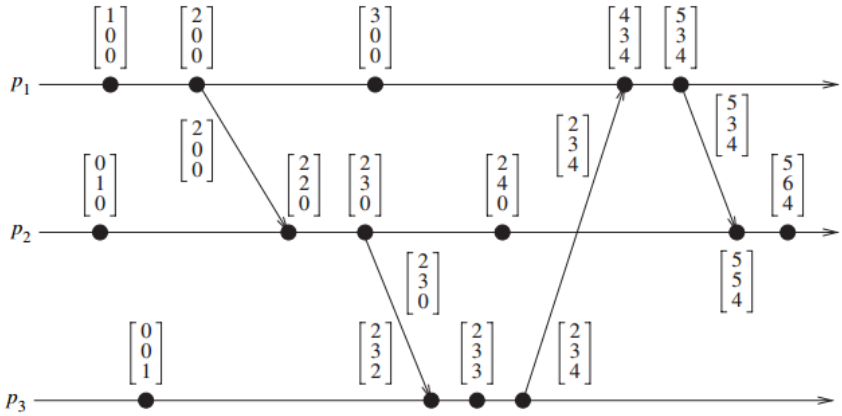
第三章：逻辑时钟

- 分布式系统不存在全局共享的内存和物理时钟，使用逻辑时钟判断分布式系统的因果单调性
- 标量时钟和向量时钟都用于描述分布式系统中事件发生的因果顺序，信道中传输时间戳

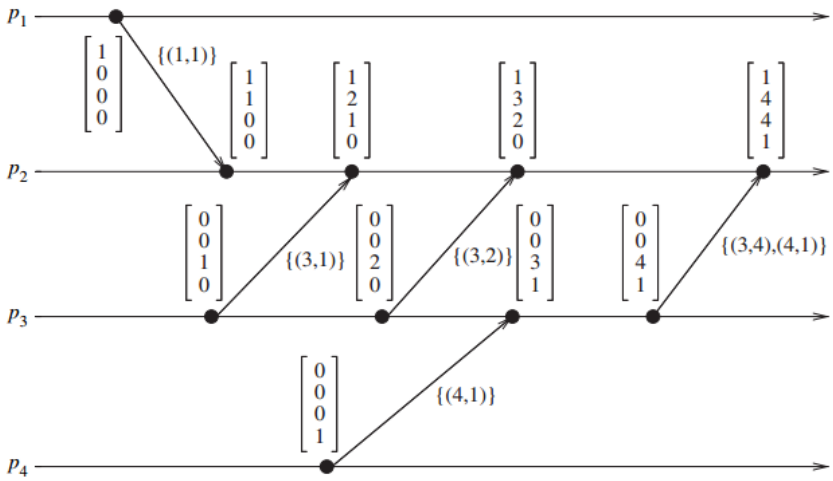
- 标量时钟



- 向量时钟



- SK算法优化向量时钟的存储时间复杂度



- **虚拟时间的local control机制**：不理解
- 虚拟时间的消息包含：发送进程PID、接收进程PID、发送时间、接收时间（在该时间前，消息必须被接收）
- 虚拟时间的antimessage和rollback：反消息和正消息只有符号差别

  `**反消息**：与消息内容相同，符号相反。传递消息时，该消息的一个拷贝进入接收方的输入队列，一个负拷贝留在发送方的输出队列以便发送方回滚。

  当一个消息与其反消息出现在同一个队列中时，它们立刻无效。

  回滚的触发：如果消息时间戳 < 接收方本地时间，则接收方必须进行回滚。

  回滚机制：

  1. 搜索状态队列，找出最大的并且 < 消息时间戳的状态，并从状态队列中去除该事件之后保存的所有状态，然后从该点恢复向前执行。为了收回一条消息，只需要发送其反消息即可。
  2. 考虑反消息的接收方：
  3. 如果原消息已经到达，但还未被处理，则它的虚拟接收时间必定大于接收方的虚拟时间。反消息到来后不会引起回滚，它将直接与原消息一同无效。

4. 如果原消息已经开始被处理，反消息的到来会使得接收方回滚到原消息被接收的虚拟时间，然后废除原消息，使接收方不保留原消息的记录。这一回滚可能引发连串回滚。
5. 如果反消息先于原消息到达接收方，它只是被加入输入队列。接收方执行输入队列时将跳过反消息。

`

- antimesage：PA向PB发送一个消息mA，mA的反消息-mA存放到PA的输出队列；PB接收mA，若mA的虚拟虚拟接收时间大于PB当前的虚拟时间(mA没有迟到)，PB正常将其放入输入队列等待处理，处理结束后放入PB的输出队列；若小于PB当前虚拟时间，PB回退到mA的时间.若PA发现mA出错，则将输出队列的反消息-mA发给PB，PB输入队列中同时有mA和-mA相互湮灭，达到回滚效果
- rollback：

  1.找到PA小于接收消息mB虚拟接收时间的最大状态s1并设置；

  2.将PA当前虚拟时间设置为mB的虚拟接受时间s2,并将大于该时间的状态全部删除；

  3.撤销PA在状态s1和s2之间发送的消息：发送反消息；

  4.接收到反消息的：反消息时间小于进程虚拟时间,进程状态回滚；大于的,反消息和消息都存在,湮灭
- anitimessage-rollback：
  - 反消息晚于正消息到：反消息被接收时,正消息未被处理,相互湮灭；

    反消息被接收时,正消息正在或已经被处理,进程回滚到反(正)消息的虚拟接收时间，正反消息湮灭；
  - 反消息早于正消息到：等正消息到,相互湮灭(轮到反消息执行就空转)

- 物理时钟同步对齐：
  - 时钟偏移和往返延迟
  - NTP协议：基于时钟偏移和往返延迟估计两个节点时钟偏差，具体见例子即可

Ca和Cb是开始计算的时间参考点，A->B时延取平均δ/2

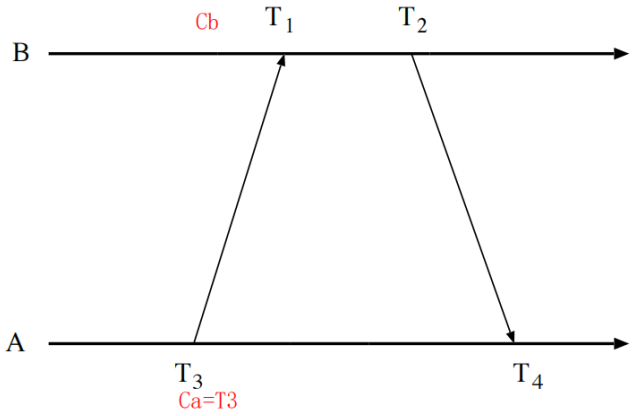


Figure 3.6: Offset and delay estimation.

时钟偏移的概念:a相对于b偏移:Ca-Cb

- Let $a = T_1 - T_3$ and $b = T_2 - T_4$.
- If the network delay difference from A to B and from B to A, called *differential delay*, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T_4 are approximately given by the following.

时钟偏移 往返时延

B相对于A的时钟偏移

Ca=T3

Cb=T1-δ/2

Cb-Ca=

$$\theta = \frac{a+b}{2}, \quad \delta = a - b \qquad (2)$$

=（T4-T2）+(T3-T1)

和选择最小时间延迟比,选择平均时延计算:Cb>Ca，则偏小；Cb<Ca则偏大

- Each NTP message includes the latest three timestamps T_1, T_2 and T_3, while T_4 is determined upon arrival.
- Thus, both peers A and B can independently calculate delay and offset using a single bidirectional message stream as shown in Figure 3.7.

第四章：全局一致快照算法

- 分布式系统中三种事件：进程内部事件、消息发送事件、消息接收事件
- 全局一致状态：进程状态和信道状态的集合
  - 定义：进程本地状态（寄存器状态、内存、变量等）+进程之间通信状态（消息发送和接收）；满足因果性：一个事件被接收，则其一定被发送
- Channy-Lamport全局快照算法：针对FIFO信道；
  - 全局一致的充要条件C1和C2：消息只要被发送,则一定在信道或接收进程中
  - 基本思想：记录进程的本地状态+信道接收消息；marker用于分隔当前快照，记录两次marker之间的进程状态和信道状态；直到所有的进程都收到marker

        - **启动**：任意一个进程P0启动快照，记录p0本地状态后，向所有与其相邻的进程发送marker；
        - **接收**：接收进程Pi第一次收到marker，则记录其本地状态，并接收发送而来的消息，向所有与其相邻的进程发送marker；第二次接收到marker，停止接收消息
        - **完成**：所有进程都收到两次marker：节点将其本地状态和记录的传入消息发送给中央收集器，或者将其保存在本地
    - 总结
        - **快照边界**：标记消息在通道中作为“分隔符”，确保快照记录的是标记消息前的状态。
        - **通道状态记录**：
            - 若marker到达时节点尚未记录进程状态，则该通道状态为“空”。
            - 若marker到达时节点已记录进程状态，则需要将从状态记录到收到marker之间的消息记录为通道状态。
- **SK全局快照算法**：Channy-Lamport改进，FIFO信道
- **判断全局一致的快照：zigpath,不理解**
    - 全局一致性快照的充要条件：两个检查点之间没有zigzag路径

      全局一致性快照的充要条件：一个检查点能够成为一致性快照的一部分，当且仅当该检查点不属于 zigzag 环路（zigcycle）

      全局一致性快照的必要条件：两个检查点之间没有因果路径，即：全局一致性快照检查点之间没有因果路径，但没有因果路径不能作为判断的充分条件
    - zigzag path：
        - 判断准则：检查点Cx,i包含进程Px的第i个检查点区间[Cx,i-1 , Cx,i)检查点区间所有信息

          m1在Cx,i之后发送,Cx,i没有包含m1；

          中间进程接收mk,发送mk+1,mk和mk+1可以在接收mk的同一个检查点间隔或之后的任何检查点间隔;mk+1可以早于mk；

          mn在Cy,j之前接收,Cy,j包含mn
        - 特点：zigzag path是一条因果路径,允许另一个消息在消息被接收时发送,如：m3被p2接收，m4被p2发送，m4早于m3
            - m4可以在m3的同一个检查点或后面任意一个检查点区间
    - zigcycle：存在从检查点C出发，然后回到检查点C的zigzag path

第六章 术语和基本算法

- **同步算法和异步算法**：

  同步算法和异步算法的区别主要在于**节点间是否需要统一的全局时钟**。同步算法通过全局时钟以轮次推进，**节点步调一致（消息发送，消息接收，信息更新），消息在固定时间内到达**，设计简单但对同步依赖强；而异步算法没有全局时钟，**节点基于消息触发独立运行（消息的发送和接收没有固定时间）**，但设计和分析更复杂，可能增加消息冗余。

- Basic Distributed Graph Algorithms: Listing

  - Sync 1-initiator ST (flooding)
  - Async 1-initiator ST (flooding)
  - Async conc-initiator ST (flooding)
  - Async DFS ST
  - Broadcast & convergecast on tree
  - Sync 1-source shortest path
  - Distance Vector Routing
  - Async 1-source shortest path
  - All sources shortest path: Floyd-Warshall
  - Sync, async constrained flooding
  - MST, sync
  - MST, async
  - Synchronizers: simple, α, β, γ
  - MIS, async, randomized
  - CDS
  - Compact routing tables
  - Leader election: LCR algorithm
  - Dynamic object replication

第七章 消息序和组通信

- **RSC < CO < FIFO < A异步**
- FIFO判断：s1~s2，r1~r2，s1<s2则r1<r2

  e.g, TCP，non-FIFO上实现FIFO使用序列号和确认id
- CO判断：找在同一进程的接收事件r1~r2且r1<r2，若s2<s1不成立（无法判断也可以，不要求s1和s2在同一进程），则满足CO

  s1~s2的CO满足FIFO

  Causal Order: Definition

  Causal order (CO)

  r1和r2在同一个进程,s1因果早于s2,(不要求s1和s2在同一个进程),则r1因果早于r2

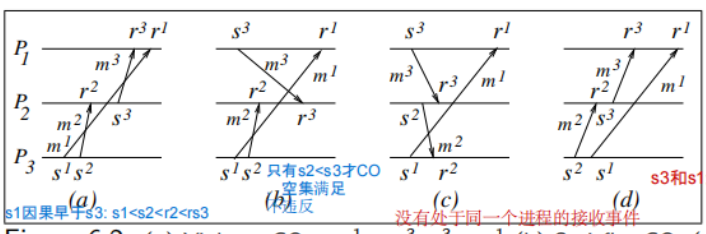
  A CO execution is an A-execution in which, for all (s, r) and $(s', r') \in \mathcal{T}$, $(r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'$

  m1因果早于m2

  事件按照因果性排序, 不按照物理事件排序

  - If send events s and s' are related by causality ordering *(not physical time ordering)*, their corresponding receive events r and r' occur in the same order at all common dests.
  - If s and s' are not related by causality, then CO is vacuously satisfied.

  空集满足, 条件不存在, 则自动满足CO

  找在同一个进程上的接收事件的
  发送事件是否违背对应规则：
  无法确定因果性也算CO(空集满足)

  Figure 6.2: (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.
- 消息序MO判断CO：s1<s2，若r2<r1不成立，则满足MO （**找因果发送事件**）

Causal Order: Other Characterizations (1)

传输过程一定不会被打断

消息序

先找因果发送事件：s1因果早于s2, 则r2一定不会因果早于r1

Message Order (MO)

A-execution in which, for all (s, r) and $(s', r') \in \mathcal{T}$, $s \prec s' \Longrightarrow \neg(r' \prec r)$

- Fig 6.2(a): $s^1 \prec s^3$ but $\neg(r^3 \prec r^1)$ is false $\Rightarrow$ MO not satisfied
- m cannot be overtaken by a chain

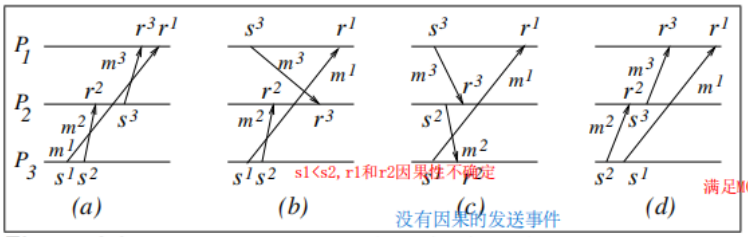


Figure 6.2: (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.

- SYNC同步系统：所有的发送和接收线均可以转换为垂直线，则为SYNC
- **异步程序（A-execution）使用同步原语执行会造成死锁**

  异步程序可以使用同步通信RSC（Realiable Synchronized Commnucation）实现

  分离线性拓展：(s,r)之间允许第三个事件发生,即s<x<r,<是因果早于；

  不可分离的线性拓展则是：s<r之间没有其他事件

- 异步执行和RSC等价充要条件：**异步执行是同步通信执行的充要条件：存在非分离线性拓展**
  - 实际判断A-execution和RSC是否等价使用crown判断，不使用查询：时间开销大
  - 异步系统转同步系统：crown

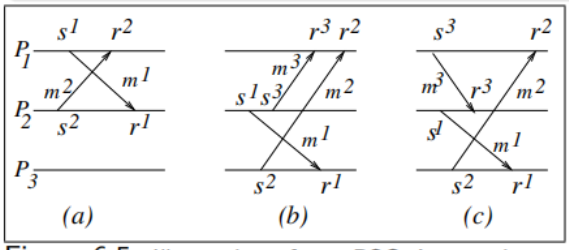
Crown: Definition

实际判断A-execution和RSC是否等价使用crown判断

Crown

构成一个crown，就是非同步

Let E be an execution. A crown of size k in E is a sequence $\langle (s^i, r^i),\ i \in \{0, \ldots, k-1\} \rangle$ of pairs of corresponding send and receive events such that:
$s^0 \prec r^1,\ s^1 \prec r^2, \ldots\ldots s^{k-2} \prec r^{k-1},\ s^{k-1} \prec r^0$.



例子

Figure 6.5: Illustration of non-RSC A-executions and crowns.
Fig 6.5(a): crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(b) (b) crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(c): crown is $\langle (s^1, r^1), (s^3, r^3), (s^2, r^2) \rangle$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$
Fig 6.2(a): crown is $\langle (s^1, r^1), (s^2, r^2), (s^3, r^3) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.

  - crown特点：

    crown中, *s_i* and *r_i+1* 可以或不在同一个进程中

    Non-CO execution 一定有一个crown；不是同步的CO有crown

    CO executions (that are not synchronous) have a crown (see Fig 6.2(b))

Cyclic dependencies of crown => cannot schedule messages serially => not RSC

第五章：微服务

1. 微服务:一系列进程，进程之间使用HTTP通信；服务之间松耦合，独立开发部署，使用网络连接；每个服务代码量小，可以独立开发
2. 微服务前提：快速交付部署，状态监控，代码复杂度高
3. 每个微服务单独使用一个数据库，因为微服务部署在容器中，而容器部署在内存中，容易被断掉，数据丢失(容器易挥发)
4. 微服务系统解藕：按照业务划分，不是代码
5. 微服务之间数据通信只能通过API，微服务对于多语言开发有优势(单语言可以简单解决的不需要微服务)
6. 微服务安全更复杂，需要使用多级检查，开始的API Gateway ,内部也要检查鉴权。传统单体应用只需要入口监测即可。
7. 总结：
   微服务是什么：一系列松耦合的服务，服务之间使用网络通信
   使用微服务前提：快速交付部署，需要多语言支持的复杂系统
   如何开发微服务应用：使用开发框架：业务拆分为多个微服务，每个微服务使用一个容器部署(容器中包含数据库的一个备份)
   微服务的debug：在请求头设置label，跟踪请求所经过的微服务的路径