并行与分布式计算 - June 21'th, 2018

Parallel System Concepts

https://github.com/rh01

分布式计算

Ex. 分布式计算是利用分布在网络上的各种软、硬计算机资源进行信息处理的过程，它是通过基于网络的分布式系统实现的

- 信息处理：信息共享，数据存储/分析/挖掘/管理/集成，流程协作

- 分布式计算的形态
  - 挖掘计算机的计算、存储等资源
    * 对等计算 (Peer to Peer Computing)
      · 利用边缘化的计算机进行信息处理，提供信息服务。
      · P2P 系统通常构建有高效的覆盖网，允许结点动态地加入和离开；每个结点在功能上是平等的；
    * 云计算 (Cloud Computing)(网格计算、集群计算)
      · 作为一种资源利用模式，能以简便的途径和按需的方式通过网络为用户提供可配置的海量信息计算资源
      · 特点：超大规模，虚拟化，按需自助服务，高可伸缩性，高可靠性，廉价/灵活计费
    * 雾计算 (Fog Computing)
      · 利用网络边缘的设备进行信息处理，使得用户能在本地得到信息服务
  - 以人为本 (消失的计算，侧重用户体验)
    * 移动计算、普适计算、社会计算 (社交网络)
    * 移动计算：分布式计算在移动维度的扩展
    * 普适计算：无时无处不在而又不可见的计算
  - 物联网 (Internet of Things; Cyber Physical Systems)
    * 无线传感网 (Wireless Sensor Network)
    * 车联网 (Internet of Vehicles, Vehicular Ad hoc NETwork /VANET)
  - 强调计算的某个特征
    * 可信计算，服务计算 (互操作性)
    * 效用计算 (Utility Computing)、弹性计算 (Elastic Computing) 、自治计算 (Autonomic Computing)

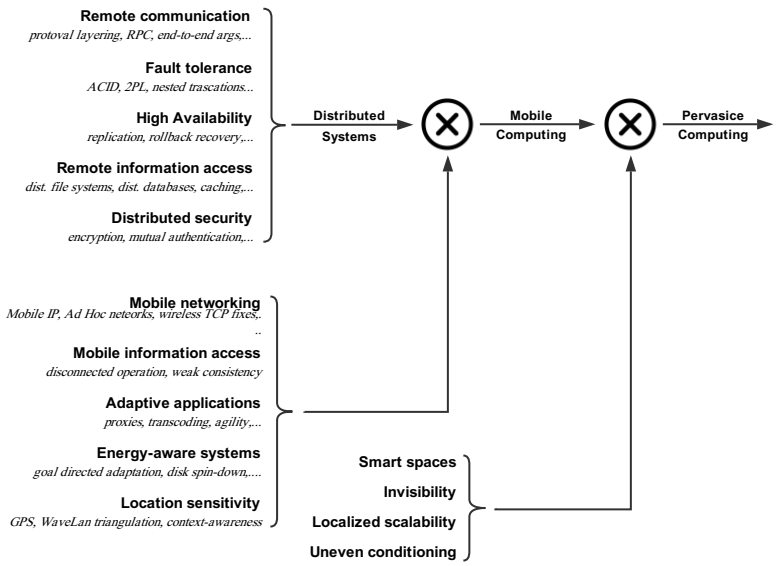

图 1：分布式计算的多种形态

并行计算硬件环境 vs. 分布式计算硬件环境

- 并行环境
  - 共享存储的多处理器系统 (PVP Parallel Vector Processor，SMP Symmetric MultiProcessor，DSM Distributed Shared Memory；)
  - 多机系统 Multicomputers：MPP Massively Parallel Processors，Cluster of Workstations
    * 天河二号
    * 神威 • 太湖之光
- 分布式环境
  - 同构或异构的多机系统 (**集群**)

并行计算软件环境 vs. 分布式计算软件环境

- 操作系统
  - DOS (Distributed Operating Systems) 用于管理多处理器系统和同构的多机系统：提供全系统资源统一视图
  - NOS (Network Operating Systems) 用于异构的多机系统；
  - 分布式系统基础架构：Hadoop v1/v2，SPARK
- 软件设计：并行计算
  - 并行算法的设计技术包括
    * 平衡树设计技术
    * 流水线设计技术
    * 分治设计技术
  - 典型应用：
    * 并行数值算法 (线性方程组的求解，快速傅里叶变换，串行算法的并行化)

– 使用并行编程接口 OpenMP、MPI、CUDA (Compute Unified Device Architecture)) 进行并行计算

分布式系统

Ex. 分布式系统是使得基于网络互相连接的若干计算单元进行协同计算的软件。对用户而言，好像是面对一个单一的系统

- 开发分布式系统的动力
  - 许多应用本身是固有分布的
  - 同集中式计算环境相比，分布式计算环境具有很好的性能价格比，可以提供单个大型主机所不能提供的并发计算的能力
  - 分布式系统具有很好的可靠性和可用性

分布式系统的基本属性

- 一个分布式系统拥有一定数目的计算单元和一定数目的进程
  - 物理资源：计算单元
  - 逻辑资源：进程
- **进程之间通过消息传递进行通信**
- **进程之间以协作的方式交互**
- **通信延迟不可忽略**
- **任何单个逻辑或物理的资源故障不会导致整个系统的崩溃**
- **在资源故障的情况下系统必须具有重新配置和故障恢复的能力**

Example

- 企业应用：金融交易 (银行网点)
- 互联网应用：WWW，搜索引擎

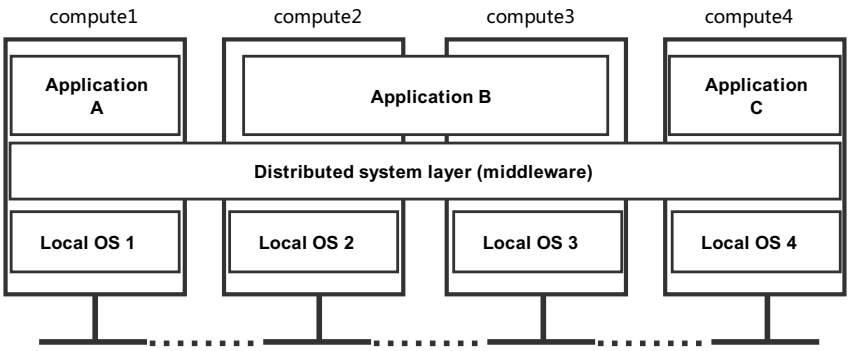

图 2：Example

分布式系统的分类

- 基于网络：有线网 (LAN/Internet)；无线网 (3G/4G/WSN/VANET)

- 使用规模：个人用户/企业用户/互联网应用
- 从系统设计的角度，系统分类的最重要因素：对时序的假设 $\longrightarrow$ 其次：故障假设；
- 所使用的进程间通信机制 (Nancy Lynch)
  - 同步系统：已知时钟漂移率的范围、最大的消息传输延迟和进程每一步的执行时间。一个实际的分布式系统可以模拟成一个同步系统
  - 异步系统
  - 部分同步系统

分布式系统的分类 (Andrew Tanenbaum, 2016)

- Distributed Computing Systems
  - Cluster computing systems
  - Grid computing systems
  - Coulds: IaaS, PaaS, SaaS
- Distributed Information Systems
  - Transaction processing systems: ACID
  - Enterprise application integration
- Distributed Pervasive Systems
  - Ubiquitous computing systems
  - Mobile computing systems
  - Sensor networks

Cluster Computing Systems

- A collection of similar workstations or PCs, connected by means of a high-speed local-area network,
- Each node runs the same operating system

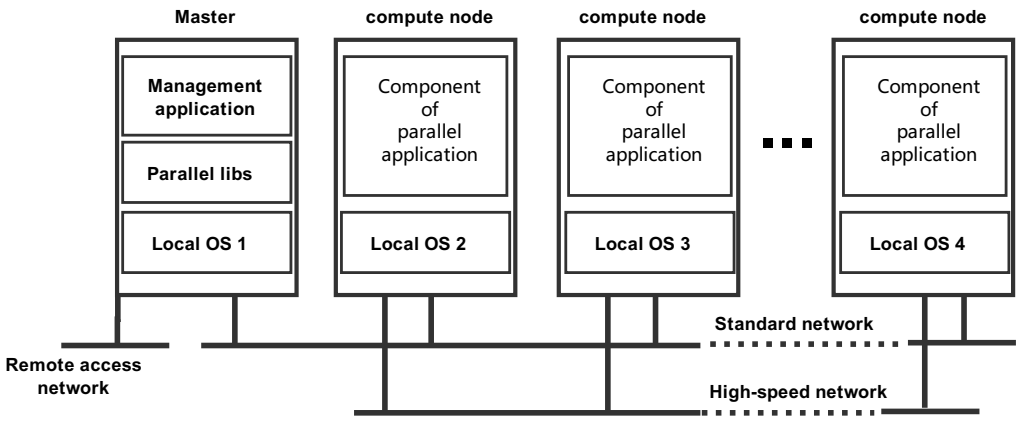

图 3：Cluster Computing Systems

Grid Computing Systems

Ex. A layered architecture for grid computing systems

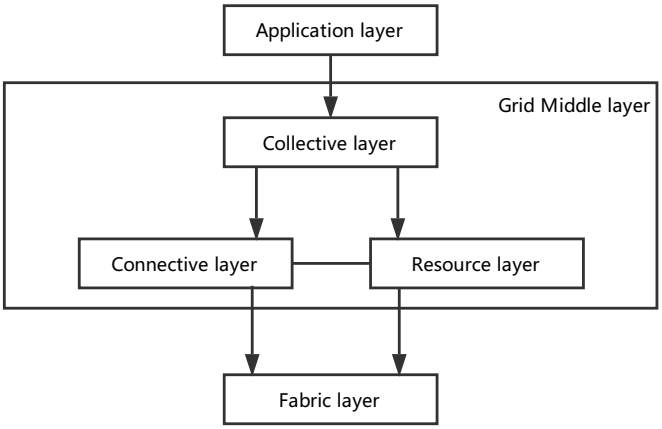

图 4：Grid Computing Systems

Clouds

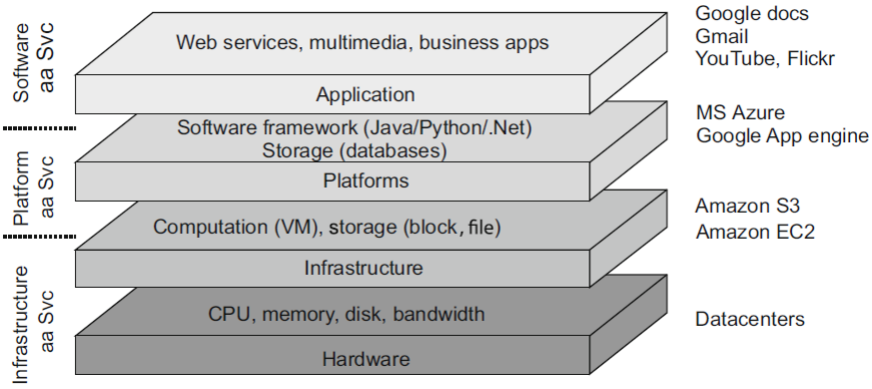

图 5：Clouds

Transaction Processing Systems

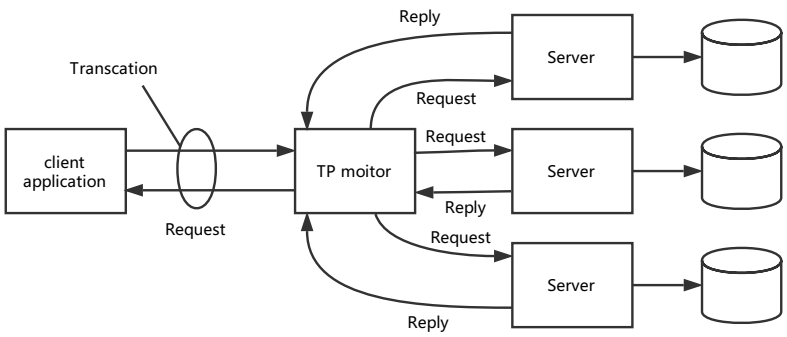

图 6：Transaction Processing Systems

Enterprise Application Integration

- Communication middleware:
  - remote procedure call (RPC),
  - remote method invocation (RMI),
  - message-oriented middleware,
  - publish/subscribe middleware

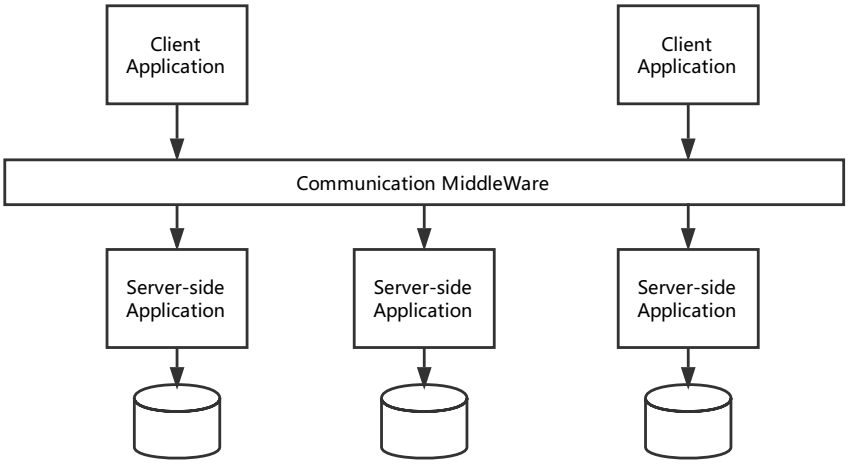

图 7：Enterprise Application Integration

普适计算系统

- Distribution: Devices are networked, distributed, and accessible in a transparent manner
- Interaction: Interaction between users and devices is highly unobtrusive
- Context awareness: The system is aware of a user's context in order to optimize interaction
  - the where, who, when, and what
- Autonomy: Devices operate autonomously without human intervention, and are thus highly self-managed
  - Allocating network addresses: Dynamic Host Configuration Protocol (DHCP)
  - Adding devices: Universal Plug and Play Protocol (UPnP)
  - Automatic updating
- Intelligence: The system as a whole can handle a wide range of dynamic actions and interactions

分布式系统的设计目标

- 连接用户和资源
  - 以一种安全、可靠的方式进行资源共享和用户协作
- 透明性
- 开放性
- 可伸缩性 (scalability)

透明性	描述
访问	隐藏数据表示上的不同和访问资源的方式
位置	隐藏资源的定位方式
迁移	隐藏资源移动
重定位	允许资源可以在使用的时候移动位置
复制	隐藏一个资源是复制的
并发	隐藏一个资源被几个用户共享
故障	隐藏一个资源的故障和恢复
持久	隐藏一个资源是在内存还是在磁盘

表 1: 分布透明性

处理器的数据表示

- 大序法 (Big-endian)：整数的最高位存在最低的地址上
  - Motorola 68000,System/380, SPARC (SUN Solaris) 处理器
- 小序法 (Little-endian)：
  - Intel x86 处理器，AMD64
- 双序法 (Bi-endian)，缺省情况是大序法
  - 大多数的 PowerPC 系统，运行 IRIX 的 MIPS，
- 双序法，缺省情况是小序法
  - 大多数 Alpha 处理器

开放性

- 它提供的服务的规约应该是完整和中性的
- 系统应该是灵活的、可扩展的

可伸缩性

可伸缩性的三个尺度

- 在规模上可伸缩/vertical scalability：可以增加更多的用户和资源
- 在地理上可伸缩/horizontal scalability：用户、资源都可以相距很远
- 在管理上可伸缩：能很容易地管理相互独立的组织

分布式算法与集中式算法的区别

- 没有全局时钟，分布式算法是以没有全局时钟为前提的
- 没有任何一台机器具有系统完整的状态信息
- 每台机器仅根据本地信息进行决策
- 一台机器出了故障，不会使整个算法崩溃

具有可伸缩性的系统实例：**DNS**

改善系统可伸缩性的方法

- 在**体系结构层面**：发现分布的可能性
  - 对数据、服务进行克隆；
  - 对不同的数据、服务进行拆分，分布到不同的地点
  - 对客户进行拆分，分布到不同的地点；
- 在**通信层面**：利用异步通信，避免消息总线过度拥挤；减少通信
- 在**容错层面**：采用能隔离故障的设计，使得故障的影响面有限，包括避免级联模块的设计，避免形成单点故障的设计
- 在**数据层面**：使用复制和缓存；努力实现无状态；如果不能避免，那么或者尽可能在浏览器端维护会话，或者利用分布式缓存存放状态

分布式系统的时间

- 时间的用途：很多算法依赖时间及时间同步
  - 事件排序；基于时间戳的并发控制；程序编译 (Make)
- 时间的获取
  - 时钟漂移 (clock drift)：时钟的绝对偏差
    - 漂移率：10^{-6} 秒/秒 (石英钟)，即每 100 0000 秒或 11.6 天有 1 秒的差别
  - 时间偏移 (clock skew)：两个时钟在读数上的瞬间不同
  - UTC (Universal Time Coordinated) $\Longrightarrow$ 协调世界时，世界标准时间

计算机时钟

- 计算机上的硬件时钟 H(t)
- 计算机上的软件时钟 $C(t) = \alpha \cdot H(t) + \beta$
- 分布式系统中的每个计算机有它自己的时钟
  - 本地进程可用它获得当前时间
  - 不同计算机上的进程能用本地时间给事件打时间戳
  - 不同计算机上的时钟可能给出不同的时间
  - 计算机时钟与完美时间有漂移，它们的漂移率个不相同
- 即使分布式系统中所有计算机的时钟被设成相同时间，它们的时钟如果不做校正最终也会相差很大

时间同步

- 时钟正确性：一个硬件时钟 H 是正确的，如果它的漂移率在一个已知的范围 $\rho > 0$ 内
  - 实际时间为 t'、t $(t' > t)$，$H(t')$、$H(t)$ 为硬件时钟值
  - 误差有界性：$(1-)(t'–t) \leq H(t')–H(t) \leq (1+)(t'-t)$
  - 单调性：$t' > t \Longrightarrow C(t') > C(t)$
  - 千年虫问题违反了单调性
- 在计算机系统中，如何解决时间快了或慢了
  - 快了回拨：违反了单调性
  - 慢了拨快：在同步点拨快
- 外部同步：用权威的外部时间源同步进程的时钟

- 内部同步：指时钟相互同步
  - 内部同步的时钟未必是外部同步的，因为即使它们相互一致，它们整体上也可能都与时间的外部源有偏差
  - 如果系统在范围 D 内是外部同步的，那么同一系统在范围 2D 内是内部同步的

同步系统 vs. 异步系统

- 同步系统
  - 已知时钟漂移率的范围
  - 已知最大的消息传输延迟
  - 已知进程每一步的执行时间
- 异步系统
  - 在进程执行时间、消息传递时间和时钟漂移上没有限制的
  - 不知道最大的消息传输延迟，最大的消息传输延迟可能是无穷
  - 最小的消息传递延迟是可知的
  - 例子：Internet

同步系统的时间同步

- 同步系统：最大的消息传输延迟是已知的，用 min 表示最小传输延迟，用 max 表示最大传输延迟
- 内部同步方法
  - 一个进程在消息 m 中将本地时钟的时间 t 发送到另一个进程
  - 接收进程能将它的时钟设成 t + Ttrans，其中 Ttrans 是传输 m 所花的时间
  - 如果接收方将它的时钟设成 t + min，那么时钟偏移至多 max - min ，因为事实上消息可能花了 max 时间才到达。
  - 如果将时钟设成 t + max，那么时钟偏移可能与 max - min 一样大
  - 如果将时钟设成中间点 t + （max + min）/2，那么时钟偏移至多 (max - min)/2
- 同步 N 个时钟?

*** 同步时钟的 Cristian 方法：一种外部同步方法**

- Cristian 算法：使用一个时间服务器 S，它连接到一个接收标准时间信号的设备上，用于实现外部同步
- 进程 p 在发给 S 的消息 m_r 中请求获得准确的时间，然后，从消息 m_t 中接收时间值 t。进程 p 记录了发送请求 m_r 和接收应答 m_t 的整个往返时间 T_{round}。进程 p 设置它的时钟时间为 $t + \frac{T_{round}}{2}$
- 如果 p 和 S 之间的最小传输时间 min 的值是已知，则应答消息到达时，S 的时钟的时间在 $[t+min, t+T_{round}-min]$ 范围。这个范围的宽度是 $T_{round} - 2min$，所以精确度是 $\pm(\frac{T_{round}}{2} - min)$

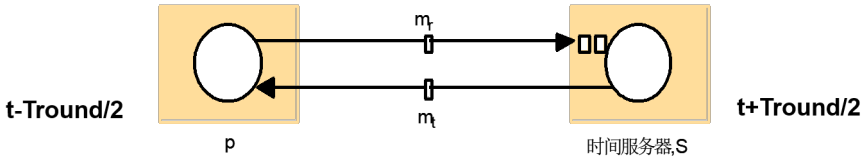

图 8：同步时钟的 Cristian 方法

*** 同步时钟的 Berkeley 算法：一种内部同步方法**

- 对一群运行 Berkeley UNIX 的计算机群，选择一台用作主机 (Master)。它周期性地轮询其他要进行时钟同步的计算机 (Slave)。Slave 将它们的时钟值返回给 Master。Master 通过观察往返时间 (类似 Cristian 的技术)，估计它们的本地时钟时间，并计算所获得值 (包括它自己时钟的读数) 的平均值
- Master 发送每个 Slave 的时钟所需的调整量
- Master 采用了容错平均值，在时钟中选择差值不多于一个指定量的子集，仅根据这些时钟的读数计算平均值，消除异常的往返时间
- Master 通常通过选举算法产生，也就是允许 Master 崩溃
- 同步的精确度取决于主-从之间的最大往返时间

网络时间协议-NTP

- NTP (Network Time Protocol) 定义了时间服务的体系结构和在 Internet 上发送时间信息的协议
- NTP 的设计目标：
  - 提供一个服务，使得 Internet 上的用户能精确地同标准时间同步：用统计方法来过滤时序数据
  - 使得客户能经常有效地重新同步以抵消在大多数计算机中存在的漂移率：该服务具有对大量客户和服务器的可伸缩性
  - 在不可靠的通信链接上提供一个可靠的服务：通过提供冗余的服务器以及服务器之间冗余的路径。如果其中一个不可达，服务器能重配置以便继续提供服务
  - 能防止对时间服务的干扰，无论是恶意的还是偶然的：使用认证技术
- 体系结构
  - 主服务器 (在根上) 直接连到外部时间源，例如短波时间接收器、GPS 接收器之类的设备，层次 (stratum)2 的服务器与主服务器同步，层次 3 的服务器与层次 2 的服务器同步等等
  - 同步子网是用 Bellman-Ford 路由算法的变种来组织的，构建以主服务器为根的最小权重的支撑树
- 故障处理
  - 如果主服务器的外部时间源出现故障，那么它能变成层次 2 的二级服务器

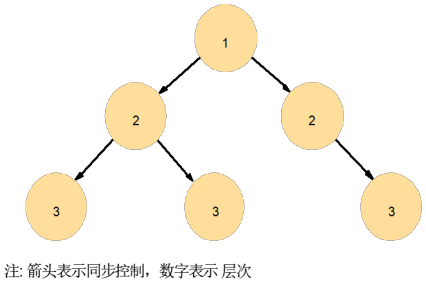

图 9：故障处理

- 同步方式：NTP 服务器按三种模式中的一种相互同步：组播、过程调用、对称模式
  - 组播模式用于高速 LAN 环境。一个或多个服务器周期性地将时间组播到其他服务器，并设置它们的时钟 (假设延迟很小)
  - 过程调用模式类似上述 Cristian 算法的操作。一个服务器接收来自其他计算机请求，并用时间戳 (本地的当前的时钟读数) 应答。这个模式适合精确性要求比组播更高的地方，或不能用硬件支持组播的地方

  - 对称模式用于在 LAN 中提供时间信息的服务器和同步子网的较高层 (较小的层次数)，即要获得最高精确性的地方。一对服务器相互交换有时序信息的消息。
- 消息传递都用 UDP
- 每个消息携带 3 个时间戳：发送前一个 NTP 消息的本地时间、接收前一个 NTP 消息的本地时间、发送当前消息的本地时间
- m 和 m' 实际的传输时间分别是 t 和 t'，di (roundtrip delay) 是两个消息整个的传输时间，设 B 上时钟相对于 A 的真正偏移量是 o，oi 是偏移 (clock offset) 的估计
- $T_{i-2} = T_{i-3} + t + o \; T_i = T_i - 1 + t' - o$
- $di = t + t' = T_{i-2} - T_{i-3} + T_i - T_i - 1$
- $o = oi + (t' - t)/2$ ，where $oi = (T_{i-2} - T_{i-3} + T_{i-1} – T_i)/2$
- 有 $oi – \frac{di}{2} \leq o \leq oi + \frac{di}{2}$

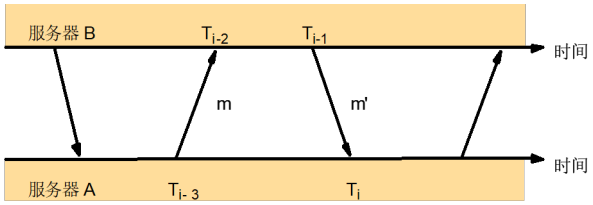

图 10：Data filter algorithm improve the offset estimate for a single clock
选具有最小 di 值的 oi 值用于估计 o

Lamport 算法

事件排序

- Lamport 的发生在先 (happened-before) 关系 (用 $\Longrightarrow$ 表示) 用于事件排序：
- HB1：如果 $\exists$ 进程 pi：$e \Longrightarrow_i e'$，那么 $e \Longrightarrow e'$
  - $\Longrightarrow i$：表示同一个进程中两个事件的顺序发生关系
- HB2：对任一消息 m，$send(m) \Longrightarrow receive(m)$
  - 其中 $send(m)$ 是发送消息的事件，$receive(m)$ 是接收消息的事件
- HB3：如果 e，e' 和 e'' 是事件，且有 $e \Longrightarrow e'$ 和 $e' \Longrightarrow e''$，那么 $e \Longrightarrow e''$

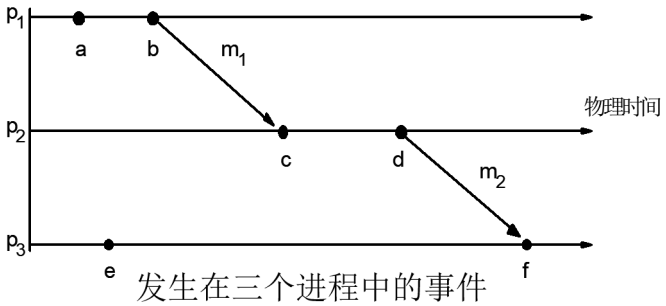

图 11：事件排序

发生在先关系：几点说明

- 不能由 $\Longrightarrow$ 排序的事件是并发的事件
  - a$\Longrightarrow$e 和 e$\Longrightarrow$a 都不成立，因为它们发生在不同的进程中，且它们之间没有消息链

- 关系 $\Longrightarrow$ 捕获了以消息传递方式表示的数据流
  - 不能对非消息传递方式的数据流动关系建模
- 如果发生在先关系在两个事件之间成立，那么第一个事件可能或不可能实际地引起了第二个事件
  - 关系 $\Longrightarrow$ 捕获可能的因果关系，两个事件即使没有真正的联系，也可以有 $\Longrightarrow$ 关系

***Lamport 逻辑时钟**

- 定义：一个逻辑时钟是一个单调增长的软件计数器
- 每个进程 pi 维护它自己的逻辑时钟 Li，进程用它给事件加时间戳
- 用 Li(e) 表示进程 pi 的事件 e 的时间戳，用 L(e) 表示发生在任一进程中的事件 e 的时间戳
- 逻辑时钟计算规则
  - LC1：在进程 pi 发出每个事件之前 Li 加一：$L_i := L - i + 1$
  - LC2：
    - 当进程 pi 发送消息 m 时，在 m 上捎带上值 t L_i
    - 在接收 (m, t) 时，进程 pj 计算 $L_j := max(Lj, t)$，然后将 receive(m) 事件的时间戳设置为 $L_j := L_j + 1$
- 逻辑时钟和事件的发生在先关系的关系
  - $e \longrightarrow e' \Longrightarrow L(e) < L(e')$
  - $L(e) < L(e')$，不能推出 $e \longrightarrow e'$

Example: 逻辑时钟

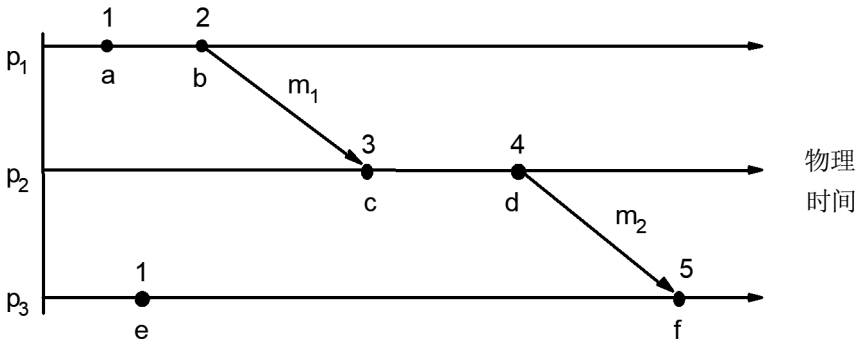

图 12：进程 p1，p2 和 p3 的逻辑时钟初始值为 0

L(e)< L(b) ，但没有 e⇒b，而是 b||e

向量时钟

- 定义：系统中有 N 个进程，每个进程 p_i 和一个向量 V_i [1..n] 相关联，用于给本地事件加时间戳
- 对向量时钟 V_i，V_i[i] 是进程 p_i 已经加了时间戳的事件的个数
- V_i[j](j⇒i) 是在 p_j 中发生的可能会影响 p_i 的事件的个数，进程 p_j 中在因果关系上处于当前 p_i 之先的事件计数
- 向量时钟计算规则：
  - VC1：初始，V_i[j] = 0，对 i,j=1,2,⋯, N
  - VC2：p_i 在给一个事件加时间戳之前，设置 $V_i[i] := V_i[i] + 1$

- VC3：p_i 在它发送的每个消息中包括值 t = V_i
- VC4：当 pi 接收到消息中的时间戳 t 时，设置 $V_i[j] := max(V_i[j], t[j])$(j=1,2,⋯,N)

Example：向量时钟

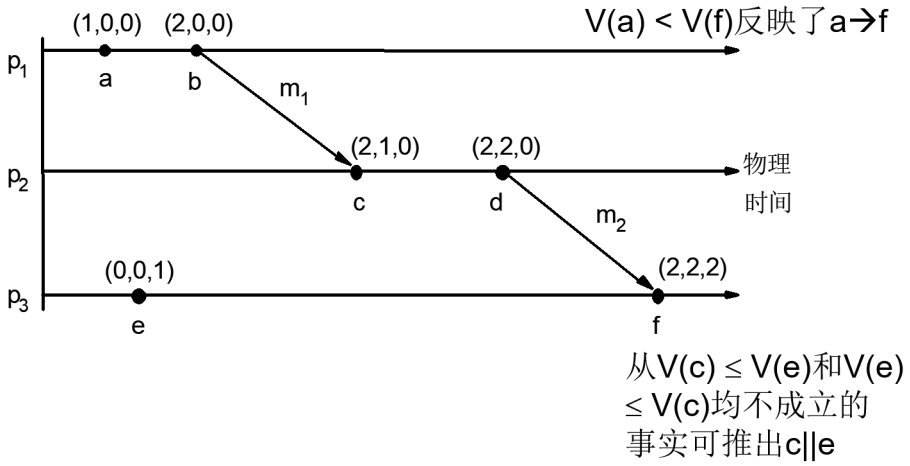

图 13：Example: 向量时钟

- 比较事件的向量时间戳的方法：
  - V=V' iff V[j] = $V'[j]$ (j=1,2,⋯,N)
  - V≤V' iff V[j] ≤ $V'[j]$ (j=1,2,⋯,N)
  - V<V' iff V ≤ $V' \wedge$ V ≠ V'
  - 可以在发生在先关系和向量时钟之间建立等价关系：
  - e→e' ←→ $V(e) < V(e')$

分布式系统的状态

- 观察分布式系统的全局状态的必要性
- 分布式无用单元收集：对象的引用情况是对象的一个状态
- 分布式调试：例如，判断某个进程 pi 的变量 x 和另一个进程 pj 的变量 y 的差值大于
- 死锁判定和程序终止判定：是一个全局状态谓词
  - Stable Property
  - Stable Property Detection
- 检查点 (Checkpointing)

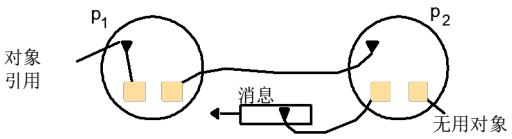

图 14：Example: 分布式系统的状态

全局状态

- 全局状态定义：分布式系统可以看作是一系列协同工作的进程集合 P=$\{P_1, P_2, ..., P_n\}$，进程可以是物理分布的，它们之间通过消息通信实现互操作。分布式系统的全局状态由局部状态集和消息通道状态集组成：

  - 一个局部状态是系统中的一个进程的全部变量集合
  - 消息通道状态是在消息传输中的消息序列
- 观察系统全局状态的困难性：缺乏全局时间
  - 如果所有进程有完全同步的时钟，那么就能规定一个让每个进程记录下它的状态的时间，结果就能获得系统实际的全局状态
- 能不能从在不同时间记录的本地状态汇总出一个有意义的全局状态？
  - 一个正常执行的分布式系统，它的执行可以描述成在系统全局状态之间的一系列变迁：S0 ⇒S1 ⇒S2 ⇒ ⋯。系统通过一致全局状态这种方式逐步发展
- 割集 (Cut) 用来说明什么是一致的全局状态

割集

- 进程的历史：每个进程的执行是在进程中发生了一系列的事件
  - 事件：进程内部事件，发送消息，接收消息
  - $history(p_i) = h_i =< e_i^0, e_i^1, e_i^2, ... >$
- 进程历史的任何一个有限前缀：
  - $h_i^k =< e_i^0, e_i^1, ..., e_i^k >$
- 全局历史：单个进程历史的并集，设总共 N 个进程
  - $\mathrm{H} = h_0 \cup h_1 \cup \cdots \cup h_{N-1}$
- 割集 C 是系统全局历史的子集，表示系统的执行状态：
  - $\mathrm{C} = h_1^{c1} \cup h_2^{c2} \cup \cdots \cup h_N^{cn}$
- 割集的边界：由系统中的进程 p_i 处理的最后一个事件的集合

一致的割集

- 一致的割集：如果对它包含的每个事件，它也包含了所有在该事件之前发生的所有事件
- 一致的全局状态：指相对于一致割集的状态

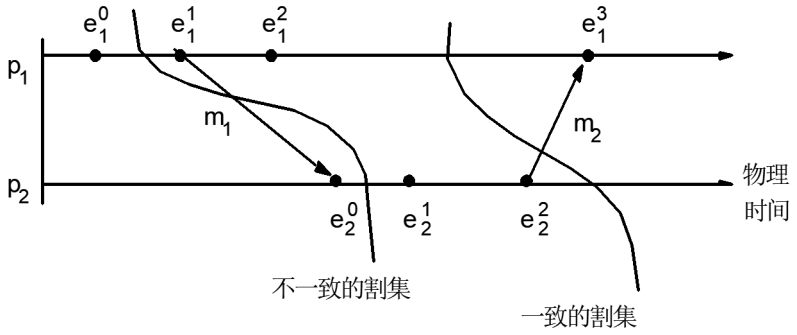

图 15：一致的割集

***Chandy 和 Lamport 的快照算法**

- 记录进程集的进程状态和通道状态
- 算法所记录的状态组合可能并没有在同一时间发生，但所记录的全局状态是一致的
- 算法假设：

  - 不论是通道还是进程都不出故障；通信是可靠的，每个发送的消息最终被完整地接收到一次
  - 通道是单向的，提供 FIFO 顺序的消息传递
  - 描述进程和通道的图是强连通的 (在任两个进程之间有一条路径)
- 算法的优点：
  - 任一进程可在任一时间开始一个全局快照
  - 在照快照时，进程可以继续它们的执行，可以发送和接收正常的消息
- 进程 pi 的标记接收规则

```
pi接收通道c上的标记消息(Marker Message):
if（pi还没有记录它的状态），
    pi记录它的进程状态;
    将c的状态记成空集;
    开始记录从其他的接入通道上到达的消息
else
    pi把c的状态记录成保留它的状态以来它所接收到的消息集合
end if
```

- 进程 pi 的标记发送规则

```
在pi记录了它的本地状态之后，对每个外出通道c:
（在pi 从c上发送任何其他消息之前）
pi在c上发送一个标记消息
```

- 算法启动时，认为从一个不存在的通道上接收到了一个标记
- 每个进程记录它的进程状态，进程还负责记录发送给每个接入通道的消息
- 进程记录每个接入通道的状态为：在进程自己记录下状态之后和在发送方记录下它自己状态 (由通道中的标记标识“发送方记录自己状态”这个行为) 之前到达的任何消息
- 关于这个算法的具体执行：算法可以在进程本地记录状态；也可以让所有进程把它们记录的状态发送到一个指定的进程进行状态收集

快照算法-执行举例

- 应用程序的功能：进程 p1 通过 c2 向 p2 发送窗口小部件的订单，并以每个窗口小部件 10 美元附上付款。过了一些时间，进程 p2 沿通道 c1 给 p1 发送窗口小部件。
- 当前状况：进程 p2 已经接收到五个窗口小部件的订单，它将马上分发窗口小部件给 p1。

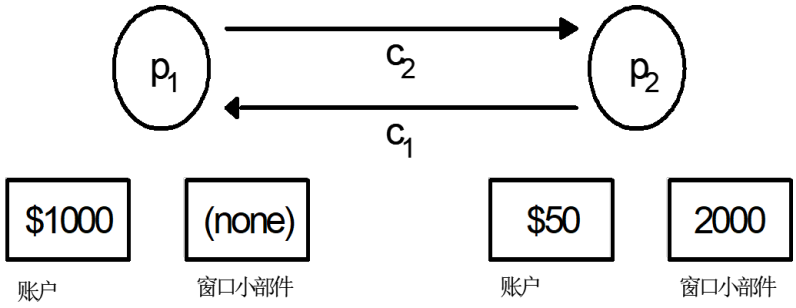

图 16：初始状态

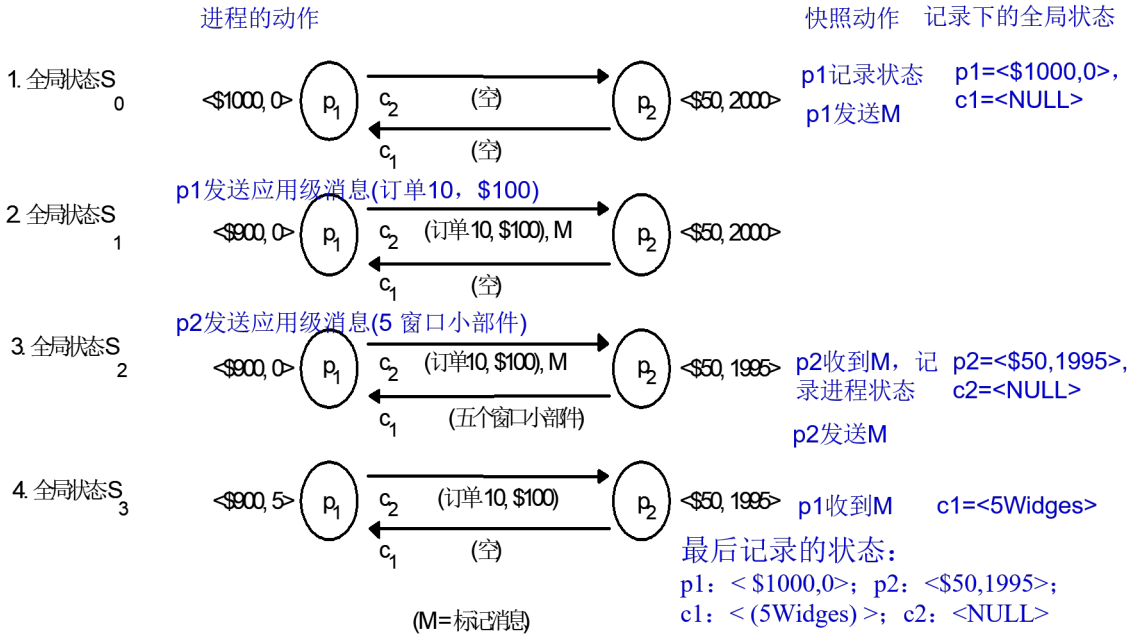

图 17：算法执行步骤

快照算法的性质

- 算法的终止性：如果某个进程发起记录它的状态，那么与它相关的所有进程会在有限的时间内记录它们的状态以及接入通道的状态
- 算法从执行历史中获得了一个一致的割集
  - 如果 ei⇒ej 并且 ej 在割集中，那么 ei 也在割集中
- 状态的可达性：Ssnap 从 Sinit 可达，Sfinal 从 Ssnap 可达
- a pre-snap event at process pi 是在 pi 记录其状态前发生的事件

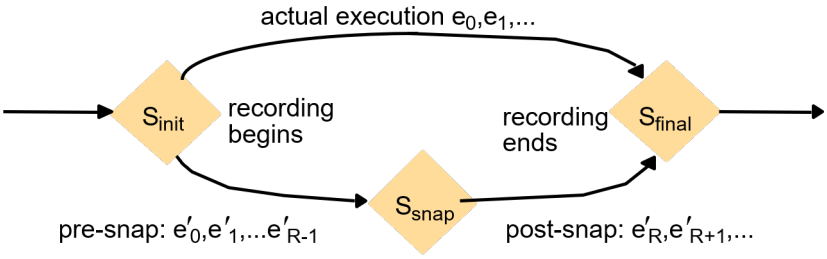

图 18：算法状态的可达性