引言

1 定义

分布式系统是独立的计算机的集合，对这个系统的用户来说，系统就像一台计算机一样。

2 为什么采用分布式的形式?

1. 性价比： 微处理器比大型机性价比高
2. 性能：分布式系统整个计算能力比单个大型主机要强
3. 固有的分布性：有些应用涉及到空间上分散的机器
4. 可靠性：如果其中一台机器崩溃，整体系统仍然能够运转
5. 可扩展性：计算能力可以逐渐有所增加

3 分布式系统的目标（需要详细解释）

1. 使资源可用
2. 透明性
3. 开放性
4. 可扩展性：

3.1 透明性

分布式系统中的透明性包括以下七个方面：

1. 访问透明：隐藏数据表示形式以及访问方式的不同
2. 资源位置透明：隐藏数据所在位置
3. 迁移透明：隐藏资源可能被移动到另一个位置
4. 重定位透明：隐藏资源可能在使用中被移动到另一个位置（即热迁移）
5. 多副本透明：隐藏资源是否已被复制（重点是多副本的一致性问题）
6. 并发透明：隐藏一个资源可能被多个用户竞争使用
7. 故障透明：屏蔽资源的故障和恢复

3.2 分布式系统的开放性

1. 能够与来自其他开放系统的服务交互，而不用考虑底层环境

- 一个良定义的接口
- 支持可移植应用（Portable Application）
- 易于互操作

2. 本身可以适配不同的操作系统，至少要独立于底层环境的异构性

3.3 分布式系统的可扩展性

1. 规模的可扩展性（用户或节点数量）
2. 地理位置的可扩展性（节点间的最大距离）
3. 管理域的可扩展性（管理域的数量）

4 策略与机制

机制是实现功能的具体执行结构的设计，是底层的设计，策略是达到目标功能而选择的参数、算法等抽象设计。从代码角度去理解，机制就是接口，策略就是对接口的具体实现。

5 分布式系统的分类

1. 分布式计算系统
2. 分布式信息系统
3. 分布式普适系统

5.1 分布式计算系统

5.1.1 集群计算

通过相对高速的本地网络形成的一个高性能的集群（通常采用计算存储分离的架构。

特点：同构性（相同的OS，几乎相同的硬件）、单一管理节点

5.1.2 网格计算

各地的限制离散资源组成的一个算力网络。

特点：异构性、分布在多个组织、轻松跨越广域网

5.1.3 云计算

IaaS、PaaS、SaaS

5.2 分布式信息系统

如分布式事务处理系统。

5.2.1 事务

事务是对对象的状态进行的操作的集合，满足ACID属性：原子性 Atomicity, 一致性 Consistency, 隔离性 Isolation, 持久性 Durability

5.3 分布式普适系统

移动计算系统
传感器网络

架构

1 分布式系统架构风格

1. 层次式（用于客户端-服务器系统中）
2. 对象式（用于分布式对象系统）
3. 总线式（这是一种松耦合的系统架构，采用事件触发），包括两种形式，一种是订阅-发布（空间上解耦），另一种是共享数据空间（时空上解耦）

2 分布式系统的组织形式

1. 集中式
2. 非集中式
3. 混合式

2.1 集中式架构（基本的C/S模型）

2.1.1 特点

1. 有提供服务的进程（服务器）
2. 有使用服务的进程（客户端）
3. 客户端和服务器可以位于不同的机器上
4. 客户端在使用服务时遵循请求/应答模式

2.1.1.1 具体形式

多客户端-单服务器架构存在的问题：

1. 服务器存在性能瓶颈
2. 服务器容易产生单点故障
3. 规模难以扩展

多客户端-多服务器架构

基于Web代理服务器的架构

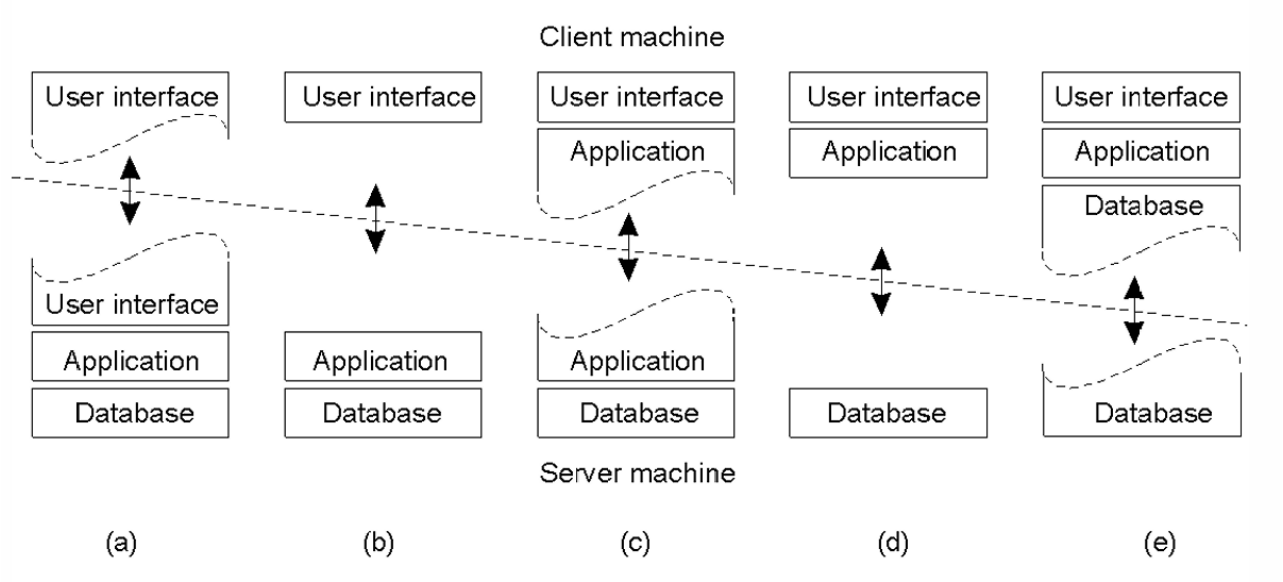
Web Applets：客户端请求服务器并下载Applets代码到本地，然后客户端和本地的Applets代码交互

2.1.2 应用的分层

传统的应用（许多的分布式信息系统，即利用数据库的应用程序）分为三层：

- 用户界面层提供应用程序的图形用户界面
- 处理层提供应用程序的功能，但不与具体的数据打交道
- 数据层提供客户端希望操作的数据

2.1.3 分层架构（就是把上面的三层怎么分配到客户端和服务器）



2.1.3.1 单层架构

系统的所有功能集中在一个设备上运行，通常由“哑终端”与主机组成，哑终端只负责显示和输入，不进行任何逻辑处理；所有逻辑运算和数据存储都在主机上完成。（典型的应用就是瘦客户端）

2.1.3.2 两层架构

由客户端和服务器组成，客户端提供用户界面并处理部分应用逻辑，服务器负责主要的业务逻辑和数据管理（如数据库）。

2.1.3.3 三层架构

每一层都在独立的机器上

2.2 非集中式架构（P2P）

2.2.1 分类

1. 结构化P2P：节点按特定数据结构组织
2. 非结构化P2P：节点随机选择邻节点
3. 混合式P2P：部分提供特定功能的节点按特定结构组织

2.2.2 结构化P2P

结构化P2P通常采用logical ring或hypercube的overlay来组织，每个节点提供根据ID提供特定的服务

2.2.3 非结构化P2P

大多数非结构化的P2P组成随机的overlay，两个节点之间按p的概率连接，查找数据时使用洪泛或随机游走。另有一种非结构化的P2P采用了簇结构，簇头间直接形成全连接。

2.2.4 混合式P2P

BitTorrent: Trackers（一旦某个节点确定了从哪里下载文件，它就会加入一群下载者，这些下载者并行地从源头获取文件块，但也将这些块相互分发）

Edge-Server结构（CDN：内容分发网络）

2.2.5 结构化 vs 非结构化

结构化的优势在于能够快速找到信息，可以设计一种算法在有界代价下找到信息，但是结构的维护成本太高了。

3 中间件

中间件是分布式系统中的一个重要组成部分，它负责不同组件之间的通信、协调与数据管理。

在许多情况下，分布式系统/应用程序是根据特定的架构风格开发的。然而，所选择的架构风格并不一定在所有场景下都最优，因此需要（动态地）调整中间件的行为。

例如：使用拦截器在调用远程对象时**拦截正常的控制流**，执行额外的操作或逻辑。

正常流程:
客户端 → 远程对象调用 → 执行方法 → 返回结果

有拦截器的流程:
客户端 → **拦截器（执行附加逻辑）** → 远程对象调用 → **拦截器（执行附加逻辑）** → 返回结果

4 自我管理的分布式系统

在许多场景下，自我管理的系统被建模为一个负反馈系统

进程

1 进程

1.1 定义

进程是进程状态上下文中的执行流

执行流包括：

- 执行指令流
- 运行代码片段
- 连续的指令序列
- “控制线程”

进程状态是运行代码可以影响或受其影响的所有内容

1.2 进程和程序的区别

程序是静态的代码和数据，进程是代码和数据的动态实例；

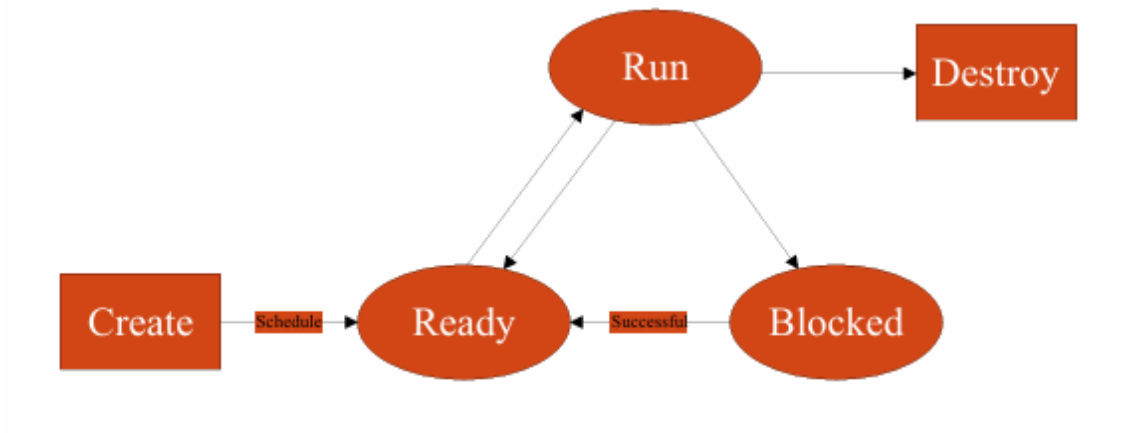
程序和进程之间没有一一对应地关系，一个程序可以有多个进程，一个进程可以调用多个程序

1.3 进程的三态

运行态：进程正在 CPU 上执行（在单核处理器上一次只能有一个进程处于此状态）。

就绪态：进程已准备好执行，但由于 CPU 正在被其他进程占用，因此等待 CPU 的分配。

阻塞态：进程正在等待 I/O 操作或某些同步操作完成，无法继续执行。



1.4 为什么进程是低效的

源于进程的资源管理，进程创建时需要分配地址空间，复制数据；调度时，需要决定进程在哪个CPU上执行；上下文切换时需要切换CPU上下文和存储上下文；进程间写作需要通过IPC或者共享内存的方式。

2 线程

2.1 定义

线程是一个最小的软件处理单元，在其上下文中可以执行一系列指令。线程是执行单元，保存线程上下文允许线程暂停执行并在未来某个时间点继续运行。

2.2 线程与进程的区别

线程是轻量级进程，一个进程包含多个线程，共享数据(地址)空间，而进程之间不共享代码和数据空间。进程是分配资源的基本单位，而线程是独立运行和独立调度的基本单位。线程的切换比进程效率高。

2.3 线程的分类

2.3.1 用户级线程

线程在用户进程的地址空间中创建，虽然效率高，但在多线程调度和阻塞处理上存在局限性。

2.3.2 内核级线程

将线程管理移到内核中，虽然增强了线程调度和阻塞处理的能力，但可能会因为频繁的系统调用导致效率下降。

3 虚拟化

3.1 为什么需要虚拟化

1. 硬件变化速度快于软件，有助于硬件和软件的解耦
2. 便于移植和代码迁移
3. 隔离故障或受攻击的组件

3.2 虚拟机的分类

3.2.1 进程虚拟机

进程虚拟机是一种程序执行环境，程序首先被编译成中间代码（便于移植的代码），然后由运行时系统（如虚拟机）执行。如：Java虚拟机

3.2.2 虚拟机监视器

虚拟机监视器是一个独立的软件层，它模拟硬件的指令集，从而支持一个完整的操作系统及其应用程序。如：Xen、KVM、VMware

4 代码迁移

4.1 代码迁移的内容

代码段：存储程序的实际代码或指令。

数据段：存储程序执行时所需的状态数据。

执行状态：包含当前执行线程的上下文信息

4.2 强迁移和弱迁移

弱迁移只迁移只迁移代码段和数据段，迁移后需要重启，最后被目标进程或者另外一个独立的进程执行。

强迁移需要迁移代码段、数据段和执行状态。有两种方式实现，迁移：将整个对象从一台机器移动到另一台机器；克隆，启动一个克隆，并将其设置为相同的执行状态。

4.3 本地资源

对象使用的本地资源在目标机器上不一定可用

4.3.1 本地资源的类型

固定资源：无法迁移的资源

固定绑定资源：原则上可以迁移，但迁移成本高的资源

未绑定资源：可以轻松迁移的资源，如缓存

4.3.2 对象与资源的绑定

按标识：对象需要特定实例的资源，例如特定的数据库或硬件设备。

按值：对象只需要资源的具体值，而不关心资源的来源，例如缓存中的一组数据条目。这类需求只需提供相应的数据内容即可。

按类型：对象只要求特定类型的资源可用，而不在乎具体实例。例如，需要一个颜色显示器，但不限定具体品牌或型号。

4.3.3 迁移方法

	未绑定	固定绑定	固定
标识	移动或全局引用	全局引用或移动	全局引用
值	值复制、移动或全局引用	全局引用或值复制	全局引用
类型	重绑定到本地可用资源、移动或全局引用	重绑定到本地可用资源、移动或全局引用	重绑定到本地可用资源或全局引用

4.4 异构系统上的迁移

4.4.1 主要问题

- 目标计算机可能不适合执行迁移的代码
- 进程/线程/处理器上下文的定义高度依赖于本地硬件，操作系统和运行时系统

4.4.2 解决方案

使用抽象机器的概念，能够在不同平台上实现统一的程序执行环境：

- 解释型语言（通常通过自己的虚拟机）
- 应用在虚拟机上运行，虚拟机负责屏蔽底层的差异

4.5 代码迁移与虚拟迁移的比较

代码迁移：优点在于迁移的数据量较小，仅限于代码和相关依赖。但需要适配目标环境，可能涉及依赖解析、状态同步以及平台兼容性问题。

虚拟迁移：通过虚拟机技术，将整个虚拟机环境（包括操作系统、应用程序和运行时状态）迁移到另一个节点，而无需直接处理代码或状态的迁移细节。优点在于目标系统不需要预适配，因为整个环境被迁移。但迁移过程可能占用更多的带宽和存储资源。

通信

1 通信的类型

1.1 分类

瞬时通信：发送者将消息放在网络上，如果无法传递给发送者或下一个通信主机，则消息会丢失。

持久通信：信息存储在通信系统中，直到信息传递给接收者。

异步通信：消发送方在执行发送操作后立即继续执行截下来的步骤，消息被存储在发送方的本地缓冲区或第一个通信服务器（路由器或消息中转节点等）。

同步通信：发送方将被阻塞，直到其消息存储在接收主机的本地缓冲区中或传送给接收方。

1.2 以C/S架构为例

C/S架构通常使用瞬时同步通信模型，客户端和服务器必须在通信时都处于活跃状态，才能完成请求-响应过程。客户端发出请求后，会暂停（阻塞）当前操作，直到服务器返回响应结果为止。服务器一般处于空闲状态，等待客户端请求，然后对请求进行处理并返回结果。

局限性：

客户端无法在等待期间执行其他任务

需要立即处理故障，如果服务器发生故障或者通信中断，客户端会长时间等待，导致系统性能下降，甚至完全失效。

模型不适用于某些场景，如**邮件**、**新闻推送**等场景，不需要即时响应的任务，瞬时同步通信显得不合适。

1.3 消息传递

消息传递是一种高层次的**持久异步通信**方式，允许进程之间通过消息队列进行信息交换。发送者将消息发送到**消息队列**中，消息会被存储。发送者发送消息后，不会阻塞等待回复，可以立即继续执行其他任务，提高系统并行性和效率。通常由**消息中间件**管理消息队列，提供**故障容错**能力，确保消息的可靠传递。

2 远程过程调用（RPC)

2.1 RPC的过程

客户端以正常的方式调用一个函数，这个调用实际上被重定向到客户端的存根（Client Stub）程序

客户端存根构建消息并调用本地操作系统的网络服务

客户端操作系统将消息发送到远程操作系统

远程操作系统将消息交给服务器存根（Server Stub）

服务器存根解包参数并调用服务器端程序

服务器执行请求并返回结果

服务器存根打包结果并调用本地操作系统

服务器操作系统将消息传递给客户端操作系统

客户端操作系统将消息传给客户端存根

客户端存根解包结果并返回给客户端

2.2 参数传递

在RPC中进行参数传递，需要解决以下问题：

1. **数据表示统一**：通过序列化确保客户端和服务器之间的数据兼容性。
2. **Copy in/Copy out**：采用值传递，避免传递引用带来的问题。
3. **访问透明性**：完全的访问透明无法实现，但可以通过**远程引用**机制提升远程数据访问的透明度。

2.3 故障处理

2.3.1 客户端无法定位服务器

- 抛出异常或使用特殊的返回值

2.3.2 客户端向服务器发送的请求消息丢失

- 超时则重新请求
- 对于不幂等的请求，编上序号让server能识别重复请求

2.3.3 服务器向客户端发送的回复消息丢失

- 超时则重新请求
- 对于不幂等的请求，编上序号让server能识别重复请求

2.3.4 服务器在收到请求后崩溃

两种情况: 执行前崩溃和执行后崩溃

- 重启server并重新进行处理：保证至少执行一次(at-least-once)
- 立即放弃并报告错误：保证至多执行一次(at-most-once)
- 什么都不做

2.3.5 客户端在发送请求后崩溃

当客户端发生崩溃时，**RPC（Remote Procedure Call）** 可能会留下**孤儿进程**（Orphan Computations），这些进程在服务器端继续运行，但客户端已无法控制它们。这种情况可能导致资源浪费或不一致性。因此，需要一种机制来管理和清理这些孤儿进程。

- 灭绝法：在客户端存根发送 RPC 请求之前，它会在**安全存储**（如磁盘日志）中记录即将执行的操作。如果客户端崩溃并重启，系统会检查日志，并**显式终止**那些可能成为孤儿的远程进程。
- 重生法：将时间划分为一系列**编号的时期（Epochs）**。当客户端重启时，它会**广播**一个新时期开始的消息。收到广播的所有远程服务器会立即**终止当前的远程计算**。
- 温和重生法：当新时期的广播消息传入时，每个机器会尝试**定位远程计算的所有者**（客户端）。如果无法找到计算的所有者，才会**终止**该远程计算。
- 超时法：为每个 RPC 设置一个固定的**超时时间 T**。如果 RPC 在 T 时间内未完成，必须**显式请求**额外的时间量（Quantum）来继续执行。超时后，远程计算会自动终止。

2.4 客户端如何定位服务器

2.4.1 硬编码

将服务器地址硬编码到客户端快速但不灵活

2.4.2 动态绑定

服务器在启动时会向一个绑定服务注册自己。这个绑定服务会跟踪活动服务器及其地址。

优点：

- 灵活
- 支持多服务器提供相同的接口（负载均衡、自动取消注册失败的服务器以达到一定的容错性、身份认证）
- 绑定服务可以识别是否客户端和服务器使用的是相同版本的接口

缺点：

- 导入/导出接口会带来额外的开销
- 绑定服务可能成为大型分布式系统的瓶颈

3 基于消息的通信

3.1 持久化消息（消息队列）

持久化消息系统通常被称为消息队列系统，主要特点有：

1. 发送端和接收端都有一个队列
2. 发送与接收解耦，当发送方将消息放入队列时，接收方无需在线，同样，接收方在从队列中取出消息时，发送方也不需要在线。

消息队列系统假设使用一个共同的消息协议：所有应用程序都必须同意消息的格式，包括消息的结构和数据表示方式。这样，系统之间的消息传递可以保证兼容和一致性。

消息代理是消息队列系统中的集中组件，负责转换传入的消息为目标格式、充当应用程序网关、提供基于主题的路由功能以支持企业应用集成等

3.2 基于消息中间件的异步持久通信

通过中间件级队列可以实现异步持久化通信，在这种通信方式中，队列被用作缓冲区，存在于中间件或通信服务器中，支持异步和持久化的消息传递。

4 面向流的通信

4.1 连续媒体与离散媒体

离散媒体指的是与时间无关的数据

连续媒体指的是音视频、动画、传感器等时间有关的数据

4.2 不同传输方式的时间保证

异步通信（离散媒体）：没有关于数据交付时间的限制。

同步通信（连续媒体）：定义了数据包的最大端到端延迟。

等时性通信（连续媒体）：定义了数据包的最大和最小端到端延迟

4.3 流

4.3.1 定义

数据流是一种面向连接的通信方式，支持等时性数据传输。

4.3.2 特性

1. 流是单向的
2. 通常有一个源和一个或多个接收端
3. 源或接收端通常是硬件的封装（例如对相机的数据进行封装获取视频流）
4. 流分为简单流和复杂流。简单流是指仅包含单一数据流的情况，如音频流、视频流。复杂流包含多个数据流。如立体声流、音视频流。

4.3.3 流的QoS

流的质量服务（QoS） 主要关注数据的及时交付，通常通过以下几个基本参数来定义和衡量：

1. 数据传输所需的比特率
2. 最大会话建立延迟（在流开始传输之前，从发起数据流的应用程序到会话完全建立所需的最大时间）
3. 最大端到端延迟（数据从发送端到接收端的最长传输时间）
4. 最大延迟抖动
5. 最大往返延迟

4.3.3.1 如何保证QoS

1. 差异化服务，对数据包进行优先级管理，确保高优先级的数据包能够在网络中获得优先传输
2. 利用缓冲区减少延迟抖动
3. 使用交错传输来降低丢包的影响（丢包是随机间隔着丢，而不是把一整块全丢了）

4.3.4 流同步

流同步是指在复杂流中保持多个子流之间的同步，以确保它们能够协调一致地传输数据。

实现同步的方式：

1. 复杂流中的所有子流先通过多路复用的方式合并成一个单一的数据流。
2. 在接收端，数据流会被解复用分离出各个子流
3. 对分离的各个子流进行同步处理

4.4 组播通信

4.4.1 应用层组播

将分布式系统的节点组织成Overlay，并使用Overlay传播数据

4.4.2 基于病毒扩散的传播方式

4.4.2.1 两种机制

4.4.2.1.1 反熵

每个副本会定期随机选择另一个副本，并交换它们之间的状态差异。交换完成后，这两个副本会达到一致的状态。

4.4.2.1.2 Gossip

当某个副本的数据被更新后，它会主动将这次更新告诉一定数量的其他副本，从而将更新逐步传播到整个系统中的所有副本。

4.4.3 数据删除的注意事项

不能直接从服务器中删除旧值，否则删除操作会被算法撤销，需要插入死亡证明来将删除作为特殊的更新。

同步

1 为什么要进行同步

- 保证多个进程不会同时访问共享资源
- 保证多个进程可以相互达成一致

2 单机系统与分布式系统中的同步

单机系统中同步问题可以通过信号量等方法解决，然而，这些方法无法在分布式系统中使用，因为它们隐式依赖于共享内存的存在。如果在分布式系统中发生两个事件，很难确定哪个事件先发生（因为两个机器的本地时钟可能不同）。

3 逻辑时钟与物理时钟

逻辑时钟：关注的是事件发生的顺序，而不是时间的绝对值。

物理时钟：要求时钟不仅一致，而且与真实时间保持同步。

4 物理时钟的同步

4.1 基本原理

假设在一个有UTC接收器的分布式系统中，需要将时间分发到每个机器。每台机器都有一个定时器，每秒生成H次中断。每台机器上有一个时钟，记作C(t)，其中t是UTC时间，理想情况下，对于每台机器p，C(t) = t。实际中，时钟并不完美同步，可能会有轻微的偏差。$1-\rho \leq dC/dt \leq 1+\rho$，其中$\rho$代表时钟偏差（每秒慢或者快$\rho$的时长，所以经过T秒，最坏情况下差距就是$T*2\rho$）。为确保所有机器的时钟保持相对同步，两台时钟之间的差异永远不应超过δ时间单位。因此，时钟应至少每隔$\delta/(2\rho)$秒进行一次同步，以保持两台时钟之间的差异在可接受范围内。

4.2 Cristian's 算法

假设有一个时间服务器，其他所有机器与时间服务器保持同步。

Cristian's的算法基于客户端向时间服务器请求时间并通过传播延迟（propagation delay）来估算准确时间的方式。

1. 客户端在本地$T0$时刻向时间服务器发起请求
2. 时间服务器经过I秒的处理后响应返回UTC时间，客户端在本地$T1$时间得到响应
3. 传播时延可以被估计为$(T1-T0-I)/2$

注意，由于时间不可以倒退，所以不能直接把客户端的时间设置成UTC时间，而是逐步地进行调整。

4.3 Berkeley算法

适用于没有精确时钟的情况，time daemon 主动询问其他所有机器的时间，计算平均时间作为标准，然后广播给每台机器让每台机器调整时钟

4.4 Averaging算法

首先将时间划分成长度为R的同步间隔，每个间隔从T0+iR开始到T0+(i+1)R结束，T0提前商议，R是系统参数。

每台机器在间隔开始的时候广播其本地时钟

一旦机器广播了时钟，就启动一个定时器，在时间间隔内收集来自其他机器的所有广播

当所有来自其他机器的广播到达后，根据这些广播计算新时间（最简单就是平均，变体比如估算传播时间、丢弃极端值）

4.5 NTP协议

基于分层客户端-服务器模型的，并使用 UDP 消息传递。

服务器被分为多个等级（Strata）。**Strata 1** 级别的时间服务器直接与标准时间源（例如原子钟或 GPS 信号）同步，而 **Strata 2** 级别的服务器则与 **Strata 1** 级别的服务器同步，以此类推。随着层级的增加精度会稍微降低。

容错性：如果一个 Strata 1 级别的服务器失败，它可以退回为一个 **Strata 2** 级别的服务器，并通过另一个 Strata 1 级别的服务器进行同步。

同步模式：

组播模式：一台计算机定期广播时间信息给网络中的多个客户端

过程调用模式：类似Cristian 's算法

对称模式：通常用于需要高精度时间同步的系统

5 逻辑时钟的同步

5.1 基本原理

为了在分布式系统中维护一个全局视图，且该视图与 Happened-Before*关系一致，我们可以为每个事件附加一个时间戳 C(e)，并遵守以下性质：

1. 如果a和b是同一进程中的两个事件，如果a比b先发生，则C(a) < C(b)
2. 如果a是发送消息m的事件，b是接收该消息的事件，则 C(a) < C(b)

在没有全局时钟的情况下，我们无法直接使用物理时钟来为事件添加时间戳。因此，我们需要维护一组一致的**逻辑时钟**，每个进程都有自己的逻辑时钟。进程内部发生事件时，进程的逻辑时钟递增。对于消息传递，接收方进程的逻辑时钟会根据接收到的时间戳调整，以保持一致。

5.2 Lamport时钟

每个进程使用一个本地计数器来实现逻辑时钟，并按照以下规则调整该计数器：

1. **同一进程中的事件递增**：对于进程 P_i 中的任何两个连续事件，C_i（本地计数器）会递增 1。
2. **发送消息时附加时间戳：**每当进程 P_i 发送消息 m 时，消息会被附加一个时间戳 $t(m) = C_i$
3. **接收消息时调整时间戳**：当进程 P_j 接收到消息 m 时，进程 P_j 会调整它的本地计数器 C_j 为 $\max\{C_j, t(m)\}$，即取接收到的消息时间戳与本地计数器中的较大值，然后递增本地计数器。

即使应用了这些规则，也可能发生两个事件的时间戳相同的情况（例如，两个进程同时发送消息）。为了避免这种情况，可以通过使用 进程 ID 来打破平局，使得即使时间戳相同，事件也可以根据进程的唯一标识符（ID）进行排序。

注意，Lamport时钟得到的是偏序关系，而没有因果关系，不能直接通过比较时间戳的大小来判断两个事件发生的先后顺序。

5.3 向量时间戳

每个进程Pi维护一个向量VCi，其中VCi[j]表示进程Pi已知的在Pj中发生的事件数量。

- 初始时所有时钟都为0；
- 每一次处理内完内部事件，将本进程对应的逻辑时钟+1；
- 每一次发送一个消息的时候，需要将自己的向量时钟和消息一起发送；
- 每一次接收到一个消息，需要将自己的逻辑时钟+1，同时更新每一个逻辑时钟，更新规则为取本地逻辑时钟和收到的逻辑时钟的最大值。

向量时钟算法当中，我们定义如果事件A在事件B之前，那么需要满足两个条件：

- 对于所有的下标K，都有VCA[K] <= VCB[K]
- 存在下标K0，使得VCA[K0] < VCB[K0]

6 分布式系统中的资源互斥

多个进程需要独占访问某个资源时，可以通过以下方案实现互斥：

6.1 集中式

方法：

- 使用单个决策进程，称为协调者
- 请求资源的进程向 协调者 请求 permission
- 若资源被占用，可能 block，也可能返回错误消息

优点：

- 保证互斥访问
- 公平性
- 无饥饿现象
- 易于实现（只需要三种：请求、许可、释放）

缺点：

- 协调者易产生单点故障
- 协调者成为性能瓶颈
- 如果进程在发送请求后一直阻塞，无法区分是因为协调者崩溃还是“访问被拒绝”

6.2 分布式算法

方法

- 当某进程希望进入临界区时，将请求和时间戳作为消息发送给所有其他进程
- 当某进程收到另一个进程的请求消息时，如果接收方未在临界区，也不想进入临界区，向请求方发送 OK 消息；如果接收方已经在临界区，不回复请求，而是将该请求排队等待处理；如果

接收方想进入临界区但尚未进入，比较接收到的消息的时间戳与自己发送给所有人的消息中的时间戳，如果接收到的消息的时间戳较小，接收方向请求方发送 OK 消息，否则，将该请求排队，且不发送任何消息。

存在的问题

- 单点故障被替换为多个故障点的影响，如果任何一个进程崩溃，它将无法响应请求。这种**沉默**会被错误地解释为“拒绝许可”，从而导致**所有进程的后续尝试全部被阻塞**，无法进入任何临界区。
- **组成员管理的复杂性**：如果没有可靠的组播机制，每个进程必须自己维护组成员列表，包括：新进程加入组、进程退出组、进程崩溃。
- 比**集中式算法**更加**缓慢、复杂、昂贵**，且**鲁棒性更差**

6.3 基于令牌环的方法

系统中的所有进程按照逻辑顺序排列成一个环，环中传递一个特殊的“令牌（Token）”，持有令牌的进程有权进入临界区（如果它需要进入）。

但需要额外机制检测令牌是否丢失，并在必要时重新生成令牌

6.4 三个算法之间的比较

算法	平均进入/退出的消息数	进入前的延迟（消息数）	问题
集中式	3	2	协调者崩溃
分布式	2*(n-1)	2*(n-1)	任一节点崩溃
令牌环	1~∞	0~n-1	令牌丢失或进程崩溃

7 选举算法

在分布式系统中，通常需要某个进程充当**协调者**（Coordinator）。选举算法的目标就是动态选择这个协调者进程。

7.1 霸凌式选举

在这个算法中，每个进程都有一个与之关联的优先级（或称为权重）。目标是选举出权重最大的进程作为协调者。问题是：如何在一开始不知道各个进程的权重的情况下，找到权重最大的进程?

- 启动选举：任意一个进程可以主动启动选举过程，向所有其他进程发送选举消息。
- 接收到选举消息的进程：如果一个进程P_{high}收到来自一个权重较低的进程P_{low}的选举消息，它会向P_{low}发送一个**接管**（take-over）消息“OK”，表示 P_{low}被淘汰出局。
- 如果某个进程没有收到任何接管消息，它就认为自己是权重最大的，因而胜利，并向所有其他进程发送**胜利消息**。

7.2 环算法

在环形选举算法中，进程按照某种逻辑形成一个环形结构。

- 启动选举：任意一个进程可以启动选举，它向自己的后继进程发送一个选举消息。
- 处理消息传递：如果某个进程的后继进程故障（down），则选举消息将传递给下一个后继进程，直到消息到达一个存活的进程。每当一个进程接收到选举消息时，它会将自己加入到消息的列表中。消息会继续传递，直到回到启动选举的进程。

- 返回选举结果：一旦消息回到最初的发起进程（即发送选举消息的进程），这时所有存活的进程都已把自己列入了消息列表中。根据优先级发送 COORDINATOR 消息通知所有人谁是领导者。

如果两个进程同时开始选举，不影响时间复杂度，只是占用带宽增加

副本与一致性

1 副本的优点和缺点

优点：

- 可靠性，可以避免单点故障
- 性能：在服务器数量和地理区域上具有可扩展性

缺点：

- 副本的透明性实现复杂
- 存在一致性问题，更新开销大，并且不小心会影响可用性

2 副本一致性模型

2.1 数据为中心的一致性模型

描述了逻辑数据存储的总体组织形式，其特点是**物理上分布并在多个进程中实现复制**。

2.1.1 非同步的一致性模型

2.1.1.1 严格一致性

2.1.1.1.1 定义

任意`read(x)`操作必须返回最近一次`write(x)`操作写入的值

2.1.1.1.2 要求

依赖**绝对的全局时间**。

所有写操作必须立即对所有进程可见。

保证操作的全局时间顺序一致。

2.1.1.1.3 局限性

在**分布式系统中无法实现**

2.1.1.2 线性一致性

线性一致性也叫严格一致性（Strict Consistency）或者原子一致性（Atomic Consistency），它的条件是：

2.1.1.2.1 要求

所有进程的读写操作以某种**串行顺序**执行，且每个进程的操作顺序与程序指定的顺序一致。

如果操作 `op1(x)` 的时间戳小于 `op2(y)`，则 `op1(x)` 在全局顺序中必须出现在 `op2(y)` 之前。

2.1.1.2.2 特点

需要基于全局时间戳进行同步，代价高昂。

主要用于程序的**形式化验证**，而不适用于实际分布式系统的实现。

2.1.1.3 顺序一致性

2.1.1.3.1 定义

类似于线性一致性，但**不要求遵守严格的时间戳顺序**。顺序一致性使用的是逻辑时钟来作为分布式系统中的全局时钟，进而所有进程也有了一个统一的参考系对读写操作进行排序，因此所有进程看到的数据读写操作顺序也是一样的。

2.1.1.3.2 要求

所有进程对数据存储的读写操作必须按照某种**顺序执行**；

每个进程的操作在全局顺序中必须与其程序中的操作顺序一致。

2.1.1.3.3 效果

所有进程能够观察到相同的操作交错顺序，类似于**可串行化**。

2.1.1.4 因果一致性

因果一致性是一种弱化的顺序一致性模型，因为它将具有潜在因果关系的事件和没有因果关系的事件区分开了。

2.1.1.4.1 条件

写操作之间如果存在**潜在的因果关系**，那么所有进程必须以**相同的顺序**观察到这些写操作。

如果写操作是**并发的**，不同机器可以以**不同的顺序**观察这些写操作。

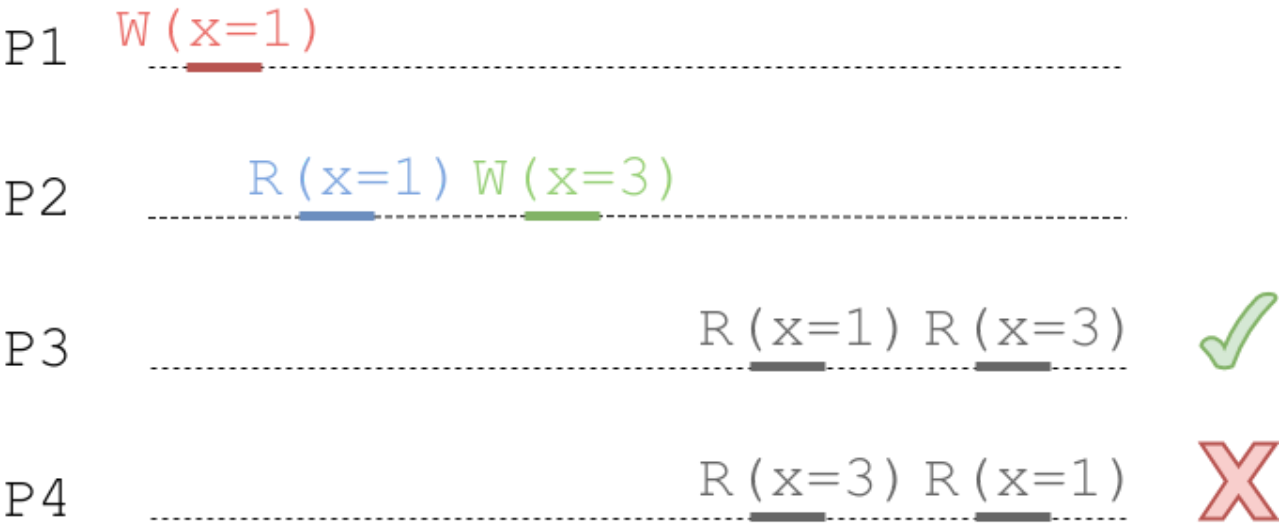
什么是潜在的因果关系?

如果事件 B 是由事件 A 引起的或者受事件 A 的影响，那么这两个事件就具有因果关系。

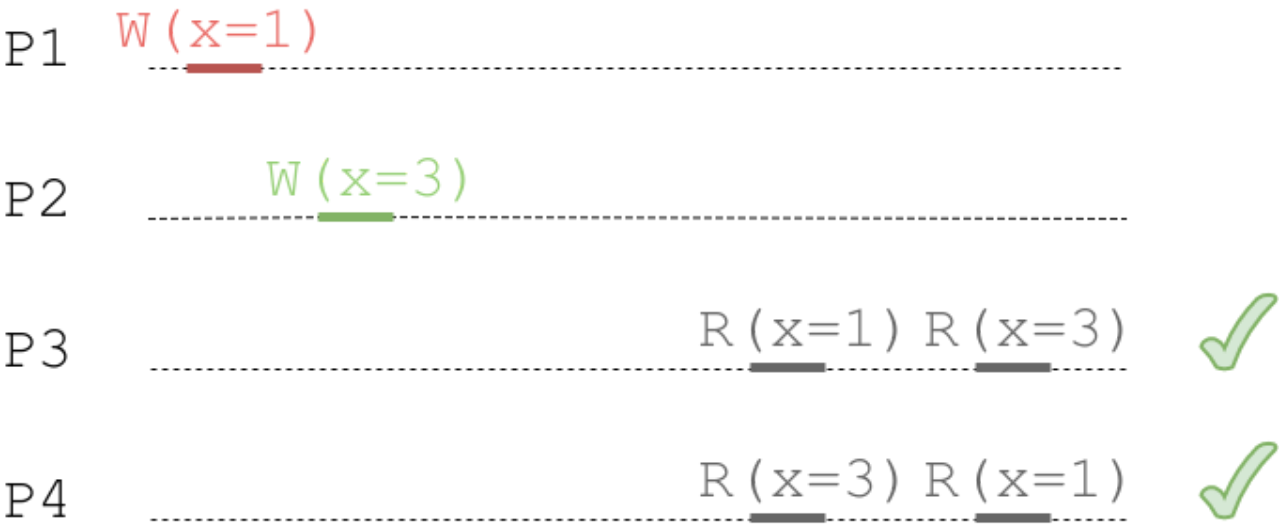
假设进程 P1 对数据项 x 进行了写操作，然后进程 P2 先读取了 x，然后对 y 进行了写操作，那么对 x 的读操作和对 y 的写操作就具有潜在的因果关系，因为 y 的计算可能依赖于 P2 读取到 x 的值（也就是 P1 写的值）。

2.1.1.4.2 举个例子

比如下图中，我们认为 **P2 写入的 3 是基于它读出来的 1 计算出来的**，它读出来的 1 又是由 P1 的写入产生的，因此认为 P1 写入 1 和 P2 写入 3 具有因果关系。P4 没有观测到这个因果关系，所以这个系统不具备Causal Consistency。

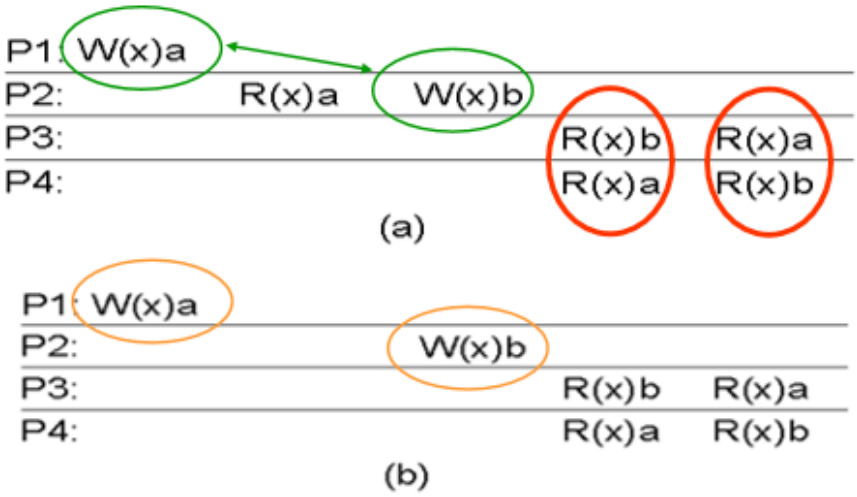


而下图中，认为P2写入3和P1写入1不具有因果关系，则P4和P3可以以任意顺序观测到它们。这个系统仍然可以说具有Causal consistency，但是不具备Sequential Consistency。



再例如下面这个图，（a）中，P2 先读了x，读出来的结果是a。我们认定接下来写入的 x=b 的结果是基于之前读出来的 a 计算出来的，因此 P1 与 P2 就有了因果关系，则 P3 和 P4 读到的结果顺序则应当遵循因果一致性，即先读到 x=a，再读到 x=b。

而（b）中，则认为 P1 与 P2 是并发的，则 P3 和 P4 读到什么顺序的结果都是无所谓的。



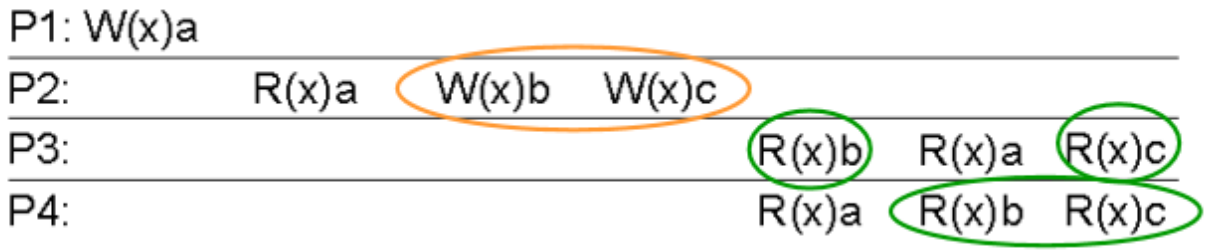
a) A violation of a casually-consistent store. $W_2(x)b$ may be related to $W_1(x)a$
b) A correct sequence of events in a casually-consistent store. $W_1(x)a$ and $W_2(x)b$ are concurrent

2.1.1.5 FIFO一致性

2.1.1.5.1 条件

- 一个进程所做的所有的写操作是被其他的进程按照这些写操作发出的顺序看到的。
- 但来自**不同进程的写操作**可能被**不同的进程**以**不同的顺序**看到的。

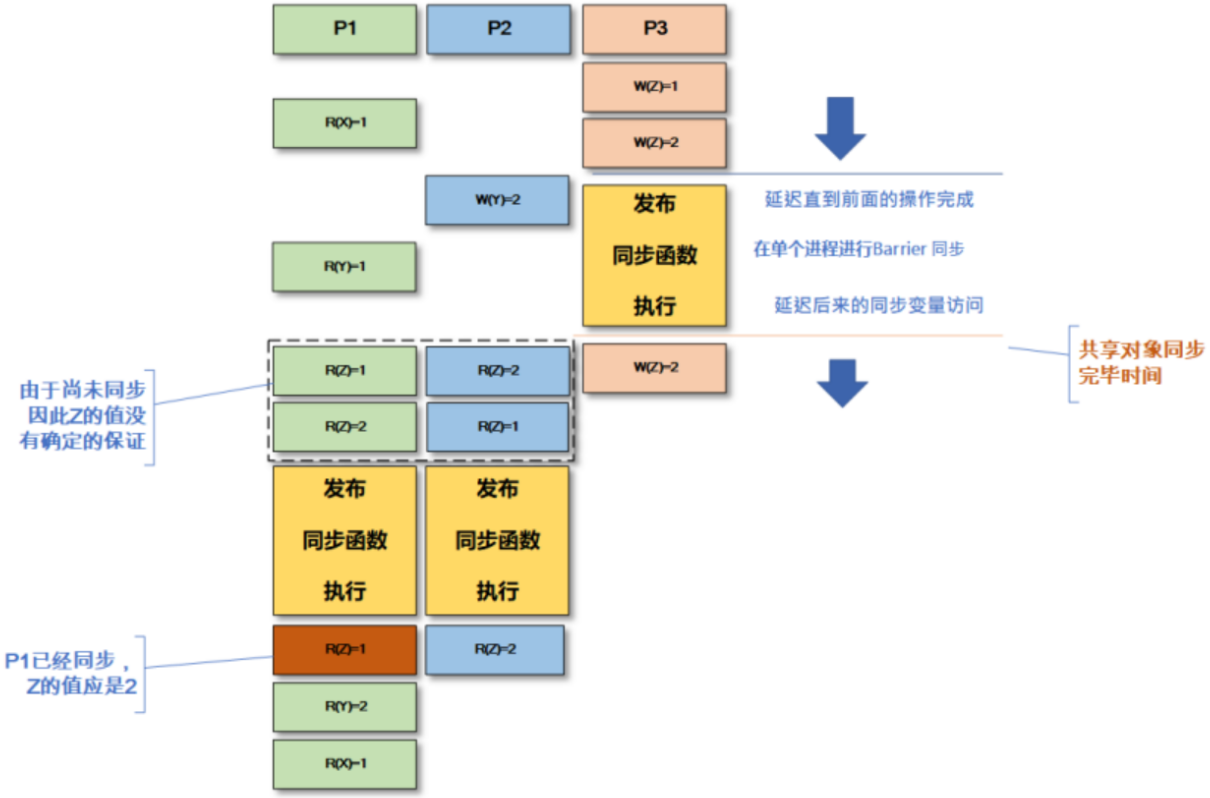
2.1.1.5.2　　举个例子



2.1.2　　基于同步的一致性模型

2.1.2.1　　弱排序一致性

1. 在一个进程完成所有先前针对各个共享存储数据（变量）的读/写入操作之前，不允许其他进程从该同步变量得到访问权限并执行任何操作。（即，其他进程同步等待所有对共享存储数据的正在进行的访问完成）
2. 在某一个进程（客户端）能够得到对同步变量的访问权限之前，不允许其对各个共享存储数据执行读取或写入操作，且能够得到先决条件是没有任何其他持有这个共享存储数据（变量）。（即，对共享存储数据所有新访问必须等待同步执行）
3. 与所有共享存储数据关联的同步共享变量的访问必须符合顺序一致性。
4. 整个系统在同步的时候“暂停”（需要顺序一致性）。



2.1.2.2 释放一致性

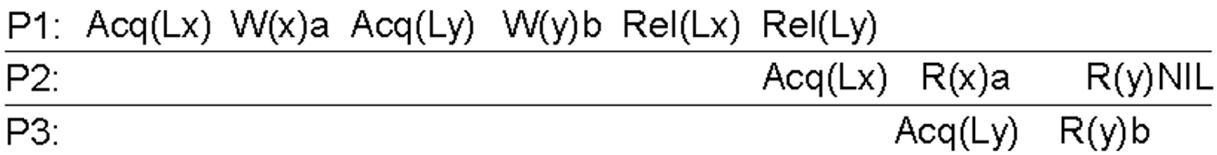
读取或写入共享数据前：
在对共享数据执行**读或写操作**之前，进程之前的所有**获取操作**（acquires）必须成功完成。

释放操作前：
在允许执行**释放操作**（release）之前，进程之前的所有**读写操作**必须完成。

同步变量的访问：
对同步变量的访问满足**FIFO一致性**，但**不要求顺序一致性**。

2.1.2.3 进入一致性

和释放一致性类似，但是Release和Acquire的作用对象是单个共享数据项。



2.2 客户为中心的一致性

一种**更宽松**的一致性形式，仅要求副本**最终达到一致**。根据最终一致性，只要保证更新会被传播，在没有进一步更新的情况下，所有副本都会收敛到彼此相同的副本。

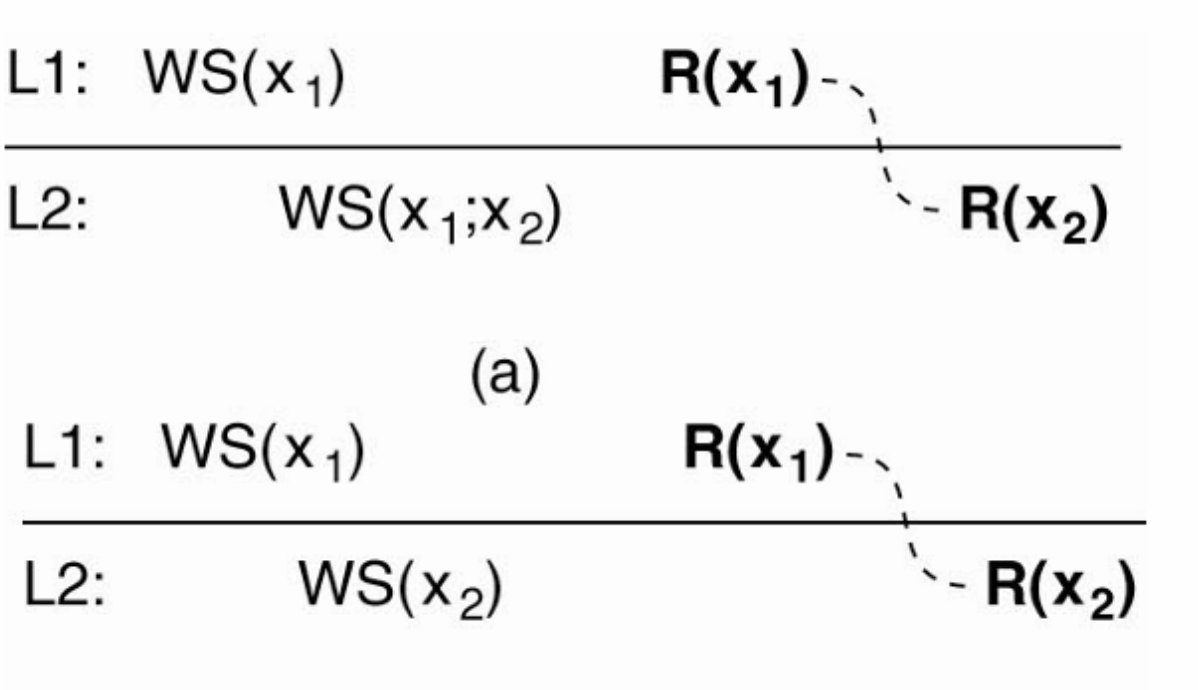
在保证一致性的过程中：

- **易于实现：** 当用户始终访问**同一个副本**时。
- **较难实现：** 当用户访问**不同副本**时，可能会出现数据不一致的情况。

核心思想： 为单个客户端提供一致性的保证，使其在数据存储中的访问结果更加可靠。

2.2.1 单调读

如果某个进程读取了数据项 `x` 的值，那么该进程之后对 `x` 的任何读取操作，都不会返回比之前读取更旧的值。

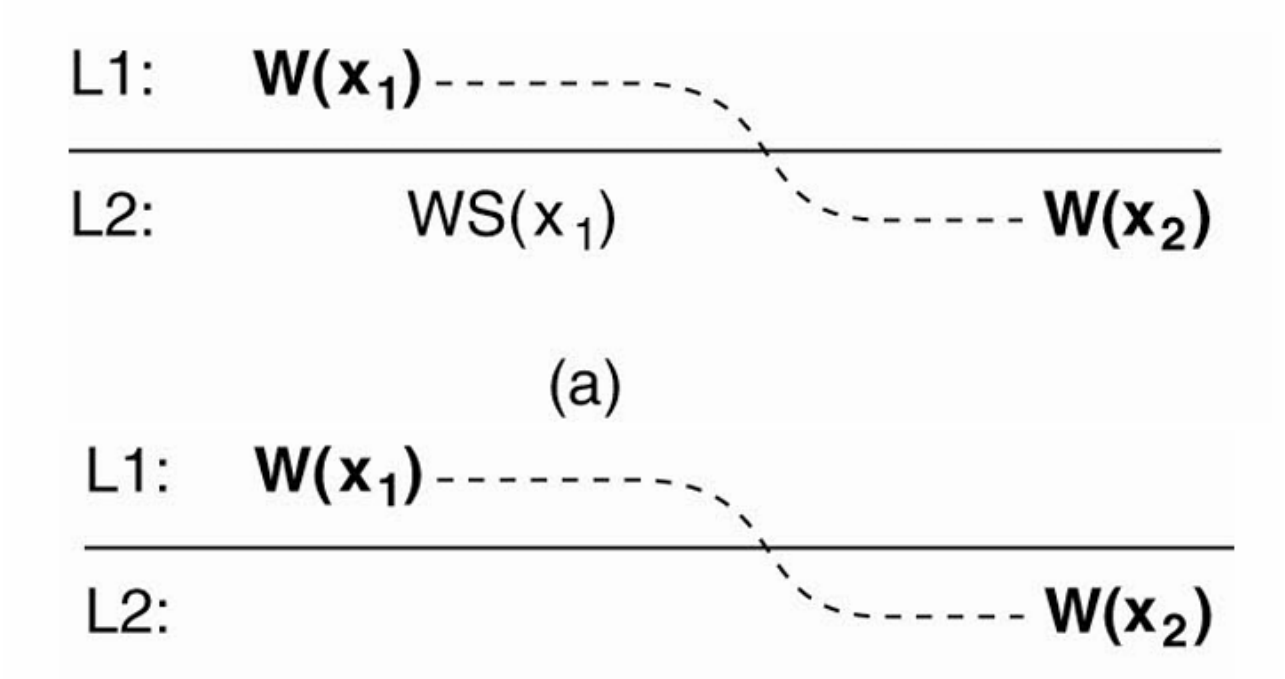


其中$WS(x_i)$表示在站点上i写数据项x的副本（Write Series on data item *x* on site *i*），而$WS(x_i;x_j)$表示已经将副本x_i复制到了本地副本x_j上。

前者满足单调读，因为L2先把L1的数据复制下来，然后再写入。

2.2.2 单调写

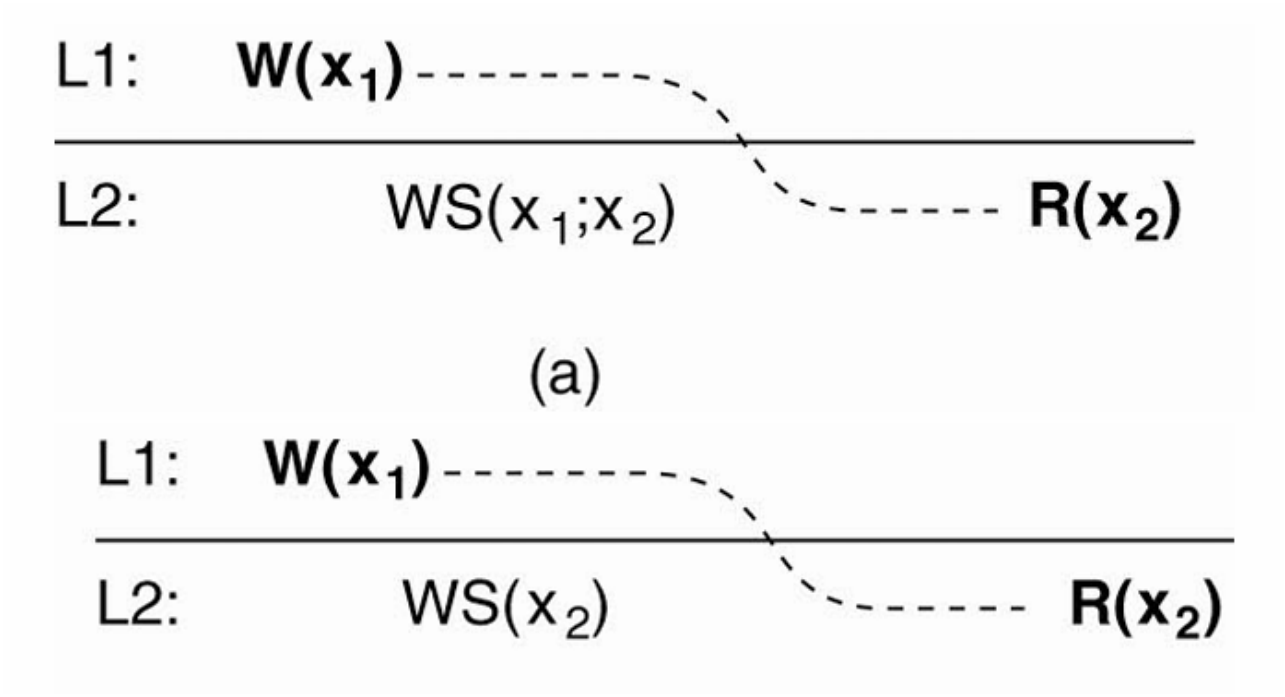
同一个进程对数据项 **x** 的写操作必须按顺序完成，之前的写操作必须完成后，后续写操作才能进行。



前者先把数据从站点1把数据复制下来，然后再写。

2.2.3 写后读

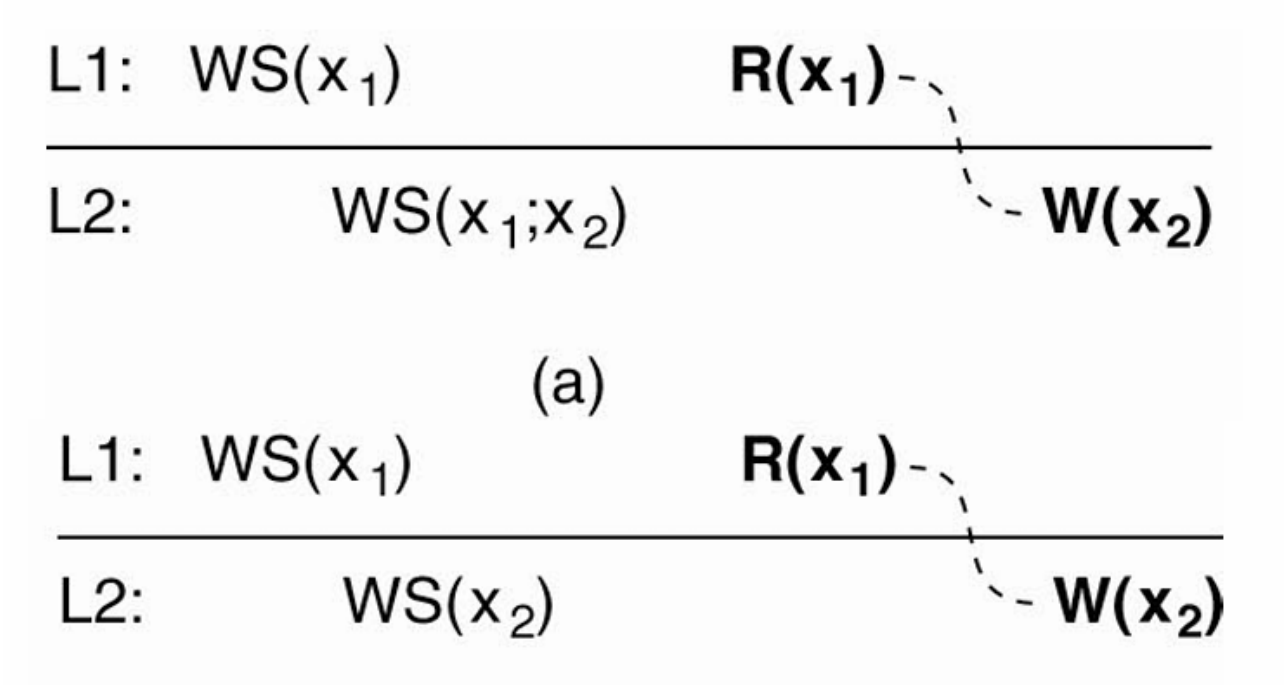
进程对数据项 x 的写操作产生的结果，对同一进程后续对 x 的读取操作是可见的。



显然前者满足，后者不满足

2.2.4 读后写

如果某个进程在对数据项 x 进行**读取**之后进行写操作，则该写操作将基于**读取的值或更新的值**进行。



显然前者满足，后者不满足

3 副本服务器放置

副本服务器放置 (Replica-Server Placement) 的核心是**选择合适的单元格大小**（Cell Size）来进行服务器放置。合适的单元格大小需要在**访问延迟、服务器负载** 和 **成本** 三者之间取得平衡。

数据存储副本的逻辑组织可以通过**三个同心圆（Concentric Rings）**的模型来描述。副本从内到外分为**永久副本、服务器发起的副本**和**客户端发起的副本**。

- **内层**（Permanent）：确保数据完整性和可靠性。如主数据库节点，确保所有数据的可靠存储。
- **中层**（Server-initiated）：提高读取性能，减少主副本压力。服务器根据策略从永久副本复制。如缓存服务器或数据中心内部的多个拷贝，用于负载均衡。
- **外层**（Client-initiated）：提供最低访问延迟，适用于临时需求。客户端主动从服务器发起的副本请求数据并缓存（Client-initiated）。如移动设备缓存、客户端的本地数据副本，减少延迟。

4 复制协议

4.1 复制状态和复制操作的对比

仅传播更新通知：只通知系统中的其他副本某个更新已经发生，而不传输实际数据或更新内容。

数据复制：将一个副本中的数据传输到另一个副本，确保多个副本的数据一致性。

传播更新操作：不仅传播更新通知或数据，还将更新的具体操作（如增、删、改）传播到其他副本，使得操作本身也被同步。

4.2 Pull和Push协议的对比

Push-based：服务器主动向客户端推送更新。在服务器状态方面，服务器需要维护一个**客户端副本和缓存列表**，用于主动向客户端推送更新；发送的消息方面，服务器会发送更新消息（Update），也可能在某些情况下，客户端稍后需要额外获取数据（fetch update）；客户端响应时间方面，客户端可以**立即**收到服务器推送的更新，或者在某些情况下，响应时间取决于更新获取（fetch-update）时间。适合实时性要求高的场景，但需要维护客户端列表，可能会产生额外的网络负担。

Pull-based：客户端主动轮询服务器获取更新，服务器状态方面，服务器不需要维护任何客户端状态，客户端自行决定何时向服务器请求数据；发送的消息方面，客户端通过**轮询（Poll）** 向服务器询问是否有更新，然后拉取更新；客户端响应时间方面，客户端的响应时间完全依赖于客户端轮询并获取更新的时间（fetch-update time）。适合状态变化不频繁的场景，但存在一定的延迟，因为客户端需要等待轮询周期。

4.3 分类

4.3.1 线性一致性

主备份：一个主节点（Primary）处理所有写请求，并将数据更新传播到备份节点（Backup）。

链式复制：数据副本按照链式结构排列，写请求从链头开始依次传递到链尾。

4.3.2 顺序一致性

4.3.2.1 基于主节点的协议

4.3.2.1.1 远程写协议

- 读操作：从本地的拷贝中读取
- 写操作：发送写请求到一个固定的主服务器上，让它执行写操作

4.3.2.1.2 本地写协议

- 读操作：从本地的拷贝中读取
- 写操作：把主服务器移动到接收到写操作的那个服务器上，然后执行本地的写操作

基于主服务器的本地写入协议：

数据项的拷贝在多个进程间传递，同一时间只有一个服务器持有数据项的拷贝。

基于主备份的协议：

负责数据项的主服务器在多个服务器之间切换，每个副本都维护数据项的 拷贝。使用一个固定的中心服务器来实现非阻塞的更新传递。

4.3.2.2 复制写协议

4.3.2.2.1 主动复制

每个副本需要一个进程来执行更新，需要保证更新顺序：

为了确保副本的一致性，必须有一种机制来强制执行更新操作的顺序，以避免由于不同副本执行顺序不一致而导致的数据不一致。可以采用全序组播原语、Lamport时间戳、序列器等。

存在的问题：

多次调用的问题：如果对象 A 调用对象 B，那么 A 的所有副本都会同时调用 B，从而可能导致重复的调用。

4.3.2.2.2 基于法定人数的协议

1. 为每个复制对象的副本分配一个投票权（记作V_i），总投票权之和为V。
2. 每个操作必须达到读取法定人数V_r才能读取对象，达到写入法定人数V_w才能读取对象。
3. 在确定法定人数时，必须遵守以下规则：$V_r + V_w > V$确保同一对象不会被两个事务同时读取和写入；$V_w > V/2$确保两个事务的写入操作不会同时发生在同一对象上。

一种简单有效的方式是ROWA，读取操作只需要从任意一个副本读取数据即可。这意味着读取法定人数为1；写入操作必须将数据写入所有副本。这意味着写入法定人数为V。

容错

1 可信系统的特征

1.1 可用性

系统满足其规范的时间比例，即系统在某一给定时刻t处于正常运行状态的概率。

1.2 可靠性

衡量系统遵循其行为规范的成功程度。系统在给定时间段内没有发生任何故障的概率。通常用于描述无法修复的系统或那些系统连续运行至关重要的情况。（系统在给定时间内没有发生故障）

1.3 安全性

系统在故障发生时，不会引发严重事故或损害

1.4 可维护性

衡量系统在故障时修复的难易程度

2 概念的辨析

- Failure：系统的行为与它应当表现出来的规范不一致，简单来说，就是系统没有按照预期工作。
- Error：系统的错误状态，可能导致 failure。错误状态是指系统处于一种内部状态，在这种状态下，如果继续按正常方式运行，系统会出错，导致故障的发生。不过，这种故障并不是因为之后出现的问题（比如外部干扰），而是由当前状态本身导致的。
- Fault：造成 error 的原因

2.1 Failure的分类

崩溃故障

遗漏故障

超时故障

错误响应故障

值故障

状态转换故障

随机故障

3 如何提高可信度

3.1 通过冗余掩盖故障

- 信息冗余：通过增加额外的数据位（纠错码）来检测和恢复数据传输中的错误
- 时间冗余：在发生错误时能够重新执行操作
- 物理冗余：通过增加硬件或软件的冗余来提高系统的可用性和可靠性：通过部署多个**硬件**实例来提高系统的容错能力、通过在软件层面增加冗余来增强容错能力例如通过**进程冗余**来确保当一个进程出现故障时，其他进程可以继续执行。

3.1.1 硬件冗余的手段

1. 备用系统：在系统中使用两台计算机来运行一个应用程序，其中一台充当主计算机，另一台作为备用计算机。如果主计算机发生故障，备用计算机可以立即接管，确保系统不受影响。这种方式非常昂贵，因为需要额外的硬件支持，但对于高可用性要求的系统（如金融交易系统、航空控制系统）来说，这是一种非常有效的解决方案。
2. 更细粒度的冗余规划：冗余个别服务器、可用于非关键任务的冗余硬件（当系统没有发生故障时，冗余硬件可以非核心任务）、冗余网络路由

3.1.2 进程冗余的手段

进程组内全部成员都可以接收到发往进程组的消息，一旦一个进程故障时，组内其他进程可以接管它的工作。进程组成员可以动态加入退出，一个进程可以有多个身份。

两种结构：扁平结构（所有进程都是平等的（全连接的），每个进程都与其他进程直接通信）、层次结构（组内进程根据某种规则划分为不同层级、Worker与协调者通信）

进程副本的管理：

- 主副本：将一个进程指定为主进程，负责协调所有更新，而其他副本则作为备份进程存在。当主进程失败时，备份进程接管工作。进程被组织成层次结构。
- 复制写：在复制进程中，写操作可能需要在多个副本之间同步。常见的同步方法包括**主动复制**和**基于法定人数的协议**。这是一种扁平化的结构，且没有单点故障。

副本数量的设计：

对于K容错的设计，如果当进程发生故障时，它无法发出任何错误的响应或信号，至少需要K+1个副本；如果考虑拜占庭故障（进程可能表现出任意错误行为，包括发送错误的、欺骗性的信息，甚至可能是恶意的），需要至少2K+1个副本。

如果管理组和组成员：

集中式：通过一个中央的组服务器来管理所有进程和组成员的信息

分布式：需要一种全序可靠组播协议

4 故障系统中的一致性协议

4.1 目标

所有非故障的处理器（进程）在有限步数内就某个问题达成一致。

4.2 两种故障类型

4.2.1 通信故障

通信故障通常涉及到网络中的消息丢失、延迟或错误。最典型的例子是**两军问题（Two-Army Problem）**。在不可靠通信的情况下，两个进程之间不可能达成一致。TCP/IP是两军问题的一个工程解（可靠性已经很高且即使通信失败了最多就重发，没有太大的损失）。

4.2.2 处理器故障

处理器故障通常指的是进程出现错误或者变得不可预测，特别是拜占庭故障（Byzantine Faults）。最典型的例子是**拜占庭将军问题（Byzantine Generals Problem）**。

如何在拜占庭将军问题中达成一致：

1. 每个将军向其他 n-1 个将军告知自己的兵力（真实或说谎）
2. 每个将军将收到的消息组成一个长度为 n 的向量
3. 每个将军将自己的向量发送给其他 n-1 个将军
4. 每个将军检查每个接收到的向量中的第 i 个元素，将其众数作为其结果向量的第 i 个元素

因此，在一个有 m 个故障处理器的系统中，只有当有 2m + 1 个处理器正常工作时，才能确保达成协议。也就是说，总共有 3m + 1 个处理器，才能保证系统在部分处理器故障的情况下达成一致。（为什么不是2m+1？假定节点总数是N，作恶节点数为f，那么剩下的正确节点数为N - f，意味着只要收到N - f个消息且N - f > f就能做出决定，但是这N - f个消息有可能有f个是由作恶节点冒充的（或因网络延迟导致f个恶意节点的消息先被收到），那么正确的消息就是N - f - f个，为了多数一致，正确消息必须占多数，也就是N - f - f > f，所以N最少是3f + 1个。）

5 分布式共识

在分布式系统中，**共识**是指系统中的多个处理器就某个决策（例如值、任务、选举等）达成一致的过程。考虑以下情况：

- 有 n 个独立的处理器，其中最多有 m 个可能发生故障。
- 系统无法知道哪些处理器会发生故障。
- 这些处理器之间只能通过 **双向消息传递**（two-party messages）进行通信。
- 通信媒介被假设为**容错的**（即不会丢失消息）且延迟可忽略不计。
- 消息的发送者始终能被接收者识别。
- 每个处理器 p 都有一些私有的信息值 Vp

6 可靠的客户端服务器通信

（处理方法见“通信-RPC”）

7 可靠组播

7.1 基本的可靠组播

一个发送者发送消息后，所有接收者确认无误后返回确认，如果丢了就告诉丢了哪个。（效率有点低）

7.2 无等级的反馈控制

如果收到没问题就不回复，只有收到有问题再回复（也不是马上回复，而是等待一段时间，如果receiver在等待时间里收到其他人发的重传请求，就不发了）。

7.3 分等级的反馈控制

节点组成一个域，本地协调者把消息发给自己的子节点，本地协调者负责处理重传请求。

7.4 原子多播

- 消息要么发送给所有进程，要么一个也不发送
- 通常需要所有的消息都按相同的顺序发送给所有的进程

需要虚拟同步机制

8 分布式提交

8.1 核心问题

针对分布在一个进程组中的计算，我们如何确保要么所有进程都提交最终结果，要么没有任何进程提交（原子性）？

8.2 两阶段提交

由发起计算的客户端担任协调者；需要提交的进程称为参与者。

阶段 1a：协调者向参与者发送投票请求（也称为预写请求）。

阶段 1b：参与者接收到投票请求后，返回 `vote-commit`（同意提交）或 `vote-abort`（拒绝提交）给协调者。如果返回 `vote-abort`，则中止其本地计算。

阶段 2a：协调者收集所有投票；如果所有投票均为 `vote-commit`，则向所有参与者发送 `global-commit`（全局提交）；否则发送 `global-abort`（全局中止）。

阶段 2b：每个参与者等待 `global-commit` 或 `global-abort` 并据此执行相应的操作。

8.3 三阶段提交

三阶段提交协议通过引入一个额外的阶段（预准备阶段）来解决两阶段提交中的阻塞问题。

1. **CanCommit阶段**：协调者向所有参与者发送CanCommit请求，询问是否可以执行事务提交操作。参与者如果可以执行事务，则返回Yes响应，否则返回No响应。
2. **PreCommit阶段**：如果所有参与者的反馈都是Yes，协调者会向所有参与者发送PreCommit请求。参与者接收到请求后，执行本地事务操作，并写入Undo和Redo日志。如果任何一个参与者返回No响应或超时，协调者会发送中断请求。
3. **DoCommit阶段**：如果所有参与者都成功执行了事务操作并返回ACK响应，协调者会发送DoCommit请求，参与者完成事务提交并释放资源。如果任何参与者返回No响应或超时，协调者会发送中断请求，参与者回滚事务。

9 分布式故障恢复

9.1 什么是故障恢复

当发生故障以后，使崩溃的进程恢复到正确的状态。

9.2 两种形式的故障恢复

9.2.1 回退恢复

- 从当前的错误状态回退到先前的正确状态
- 定时记录系统的状态，称为**检查点**

9.2.2 前向恢复

- 尝试从某点继续执行，把系统带入一个正确的新状态
- 关键在于必须预先知道会发生什么错误

9.3 检查点

9.3.1 独立检查点

每个进程独立地保存自己的本地检查点，而依赖关系被记录下来，以便在发生错误或故障时，多个进程能够协同回滚到一个一致的全局状态。

9.3.2 协调检查点

所有进程同步地共同将其状态写入本地稳定存储，从而形成一个全局一致的状态。两种算法，一种是非阻塞的分布式快照算法，一种是两阶段阻塞算法。

9.3.2.1 两阶段阻塞算法

协调者通过广播Checkpoint_request消息来启动检查点过程。每个接收到消息的进程会保存本地检查点，并暂停处理后续消息，直到向协调者发送确认。当所有进程都完成确认后，协调者广播Checkpoint_done消息，解除进程的阻塞，允许它们继续执行。