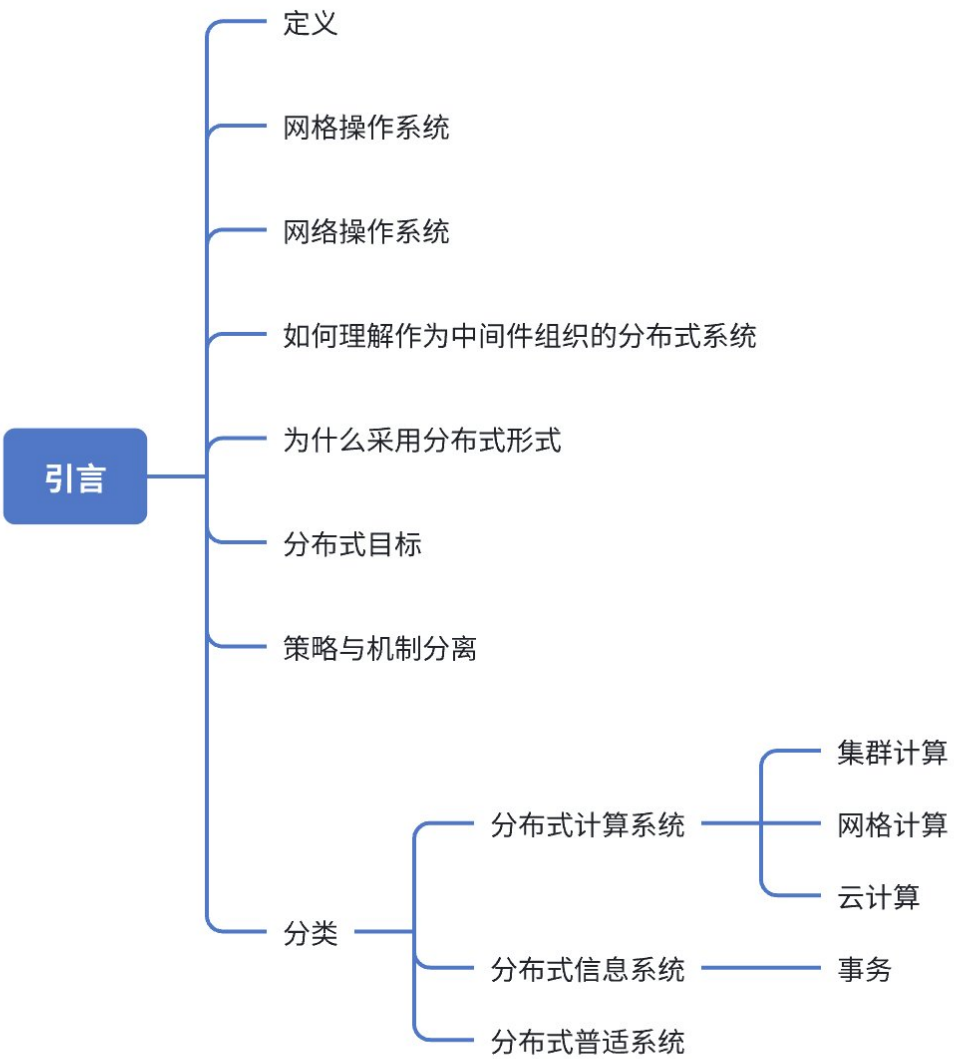
分布式系统

引言



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。

网格操作系统

网格操作系统是一种用于管理和协调分布在不同物理位置、具有异构硬件和软件环境的计算资源的操作系统。

网络操作系统

网络操作系统是一种在用网络连接的多台计算机之间实现资源共享的操作系统，多台计算机具有各自的独立性。

如何理解作为中间件组织的分布式系统?

分布式系统通常通过一个中间件层将应用程序和底层操作系统组织起来，中间件层延伸到多台机器上，为每个应用程序提供相同的接口。

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

分布式系统的目标（需要详细解释）

1. 使资源可用：使用户和应用程序能够轻松访问远程资源，并以受控形式共享
2. 透明性：分布式系统能在用户和应用程序前呈现为单个计算机系统。
3. 开放性：提供服务并遵循标准规则的系统，这些规则描述了服务的语法和语义。
4. 可扩展性：可以应对不断增加的用户和资源，同时保证性能和可管理性。

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

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. 要做到策略与机制分离，系统应当由较小的、容易替换或适配的组件构成。

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（即使跨越多个管理域，系统仍易于管理）

策略和机制分离

机制就是一个工作模式，能够做的事情，类似于接口；策略指的是实现某种功能选取的具体参数、算法，使机制工作的最好，类似于实现。

分布式系统的分类

分布式计算系统

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

网格计算

各地的闲置计算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

云计算

IaaS、PaaS、SaaS

分布式信息系统

如分布式事务处理系统。

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

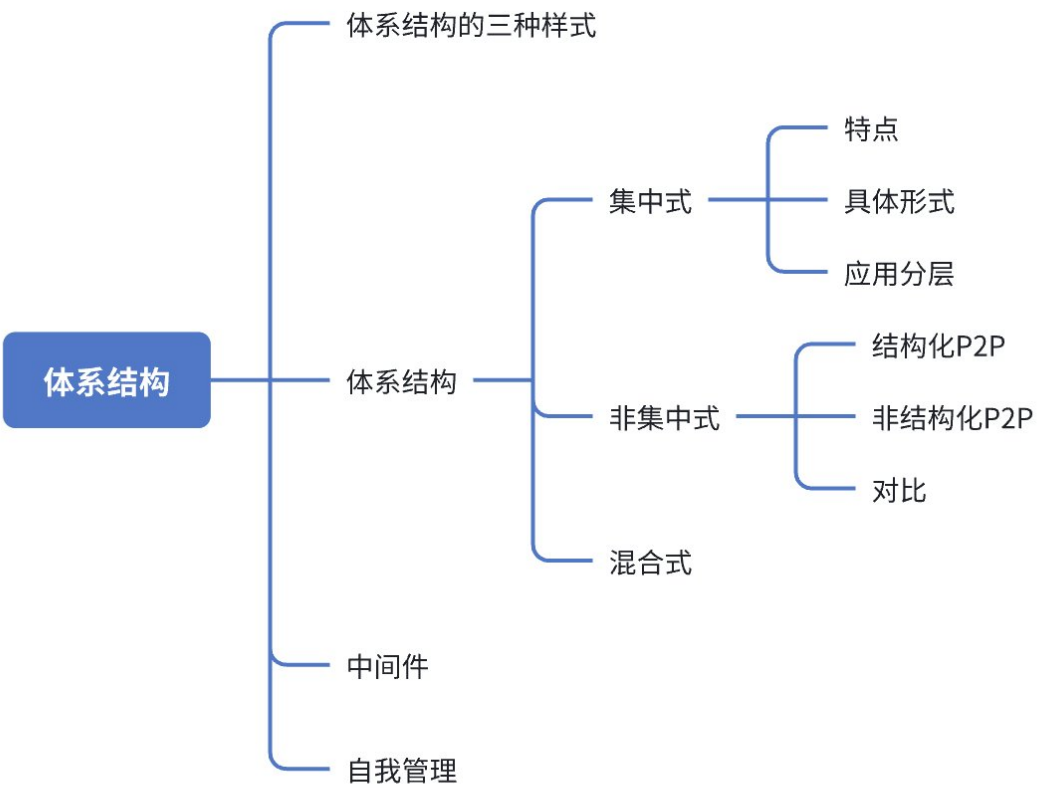
分布式普适系统

下一代分布式系统，节点很小、可移动且通常嵌入在较大的系统中，其特点是系统自然地融入用户的环境，如：

移动计算系统

传感器网络

体系结构



体系结构的样式

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）：每个对象对应一个组件、组件通过RPC连接
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）

体系结构

集中式结构

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

具体形式

1. 多客户端-单服务器架构：服务器存在性能瓶颈、服务器容易产生单点故障、规模难以扩展
2. 多客户端-多服务器架构
3. 基于Web代理服务器的架构
4. 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

应用分层

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

最简单的组织结构是只有两种类型的机器，客户端只包含用户接口的程序（哑终端），服务器包含了其余的部分。

非集中式架构

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

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——Superpeers采用了簇结构，簇头间直接形成全连接。

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。

混合式架构

BitTorrent（集中式的文件目录服务和P2P的文件传输）：首先访问全局目录，确定从哪里可以下载数据块，一旦确定，就被强制为其它节点提供帮助。

边界服务器结构（CDN：内容分发网络）：边界服务器一方面从内容提供商获取数据，另一方面数据在边界服务器间流动

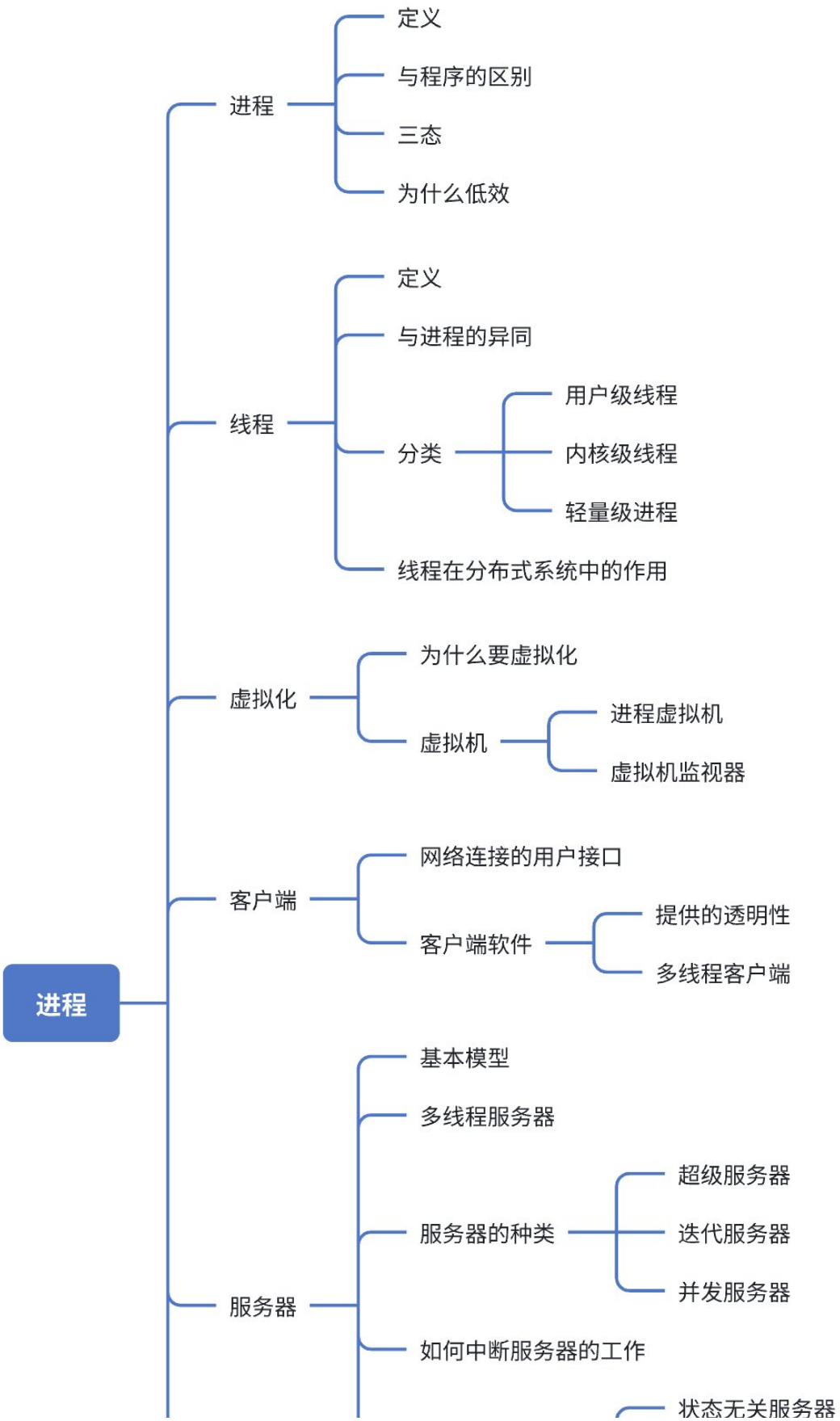
中间件

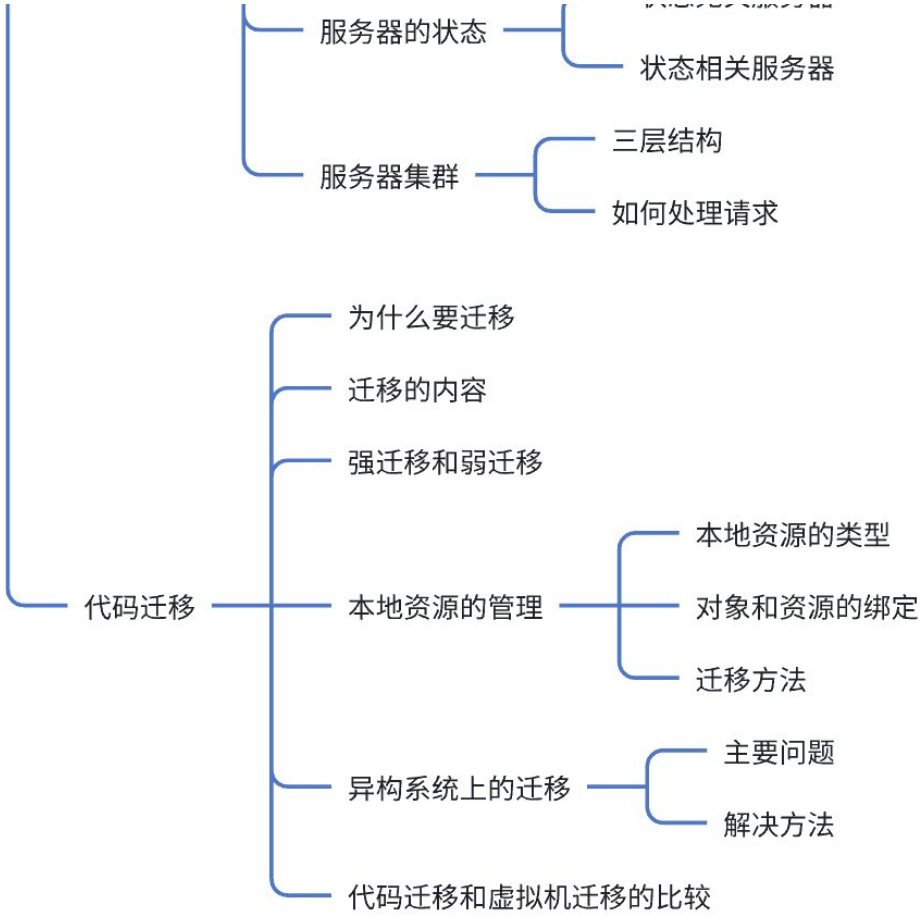
中间件是分布式系统中的一个重要组成部分，它负责不同组件之间的通信、协调与数据管理，以提供分布式的透明性。

分布式系统的自我管理

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





进程

定义

进程是进程状态上下文中的执行流

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

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。

进程为什么低效

1. 资源管理：当创建新进程时需要分配地址空间、拷贝数据
2. 调度：需要同时切换CPU上下文和存储上下文
3. 协作：需要通过IPC或者共享内存

线程

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。

线程与进程的异同

进程和线程都是执行流。一个进程包含多个线程，他们共享数据(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。

线程的分类

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。

内核级线程

将线程管理移到内核中，虽然增强了线程调度、阻塞处理的能力并且处理外部事件变得容易，但可能会因为频繁的系统调用导致效率下降。

轻量级进程

将用户级线程和内核级线程混合

线程在分布式系统中的作用

1. 隐藏网络时延
2. 提高性能
3. 可以使用阻塞调用以获得更好的结构

虚拟化

为什么要虚拟化

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

虚拟化的两种方式

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

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

客户端

网络连接的用户接口

例如X Window系统，提供一个用户接口来对远程服务的直接访问

客户端软件

透明性

- **访问透明性**：通过客户端存根实现，使远程调用看起来像本地调用。
- **位置/迁移透明性**：客户端软件跟踪服务器位置，支持服务器迁移。
- **故障透明性**：客户端处理故障，掩盖服务器和通信故障。
- **复制透明性**：客户端存根处理请求复制，管理多个服务器副本。

多线程客户端

如Web浏览器，每个线程建立一个单独的连接从服务器获取数据，减少延迟；显示线程和数据获取线程分离，可以先把文本显示出来，避免等待图片等阻塞。

服务器

基本模型

服务器是一个在特定传输地址等待传入服务请求的进程。端口和服务之间存在一对一的映射。

多线程服务器

一种流行的实现是通过dispacher线程将请求分发给空闲的worker进行处理。

服务器的种类

1. 超级服务器：监听多个端口的服务器，提供多个独立服务。当服务请求到达时，启动一个子进程来处理。
2. 迭代服务器：迭代服务器自己处理请求，同时只能处理一个客户
3. 并发服务器：并发服务器将请求传递给某个独立线程或进程，可以同时处理多个客户

如何中断服务器的工作

采用带外通信。使用单独的端口和单独的进程/线程来处理紧急的消息，一旦紧急消息到达，其他常规数据将会被搁置。此外，TCP提供了在同一连接中发送紧急消息的机制，操作系统的信号技术也可以用来捕获紧急消息。

服务器的状态

状态无关服务器

状态无关服务器不保存客户的状态信息，也不将自己的状态变化告诉客户。客户端和服务器时完全独立的，客户端或服务器崩溃导致的状态不一致会减少，但会带来性能损失，因为服务器无法预测客户端的行为

状态相关服务器

状态相关服务器一直保存客户端的状态。只要允许客户端保留本地副本，状态相关服务器的性能可以非常高。事实证明，可靠性并不是一个大问题。

服务器集群

三层结构

1. 逻辑交换机分配请求给服务器
2. 应用/计算服务器进行实际计算
3. 分布式文件/数据库系统提供缓存等

如何处理请求

如果让第一层处理所有的请求可能会存在瓶颈，通常使用TCP转发的方式。交换机收到一个TCP连接请求时，找到最佳服务器，并把请求包转发给这个服务器。服务器反过来发送一个应答给客户，但把交换机的IP地址插入到TCP的源地址域中。

代码迁移

为什么要迁移

把进程从负载较重的机器转移到负载较轻的机器有助于提升系统整体性能

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

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。

本地资源的管理

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

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。

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

异构系统上的迁移

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

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：

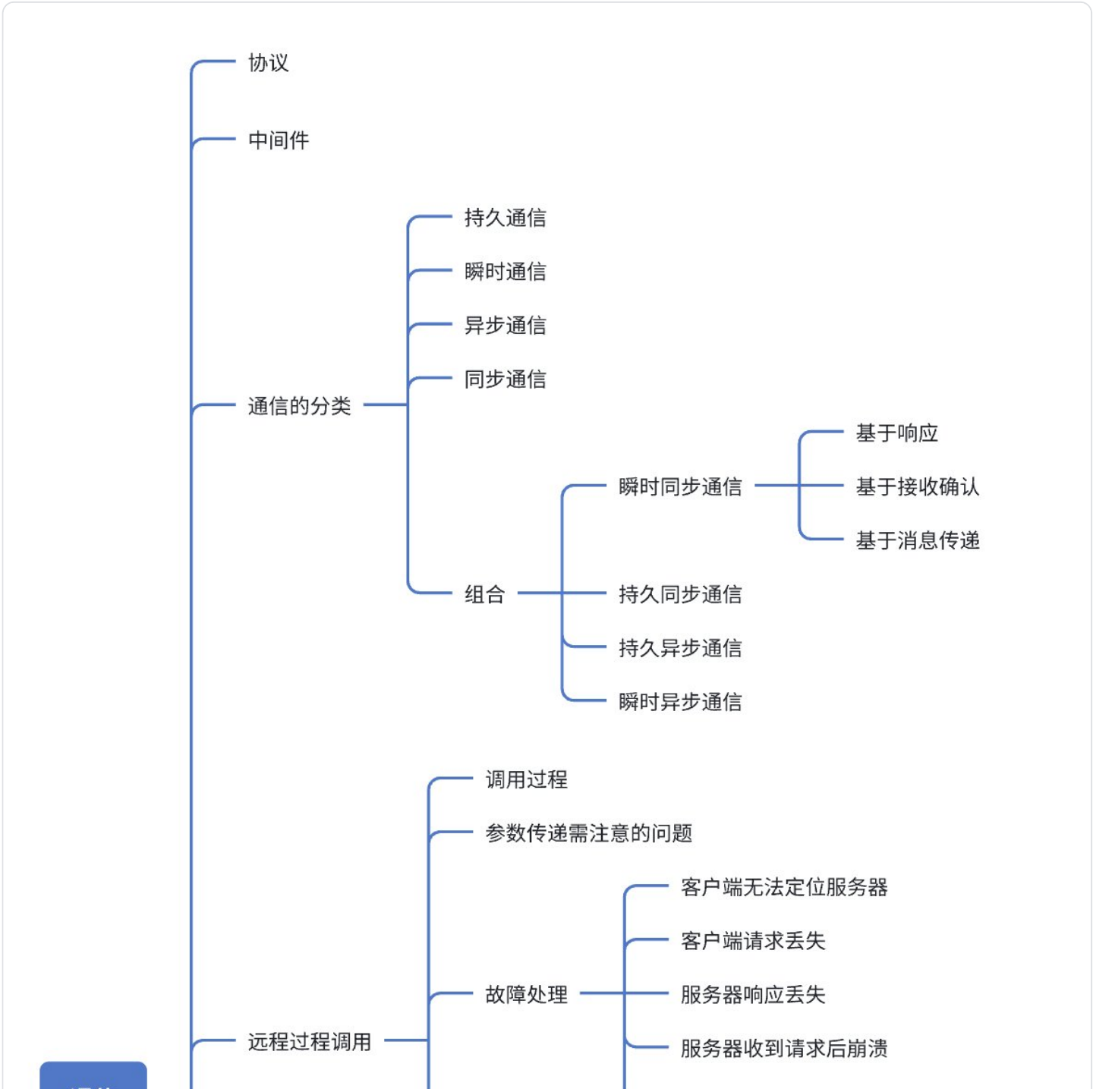
- 解释型语言（通常通过自己的虚拟机）
- 应用在虚拟机上运行，虚拟机负责屏蔽底层的差异

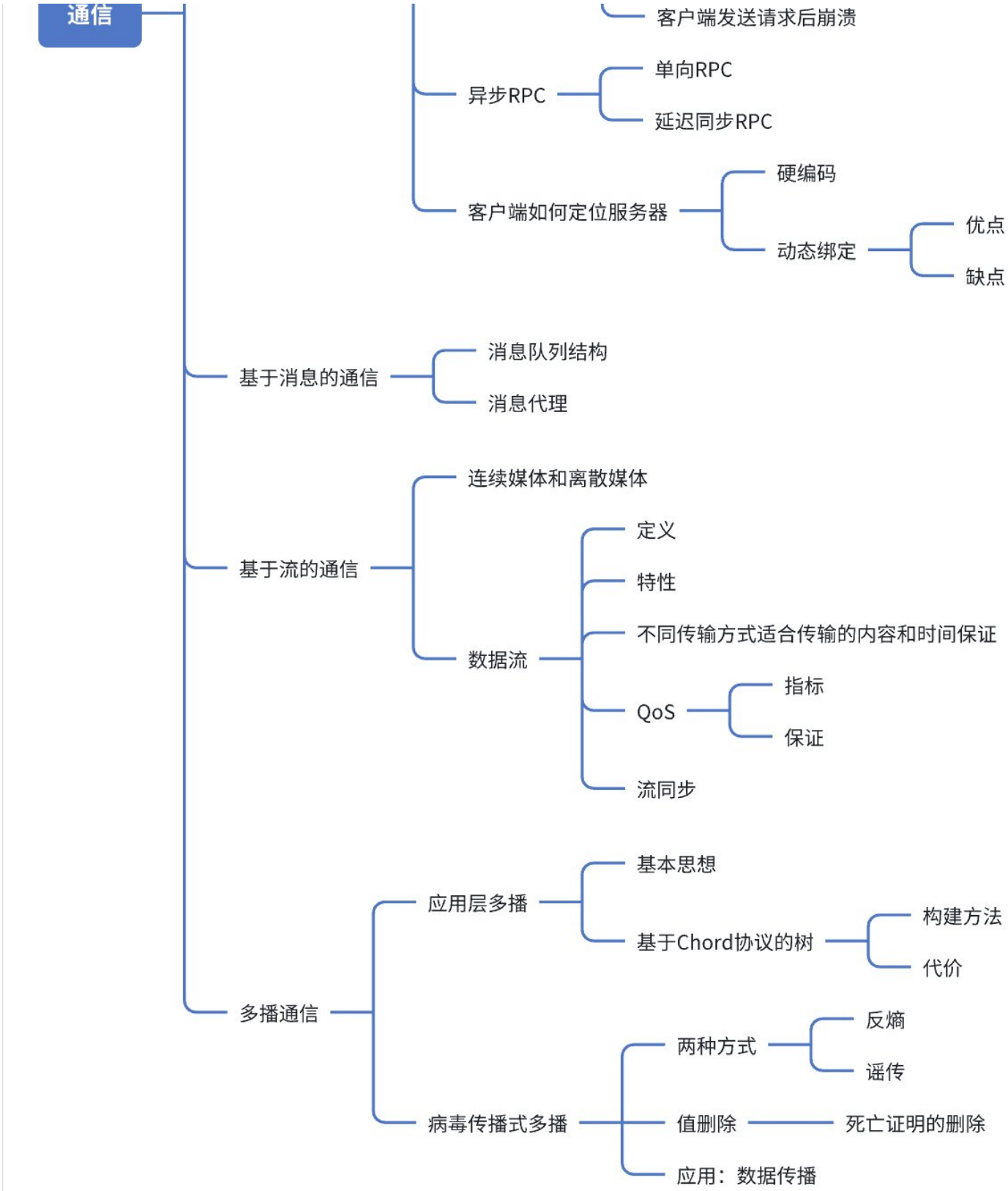
代码迁移与虚拟机迁移的比较

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





协议

对发送接收的消息格式、规则进行归纳总结，并加以形式化

中间件

中间件提供通用服务和协议，这些服务和协议可以被许多不同的应用程序使用。

通信的分类

瞬时通信：发送者将消息放在网络上，如果无法传递给接收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：消息发送方在执行发送操作后立即继续执行接下来的步骤，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。

组合

基于响应的瞬时同步通信

通信过程

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。

基于接收确认的瞬时同步通信

A 发送请求并等待确认；B 接收请求后立即发送确认（ACK），然后继续处理其他任务；A 在接收到确认后才继续；适用于需要确认消息成功送达但不急于立即处理的场景

基于消息传递的瞬时同步通信

A 发送请求并等待，直到消息被 B 接收；B 接收到请求后等到任务开始处理时返回“已接收”；A 在收到确认后继续运行，而 B 在后台处理请求；适用于需要确认消息已开始处理，但对处理的延迟容忍度较高的场景

持久异步通信

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。常用于需要高可靠性传输的系统，比如电子邮件。

持久同步通信

A 发送消息后等待 B 接收确认；如果 B 未运行，消息会被存储在 B 的位置（或中间件）中，直到 B 恢复运行并接收消息。A 在接收到 B 的确认后才继续运行。通常用于需要强一致性和可靠性确认的系统，如银行事务处理或订单处理。

瞬时异步通信

A 发送消息后立即继续自己的工作（非阻塞）；消息只有在接收方 B 正在运行时才能被接收；没有确认机制，A 无法知道 B 是否成功接收消息；适用于需要快速发送消息且不需要可靠确认的场景

远程过程调用

调用过程

1. 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
2. 客户端存根构建消息并调用本地操作系统
3. 客户端操作系统将消息发送到远程操作系统
4. 远程操作系统将消息交给服务器存根（Server Stub）
5. 服务器存根解包参数并调用服务器端程序
6. 服务器执行请求并返回结果
7. 服务器存根打包结果并调用本地操作系统
8. 服务器操作系统将消息传递给客户端操作系统
9. 客户端操作系统将消息传给客户端存根
10. 客户端存根解包结果并返回给客户端

参数传递需要注意的问题

1. 传递值参数：通过与机器无关的序列化方法确保客户端和服务器之间的数据兼容性
2. 传递引用参数：两种方法：①采用值传递，避免传递引用带来的问题②通过远程引用机制提升远程数据访问的透明度

故障处理

客户端无法定位服务器

抛出异常或使用特殊的返回值

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

服务器向客户端发送的回复消息丢失

使用ACK机制，如果未收到确认，则重发响应。请求重传时不增加请求序列号，避免重复处理请求。要求服务器在一定时间内保留旧的响应结果。

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

- 等待服务器重启并重新进行处理：保证至少执行一次(at-least-once)
- 立即放弃并报告错误或者服务器在持久化存储中保留所有之前的回复，然后重传所有回复：保证至多执行一次(at-most-once)
- 什么都不做

客户端在发送请求后崩溃

当客户端发生崩溃时，可能会留下孤儿进程，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，它会在稳定存储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Epoch。当客户端重启时，它会广播一个新Epoch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。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来继续执行。超时后，远程计算会自动终止。

异步RPC

客户端调用远程过程后，只需等待请求被接受即可，不需要等待结果。

单向RPC

客户端发送请求时，不需要等待服务器的接受确认，也不需要返回结果。常用于只需通知服务器执行任务的场景，而不关心结果。

延迟同步RPC

客户端发送请求并等待服务器接受。然后做自己的事情，服务器在完成计算后，通过另一个RPC返回结果。

客户端如何定位服务器

硬编码

将服务器地址硬编码到客户端快速但不灵活

动态绑定

服务器在启动时会导出自己的接口，并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。客户端通过查询这个绑定服务获取地址。

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

基于消息的通信

消息队列的结构

发送方把消息放入队列层，队列通过地址查询数据库解析网络传输层的实际地址，并发送给接收方，接收方从队列中取出消息并交给本地应用程序。消息可能需要通过多个路由器进行中转，最终送达目标接收端。

消息代理

源客户端发送消息到消息代理，消息代理接收消息并处理，目标客户端从消息代理接收处理后的消息。

消息代理作为集中化组件处理消息队列系统中的应用异构性：

- **将接收到的消息转换为目标格式**：消息代理可以将不同格式的消息转换为目标格式，以便目标客户端理解和处理。
- **通常作为应用网关**：消息代理经常充当应用网关，连接不同的应用程序。
- **可能提供基于主题的路由功能**：消息代理可能提供基于主题的消息路由功能，从而实现企业应用集成。

基于流的通信

连续媒体和离散媒体

连续媒体表示不同数据项在时间上有联系的数据；离散媒体指的是与时间无关的数据

数据流

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。数据流是数据单元的序列，既可以用于离散媒体，也可以用于连续媒体。

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。

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

流的QoS指标

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

如何保证流的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）

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

多播通信

应用层多播

基本思想

将节点组织成为一个覆盖网络，然后利用它来给成员传递消息。

基于Chord协议的树

构建方法

1. 首先，发起节点生成一个唯一的多播标识符
2. 在Chord环中查找负责该多播标识符的后继节点（succ）
3. 请求被路由到负责该标识符的后继节点，这个节点将成为树的根节点。
4. 节点P如果想加入多播组，会向根节点发送加入请求。
5. 当请求到达节点Q时，如果Q之前没有看到过加入请求：Q将成为转发者，P成为Q的子节点。加入请求继续被转发；如果Q已经知道树的结构：P成为Q的子节点，不再需要继续转发加入请求。

代价

1. 链路压力：ALM消息在同一个物理链路上经过的频率
2. 路径伸展：ALM级别路径的延迟与网络级别路径延迟的比值

病毒传播式多播

假设，没有写写冲突，更新操作仅在单一服务器上执行，副本只将更新后的状态传递给少数邻居，更新传播是惰性的，即不是立即传播，最终，每次更新都应到达每个副本。

两种方式

反熵

每个副本定期随机选择另一个副本，并交换状态差异，最终使两者的状态一致。

三种形式：

Push：P只给Q发送更新

Pull：P只接受来自Q的更新

Push-Pull：P 和 Q 交换相互的更新

在Push-Pull中，需要 $\sigma log(N)$ 轮才能把更新传播到所有 N 个节点，一轮表示每个节点都主动发起了一次交换时所经历的时间，σ 表示每轮传播中涉及的平均操作复杂度。

谣传

当服务器S有更新时，他会联系其他服务器。如果S联系到的服务器已经接收到这个更新，则S以1/k的概率停止传播。假设s是未感知到更新的服务器的比例，则在大量服务器下，s可以利用以下公式计算：

$$s = e^{-(k+1)(1-s)}$$

值删除

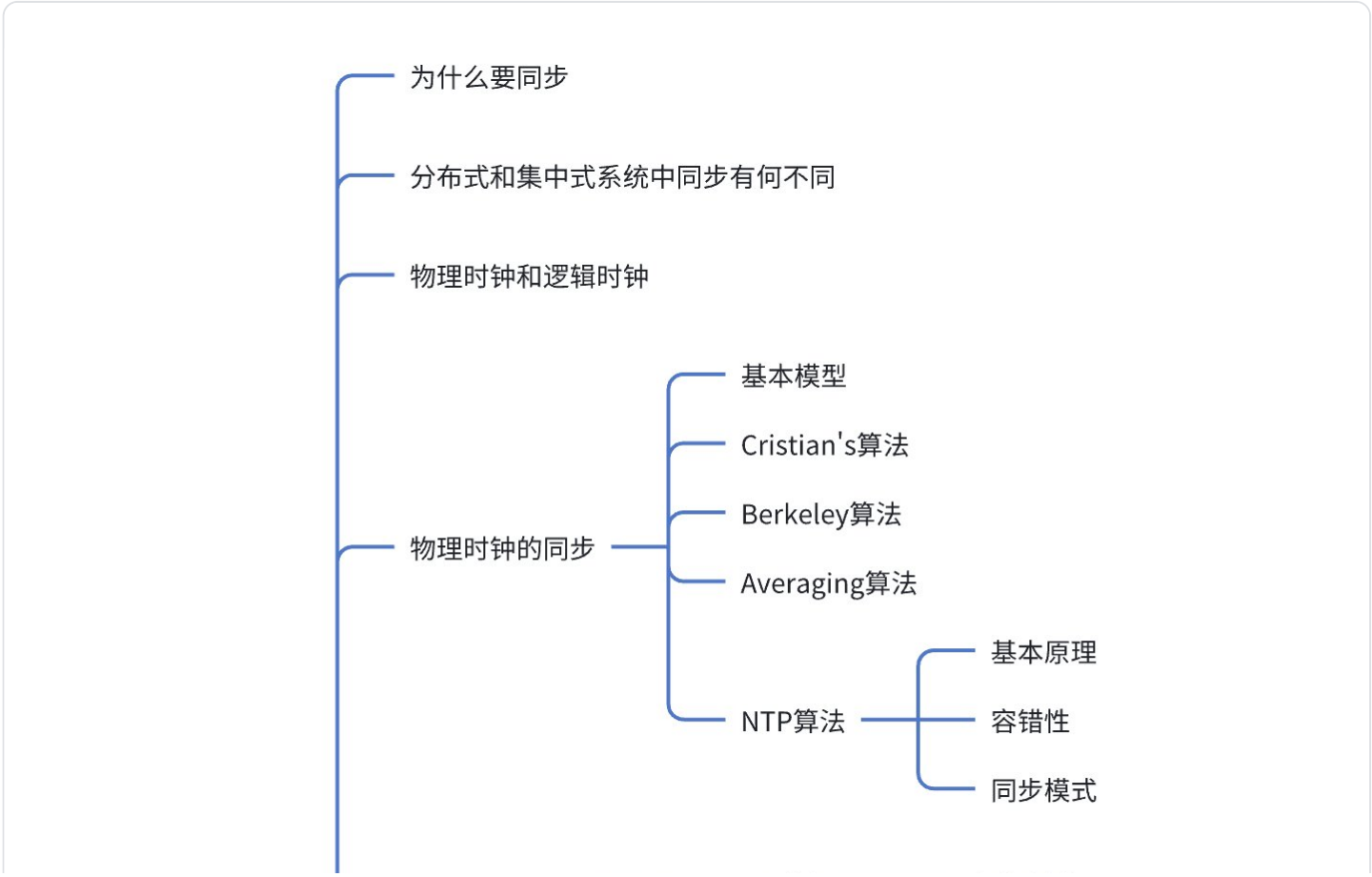
需要将值删除作为另一次更新，通过传播死亡证明来实现。否则删除的旧值通过一段时间的传播后又会出现。

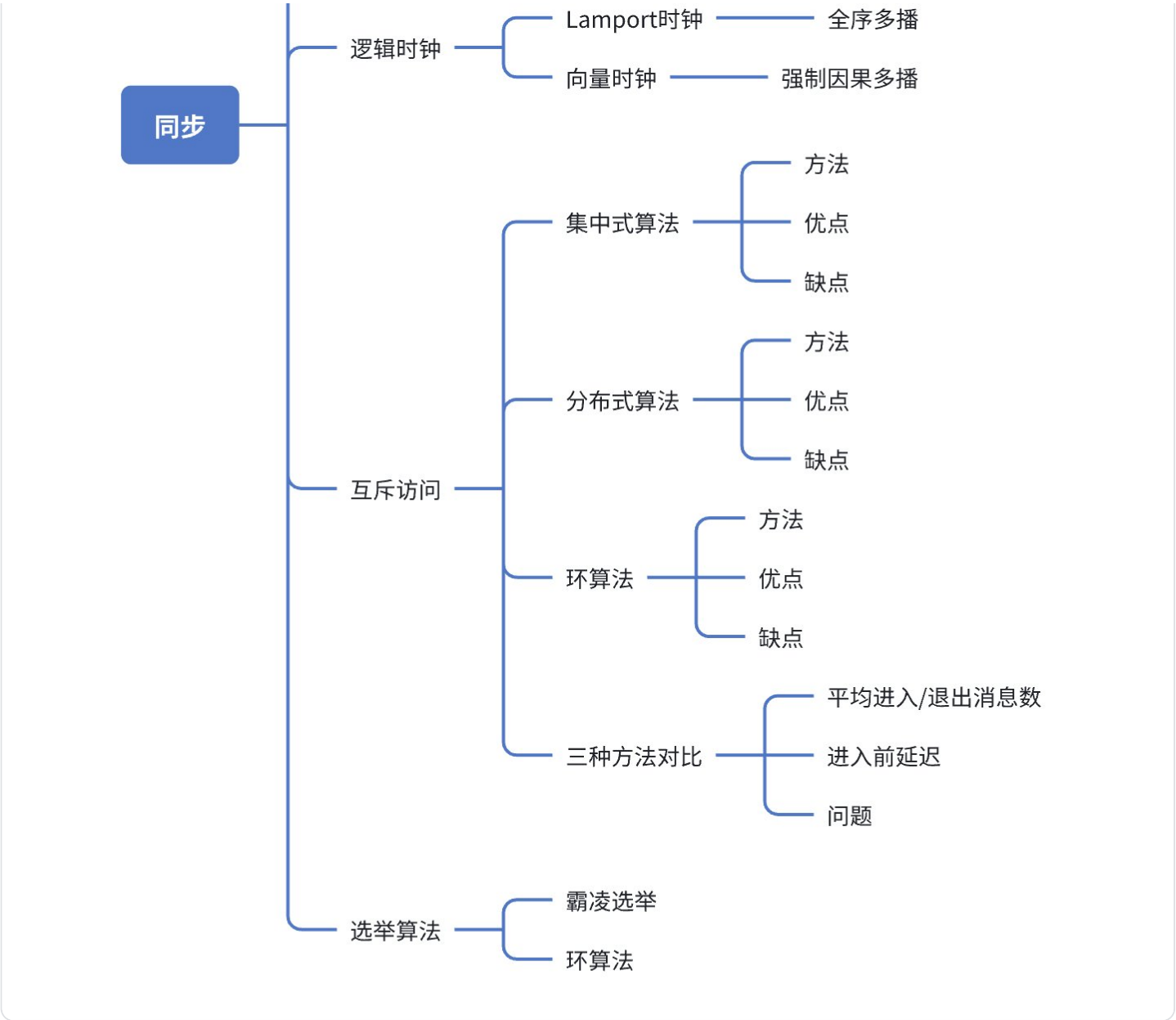
死亡证明何时删除？给每个死亡证明加上时间戳，假定在已知的有限时间里，更新信息可以传播到所有节点，那么在最大传播时间后就删除它。但是，为了确保删除信息真的传播到了所有节点，需要很少的一些节点维护休眠的死亡证明，当废弃更新到达时，再次传播即可。

应用：数据传播

假设每个节点 i 维护一个变量 x_i ，当连个节点进行gossip的时候，将其各自值更新为 $x_i, x_j = \frac{x_i + x_j}{2}$ ，最终，每个节点都会计算出所有节点变量的平均值。

同步





为什么要同步

很多程序需要依赖于程序创建和修改的先后顺序进行处理，所以在不同的系统或是进程间需要确定先后关系，也即需要时间同步。而且每台机器上有不同的时钟偏移。

分布式系统中和集中式系统中同步的不同

在集中式系统中，时间是明确的，如果进程想知道实践，可以执行一次系统调用，然后系统内核会告诉它，这依赖共享内存（可以从一个公共的模块获取时间）。但在分布式系统中，缺少这种全局时间。

物理时钟和逻辑时钟

物理时钟关心的是时间的绝对值，逻辑时钟关心时间发生的先后次序。

物理时间的同步

基本模型

假设时钟的最大偏移率是 ρ ，要保证两个时钟之间最大差值不超过 δ ，那么必须至少每 $\delta/2\rho$ 同步一次。

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。

1. 客户端在本地 T_0 时向服务器发起请求；
2. 服务器经过 I 时长的处理后响应UTC时间，客户端在本地 T_1 时间得到响应
3. 传播时间可以被估计为 $(T_1 - T_0 - I)/2$ ，本地时钟应该被矫正为 $T_{client} = T_{server} + (T_1 - T_0 - I)/2$

注意，这种修改必须逐步进行，因为时钟不能倒退。

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

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 $T_0 + iR$ 开始到 $T_0 + (i+1)R$ 结束， T_0 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）

NTP算法

基本原理

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

逻辑时钟

Lamport时钟

每个进程 P_i 维护一个局部计数器 C_i ，该计数器按以下规则更新：

1. 在执行一个事件之前，计数器+1
2. 发送消息时，把计数器的时间戳附在消息上
3. 接收消息时，调整本地计数器为本地计数器和接收时间戳中较大的一个，然后+1

基于Lamport的全序多播

一次将所有的消息按顺序传送给每个接收者叫全序多播。

进程接收一个消息之后，先把它放在一个本地队列中，按时间戳排序。然后向其他所有进程广播一个确认消息。只有当队列中一个消息位于队列头，且已经被其他所有进程确认后，进程才可将其传送给应用程序。

向量时钟

VC[i]表示到目前为止进程 P_i 发生的事件数量。

1. 在执行一个事件之前，将本进程对应的逻辑时钟+1；
2. 发送消息时，把向量时间戳附在消息上；
3. 接收消息时，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。

基于向量时钟的强制因果有序多播

假设进程 P_j 从进程 P_i 接收到一个时间戳为ts(m)的消息，把该消息传送到应用层将被延时，直到满足以下两个条件：

1. ts(m)[i] = VC[i]+1
2. ts(m)[k] <= VC[k],k != i

消息m希望是进程 P_j 从进程 P_i 接收到的下一个消息，并且已经接收到了所有来自其他进程的消息。

互斥访问

集中式算法

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和性能瓶颈
- 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”

分布式算法

方法

- 当某进程希望进入临界区时，将要访问的资源名、进程号和当前逻辑时间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。

优点

- 也不存在死锁或者饥饿现象

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

令牌环算法

方法

系统中的所有进程按照逻辑顺序排列成一个环，环中点对点地传递一个令牌，持有令牌的进程有权进入临界区（如果它需要进入）。

优点

- 也不存在死锁或者饥饿现象，最多就是进入前需要等待其他进程都使用一遍该资源

缺点

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

对比（平均进入/退出消息数；进入前延迟；问题）

A	B	C	D
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

选举算法

霸凌选举

当一个进程发现协调者不响应请求时发起选举。进程P按如下过程主持一次选举：

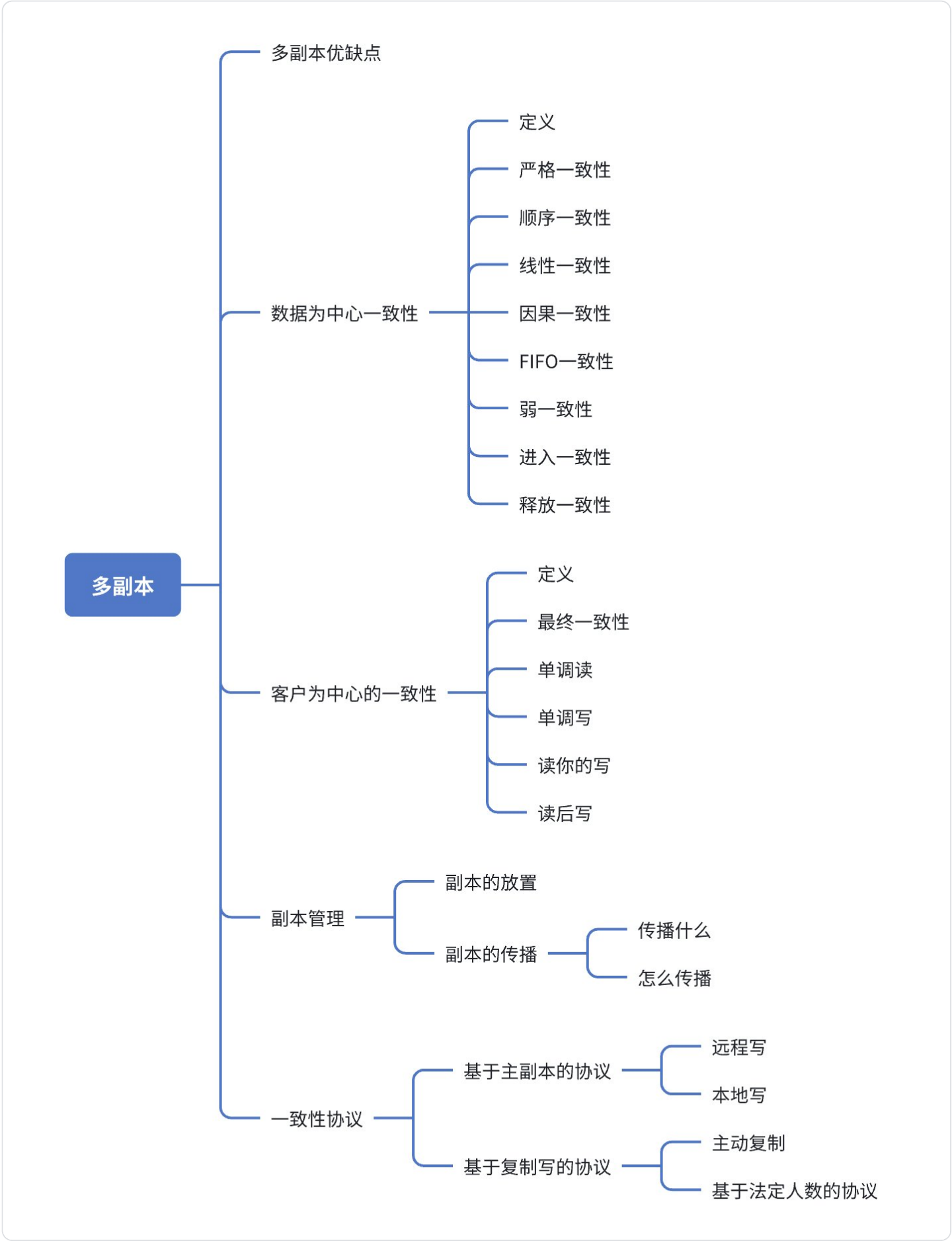
1. P向所有进程发送选举消息；
2. 收到选举消息的节点，如果当前节点比它权重高，则回送一个OK地接管指令，表示P被淘汰出局
3. 如果一段时间内没有收到接管消息，则向所有节点发送胜利消息

环算法

1. 当一个进程发现协调者不响应请求时发起选举。就发送一个带有自己优先级的选举消息给后继节点；
2. 如果某个进程的后继进程故障，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，它会将自己的优先级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。
3. 一旦消息回到最初的发起进程，这时所有存活的进程都已把自己的优先级列入了消息列表中。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。

多副本与一致性


多副本
多副本优缺点
数据为中心一致性
定义
严格一致性
顺序一致性
线性一致性
因果一致性
FIFO一致性
弱一致性
进入一致性
释放一致性
客户为中心的一致性
定义
最终一致性
单调读
单调写
读你的写
读后写
副本管理
副本的放置
副本的传播
传播什么
怎么传播
一致性协议
基于主副本的协议
远程写
本地写
基于复制写的协议
主动复制
基于法定人数的协议

多副本的优缺点

优点

1. 提高系统的可靠性，避免单点故障
2. 提高系统的性能，使系统在规模和地理位置上可扩展

缺点

1. 多副本的透明性难以实现
2. 多副本会带来一致性问题

数据为中心的一致性

定义

描述多个进程同时访问共享资源时的并发访问语义

严格一致性

任意 `read(x)` 操作必须返回最近一次 `write(x)` 操作写入的值，要求依赖绝对的全局时钟，在分布式系统中不可实现

顺序一致性

当进程在多台机器上并发运行时，任何读写操作的有效交叉是可接受的，但所有进程都看到相同的操作交叉。

线性一致性

当进程在多台机器上并发运行时，所有进程看到的读写操作的顺序必须与某种时间戳吻合。

因果一致性

因果一致性是一种弱化的顺序一致性模型，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。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。

FIFO一致性

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。

弱一致性

只有在进行同步操作后才会一致

释放一致性

临界区退出后实现一致

进入一致性

对变量的锁，如果没有拿到这个锁，读到的就是NIL

客户为中心的一致性

定义

以客户为中心的一致性中大部分操作是读操作，目标是为单个客户提供访问数据时的一致性保障。

最终一致性

如果在很长的时间内没有更新操作，那么所有的副本将逐渐变成一致。如果客户总是访问同一副本，最终一致性工作的很好。

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。

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，即使这些写操作发生在其他副本上。

读你的写

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

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。

副本管理

副本的放置

逻辑上组织为三类：

1. 永久副本：构成分布式数据存储的副本的初始集合。
2. 服务器发起的副本：初始化数据存储的所有者时创建的副本（说人话就是，拥有副本的服务器根据需要让其他副本服务器创建副本）
3. 客户端发起的副本：客户初始时创建的副本

内容的分发

传播什么

1. 只传播更新的通知：例如无效化协议，通知副本过期了
2. 把一个副本直接传送给另一个副本
3. 把更新操作传播给其他副本

怎么传播

1. 基于Push的协议：需要实时或近实时数据更新的应用。
2. 基于Pull的协议：数据更新较少或不需要实时更新的场景。

一致性协议

基于主备份的协议

远程写

要在数据项x中执行一个写操作的进程，把操作转发给x的主服务器，主服务器在其x的本地副本上执行更新操作，随后把更新转发给副本服务器。每个副本服务器执行这个更新操作，并向主服务器发送一个确认消息。当所有副本服务器都更新了它们的本地备份后，主服务器发送一个确认消息给初始进程。读操作可以在本地执行。

缺点是启动更新的进程在运行继续执行前，需要阻塞等待较长一段时间。

本地写

主副本在执行写操作的本地副本间迁移。当某个进程要更新数据项x时，先定位x的主副本，然后将其迁移到自己的位置上。优点是，可以在本地执行多个连续的写操作，而读操作仍可以访问其本地副本。但是只有使用非阻塞协议，在主备份完成更新后，通过该协议将更新传播到其他副本时才有该优点。

基于复制写的协议

主动复制

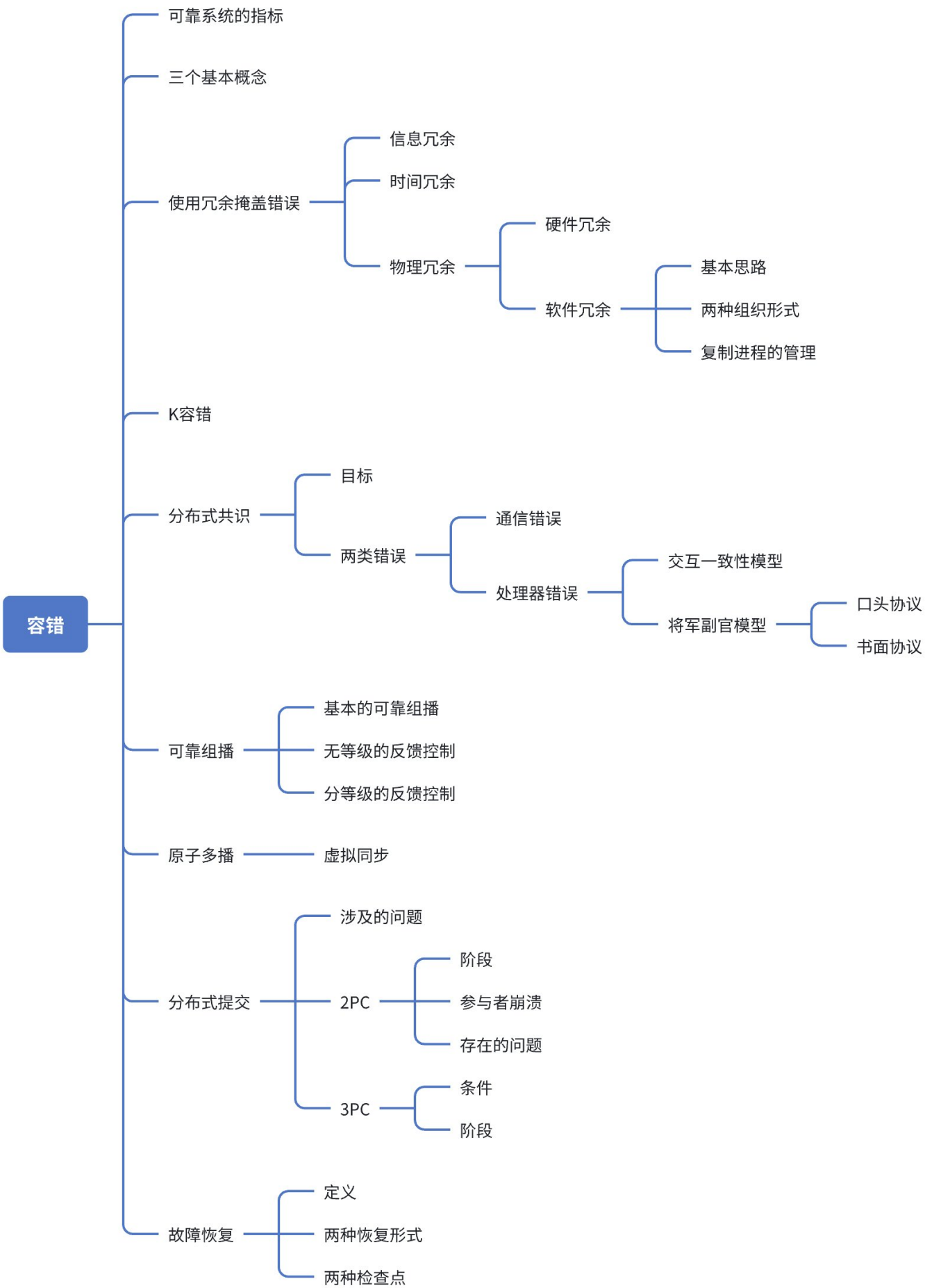
每个副本有一个关联的进程，该进程负责执行更新操作。要保证各地按照相同的顺序执行更新操作，需要全序的多播协议，或者是使用中心协调器（定序器）。

基于法定人数的协议

要求客户在读或者写一个副本的数据项前向多个服务器提出请求，并获得它们的许可。

1. 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。
2. 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。

容错


容错
可靠系统的指标
三个基本概念
使用冗余掩盖错误
信息冗余
时间冗余
物理冗余
硬件冗余
软件冗余
基本思路
两种组织形式
复制进程的管理
K容错
分布式共识
目标
两类错误
通信错误
处理器错误
交互一致性模型
将军副官模型
口头协议
书面协议
可靠组播
基本的可靠组播
无等级的反馈控制
分等级的反馈控制
原子多播
虚拟同步
分布式提交
涉及的问题
2PC
阶段
参与者崩溃
存在的问题
3PC
条件
阶段
故障恢复
定义
两种恢复形式
两种检查点

可靠系统的指标

1. 可用性：在任何给定的时刻，系统都可以正确地操作；
2. 可靠性：系统可以无故障地持续运行；
3. 安全性：系统偶然出故障的情况下能正确操作而不会造成任何灾难；
4. 可维护性：发生故障的系统被恢复的难易程度。

基本概念

1. failure：系统不能兑现其承诺
2. error：系统状态的一部分，可能导致failure
3. fault：造成error的原因

使用冗余掩盖故障

信息冗余

添加额外的位来使错乱的位恢复正常

时间冗余

执行一个动作，如果需要就再次执行

物理冗余

通过添加额外的设备或者进程使系统作为一个整体来容忍部分组件的故障

硬件冗余

1. 使用两台计算机来运行一个应用程序，互为主备。开销较大，但通常很管用。
2. 在更细粒度上进行冗余：冗余个别服务器、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

一个例子是三倍模块冗余：每个模块被复制三份，每一级有三个表决电路

进程冗余

基本思路

把同样的进程组织到一个进程组中。当消息发送到进程组中时，所有组成员都接收，如果一个进程失效，其他组成员可以接替它。

两种组织形式

1. 平等组：没有单点故障，但决策需要表决，导致一定的延迟和开销
2. 等级组：协调者故障会造成单点故障，但是做出决策比较方便

复制进程的管理

1. 主副本：采用等级组，包含一个主副本和若干个备份副本。协调者充当主副本，负责所有更新。如果协调者出错，备份副本通过选举接管。
2. 复制写：采用平等组，所有副本都参与处理请求，采用主动复制或者基于法定人数的协议来避免读写冲突和写写冲突。

K容错

如果进程是沉默故障，只需要K+1的进程既可以实现K容错；如果是拜占庭故障，需要2K+1个进程。

要求：在集中式系统中，需要有一个组服务器来收集，在分布式系统中，需要有可靠的全序多播协议（所有请求按相同的顺序到达所有服务器）。

分布式共识

目标

所有非故障进程在有限的步骤内达成共识

两类错误

通信错误：两军问题

在不可靠通信中，两军问题无解。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）。

处理机错误：拜占庭将军问题（和容错不一样的是，这里传递的是私有的消息）

交互一致性模型

对于给定的 m，n>0，是否可以设计一种基于消息交换的算法，使得每个非故障处理器能够计算一个值向量，该向量对于每个n个处理器都有一个元素，满足以下条件：

1. 非故障处理器计算出的向量完全相同；
2. 该向量中对应于某个非故障处理器的元素是该处理器的私有值。

解决方案：

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

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。

将军-副官模型

上述问题的一种简化模型是将军-副官模型：

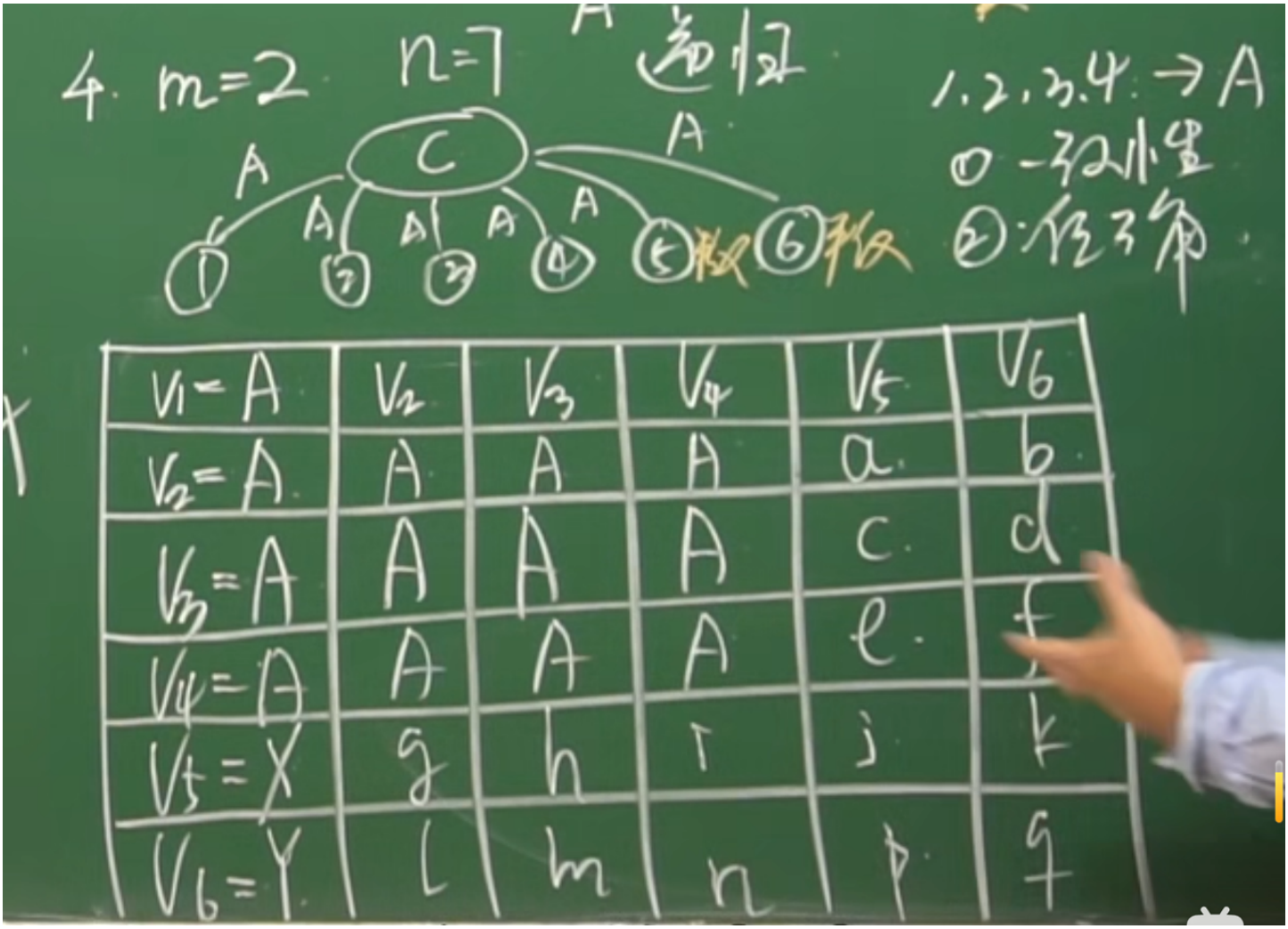
将军向n 名副官发出命令，满足：

- 所有忠诚的副官都服从同一命令；
- 如果将军是忠诚的，那么每个忠诚的副官都会服从他发出的命令

口头协议OM(m)

m表示叛徒的数量

1. OM(0)：将军把命令发给每个副官，每个副官执行将军的命令，若未收到，则执行默认命令；
2. OM(m)：将军把他的命令发给每个副官；每个副官接收到将军的命令后作为临时将军，将命令传输给其他n-2个副官，并递归调用OM(m-1)；最后，每个副官根据从其他副官收到值取众数。



以V2为例，1号节点不会直接认为2号节点发过来的A就是V2，因为他也不知道2号节点是不是叛徒，所以他就和V3~6确认，让他们告诉自己2号给他们发的是什么，然后取众数得到V2。

书面协议

假设：

1. 每条发送的消息都能被正确送达
2. 消息的接收者可以知道消息的发送者是谁
3. 可以检测到消息的缺失
4. 忠诚将军的签名无法被伪造，并且对其签名消息内容的任何修改都可以被检测到;任何人都可以验证将军签名的真实性

步骤：

1. 将军签名并发送他的命令给每个副官；
2. 对于每个副官i，如果从将军收到一个v:0的消息，且它还没有收到任何命令，就令V_i={v}，并加上自己的签名，把v:0:i发送给其他副官；
3. 如果副官接收到一个v:0:j_1:...j_k的消息，并且v不在V_i中，就把v添加到V_i里面，如果k<m就把v:0:j_1:...j_k加上自己的签名发送给不同于j_1:...j_k的副官；
4. 当副官不再接收到消息的时候，就遵循V_i的众数行动

可靠组播

基本的可靠组播通信

发送进程给每个发送的组播消息分配一个序列号，并将每个消息在本地存储。接收方收到没问题就回复ACK，否则就告诉发送方丢了哪个，要求发送方重发。存在的问题是发送方可能会被大量ACK淹没，出现反馈拥塞。

一种改进是接收方不对消息接收进行反馈，只在丢失时才反馈。可以减少反馈规模，但是发送方不得不永久保存消息。

无等级的反馈控制

当接收方发现丢失一条消息后，就给组内其他成员组播它的反馈，组播反馈可以抑制组内其他成员的反馈，因此一个组只会有一个重发请求。

分等级的反馈控制

每个子组指定一个本地协调者，负责子组内接收方的重发请求，协调者自己也有历史缓存。

原子多播

如果消息的发送方在多播期间崩溃，那么消息要么发送给剩余的其他所有进程，要么被每个进程忽略，满足这种属性的可靠多播就被称为虚拟同步。

分布式提交

涉及的问题

一个操作要么被进程组里的所有成员都执行，要么都不执行

2PC

概述

两个阶段，分别是准备阶段和提交阶段。

准备阶段：首先协调者会给所有参与者发送一个VOTE_REQUEST请求，参与者收到请求后根据自己的情况给协调者反馈是VOTE_COMMIT还是VOTE_ABORT。

提交阶段：协调者收集来自参与者的所有投票，如果都是VOTE_COMMIT就给所有参与者发送一个GLOBAL_COMMIT，否则，就发GLOBAL_ABORT；参与者阻塞等待协调者命令，根据命令执行。

参与者崩溃

1. 初始状态：参与者尚未开始处理事务，不会影响协议的进行；
2. 准备状态：参与者正在阻塞等待协调者的最终决策，需要从协调者或者其他节点联系获得最终决策；
3. 中止/提交状态：只要保证操作时幂等的就可以。

存在的问题

1. 同步阻塞：各个参与者在等待其他参与者响应的过程中，无法进行其他操作。这种同步阻塞极大的限制了分布式系统的性能；
2. 如果崩溃的参与者进入了COMMIT状态，而其他都在READY状态，就会阻塞。关键问题是协议没办法进行本地决策，而必须依赖其他进程。

3PC

条件

1. 没有一个可以直接转移到COMMIT和ABORT的状态
2. 没有一个状态，它不能做出最后决策，并且可以由它直接转移到COMMIT状态

三个阶段

1. CanCommit：协调者向所有的参与者发送一个包含事务内容的canCommit请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应。各参与者如果自身认为可以顺利执行事务，则反馈Yes响应，并进入预备状态，否则反馈No响应。
2. PreCommit：协调者在得到所有参与者的响应之后，会根据结果有2种执行操作的情况：执行事务预提交，或者中断事务。
3. DoCommit：该阶段做真正的事务提交或者完成事务回滚。

如果说，其他所有都在READY状态，那直接ABORT就完了，也没有什么损失。

总结

两阶段提交分为准备阶段和提交阶段，优点是实现简单，消息轮数少；广泛支持和使用，缺点是，单点故障问题：协调者崩溃时，参与者可能无法完成事务决策，进入“阻塞状态”。**阻塞问题**：如果参与者在“准备阶段”等待协调者的决策，无法执行其他操作，影响系统性能。

三阶段提交在此基础上增加了预提交阶段，该阶段参与者执行事务并写入日志，但不提交。优点在于缓解了两阶段提交的阻塞问题，参与者在协调者崩溃时有更多判断余地，可以独立进行回滚或提交，减少阻塞风险。但是**协议复杂度更高**：三阶段增加了额外的通信轮数和状态管理。而且，三阶段并未完全解决阻塞问题，比如说再网络分区的情况下就无法解决。

工程上仍然推荐使用两阶段算法，原因是其实现简单，成本低，通信开销小，而且两阶段阻塞的情况发生概率不高，即使发生也可以在应用层通过工程化的手段解决，无需使用更复杂的3PC。

故障恢复

是什么

当发生故障以后，使崩溃的进程恢复到正确的状态。

两种形式的故障恢复

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

前向恢复（例如前向纠错码）

- 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

检查点

独立检查点

每个进程独立地设置本地检查点，依赖项的记录使进程可以联合回滚到一致的全局状态；但每个进程回退的状态可能不一致，需要继续回退，可能造成多米诺效应。

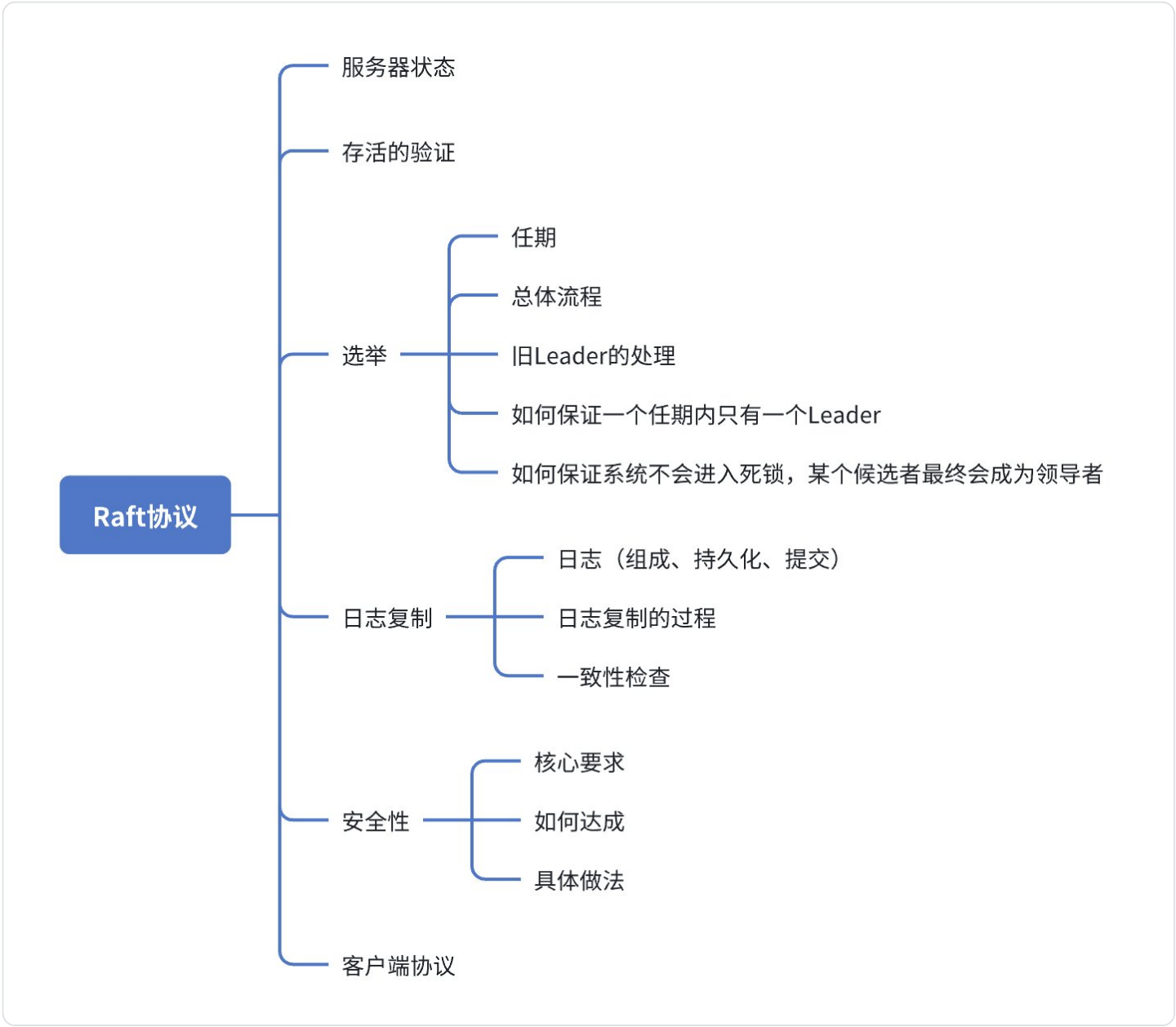
协调检查点

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。

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。

Raft协议



服务器状态

在任意时刻，每个服务器是以下三种状态之一：

- Leader：处理所有客户请求，日志复制
- Follower：被动处理
- Candidate：选举新Leader

存活的验证

- Leader定期发送心跳来表明自己存活
- 如果Follower在随机选举超时时间内没有收到任何RPC，那么就认为Leader崩溃了，并且开启新的选举

选举

任期

时间被分成若干任期，每个任期由两部分组成，分别是选举和正常操作阶段。

每个服务器维护一个当前的任期值，帮助识别和丢弃过期的信息。

总体过程

1. 发起选举的节点增加自己的任期号，将自己设置为Candidate状态，并给自己投票；
2. Candidate向所有其他服务器发送 RequestVote RPC 请求，直到：
    a. 如果候选者获得多数服务器的选票，就成为Leader，并给所有其他节点发送AppendEntries的心跳
    b. 如果收到了来自Leader的AppendEntries的RPC，回到Follower状态
    c. 如果没有服务器在选举超时时间内赢得多数选票，增加任期值，开始新一轮选举

旧Leader的处理

如果，旧领导者暂时断开连接，其他服务器选出新的领导者，旧领导者重新连接并试图提交日志条目，则可以使用任期检测过时的领导者，如果发送者的任期小于接收者的任期，接收者拒绝RPC；如果接收者的任期小于发送者的任期，接收者回退为追随者，更新任期，处理RPC。这样保证了被罢免的Leader无法提交新的日志。

如何保证一个任期内只有一个Leader

1. 每个服务器在每个任期内只能投一次票（投票情况需要持久化到磁盘）
2. 同一个任期内，两个候选者无法同时获得多数票

如何保证系统不会进入死锁，某个候选者最终会成为领导者

1. 随机选举超时时间，减少了多个候选者同时开始选举的概率，从而避免频繁的投票分裂。在网络条件正常的情况下，由于随机超时时间的差异，一个候选者会最早启动选举，并在其他服务器超时之前赢得多数票。这种情况在最小超时时间大于网络往返时间时工作得很好。

日志复制

日志

每条日志由索引号、任期号和指令组成；日志被持久化存储在磁盘中；当日志条目被大部分服务器所存储时，被认为该日志条目已提交；已提交的日志条目是持久的，不会被覆盖或丢失；已提交的日志最终会被状态机执行。

日志复制的过程

1. 客户端发送命令给Leader
2. Leader将命令放进自己的Log中
3. Leader使用AppendEntries RPC向Follower发送日志，如果Follower崩溃或反应慢，Leader会不断重试 AppendEntries RPC，直到日志成功复制。
4. 一旦某个日志条目被提交：
    a. Leader会将该指令发送给自己的状态机，并将结果发送给客户端
    b. Leader通过后续的 AppendEntries RPC 将日志的提交信息“捎带”通知给Follower
    c. Follower收到日志提交信息后，也会将对应的日志条目应用到它们的状态机，保持一致

一致性检查

1. 领导者为每个追随者维护nextIndex，表示要发送给该追随者的下一个日志条目的索引，初始化为 `1 + 领导者的最后一个日志索引`
2. Leader每次发送AppendEntries RPC都会附带nextIndex-1日志的索引和任期。
3. Follower收到了之后会检查本地的最后一条日志是否匹配Leader提供的信息，如果匹配，则接受新日志条目，并追加到本地日志中，否则，拒绝这次请求。
4. 若Follower拒绝请求，领导者会通过减少nextIndex，尝试找到与Follower日志中匹配的最近条目。
5. 一旦找到匹配位置，Follower会覆盖冲突部分。

Raft的安全性

核心要求

要求如果某个日志条目已经提交，那么该条目将出现在所有未来领导者的日志中。

具体做法

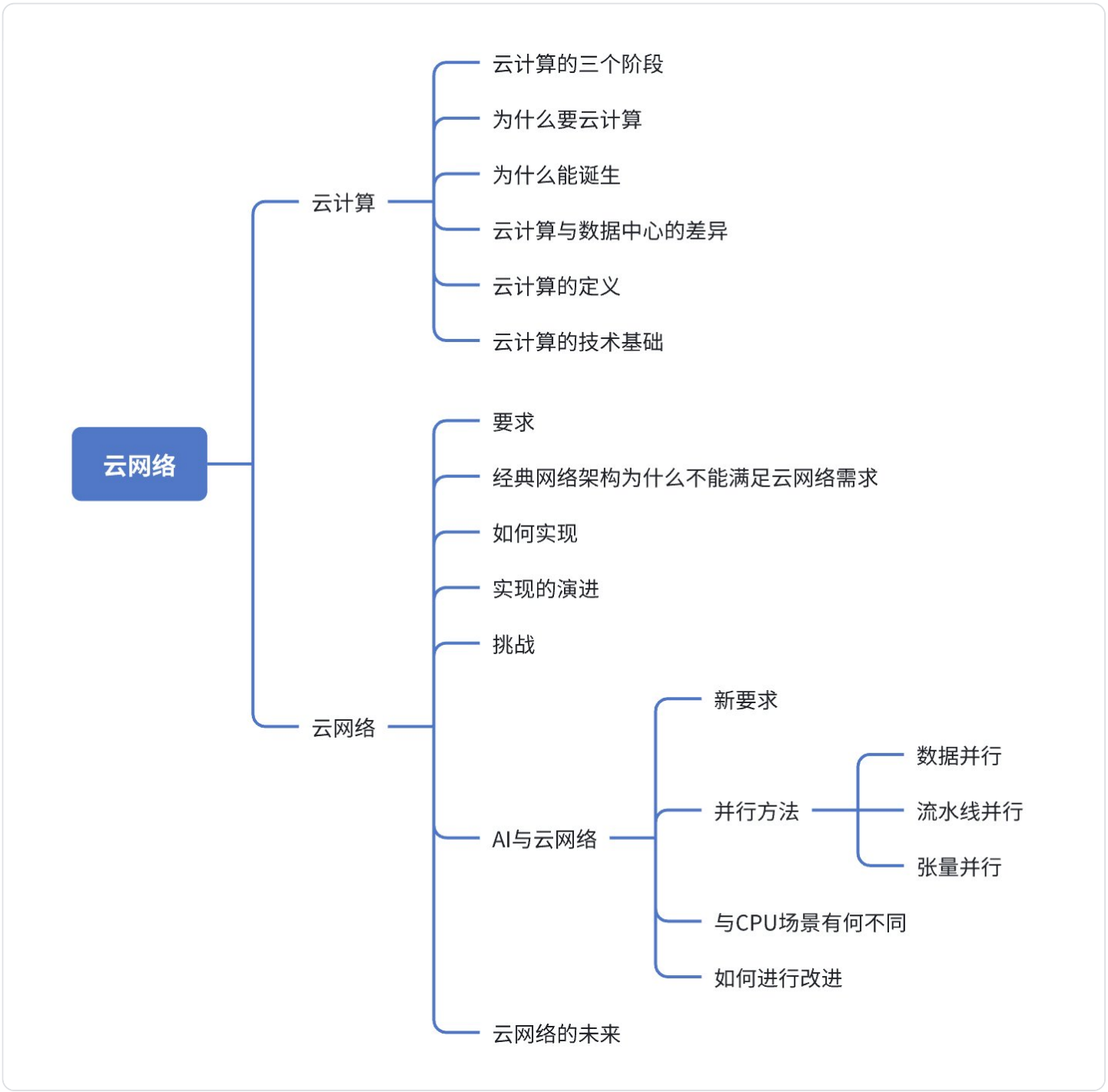
1. 一个候选者在发起选举时，必须包含自己的日志中的最后一个条目的索引和任期。一个节点在投票给候选者时，必须检查候选者的日志是否比自己的日志更新（任期更新或者任期相同索引更大）。这样，可以保证选出的领导者的日志是最新的。
2. 一个领导者在复制日志时，必须包含自己的日志中的最后一个条目的索引和任期。一个节点在接受日志时，必须检查领导者的日志是否与自己的日志匹配。这样，可以保证领导者的日志是一致的。

3. 一个领导者在提交日志时，必须保证自己的任期与日志条目的任期相同。这样，可以保证领导者不会提交过期的日志条目。

客户端协议

1. 客户端给Leader发送指令：如果不知道当前领导者，可以联系任意服务器，该服务器会将客户端重定向到当前领导者。
2. 领导者仅在命令被记录、提交并执行后才响应：确保了命令在系统中达成一致并持久化后，客户端才会收到响应
3. 如果请求超时，客户端会重新向新的领导者发送命令。
4. 为了确保即使在领导者故障的情况下也能实现“exactly-once”语义，客户端需要在命令中嵌入唯一ID，领导者在接受请求前检查日志中的ID条目，确保命令不会被重复执行。

云网络



云计算

三个阶段

1. 自建：机房水电服务器全部自己建
2. 数据中心托管：运营商提供地方和水电，客户提供服务器
3. 云数据中心：像买水电一样买云资源

为什么要云计算

1. 成本：传统IT基础设施通常需要大量的前期投资以及长期的维护费用。云计算的模式下只需为实际使用的资源付费，避免了大量的资本投入，并减少了运营和维护的成本。

2. 弹性，提高资源利用率：云计算允许用户根据需要随时增加或减少计算资源，避免了为了满足峰值需求而带来的大量资源闲置。
3. 提高灵活性和敏捷性：云计算让开发人员能够快速部署和测试新应用或服务，而无需等待传统硬件的采购和配置，同时也可以减小创业企业的沉没成本。

云计算为什么诞生

1. 客户有诉求
2. 企业有大量基础设施闲置
3. 有技术能力

云计算与数据中心的最大差异

最大的差异在于商业模式，从一次性购买转向按需购买。

云计算的定义

1. 按需无限量的供应计算资源（资源是复用的，可以削峰填谷）；
2. 消除云用户的预先承诺（我想用的时候就用，只要给你花钱，不需要提前订购）；
3. 短期使用的计算资源根据实际需要
4. 通过规模效应来降低成本
5. 通过虚拟化和复用的方式来提高资源的利用率

云计算的技术基础

1. 虚拟化将物理设备拆成一个一个的虚拟设备分配个各个租户
2. 端到端从底向上的自研技术体系（包括交换机、骨干网、操作系统、数据库……，这样才能满足客户的异构需求）

云网络

要求

带宽

时延

抖动

租户隔离（云网络跑的海量用户，每个用户要有隔离的网络）

弹性可迁移

经典的网络架构为什么不能满足云网络需求

1. 隔离性比较差（传统的是三层到核心，下面全是二层，二层靠吼）
2. 虚拟机迁移受地理位置约束（云计算中虚拟机可能会为了削峰填谷从一台机器迁移到另一台机器，但是IP不能变，这要求二层域要足够大（因为跨二层IP就会变），但是传统网络为了避免网络风暴，二层域很小）
3. 私网地址不足
4. 自组网的要求（每个租户有自己的ACL策略）

如何实现

使用VXLAN隧道技术，将不同用户的网络报文封装在隧道内，外层添加隧道头表示物理位置和所属租户。

演进

虚拟化的演进是由私有云（严格来说不是云，只是底层使用了虚拟化，没有改变商业模式）开始的，使用Linux内核实现VXLAN，性能比较差。

后来，华为、思科等设计了支持VXLAN的物理交换机，但无法用于公有云，因为这些交换机容量太小。

现在，云计算在可编程硬件（ 如网卡里的FPGA）中利用软件定义网络（SDN）来实现VXLAN。

挑战

1. 可靠性
2. 弹性
3. 体验
4. 故障
5. 在线升级

AI大模型对网络的新要求

原因：一个GPU无法满足这么大的参数量，因此需要进行分布式并行计算。

并行方法

数据并行

在数据并行中，每个计算节点GPU都持有一份完整的模型副本，但处理的数据不同。

- 将训练数据拆分成多个小批次，每个计算节点（GPU）处理一个小批次的数据。
- 所有计算节点分别计算梯度，然后将梯度进行聚合（通常使用平均值），并将结果同步更新到每个节点的模型参数。

优点：

- 易于实现，适用于大多数模型。
- 可以充分利用多个计算节点的计算能力。

缺点：

- 在大模型中可能存在显存不足问题

流水线并行

将模型的不同部分分配到多个计算节点上，以流水线的方式处理数据。例如，第一个节点负责处理模型的第一部分，计算完成后将中间结果传递给下一个节点，以此类推，直到最后一个节点完成所有计算。

优点：

- 可以平衡计算负载，充分利用所有计算节点的资源。
- 有助于减少显存需求，适合非常深的模型。

缺点：

- 流水线阶段间存在数据传输延迟。
- 每个阶段的数据处理需要一定的时间，会有一定的时间损耗。

张量并行

将模型中的张量沿特定维度进行拆分，每个计算节点只负责部分张量的计算。例如在矩阵乘法中，可以将输入或权重矩阵沿某个维度分割，然后在多个节点上并行执行乘法，再将结果组合。

优点：

- 可以有效降低单个节点的显存占用，因为每个节点只处理张量的一部分。
- 在层级内实现了计算并行化，提高计算效率 。

缺点：

- 实现复杂度较高，需要对模型进行拆分和协调计算结果。

新挑战（与CPU场景有何不同）

- GPU之间通信的流数量比较少（之前通用计算流的数量非常多），但是每个流的通信量非常大
- 对时延非常敏感，要快速进行同步
- 波峰波谷的现象（周期性）非常明显

怎么进行改进

将原来的一张网改成三张网（业务网：负责拉数据到GPU；Scale Out网络和Scale Up网络：用于GPU之间的通信，Scale Up是计算机总线网络PCIE的延伸，直接去读写另一个GPU的显存，使得单一系统更加强大；Scale Out使用RDMA的方式来构建分布式系统，让更多独立系统来共同完成任务）

（RDMA允许应用不经过内核直接通过网卡把数据发出去，跳过了许多内存的拷贝，性能比较高，时延比较低，可以提升吞吐）

云网络的未来

利用AI来为网络的运维服务

云边协同的自适应资源调度

任务调度

多个任务合理分配和各种有限资源的有效管理，从需求侧来说，希望达到速度更快、代价更低、精度更高的目标，从资源提供方的角度说，希望效率更高，成本更低，收益更好。

现代计算机体系的演进

1. 原始共享：一个机器连接多个终端
2. PC机：每个人有自己的PC机
3. 数据中心和云计算：利用分布式并行，集中化的算力计算池

资源调度

目标

需求匹配资源

理论建模

1. 任务分配问题：将N个具有固定执行时间的任务分配到M台相同的机器，这些机器同时开始执行，每台机器一个时候只能执行一个任务，目标是使得所有任务的总执行时间最小。——利用二近似求解
2. 资源部署问题：在N个地区建立工厂生产产品以满足M个消费者的需求，工厂本身有建设成本，工厂和消费者有运输成本，要求是在满足消费者需求的条件下使得总成本最低

实际的模型

资源维度

1. 计算资源
2. 数据资源：数据在哪里，计算就应该到哪里去，以前的很多场景都是计算密集型，但现在大部分是数据密集型的。

更多的优化目标

1. 时延
2. 抖动
3. 精度
4. 能耗

主要挑战

大量不确定，即使当前最下的最佳决策，未必能在未来工作的最好

数据中心的资源调度

主要做法

通过负载预测来解决不确定性

机制

虚拟机支持热迁移

调度目标

最小化代价（资源占用最小、容灾、安全性、稳定性）

策略

对这些虚拟机怎么编排到每台物理机

主要难点

怎么把多个虚拟机合理的分配到物理机，多个虚拟机的负载还是动态的，但对于数据中心资源充足，业务和资源特征相对来说还是稳定性的

怎么解决

两种方法：

1. 面向激增负载的资源预留：如果某些负载是激增的，把用户请求分成两块，一块是固定要用多少资源，一块是可能产生的激增，激增有一定的概率转移到另一个激增，产生了一个马尔可夫链的模型来刻画多个基本状态，然后把多个激增请求合并来实现资源的节省；
2. 将三层应用之间形成的虚拟网络拓扑嵌入到物理拓扑中？？？

边缘计算中的资源调度

边缘智能是使能边缘设备执行智能算法的能力，是智能应用落地的新模式

目前大量的数据和计算没有集中在数据中心，而是分散在各个终端（有些实时性的业务必须要在边缘完成），需要确定哪些任务在云端执行，哪些在边缘执行。

1. 用户请求量波动更大

2. 资源性能波动更大（特别是边缘设备）

第一类不确定性：不确定性本质是确定的，只是在当前决策前可以获知：

1. 实时获取负载状态，根据负载量调整资源配置
2. 优化长期效益，如果能预测负载变化趋势的话，可以进行提前进行布局

第二类不确定性：不确定性本质是确定的，但是仅在决策后可以获知：

（比如说你要在手机上部署一个模型，你到底部署多少参数量的模型，只有你决策部署后才能知道适不适合）

1. 利用边缘负载变化趋势来调整决策

第三类不确定性：不确定性本质是随机的，既不能在决策前获知，也不能在决策后完全确定

1. 容错机制与冗余设计