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

- 访问透明性：隐藏数据表示的差异和访问资源的方式。
- 位置透明性：隐藏资源的定位方式。
- 迁移透明性：隐藏资源移动。
- 重定位透明性：允许资源在使用时移动位置。
- 复制透明性：隐藏资源的复制。
- 并发透明性：隐藏资源被多个用户共享的情况。
- 故障透明性：隐藏资源的故障和恢复。
- 持久透明性：隐藏资源是在内存还是在磁盘。

开放性

- 系统应提供完整和中性的服务规范，提高互操作能力和可移植性。
- 系统应灵活、可扩展，可以组合不同开发者开发的组件。

可伸缩性

- 规模可伸缩：能够增加更多用户和资源。
- 地理可伸缩：用户和资源可以相距很远。
- 管理可伸缩：能容易管理相互独立的组织。

可伸缩性问题

规模伸缩受限原因

- 分布式算法没有全局时钟，没有系统完整状态信息，每台机器仅根据本地信息决策，单台机器故障不会使整个算法崩溃。

地理伸缩受限原因

- 同步通信，WAN通信不可靠，集中式服务。

管理伸缩受限原因

- 不同管理组织的资源使用、管理和安全策略冲突。

具有可伸缩性的系统实例

- DNS：通过划分名字空间来实现。

改善系统可伸缩性的方法

- 在体系结构层面：克隆数据和服务，拆分数据和服务，分布到不同地点。
- 在通信层面：利用异步通信，减少通信。

- 在容错层面：设计能隔离故障，避免单点故障。
- 在数据层面：使用复制和缓存，实现无状态或在浏览器端维护会话。

分布式系统的设计和实现

- 基于网络类型：局域网、广域网、传感网等。
- 体系结构：客户和服务器的分工与部署，通信方式。
- 故障处理：通信和硬件故障，共享数据一致性，系统可用性。
- 用户并发性：如何处理多于一个服务器能处理的客户，负载平衡。
- 系统安全性：如何保证安全。

开放性

- 服务规约应完整和中性，提高互操作性和可移植性。
- 系统应灵活、可扩展，可以组合不同开发者的组件。

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，如事件排序、基于时间戳的并发控制、程序编译等。

时间获取

- 铯原子钟定义的秒，时钟漂移（clock drift），时钟偏移（clock skew），时间的测量、发送和接收。UTC（协调世界时）。

计算机时钟

- 硬件时钟和软件时钟$C(t) = \alpha H(t) + \beta$。分布式系统中的每台计算机都有自己的时钟，不同计算机的时钟可能不同。

时间同步

- 时钟正确性：漂移率在已知范围内。软件时钟的单调性。
- 外部同步：使用权威外部时间源同步。
- 内部同步：时钟相互同步，但整体上可能与外部时间源有偏差。
- 同步算法：Cristian算法、Berkeley算法、NTP协议等。

同步系统和异步系统

- 同步系统：已知时钟漂移率、最大消息传输延迟、进程执行时间。
- 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。

时间同步算法

- 同步系统时间同步：一个进程在消息中发送本地时钟时间，接收进程调整时钟。
- 同步N个时钟的算法。

Cristian算法

设计目标：

- 使用时间服务器实现客户端和服务器之间的时间同步，使客户端的时钟尽可能与服务器时间一致。

具体思路：

- 客户端向时间服务器发送请求，记录请求发送时间。时间服务器返回当前时间戳，客户端根据请求的往返时间调整本地时钟。

实现方案：

1. **请求发送**：客户端向时间服务器发送请求，并记录发送请求的本地时间 T1。
2. **服务器响应**：时间服务器接收到请求后，立即返回当前时间 T2。
3. **接收响应**：客户端接收到服务器响应消息，并记录接收响应的本地时间 T3。
4. **计算和调整**：
    - 客户端计算请求和响应的往返时间（RTT）为 T3 - T1。
    - 客户端估计消息在网络中传输的一半时间为 RTT / 2。
    - 客户端将其时钟设置为 T2 + RTT / 2，以校正其本地时间。

优点：

- 简单易实现，适用于小规模网络环境。
- 计算复杂度低，易于部署。

缺点：

- 对网络延迟敏感，网络延迟不稳定会影响同步精度。
- 单点故障风险：如果时间服务器不可用，客户端无法进行同步。

Berkeley算法

设计目标：

- 在不依赖精确时间源的情况下，通过一个主节点（Master）与多个从节点（Slave）之间的协调，实现整个系统的时间同步。

具体思路：

- Master节点定期向所有Slave节点询问时间，并根据所有节点的时间计算一个平均时间，然后指示各个节点进行校正。

实现方案：

1. **时间询问**：
    - Master节点定期向所有Slave节点发送请求，询问它们的当前时间。
    - 每个Slave节点接收到请求后，返回其本地时间。
2. **时间计算**：
    - Master节点接收到所有Slave节点的时间后，估计往返时间，并根据所有节点的时间计算一个平均值 T_avg。
    - 计算平均值时，Master节点会考虑各个节点的时间偏差，丢弃明显异常的时间值。
3. **时间校正**：

- Master节点将计算出的平均时间 T_avg 与自身的时间进行比较，并指示各个Slave节点调整其时钟。
- Slave节点根据Master节点的指示调整其本地时钟，使整个系统的时间达到同步。

优点：

- 容错能力强，允许Master和Slave节点存在一定的时间误差。
- 适用于局域网环境，能够有效处理节点之间的时钟不同步问题。

缺点：

- 依赖于Master节点，Master节点的故障会影响整个系统的同步。
- 对网络延迟和往返时间的估计要求较高。

NTP协议

设计目标：

- 提供一个在广域网环境下精确且可靠的时间同步服务，使得网络中的所有计算机能够与标准时间源同步。

具体思路：

- 通过分层次的时间服务器架构和多种同步模式，实现时间同步。使用复杂的算法和统计方法来过滤和校正时钟偏差。

实现方案：

1. **层次结构**：
   - NTP采用分层结构，最顶层的Stratum 0服务器连接到精确时间源（如GPS或原子钟）。
   - Stratum 1服务器与Stratum 0服务器同步，Stratum 2服务器与Stratum 1服务器同步，依次类推。
2. **同步子网**：同步子网使用Bellman-Ford算法的变种，构建以主服务器为根的最小权重的支撑树。
3. **消息交换**：
   - 每个NTP消息包含三个时间戳：消息发送前的本地时间 T1、接收消息前的本地时间 T2 和当前时间 T3。
   - 服务器之间相互交换消息，使用这些时间戳来计算往返延迟和时钟偏移。
4. **时间校正**：
   - 客户端根据收到的NTP消息中的时间戳，计算时钟偏移（theta）和往返延迟（delta），调整本地时钟。
   - 计算公式：
     - 时钟偏移（theta） = (T2 - T1 + T3 - T4) / 2
     - 往返延迟（delta） = (T4 - T1) - (T3 - T2)
5. **故障处理**：
   - 提供冗余服务器和路径，确保即使部分服务器不可达，整个系统仍能继续提供服务。
   - 服务器之间相互监控，发生故障时，系统能够自动调整，选择新的同步源。
6. **安全机制**：使用认证技术防止恶意攻击，确保时间同步信息的完整性和正确性。

优点：

- 高精度：能够在广域网环境下提供高精度的时间同步。

- 高可靠性：通过冗余设计和容错机制，确保系统的可靠性。
- 可扩展性：分层结构和同步子网设计使得NTP可以扩展到大规模网络。

缺点：

- 实现复杂，配置和维护成本较高。
- 对网络带宽有一定要求，网络环境不稳定可能影响同步精度。

分布式系统的状态

分布式系统是一系列协同工作的进程集合，通过消息通信实现互操作。全局状态由两个部分组成：

- **局部状态集**：系统中每个进程的当前状态，包含进程的所有变量和数据。例如，一个进程的局部状态可以包括它正在处理的数据、变量的当前值以及进程的执行状态。
- **消息通道状态集**：消息通道中传输的消息的当前状态。它包含在消息传输过程中未被处理的所有消息序列。例如，如果进程A向进程B发送了一条消息，但进程B尚未收到该消息，那么该消息会被记录在消息通道状态集中。

观察系统全局状态的困难

缺乏全局时间：分布式系统中没有一个全局的时钟，各进程的时钟可能不同步。这导致无法通过简单的方法确定所有进程在某一时刻的状态。

- **时钟同步问题**：由于每个进程的时钟都可能有偏差，不同进程的事件记录的时间戳不一定能直接对比。例如，进程A和进程B的时钟可能有不同的偏差，即使它们记录的时间相同，实际上事件发生的顺序可能不同。
- **一致性问题**：无法保证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。即使在每个进程本地状态一致的情况下，由于消息传输延迟和处理顺序不同，整体系统状态可能不一致。

割集（Cut）

割集（Cut）是系统全局历史的一个子集，用于表示系统的某一时刻的执行状态。

- **系统全局历史**：所有进程的事件序列的集合，包括每个进程的内部事件、发送消息和接收消息的事件。例如，进程A的事件序列可以包括读取文件、处理数据、发送消息等。
- 一致性割集：如果割集中包含一个事件，则必须包含该事件之前发生的所有事件。这确保割集能够反映系统的实际状态。例如，如果进程A在割集中发送了一条消息给进程B，那么进程B在割集中必须包含接收到这条消息的事件。这样才能保证割集反映了实际的系统状态。
- **全局状态**：由割集定义的系统的状态。例如，如果割集包括进程A在发送消息和进程B在接收消息之前的所有事件，那么全局状态就包括了这些事件发生时的系统状态。

Chandy和Lamport的快照算法

设计目标：

- **记录进程状态和通道状态**：在分布式系统中捕获一致的全局状态，帮助诊断系统问题，如死锁、资源泄漏等。
- **假设**：通道和进程无故障，通信可靠。进程在快照时可以继续执行和通信。

实现方案：

1. **初始触发**：任意一个进程（称为“启动进程”）可以随时启动快照过程。它记录自己的状态，并向所有出站通道发送一个标记消息（Marker）。

2. **标记发送规则**：
    - 当一个进程记录了它的状态后，它必须在每个出站通道上发送一个标记消息。
    - 这个标记消息必须在该进程发送任何其他消息之前发送。
3. **标记接收规则。**当一个进程接收到一个标记消息时：
    - 如果这是该进程第一次接收到标记消息，它必须记录它的当前状态，并在每个出站通道上发送标记消息，然后开始记录从其他入站通道接收到的消息。
    - 如果该进程已经记录了状态，则它必须记录从收到标记消息之后到它记录状态之前，通过该通道接收到的所有消息。

具体步骤：

1. **启动快照**：启动进程 P1 记录其本地状态，并向其所有出站通道发送标记消息。
2. **其他进程响应。**其他进程（如 P2，P3）在收到标记消息时：
    - 如果是第一次收到标记消息，记录其本地状态，并向所有出站通道发送标记消息。
    - 开始记录所有从其他入站通道收到的消息。
3. **记录消息**：各进程在记录状态后，继续执行，并记录在此期间从其他进程接收到的所有消息，直到所有通道都收到标记消息。

快照算法的性质：

1. **一致性**：算法确保每个进程在记录状态的同时，确保所有消息在传输过程中都被正确记录，形成一致的全局状态。
2. **终止性**：算法在有限时间内终止。所有进程最终都会收到标记消息，并记录所有通道的状态。
3. **无干扰性**：在快照过程中，各进程可以继续执行和通信，不会中断正常操作。

假设有三个进程 P1、P2 和 P3，通过通道 C1、C2 和 C3 相互通信。

1. **启动快照**：P1 启动快照，记录其状态 S1，并向 C1 和 C2 发送标记消息。
2. **P2 接收标记消息**：P2 从 C1 收到标记消息，记录其状态 S2，并向 C3 发送标记消息，同时记录从 C3 接收到的所有消息。
3. **P3 接收标记消息**：P3 从 C2 收到标记消息，记录其状态 S3，并向 C1 发送标记消息，同时记录从 C1 和 C2 接收到的所有消息。
4. **消息记录完成**：当所有进程都记录了状态，并接收到所有标记消息后，快照过程完成。全局状态由 S1、S2 和 S3 以及通过 C1、C2 和 C3 记录的消息状态组成。

快照算法的应用

- 稳定性质（Stable Property）：系统达到某一状态后，不会再变化的性质，如死锁、终止等。
- 稳定性检测算法：记录全局状态，判断系统是否满足稳定性质。

事件排序

- Lamport的发生在先（happened-before）关系，用于事件排序。定义了事件之间的因果关系。

Lamport逻辑时钟

- 逻辑时钟是一个单调增长的软件计数器，每个进程维护自己的逻辑时钟。计算规则确保事件的发生在先关系。

向量时钟

- 系统中每个进程与一个向量相关联，向量时钟用于给事件加时间戳。计算规则确保事件的发生在先关系。