xeCJK/punctuationquanjiaoxeCJK/punctuationhangmobanjiaoxeCJK/punctuationbanjiaoxeCJK/punctuationkaimingxeCJK/
xeCJK/punctuationquanjiao

5 命名系统

名称用于表示分布式系统中的**实体**。想要操作一个实体，我们需要使用这个实体的**访问点**。访问点是一种特殊类型的实体，其名称称为**地址**。实体访问点的地址可以简称为实体的地址；一个实体可以有多个访问点，每个访问点都有一个地址。名称通常组织成**名称空间**（namespace）。名称空间是能够识别的所有有效名称的集合，例如名称"..."作为计算机的DNS 名称是不可接受的，而名称"www.another-name.net"即便不一定被绑定，也是有效的名称。

实际上很少使用地址作为实体的名称，因为这种方式不灵活、用户不友好。更好的办法是某个服务与独立的名称相关，而名称与地址无关，我们称这种名称是与**位置无关**的。

还有一类名称值得注意：**标识符**。标识符有三个特性：

(1) 一个标识符最多只能指向一个实体；

(2) 每个实体最多只能有一个标识符；

(3) 一个标识符总是指向同一个实体，即标识符不会被重用。

标识符不一定是**纯名称**（只用于比较异同的随机字符串），i.e. 标识符可以有内容。

5.1 无层次命名

本节介绍如何将标识符转换为地址。先介绍两种在局域网中适用的简单无层次命名方案。

5.1.1 广播和多播

局域网中定位实体是一件简单的事情：向局域网中的所有机器**广播**实体标识符，询问每台机器是否拥有该实体。只有拥有该实体的机器才会回复消息，并在回复消息中包含实体的地址。ARP 就是这样做的，这里的实体标识符是 IP 地址，访问点地址是 MAC 地址。

随着网络的膨胀，广播开始变得低效。一种可能的解决方法是转为**多播**，也就是只有符合条件的一组主机才会接收到请求。例如，可以允许主机加入特定的多播组，这样的组由多播地址来标识；当主机向一个多播地址发送消息时，网络层会提供最尽力的服务来把消息发送给全部组成员。

多播还有一种应用场景：为实体创建一个多播地址，让多播地址与实体的所有副本相关联。向该多播地址发送请求时，每个副本都会用它当前的 IP 地址进行响应，这样就能选择最近的实体副本。

5.1.2 转发指针

转发指针可以用于定位不断变化的实体。当实体从 A 移动到 B 时，它会在 A 留下一个指针，指向它在 B 中的新位置。客户使用传统的命名服务找到实体（的空壳）以后，就可以顺着转发指针链来查找实体的当前地址。找到实体以后，客户会更新自己的引用，从而避免下次再次遍历转发指针链。转发指针的**缺点**在于：

(1) 链可能很长，定位实体的开销会变得很大；

(2) 链中的所有中间位置都必须维护它们的那一部分转发指针链；

(3) 链很脆弱，易断开，只要有一个转发指针丢失，就无法再到达实体。

5.1.3 基于宿主位置（Home Location）的方法

宿主位置方法使用一个**宿主机**来持续跟踪实体的当前位置。这种方法为实体维护一个宿主机，宿主改变位置时将自己的位置告知宿主机。当客户需要与实体通信时，使用宿主位置联系宿主机，宿主机视情况将请求转发给实体。宿主机起到的就是代理的作用。宿主位置方法确实能能在大型网络中定位移动实体，但这种方法的**缺点**在于：

(1) 宿主需要伴随实体的整个生命周期；

(2) 宿主位置必须是固定的，这样才能保证客户能够找到宿主；

(3) 宿主位置可能与实体本身处于完全不同的位置，这样会增加通信延迟（最差的情况下，实体和客户端可能就紧挨着，但访问宿主需要远涉重洋）。

5.1.4 分布式哈希表(Distributed Hash Table, DHT)

标识符是一类键,地址是一类值,因此使用哈希表处理地址查找是很合理的。分布式环境下就要使用**分布式哈希表**,合理得不能再合理了。分布式哈希表中,最为典型的是 **Chord 系统**。

Chord 使用一个 m 位(通常是 128 或 160 位,大小 2^m)的环形标识符空间,将键和值都通过哈希函数(如 SHA-1)映射到这同一个环形空间中。令 $\text{succ}(k)$ 表示环上第一个大于等于 k 的标识符,即 k 的后继;这之后,令 $\text{lookup}(k) = \text{succ}(k)$,即可在哈希表中查找键 k 对应的值。每个 Chord 节点维护一个最多有 m 个实体的**指状表**(finger table),满足以下条件:

$$FT_p[i] = \text{succ}(p + 2^{i-1})$$

其中,FT_p 表示节点 p 的指状表。换言之,指状表中的第 i 项指向 $p + 2^{i-1}$ 之后的第一个节点。可以将指状表理解为从当前节点出发对整个空间的划分,越靠近的节点划分得越细致。这样,每次查找 k 时,节点 p 只需将请求转发给指状表中第一个大于等于 k 的节点即可。换言之,节点 p 转发给索引为 j 的节点 $q = FT_p[j]$,q 满足:

$$q = FT_p[j] \leqslant k < FT_p[j+1]$$

若 $p < k < FT_p[1]$,则 p 会将请求转发给 $FT_p[1]$。可以证明,一个查找通常需要 $O(\log(N))$ 步,其中 N 为系统中的节点数。图 ?? 演示了 Chord 查找 $k = 54$ 的查找过程,同时说明了指状表是如何将 $O(N)$ 的线性探测(只查找后继节点)降低到 $O(\log(N))$ 的。可以看到,如果把线性探测近似为环形空间的"切线",那么藉由指状表的探测就像是"弦"一样,这可能就是 Chord 名字的由来。

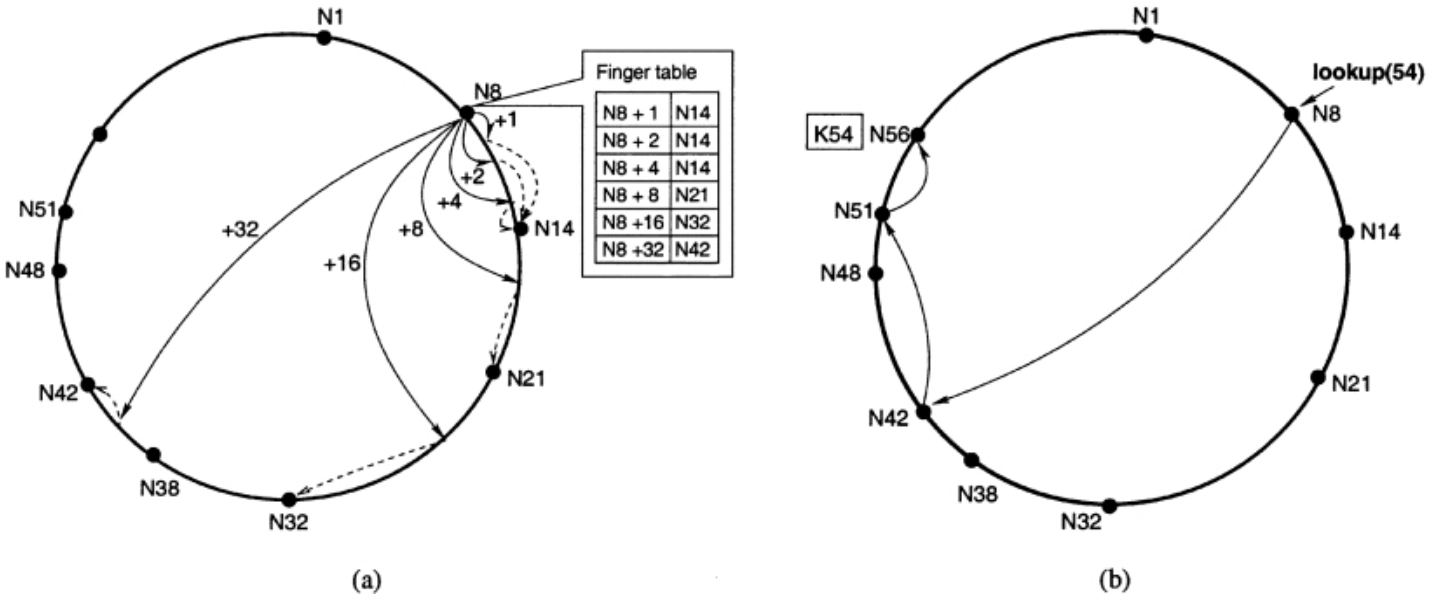


图 5.1: Chord 查找过程

在大型的分布式系统中,参与节点集可能总在变化。假设新节点 p 要加入,p 只与已有节点联系,并请求查找 $\text{succ}(p+1)$。一旦标识了该节点,p 就可以将自己插入到该环中。Chord 中,新节点加入后并不会立刻更新所有节点的指状表,而是通过后台进程定期更新。

Chord 的一个小问题是 p 和 $\text{succ}(p)$ 地理上可能相距很远,这样会导致查找的开销很大。**基于拓扑的结点标识符赋值**(标识符赋值时两个邻近结点所赋给的标识符也是靠近的)、**邻近路由**(结点维护一个转发请求的可选列表)、**邻近邻结点选择**(优化路由表,使得选择离最近的结点作为邻结点)都是为了解决这个问题。

例题集 5.1: Chord 系统

问题 1 Srini decides to move the content to a Chord DHT. He splits the lectures in 10 second chunks, which results in 65536(2^{16}) files. He uses SHA3-256 (256bit long hash) to generate the identifiers used in his DHT. Finally, he recruits 1024(2^{10}) users to participate in the Chord DHT as storage nodes.

(1) How many hops does it typically take to lookup a single chunk?

答: $\log_2 1024 = 10$

(2) How many different hosts are typically in the Chord finger table at any node?

答: $\log_2 1024 = 10$

问题 2 Which of the following are true about the Chord distributed hash table?

A. Has O(log(N)) lookup latency

B. Directly supports fuzzy or wildcard lookups for content

C. Uses content-based naming

D. Provides anonymity

答: A, C

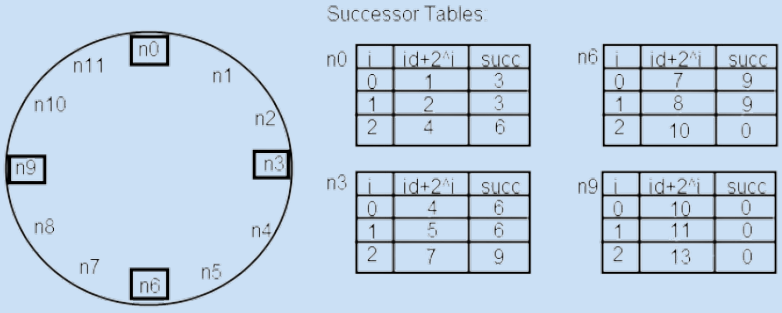
问题 3 Consistent hashing is used in a variety of distributed systems over more traditional hashing algorithms.

(1) Describe one way that consistent hashing performs better when failures occur (e.g. in DHTs or in CDNs).

答: Less content needs to be moved between nodes.

(2) Describe one way that consistent hashes perform better when nodes have differing views on the availability of servers.

答: Consistent hashing reduces the spread of requests across nodes when views are different.



问题 4 Suppose the above Chord setup:

(1) At what node do you expect to find key 10?

(2) At what node do you expect to find key 9?

(3) On average, how many hops are there when searching for a key?

(4) What route will a search for key 11 take, starting at node 3?

答: 待补充

问题 5 Suppose you want to build a large distributed system to store cookie recipes. Each cookie is stored using the recipe's name ("Grandma's Arsenic Cookie Surprise"). Unlike many P2P systems, in this one, the recipe data is small, so you wanted to store the actual data in the P2P system instead of leaving it on the original nodes. A friend says "Hey, I hear that Chord is very efficient — it doesn't take many lookups to find the node containing the data." Explain to your friend why directly throwing this data into a DHT such as chord, with key = recipe name, value = recipe data, is a really bad design decision. Brief - two sentences or so.

答: 待补充

问题 6 Consistent Hashing

(1) In a distributed hash table (DHT) using a *ring mapping*, explain how a key is mapped to a specific node?

答: Keys and nodes map to points in a ring space. A node handles all keys mapped between it and its successor.

(2) In the most basic form of DHT systems, nodes simply track their predecessor and successor. Very briefly state what problem this leads to as the network grows larger.

答: Performance problems as nodes forward a key resolution request to larger chains of neighbors.

(3) The Chord scheme overcomes this problem by having each node maintain a "finger table". Given starting

node j and destination node k, where $k > j$, how much is the distance between the two nodes reduced with each hop?

答:It decreases exponentially, or it is cut in half.

(4) In a Chord structured DHT with N nodes, how many hops would a lookup operation require?

答:$\log_2 N$

(5) Assume a Chord network in which node q is the successor of node p. During operation, node p discovers that its successor link is no longer consistent because q has updated its predecessor link. What does this change imply must have happened in the network?

答:A new node joined the network "between" p and q.

(6) An optimization to Chord involves storing several nodes for each entry in the finger table instead of just one. Explain an important benefit this optimization confers in a globally distributed DHT.

答:1) This allows routing based on proximity, which would reduce slow routes that criss-cross the globe. 2) It provides a fallback in case one of the nodes in the finger table is unreachable.

问题 7 The pseudocode for Chord'sjoin implementation, from figure 7 of the paper "Chord: a scalable peer-to-peer lookup service for Internet applications", is as follows: Ben observes that the Chord code for stabilization uses an indirect method to set the predecessor of a node, and proposes the following modifications to this pseudocode: Why is this change a bad one? (Give a brief explanation and a scenario that illustrates your explanation.)

答:If nodes with close node IDs join at the same time, only one might appear in the succ ring. For example, if a Chord ring consists of nodes 1 and 4, and nodes 2 and 3 join the ring concurrently, it's possible for both new nodes to have node 4 for a successor, and node 1 for a predecessor, and never find out about each other. In addition, if node leave the system, the ring doesn't repair because Ben deleted stabilize.

问题 8 Given the protocol in the Chord paper, is `lookup`(k) guaranteed to return the IP address of the node that stores k? (Explain your answer briefly.)

答: No. During stabilization, a node responsible for a key may not have that key yet. The application is supposed to retry when a lookup fails.

问题 9 Ben Bitdiddle is designing a location service for an Internet instant messaging system. There are lots of users, each with a computer. Each user has a unique ID. As each user moves, the user's IP address changes; the location service needs to keep its mapping from user ID to IP address up to date as these changes occur. The instant messaging application on each user's computer knows when the computer's IP address changes.

Ben is considering two different designs. One, the "centralized design", uses a single computer as a server to store the current mapping from IDs to IP addresses. When a user's computer changes IP addresses, it sends an update request to the server. When a messaging application running on user U_1's computer needs to find the current IP address of U_2, it sends a request to the server with U_2's ID, and the server responds with U_2's current IP address. Ben has in mind that the server would be an ordinary computer that never changes its IP address, perhaps his dorm-room PC. The messaging application would have the server's IP address built in.

The other design uses Chord. Each user computer would join the Chord ring as a Chord node, using a hash of the user's ID as the Chord ID. When user U_1 wants to send an instant message to U_2, U_1's messaging application would use a Chord lookup to find U_2's computer, given U_2's ID; a side-effect of this Chord lookup is that U_1's application would learn U_2's computer's IP address, to which it can address instant messages. When a user's computer changes IP addresses, it re-joins the Chord ring. Ben intends to modify Chord so that it recognizes that a join of a node with an ID that's already known but with a different IP address overwrites the old successor-list and finger-table information for that ID. Ben intends to run a few computers with well-known unchanging IP addresses to help new computers join the Chord ring.

Assume that user IDs are evenly distributed and random, that each user's computer changes IP address with about the same frequency, and that the popularity of different users (as instant message sources or recipients) is roughly uniform. No-one is trying to attack the system. Each computer in the system has a small chance of being unavailable due to temporary network or computer failures.

(1) Help Ben decide which design to use by marking each of these statements as true or false.

(1.a) Chord will result in less total network traffic than the centralized design.

答:False

(1.b) The server in the centralized design will handle more network traffic than any one computer in the Chord design.

答:True

(1.c) The work involved in serving lookups will be more evenly divided among all the computers with Chord than with the centralized design.

答:True

(1.d) Lookups are likely to complete more quickly in the Chord design than in the centralized design.

答:False

Ben builds the Chord-based design. Suppose there are a million computers in the Chord ring. For the duration of this question each computer is reliable and doesn't change IP addresses. The network is not entirely reliable: the probability of the network losing each Chord request during a lookup is 0.01 (one percent).

(2) What is the approximate probability that a Chord lookup will encounter at least one lost message? Circle the best answer.

0.01 0.02 0.10 0.18 0.90 0.99

答:0.10. It would be 0.18 but a hop is only taken to correct an ID bit, so we expect $0.5 \times \log(N)$ hops.

5.1.5 分层方法

这一节讲述了如何使用分层方法来解决无层次命名的问题。分层方法的基本思想是构建一个大型的分层搜索树,将底层网络划分为若干不重叠的域。域可以组合成更高层的(不重叠)域,以此类推。有且只有一个顶层域涵盖了整个网络。每个层次的每个域都有关联的目录结点,如果一个实体位于域 D 中,那么更高一层的域的目录结点将拥有一个指向 D 的指针。最低层目录结点存储了实体的地址,最高层的目录结点知道所有的实体。实际上这一节我一点也不想看。最好别考。

5.2 有层次命名

5.2.1 名称空间和名称解析

结构化名称的名称空间可以表示为带有标记的、具有两种类型的结点的有向图。**叶结点**(leaf node)表示一个具名实体,它没有分支边。叶结点通常存储关于它所表示的实体的信息和状态(如实体的地址、文件系统中的文件)。**目录结点**(directory node)具有一定数量的分支边,每条边用一个名称来标记。目录结点用于存储**目录表**,目录表存储了若干对(边标签, 结点标识符)。**根节点**只有分支边,没有进入边。很多命名系统只含有一个根结点,路径可以通过边标签的序列来表示。

名称解析(name resolution)是指将一个名称转换为一个实体的过程。知道如何启动以及在何处启动名称解析的机制称为**终止机制**(closure mechanism)。终止机制的难点在于,它不可能是完全显式的;必须知道一些隐式的、约定的信息,才能启动名称解析。比方说,我们知道 /home/user 是一个地址,但解析它的前提是获取根节点的地址,这就需要一些约定。再比方说,没有人能直接解析 0031 20 598 7784,除非知道这是一个电话号码。

5.2.2 名称空间的实现

大型分布式系统包含许多实体,可能会跨越很大的地理区域;在这样的系统中,需要使用多台名称服务器来完成命名服务的实现。此时,将名称空间**分成逻辑上的层**,分层组织名称空间会很方便。Cheriton 和 Mann 将名称空间分为以下三层:

(1) **全局层**(Global Layer)由最高级别的结点(即根结点以及根结点的子结点)组成。全局层通常是稳定的,即目录表很少改变。全局层的目录往往必须由多个管理机构共同管理。

(2) **行政层**(Administrational Layer)由单个组织内一起被管理的目录结点组成。

(3) **管理层**(Managerial Layer)由低层次、单个组织内、经常被修改的目录结点组成。如何将管理层的目录结点映射到本地名称服务器上是管理层的主要问题。

这些层次的特点如表 ?? 所示。

表 5.1: 名称空间的层次结构

	全局层	行政层	管理层
地理规模	全球	单个组织	单个部门
节点数量	少	多	巨大
响应时间	秒级	毫秒级	立即
更新传播	惰性	立即	立即
副本数量	多	少或无	无
客户端缓存	是	是	否

DNS 的名称空间就是这样的分层结构,如图 ?? 所示。名称空间还可以继续划分,例如 DNS 中的**区域**(Zone)。

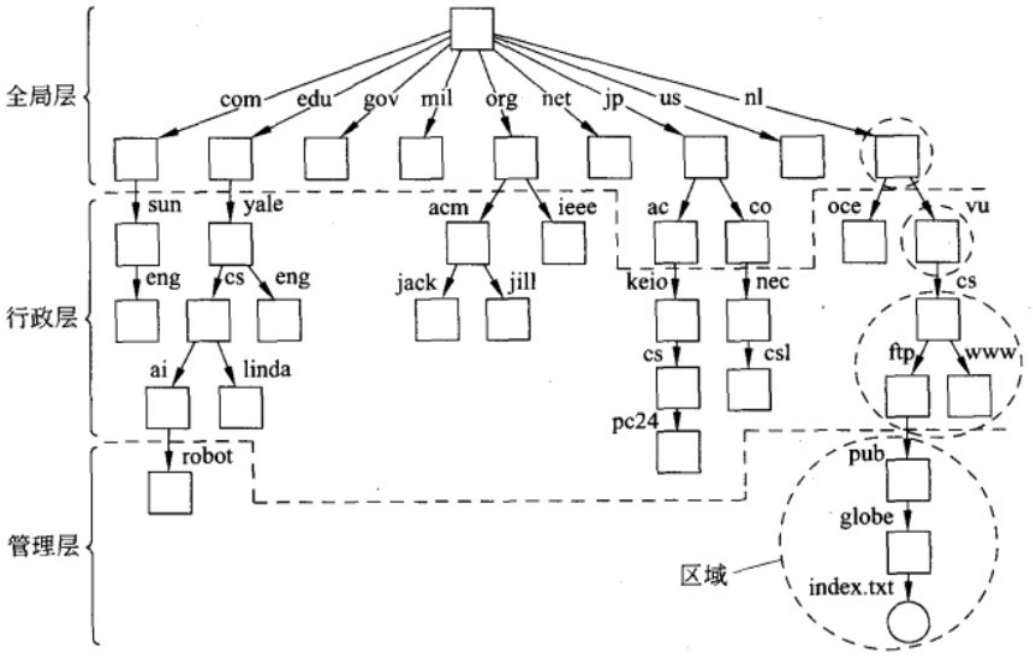


图 5.2: DNS 的名称空间

5.2.3 名称解析的实现

名称解析分为**迭代名称解析**和**递归名称解析**。后者对缓存的使用更加充分,通信开销也更小,因此更常用。

高层次名称服务器往往需要处理大量的请求。为了解决这个问题,我们可以假设高层次的目录结点很少改变,从而将目录结点映射到多个服务器上。这样,我们就可以在离客户端最近的服务器上启动名称解析。

5.2.4 示例:DNS

因特网的**域名系统**(Domain Name System,DNS)是当今使用的最大的分布式名称服务。

DNS 的名称空间是分层组织的,就像一棵有根的树。标识符是区分大小写的字符串,该字符串由字母和数字组成。标识符的最大长度是 63 个字符,完整路径名的最大长度是 255 个字符。路径名的字符串表示由一列标识符组成,从最右边开始,使用圆点“.”分隔各个标识符。根由圆点代表。举例来说,路径名 `root:(nl,vu,cs,flits)` 用字符串 `flits.cs.vu.nl.` 表示,这个字符串包括最右边的圆点,该圆点象征着根结点。为了方便阅读,我们通常省略这个圆点。

DNS 名称空间的每个结点都包含一个进入边(根结点除外,它不包含进入边),因此结点进入边的标识符也可用作该结点的名称。子树被称为**域**(domain),指向根结点的路径名称为**域名**(domain name)。注意,域名可以是绝对的,也可以是相对的,就像路径名一样。

结点的内容由一组**资源记录**(resource record)组成。有各种不同类型的资源记录,表 ?? 列出了最重要的一些记录。

表 5.2: DNS 中的资源记录

记录类型	描述
A (Address)	保存域的 IP 地址
CNAME (Canonical Name)	将一个域或子域转发到另一个域,不提供 IP 地址
MX (Mail Exchanger)	将邮件定向到电子邮件服务器
NS (Name Server)	存储 DNS 条目的名称服务器
SOA (Start of Authority)	存储域的管理信息
SRV (Service)	指定用于特定服务的端口
PTR (Pointer)	提供反向查询(根据 IP 地址查找域名)的域名

A 记录是最基础的 DNS 记录类型,它表示给定域的 IP 地址。例如,拉取 `cloudflare.com` 的 DNS 记录,A 记录当前返回的 IP 地址为 `104.17.210.9`。绝大多数网站只有一个 A 记录,但可以有多个,这是循环负载均衡技术的一部分,该技术可以将请求流量分配到托管相同内容的多个 IP 地址中的一个。**CNAME 记录**从别名指向规范名称,每个主机都假定拥有一个规范名称,或者说主名。别名由存储 CNAME 记录的结点实现,CNAME 记录包含主机的规范名称,有点像是符号链接。**MX 记录**指示如何通过 SMTP 协议对电子邮件进行路由。例如,代表域 `cs.vu.nl` 的结点拥有一条包含名称 `zephyr.cs.vu.nl` 的 MX 记录,名称 `zephyr.cs.vu.nl` 指向了一台邮件服务器。该服务器会处理所有发给域 `cs.vu.nl` 用户的邮件。一个结点中可以存储多条 MX 记录。**SRV 记录**与 MX 记录有关,SRV 记录包含处理特定服务的服务器名称。**NS 记录**指示哪个 DNS 服务器对该域具有权威性(即,哪个服务器包含实际 DNS 记录)。基本上,NS 记录告诉互联网可从哪里找到域的 IP 地址。一个域通常会有多个 NS 记录,这些记录可指示该域的主要和辅助域名服务器。**SOA 记录**存储有关域或区域的重要信息,如管理员的电子邮件地址、域上次更新的时间,以及服务器在刷新之间应等待的时间。**PTR 记录**与 A 记录完全相反,它提供与域名关联的 IP 地址。

DNS 的实现方式和之前描述的实现方式很类似。本质上,DNS 名称空间可以划分成全局层和行政层(管理层则由本地文件系统组成,形式上不是 DNS 的一部分,因此也不使用 DNS 来管理)。每个区域都由一个名称服务器来实现,从可用性考虑,这个名称服务器常常有很多副本。区域更新一般由主名称服务器完成:通过修改主服务器的 DNS 数据库,可以完成更新操作。辅名称服务器不直接访问该数据库,相反,它请求主服务器把内容传送给它,这在 DNS 术语中称为区域传送。DNS 数据库由一组(数量很少)文件实现,其中最重要的一个文件包含了特定区域中所有结点的所有资源记录。这种方法允许只用结点的域名来标识结点,这样,结点标识符就变成了一个(隐含的)文件索引。

例题集 5.2: DNS

问题 1 The DNS root and GTLD servers often come under attack. Give 2 reasons that we don't notice when they are under attack.

答:The records from the root and GTLD have long TTLs and are cached widely. The root and GTLD zones are replicated very widely and it is difficult to attack all locations at the same time.

问题 2 Which techniques are used in DNS to improve scalability?

A. Replication
B. Recursive queries
C. Caching / Soft-State
D. Partitioning of Namespace

答:A, C, D

问题 3 As the CMU's Distributed Systems class graduates ventured into the world, they spread their description of class lectures. As a result, the popularity of video recordings of the lectures skyrocketed (unfortunately, mostly for the jokes rather than the technical content) and now Srini needs your help in managing the storage and delivery of the files. Srini starts by building an Akamai-like CDN to deliver the files for his web site `www.srini-lectures.com`. As a first step, Srini:

1) registers the domain `srini-lectures.com` with two name servers `ns1.srini-lectures.com` and `ns2.srini-lectures.com`.

2) sets up two regional, low-level DNS servers - `east-ns.srini-lectures.com` and `west-ns.srini-lectures.com` - for clients on the east and west coast.

3) sets up two servers to deliver cached content with IP addresses `1.1.1.1` and `2.2.2.2`. Srini expects users to get the `1.1.1.1` address for `www.srini-lectures.com` if they are on the east coast and `2.2.2.2` if they are on the west coast.

(1) There are 6 name servers involved in the system:

- `a.root-servers.net`
- `a.gtld-servers.net`
- `ns1.srini-lectures.com`
- `ns2.srini-lectures.com`
- `east-ns.srini-lectures.com`
- `west-ns.srini-lectures.com`

For each of the following DNS records: 1) list *ALL* servers where the DNS record stored (do not list cached locations), 2) what will its TTL be (assume that the TTL can only be 1 minute or 1 day) and 3) why you chose that TTL (1-2 sentences)

(1.a) NS record that points to `ns1.srini-lectures.com`

答:`a.gtld-servers.net` with TTL 1 day

(1.b) NS record that points to `east-ns.srini-lectures.com`

答:`ns1.srini-lectures.com` and `ns2.srini-lectures.com` with TTL 1 day

(1.c) A record for `www.srini-lectures.com`

答:`east-ns.srini-lectures.com` and `west-ns.srini-lectures.com` with TTL 1 min

(2) On the first query (i.e. no cached information at the local name server) for `www.srini-lectures.com` from New York - which of the 6 name servers listed are contacted?

答: `a.root-servers.net`, `a.gtld-servers.net`, `ns1.srini-lectures.com` or `ns2.srini-lectures.com`, `east-ns.srini-lectures.com`

(3) On a second query, 5 minutes after the previous query, for `www.srini-lectures.com` from New York - which of the 6 name servers listed are contacted?

答:`east-ns.srini-lectures.com`

问题 4 True or False?

(1) Domain Name Service (DNS) servers remember which clients have cached DNS replies so that the servers can send invalidation messages when name bindings change.

答:False

(2) DNS is delegated hierarchically by having, e.g., the root nodes tell resolvers which servers to query for ".com", and so on, until the client's query can be answered.

答:True

(3) Domain Name System (DNS) resolvers use Paxos and invalidation messages to maintain the consistency

of cached records.
答:False
(4) Some Content Delivery Networks (CDNs) use DNS responses to direct clients to the closest CDN cache.
答:True
(5) DHCP is used to translate hostnames into IP addresses.
答: False. Domain Name Service (DNS) is used for name-address translation, while DHCP (Dynamic Host Configuration Protocol) is used for dynamically assigning IP addresses to networked end-hosts.

5.3 基于属性的命名

在基于属性的命名系统中,实体是由 (属性, 值) 对的集合来描述的,查询也是以这样的对来表达的,传统的解决方案就是在一个集中式数据库中进行搜索。还有其他的解决方法:其一是把 (属性, 值) 对映射到基于 DHT 的系统。其二是**语义覆盖网络**,即让结点维护一个内容相似结点的本地列表;每次查询时,首先查询相邻结点,找不到就进行有限的广播接着找。

6 同步

6.1 时间同步

> Quid est ergo tempus? Si nemo ex me quaerat, scio; si quaerenti explicare velim, nescio
>
> —— Augustine of Hippo, *Confessiones*

事件之间因果关系的概念是操作系统设计/分析、并行和分布式计算的基础。集中式系统中,每个进程都有一个共同的时间,可以通过这个时间来跟踪因果关系。例如,`make` 程序就使用文件的修改时间来判断是否需要重新编译。此外,分布式系统的正确性往往依赖于系统不变量的保证(例如,保证不存在死锁、保证分布式数据库的写访问不会授予多个进程、保证电子现金系统中所有账户的借记和自动柜员机支付的总和为零、保证对象只有在不再引用它们时才会被垃圾回收),这些不变量都依赖于时间的概念。

然而,在分布式系统中,每台机器都有自己的时钟,各个物理设备的本地时钟走时并不准确;此外,消息传递需要消耗时间,但网络的可变延迟使我们难以得知消息传递的耗时。这使得我们很难确定事件的发生顺序。这个问题的一种常见解决方法是使用**时间服务器**来同步时间,但这仍不足以解决网络可变延迟的问题,同步后的时间也未必精准。另一种解决方法是使用跳跃式的**逻辑时钟**来确定时间顺序,这种逻辑时钟充分捕获了时间与因果关系相关的单调性特性。

6.1.1 物理时钟

处理器的时钟一般依赖于石英晶体振荡器。尽管石英晶体的振荡频率相当稳定,但不同计算机中石英晶体的振荡频率仍有细微的差别。现代计时器的错误率大约是 10^{-5},即每小时产生 215 998 至 216 002 个中断。

NIST 提供了一个被称为 **WWV 短波广播**的服务,可以用来同步时钟。WWV 本身的精确度为 $\pm 1\,\mathrm{ms}$,但由于广播的延迟,最终的精确度只有 $\pm 10\,\mathrm{ms}$。GEOS 卫星能够提供 $\pm 0.5\,\mathrm{ms}$ 的 UTC,其他一些卫星提供的精度甚至更高。这些服务都需要准确地了解发送信号和接收者的相对位置,以便对信号传输延迟进行补偿。

如果一台机器上有 WWV 接收器,那么我们需要使其他所有的机器与它同步。如果没有一台机器有 WWV 接收器,每台机器都按自己的机器时间计时,这时我们的目标是尽可能使所有机器的时间保持同步。目前已经提出了很多时钟同步算法,这些算法的基础模型是相同的。假设每台机器都有一个每秒产生 H 次中断的计时器。当计时器产生中断时,中断处理程序将软件时钟加 1,软件时钟记录从过去某一约定时间开始的中断数。我们将这个时钟值称为 C。更具体地说,当 UTC 时间为 t 时,机器 P 上的时钟值为 $C_p(t)$。最理想的情况是对所有的 p 和 t,都有 $C_p(t) = t$;换言之,$\mathrm{d}C/\mathrm{d}t$ 的理想值为 1。同步算法的效果有两个衡量指标:其一是精密度 π,指对任意 p 和 q,都有 $|C_p(t) - C_q(t)| \leqslant \pi$;其二是准确度 α,指对任意 p,都有 $|C_p(t) - t| \leqslant \alpha$。接下来将介绍物理时间的同步算法。

Cristian 算法

Cristian 算法解决了一个大问题,一个小问题。大问题指的是如何在不能让时间倒退的前提下修改时间,以及如何平滑的修改时间。需要倒退时,就让每一次时钟中断增加的时间短一点,或者说让时钟走慢一点;相应地,需要让时间加快时,就让每一次时钟中断增加的时间长一点。小问题指的是传输延迟,一次时间查询的过程如图 **??** 所示,其中 T_1 是发送查询的时间,T_2 是接收到查询的时间,T_3 是发送响应的时间,T_4 是接收到响应的时间。

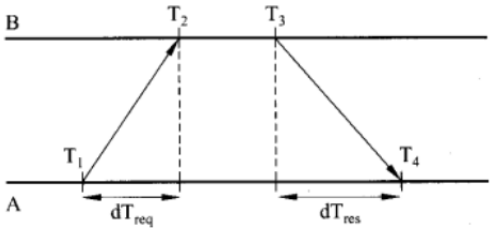


图 6.1: 时间查询的过程

Cristian 算法只返回一个时间 T,这个 T 必然介于 T_2 和 T_3 之间。令 $RTT = T_4 - T_1$,Cristian 算法令假定 T 刚好在 T_1 和 T_4 的中间,则应将时间设为 $T + \frac{1}{2}RTT$,误差为 $\pm\frac{1}{2}RTT$。

更进一步,若能估计发送时延 $\mathrm{d}T_{\mathrm{req}} = T_2 - T_1$ 和接收时延 $\mathrm{d}T_{\mathrm{res}} = T_4 - T_3$ 的下界 $\min_1, \min_2$,则应将时间设为 $T + \frac{1}{2}(RTT + \min_2 - \min_1)$,误差为 $\pm\frac{1}{2}(RTT - \min_1 - \min_2)$。

NTP 协议

网络时间协议(Network Time Protocol, NTP)是一种用于时间同步的协议,它通过在计算机网络中传输时间信息,使得连接在网络中的计算机能够获得准确的时间。NTP 有四种工作模式:**客户/服务器模式**下客户端向服务器发送时钟同步消息,服务器自动以服务器模式操作并发送回复;客户可以同步到服务器,但服务器不能同步到客户;若客户端可以同步到多个时间服务器,则选择最好的一个。**对称模式**下主动方定期向被动方发送时钟同步消息,被动方自动以对称被动模式操作并发送回复;对称模式可以同步主动方,也可以同步被动方,如果两边都同步,那么层级较高的同步到层级较低的;对称模式常用于相同层次服务器之间的同步,是所有模式中最精确的。**广播模式**下服务器向广播地址发送时钟同步消息,客户端收到第一条消息后开始与服务器交换消息以计算网络延迟,然后只有服务器发送时钟同步消息;广播模式可以同步客户端,但服务器不能同步客户端;广播模式用于一个或几个服务器和大量客户端的配置,时间精度低于客户/服务器和对称模式。**多播模式**是广播模式的多播版。

NTP 采用分层体系结构,其中存在若干层级的时间服务器,传输时使用时间戳和基于 UDP 的数据包。其架构如图 ?? 所示。NTP 使用类似 Cristian 算法但更复杂的算法来计算时间:计算相对偏差 $\theta = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]$,单向延迟 $\delta = \frac{1}{2}[(T_4 - T_1) - (T_3 - T_2)]$,则真实时间**取多次测量中 δ 最小的那个**。同步时,层数越高、同步离散程度越低的服务器优先级越高。

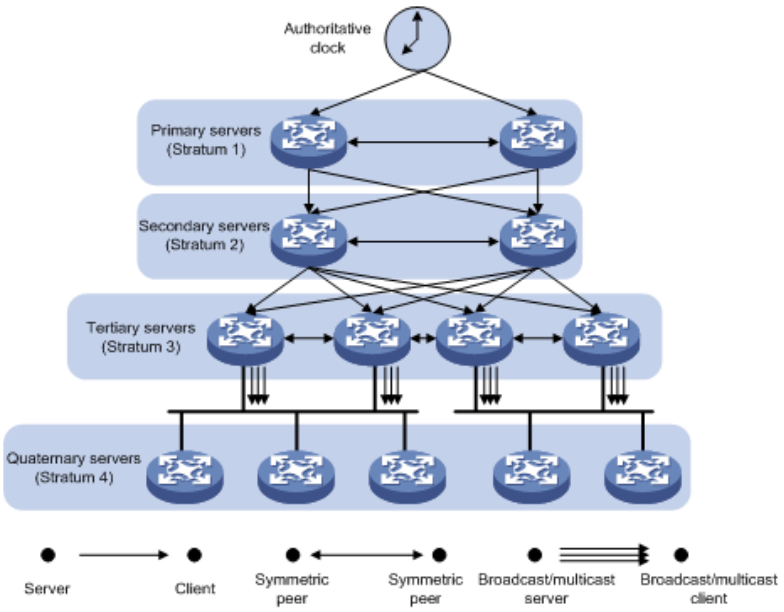


图 6.2: NTP 的架构

Berkley 算法

Berkley 算法与 Cristian 算法相反:Cristian 算法是客户端向服务器查询时间,Berkley 算法是服务器向客户端发送时间。这一算法中,服务器轮询客户端的时间,然后计算出平均值,再通知各客户端如何调整自己的时钟。

例题集 6.1: 物理时钟

问题 1 Name and briefly describe two differences between the time synchronization algorithm used in NTP and Christian's algorithm.

答: 1) NTP incorporates the server processing time. Actually a different equation. 2) NTP sends multiple

measurements to several time master servers in parallel. 3) NTP is hierarchically organized. Christian's algorithm considers only a single time master server.

问题 2 Impressed by how time synchronization works in distributed systems, John decides to rent a time server with an accurate GPS receiver installed. He plans to connect to the server with his personal computer, and there's one router between his PC and the server. Please answer the following questions.

(1) First, John wants to calculate the estimated minimum one way delay of sending a time sync request to the server. Assuming store-and-forward routing as well as the following scenarios listed below, please help John with the calculation (consider only one request message for one way):

- The link from John's PC to the router has a bandwidth of 5 Mbps, and a latency of 10 ms.
- The link from the router to the server has a bandwidth of 2 Mbps, and a latency of 100 ms.
- The time sync request has a size of 20 kb.

Note: Show your steps for full credit.

答:20 kb/5 Mbps + 10 ms + 20 kb/2 Mbps + 100 ms = 4 ms + 10 ms + 10 ms + 100 ms = 124 ms

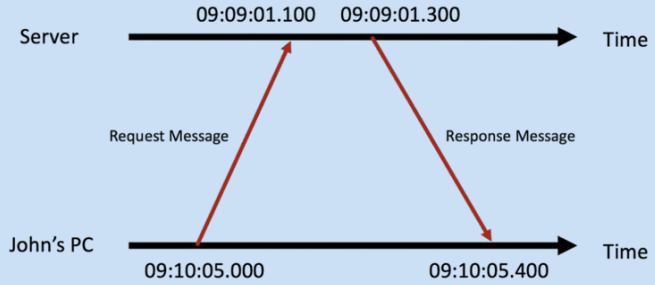
(2) John now decides to use Cristian's Time Sync algorithm to synchronize the clock on his PC with the server. He sends a sync request at 08:06:00.000. Then, at 08:06:00.400 his PC receives a response from the server, saying that the time is 08:08:10:000. What time should John set his PC to?

答:08:08:10.200

(3) Based on your answers to the above two sub-questions, what is the accuracy of the time estimate made in (2)? *Note: Show your steps for full credit.*

答:$\pm(200 - 124)$ms $= \pm 76$ ms

(4) However, after using Cristian's Time Sync algorithm several days, John realizes that this algorithm may not be very accurate if it the RTT is large. Therefore, John decides to use NTP instead. Consider the following diagram:



The four timestamps show when John sends the request, when the request is received at the server, when the server sends the response, and when John receives the response.

What is the RTT in this diagram (does NOT include server process time)? Calculate the estimated time offset of NTP.

答:200 ms. -64 s.

(5) Besides algorithms dealing with real-world time, John also wants to try logical clocks, which only care about relative order of different events. John thinks that regular Lamport clocks have some drawbacks, so he'd like to choose between total-order Lamport clocks and vector clocks. Help him by clarifying some details about each algorithm.

(5.a) How do total-order Lamport clocks achieve a "total-order"? Explain in 1-2 sentences.

答: Potential ties (possible due to concurrency) are broken using process ID, so that they become totally ordered.

(5.b) What shortcoming of Lamport logical clocks do vector clocks overcome?

答:If one Lamport timestamp is less than another, it did not necessarily happen before the other (it may be

concurrent). Vector clocks do not have this ambiguity.

问题 3 Global Position System (GPS) satellites use the Network Time Protocol (NTP) to keep their clocks synchronized. True or False?

答:False. GPS satellites have their own atomic clocks. NTP would not provide sufficient accuracy.

6.1.2 逻辑时钟

实际应用中,并不需要时钟与实际时间完全同步,只要所有的机器具有相同的时间就够了。更进一步,时钟前进的速率也不必于实际时间相同,只要机器之间保持一致即可。Lamport 指出,尽管时钟同步是可能的,但它不是绝对必要的。如果两个进程不进行交互,那么它们的时钟就无需同步;即使没有同步也觉察不出来、也不会产生问题。通常,重要的不是所有的进程在时间上完全一致,而是它们在事件的发生顺序上要达成一致。在 `make` 例子中,有用的信息是 `input.c` 的创建时间比 `input.o` 的创建时间早还是晚,而不是它们创建的绝对时间。因此,跟踪各自的事件顺序就可以了,这类算法一般将时钟称为**逻辑时钟**。

定义逻辑时钟之前,首先要定义**先发生**的关系。$e_i \to e_j$ 表示事件 e_i 先于事件 e_j 发生,这种先发生关系有两种情况:若 e_i 和 e_j 发生在同一进程中,且 e_i 发生在 e_j 之前,则 $e_i \to e_j$ 为真;若 e_i 是一个进程发送消息的事件,e_j 是另一个进程接收这一消息的事件,则 $e_i \to e_j$ 也为真。先发生是传递关系。

定义了先发生关系之后,就可以定义逻辑时钟了。**逻辑时钟**由一个时间域 T 和一个逻辑时钟 C 组成,C 将分布式系统中的事件 $e \in H$ 映射到 T 上,即 $C: H \to T$;而 T 对关系 $<$ 是偏序的。C 需满足**时钟一致性条件**:对任意事件 e_i 和 e_j,有 $e_i \to e_j \Longrightarrow C(e_i) < C(e_j)$。更进一步,若 $e_i \to e_j \iff C(e_i) < C(e_j)$,则称系统是**强一致**的。根据 T 的不同,逻辑时钟主要可以分为两类:第一类是 **Lamport 标量时钟**,时间用非负整数表示;第二类是**向量时钟**,时间用非负整数向量表示。逻辑时钟需要处理两类情况:第一类(**R1 规则**)是进程执行一个事件(发送、接受或者内部事件)时,它该如何更新本地时钟;第二类(**R2 规则**)是进程如何更新全局时钟,使得进程能够得知全局进展的信息。不同的逻辑时钟有不同的逻辑时间表示方式和不同的更新协议,但所有的逻辑时钟都实现了 R1 规则和 R2 规则,其结果就是保证了一致性条件。

Lamport 标量时钟

Lamport 是最早提出的逻辑时钟,它的时间域是非负整数,即 $T = \mathbb{N}$。Lamport 时钟是典型的中间件,实现于应用程序和网络通信之间。Lamport 标量时钟的更新规则如下:

(R1) 进程 P_i 执行一个事件(发送、接受或者内部事件)之前,它的本地时钟 C_i 加 d,$d > 0$。典型的 d 的值是 1,这足以把一个进程中每个事件的时间区分开来。

(R2) 消息发送方在发送消息时,附带它的本地时钟值。当进程 p_i 接收到带有时间戳 C_{msg} 的消息时,它将自己的本地时钟 C_i 更新为 $C_i = \max(C_i, C_{\text{msg}})$,然后执行 R1 规则,并把消息传递给上层应用程序。

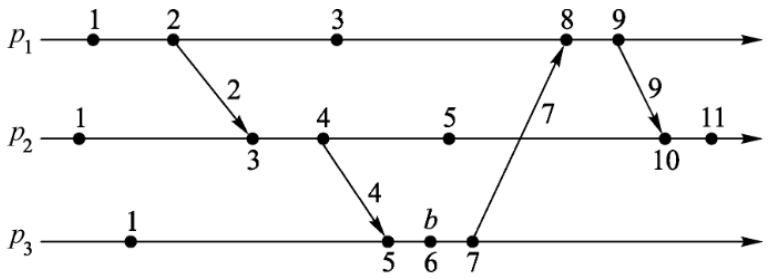


图 6.3: Lamport 标量时钟

三个进程更新标量时钟的例子如图 **??** 所示。Lamport 标量时钟的特性如下:

(1) **一致性**。如果 $e_i \to e_j$,则 $C(e_i) < C(e_j)$。

(2) **全序性**。只需略加修改(在时间的小数点后附加进程号),Lamport 标量时钟的时间域就是全序的,因此可以用来比较任意两个事件的先后顺序。

(3) **事件计数**。如果 d 的增加值总是 1,则标量时间有下面有趣的性质:如果事件 e 的时间戳为 h,那么 e 之前已经顺序产生了 $h-1$ 个事件(不管这些事件是哪些进程产生的)。以图 **??** 为例,b 的时间戳为 6,意味着在 b 之前已经顺序产生了 5 个事件。

(4) **非强一致性**。Lamport 标量时钟不是强一致的。以图 **??** 为例,事件 3.1 就晚于事件 3.2。之所以会有这种情况,是因为标量时钟是个标量,进程的时钟和本地全局时钟被压缩成一个,导致了不同进程事件之间因果依赖信息的缺失。

Lamport 标量时钟的一个应用是**全序多播**,即把消息传递给所有进程,且所有进程都按照相同的顺序接收消息。每个进程维护一个请求队列,队列按照请求的 Lamport 时间戳来排序。发送者将消息放入自己的请求队列中,然后向其他进程广播消息;接收者收到消息后,将其放入自己的请求队列中,并向发送者单播确认消息。当且仅当发送者自己的消息处于对首,且收到了其他所有进程的确认消息,发送者才可以将消息传递给应用程序。结束后,发送者从自己的请求队列中删除该消息,并向其他进程广播删除消息的请求。因为每个进程都有相同的请求队列,所以在任何地方,所有消息都以同样的顺序交付。全序多播可以用于解决多个进程对临界区的访问。

向量时钟

向量时钟中,时间由 n 维非负整数向量表示。每个进程 p_i 维护一个向量 $VC_i = (VC_i[1], VC_i[2], \ldots, VC_i[n])$,其中 $VC_i[i]$ 是 p_i 的本地逻辑时钟;$VC_i[j]$ 表示 p_i 关于 p_j 本地时间的最近信息:若 $VC_i[j] = x$,则 p_i 知道 p_j 的逻辑时间已进展到 x(p_j 已经发生了 x 个事件)。向量时钟比较的规则如下:

$$
\begin{aligned}
VC_1 = VC_2 &\iff \forall k, VC_1[k] = VC_2[k] \\
VC_1 \leqslant VC_2 &\iff \forall k, VC_1[k] \leqslant VC_2[k] \\
VC_1 < VC_2 &\iff (\forall k, VC_1[k] \leqslant VC_2[k]) \wedge (\exists k, VC_1[k] < VC_2[k]) \\
VC_1 \parallel VC_2 &\iff \neg(VC_1 < VC_2) \wedge \neg(VC_2 < VC_1)
\end{aligned}
$$

向量时钟的更新规则如下:

(R1) 进程 p_i 执行一个事件(发送、接受或者内部事件)之前,它的本地时钟 $VC_i[i]$ 加 d,$d > 0$。

(R2) 消息发送方在发送消息时,附带它的本地时钟值。当进程 p_i 接收到带有时间戳 VC_{msg} 的消息时,它将自己的本地时钟的每一项都更新为 VC_{msg} 和 VC_i 中对应项的最大值($\forall k \in [1,n], VC_i[k] = \max(VC_i[k], VC_{\text{msg}}[k])$),然后执行 R1 规则,并把消息传递给上层应用程序。

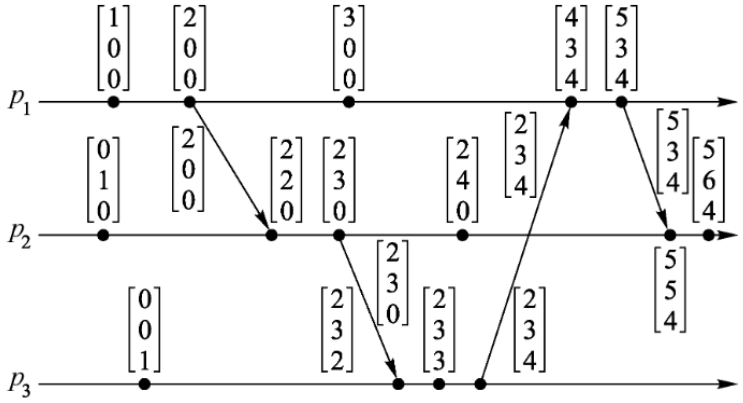


图 6.4: 向量时钟

向量时钟更新的例子如图 **??** 所示。向量时钟的特性如下:

(1) **同构**。如果两个事件 e_1 和 e_2 分别有时间戳 VC_1 和 VC_2,那么

$$
\begin{aligned}
e_1 \to e_2 &\iff VC_1 < VC_2 \\
e_1 \parallel e_2 &\iff VC_1 \parallel VC_2
\end{aligned}
$$

这样,由分布式计算产生的偏序事件集合与它们的向量时间戳之间是同构关系。

(2) **强一致性**。向量时钟系统具有强一致性。

(3) **事件计数**。如果在 R1 规则中 d 总是 1,则 $VC_i[i]$ 表示在 p_i 上知道的,该瞬间之前发生的事件数;若事件 e 有时间戳 VC,则 $VC[j]$ 表示进程 p_j 执行的,因果关系先于 e 的事件数,因果关系先于 e 的事件总数为 $\sum VC[j]-1$。

向量时钟的一个应用是强制**因果有序通信**,即保证先于某消息的所有消息接受后再发送该消息。这只需保证 p_j 接收 p_i 的消息 C_{msg} 时,$C_{\text{msg}}[i]=VC_j[i]+1$,且 $\forall k\neq i, C_{\text{msg}}[k]\leqslant VC_j[k]$。前者表示消息 msg 是进程 p_i 希望从进程 p_j 接收的下一条消息;后者表示当进程 p_i 发送消息 msg 时,进程 p_j 已经接收了所有进程 p_i 发送的消息。图 **??** 中展示了因果有序通信的一个例子。其中,$VC_0=(1,0,0)$ 时 p_0 发送 m 到其他两个进程;$VC_1=(1,1,0)$ 时 p_1 接收 m,然后发送 m*,m* 比 m 先到达 p_2;此时,m* 的传送被 p_2 延迟,直到 m 被接收且传送给了 p_2 的应用层。

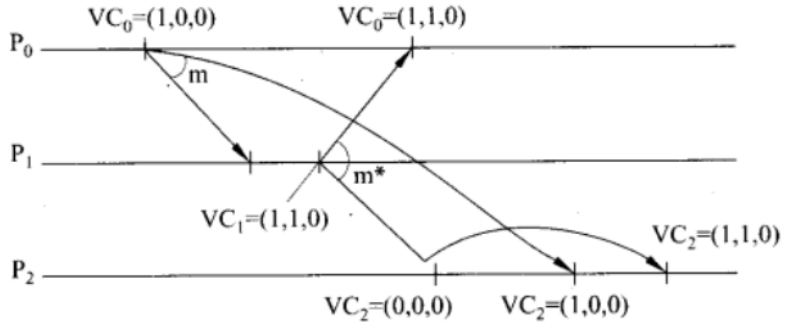


图 6.5: 因果有序通信

例题集 6.2: 逻辑时钟

问题 1 NoSQL systems like Amazon's Dynamo allows writes to continue to all machines during a network partition. Once reconnected, Dynamo nodes seek to reconcile inconsistent file versions. Which technique would you choose to detect and flag parallel events? Briefly explain why.

答:Vector clocks. Only algorithm that can properly preserve causality, so Dynamo would know that two events happened in parallel. Lamport clocks can't do this, as they create arbitrarily ordered timestamps, which might give two parallel events different timestamps. (2 points for Vector clocks and 2 points for correct reason. Note: Totally Ordered Lamport clocks is incorrect)

问题 2 True or False?

(1) Recall the global "happens-before" relationship ($\rightarrow$) that captured the logical notions of time using both the local happens-before relationship at each server and the fact that a message must be sent before it can be received. For a Lamport logical clock L, if $e_1 \rightarrow e_2$ then $L(e_1) < L(e_2)$.

答:True

(2) If the Lamport clock of event e_1 is less than the Lamport clock of event e_2, then there must be a chain of causal events by which e_1 precedes e_2.

答:False

问题 3 Consider a system with 4 machines that uses Lamport Timestamps for logical ordering. Let $L_i(m_j, \text{send})$ be the Lamport Logical Time for machine i sending message j. Let $L_i(m_j, \text{recv})$ be the Lamport Logical Time for machine i receiving message j. Each local clock is initialized to 0. Answer the questions below based on the following timestamps:

- $L_1(m_1, \text{send}) = 1$
- $L_1(m_2, \text{send}) = 2$
- $L_2(m_1, \text{recv}) = 2$
- $L_3(m_2, \text{recv}) = 3$
- $L_2(m_3, \text{send}) = 3$
- $L_3(m_4, \text{send}) = 4$
- $L_3(m_3, \text{recv}) = 5$
- $L_4(m_4, \text{recv}) = 5$
- $L_4(m_5, \text{send}) = 6$
- $L_1(m_5, \text{recv}) = x \longleftarrow x \textit{ is here}$
- $L_4(m_6, \text{send}) = 7$
- $L_1(m_7, \text{send}) = 8$
- $L_3(m_7, \text{recv}) = 9$
- $L_3(m_6, \text{recv}) = 10$

(1) What is the value of x? If it cannot be determined, write "unknown".

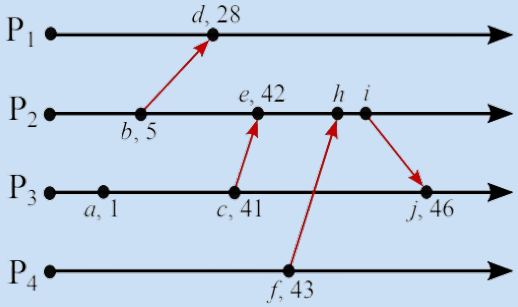
答:待补充

(2) For the following, circle which event happened first. If it cannot be determined, circle "unknown".

- $L_1(m_1, \text{send})$ $L_4(m_5, \text{send})$ unknown
- $L_2(m_3, \text{send})$ $L_3(m_4, \text{send})$ unknown
- $L_3(m_4, \text{send})$ $L_3(m_3, \text{recv})$ unknown
- $L_2(m_3, \text{send})$ $L_4(m_6, \text{send})$ unknown
- $L_4(m_6, \text{send})$ $L_1(m_7, \text{send})$ unknown

答:待补充

问题 4 The following figure depicts the timeline of multiple events happening across several different processes. The number at each event corresponds to its Lamport clock value.



(1) Based on their Lamport clock values, what is the relationship between events labelled b and c? Which event happened earlier?

答:No relationship, events b and c are parallel ($L(b) < L(c)$ does not imply that b precedes c).

(2) Ignore the Lamport clock values for this subpart. Han proposes a much simpler algorithm to determine event ordering: each time two processes communicate, the sender attaches a log of all events it has seen so far, along with their corresponding local timestamps. For example, when P_3 sends event c to P_2, it also sends event a's timestamp. When P_2 receives the message (event e), it can reconstruct a global view of event history across both P_2 and P_3. Based on these logs, can P_2 determine the ordering between events a and b? If so, what is the correct ordering from P_2's perspective up to event e? If not, why not?

答:No, because it is futile to compare timestamps across different processes. Since the machine clocks are not precisely synchronized, it is not meaningful to compare the resulting timestamps either. In this example, event a is recorded in terms of P_3's clock while b is recorded in terms of P_2's. Alternative explanations based on Lamport time will also be accepted.

(3) As you may have noticed, event h is missing a Lamport clock value. What are the possible values for the Lamport clock of event h? What are their corresponding total-ordered Lamport clock values? Assume that the maximum number of processes is 10, and that process P_2 has ID 2.

答:44 is the only possible Lamport clock value for event h. Therefore, its totally-ordered value is $M \times L_i(e) + i = 10 \times 44 + 2 = 442$.

问题 5 Suppose each computer in a distributed system keeps an approximate real-time clock R_i in addition to a Lamport-like clock L_i. A computer's real-time clock R_i is an always-increasing integer (i.e., for any two reads r_1, r_2 of R_i such that $r_1 \rightarrow_i r_2$, we have $r_2 > r_1$), but the real-time clocks at different computers may drift relative to each other (i.e., $R_j - R_i$ is non-constant for $j \neq i$). Consider the following modification, Shamport, to Lamport's partial-ordering algorithm:

(Rule 1) Before each event at computer i, set $L_i = \min(L_i + 1, R_i)$.

(Rule 2) When sending a message m, apply Rule 1 and include the time L_i as part of the message (i.e. send (m, L_i) instead of just m).

(Rule 3) When receiving a message (m, t) at computer j, set $L_j = \max(L_j, t)$ and then apply Rule 1 before timestamping the message-arrival event.

Let the Shamport global time of an event e at computer i be $S(e) = L_i(e)$.

(1) In the algorithm above, underline the difference between Shamport and Lamport's algorithm. (Underline as little as possible).

答: Shamport's algorithm uses $L_i = \min(L_i + 1, R_i)$ in Rule 1 instead of $L_i = L_i + 1$. Otherwise the two algorithms are the same.

(2) Let e, e_0 be two events at computer i such that $e \rightarrow_i e_0$. Prove that $S(e_0) > S(e)$.

答:证明题,希望别考,嘻嘻

问题 6 Three computers at CMU, A, B, and C communicate using a protocol that implements the idea of lamport clocks (they include their clock time stamp in messages). For reference, if you need a reminder, recall that the three rules of Lamport's algorithm are:

(Rule 1) At process i, increment L_i before each event.

(Rule 2) To send message m at process i, apply Rule 1 and then include the current local time in the message, i.e., send (m, L_i).

(Rule 3) To receive a message (m, t) at process j, set $L_j = \max(L_j, t)$ and then apply Rule 1 before time-stamping the receive event.

At the beginning of time, all three computers begin with their logical clock set to zero (0). Later, the following sequence of events occurs:

- A sends message M_1 to B: "hi".
- After receiving M_1, B sends message M_2 to C: "A told me hi".
- After receiving M_2, C sends message M_3 to A: "B is boring".

(1) Indicate the time included with the messages as they are sent at each step.

- Send $(M_1, ?)$
- Send $(M_2, ?)$
- Send $(M_3, ?)$

答:

- Send $(M_1, 1)$
- Send $(M_2, 3)$
- Send $(M_3, 5)$

(2) Maintaining all clock states from the previous question, three ADDITIONAL messages are sent:

- After receiving M_3, A sends message M_4 to B: "C is kind of random!".
- After receiving M_4, B sends message M_5 to A: "C is boring".
- A receives message M_5.

After all of these messages have been sent and received, what time does each computer think it is?

答:$A = 10$, $B = 9$, $C = 5$. (Remember, receiving and sending are different events!)

(3) Is this a relatively or totally ordered system?

答:This is a relatively ordered system.

Reset all clocks to zero. This time:

- A sends message M_1 to B.
- The user of machine A is talking on the phone with the user of machine B. She tells him "I just sent you a message online. Please send a message to C right now."
- B sends message M_2 to C.

(4) Once all of the messages have been sent and received, what times might C's clock be set to under which circumstance?

答:If B received the message before sending the message to C, then C's clock would read 4. Otherwise, C's clock would read 2.

(5) Explain briefly why the scenarios above could happen even though the events must have happened sequentially in real-time.

答:Because the communication over the cell phones was not timestamped using the lamport clock protocol.

Therefore, the protocol did not know that the message from A to B had to have happened before the message from B to C.

(6) Which of your two answers for C's clock is more likely to arise? Briefly (1 sentence) justify why.

答: It's more likely to be 5. Most of the time, the communication latency between two computers at CMU is likely to be in the few milliseconds range, which is much faster than the two humans can communicate commands in words. Therefore, message M_1 is likely to reach machine B well before the user instructs it to send a message to C.

6.2 互斥

实现互斥的算法主要有两类:**基于许可**的算法和**基于令牌**的算法。基于许可的算法中,进程必须先获得其他进程的许可,才能进入临界区。基于令牌的算法中,进程之间传递一个特殊的消息,称为**令牌**:只有持有令牌的进程才能进入临界区,完成后将令牌传递给其他进程;如果进程持有令牌但不需要进入临界区,就将令牌传递给其他进程。令牌方法的优点是简单,可以避免饥饿和死锁;缺点是需要处理令牌丢失的问题。

6.2.1 基于许可:集中式算法

很显然,最简单的算法就是**维护一个协调者进程**,所有进程都向它发送请求,然后由它来决定谁可以访问资源。如果当前没有其他进程访问资源,协作者就发送信息准许请求进程访问资源;收到准许信息的进程就可以访问资源。如果当前有其他进程访问资源,协作者就将请求放入队列中,不应答或者发送拒绝信息,请求进程就会阻塞。当进程释放资源后,它向协作者发送一个消息,协作者从队列中取出第一个进程,并向它发送准许信息。

集中式算法确实保证了互斥的实现,且易于实现:每个共享资源只需要 3 个消息(请求、准许、释放)。但协作者引入了单点故障和性能瓶颈问题。两相比较,其简单性带来的好处大于潜在的缺点。而且,正如后面示例所示的那样,分布式解决方法的表现还不如这个。

6.2.2 基于许可:非集中式算法

很多情况下,单点故障是不能接受的。有一种基于 DHT 的投票算法可以解决这个问题。假设临界资源一共有 N 个,则设置 N 个协调者,每个协调者负责一个资源。协调者编号为 `resource_name-i`,这一编号就作为 DHT 的键。某个进程想要访问资源时,需要获得超过半数($m > \frac{1}{2}N$)协调者的投票;若请求失败,就等待随机时间后重试。这一算法的缺点在于,如果很多进程要访问同一个资源,将导致性能大大降低:如果太多的进程去竞争获得许可,最终使得谁也得不到足够的投票,从而使得资源谁也用不了。(另一个我没有搞懂的点是,如果需要超过半数许可才能访问资源,那么岂不是有一小半的资源永远都用不上……)

6.2.3 基于许可:分布式算法

这一算法由 Lamport 提出,我们讨论的是 Ricart & Agrawala 的改进版本。这一算法使用全序 Lamport 标量时钟,使用 REQUEST 和 REPLY 两种消息:REQUEST 信息中包含资源名、进程号和时间戳。每个进程维护一个请求队列,想要进入临界区的进程发送 REQUEST 消息给所有进程(包括自己)。

收到 REQUEST 消息的进程根据自己的状态决定如何应答:

(1) 若进程不需要访问资源,就回复 REPLY 消息。

(2) 若进程已经获得访问资源的许可,就将请求放入队列中。

(3) 若进程想要访问资源但还没有访问,就比较请求自己请求的时间戳和收到请求的时间戳:若收到请求中的时间戳较早,就回复 REPLY 消息;若自己请求的时间戳较早,就将请求放入队列中,不发送任何消息。

当进程收到所有其他进程的 REPLY 消息后,就可以进入临界区了。进程退出临界区后,向队列中的所有进程发送 REPLY 消息,并从队列中删除这些请求。这种算法和集中式算法相比一无是处,更慢、更复杂、开销更高、更不健壮,唯一的研究价值就是它是分布式的。研究这种算法就像吃菠菜或者在高中学习拉丁语一样,其好处还有待挖掘。

6.2.4 基于令牌:令牌环算法

这种算法首先用软件方法将所有参与的进程排成环形。环初始化时,进程 0 得到一个令牌。令牌绕着环运行,用点对点发送消息的方式将令牌从进程 k 传到 $k+1$(以环大小为模)。进程从它邻近的进程得到令牌后,检查自己是否需要访问资源。如果要则继续完成其工作,然后释放资源;否则继续传递。在该进程完成后,沿环继续传递令牌。

该算法的正确性是显然的,且不会发生饥饿或死锁。但是,如果令牌丢失了,则需要重新生成令牌;但实际上很难确定令牌是否丢失。此外,如果有进程崩溃,令牌环就会被破坏。

6.2.5 四种算法的比较

表 ?? 比较了四种互斥算法的特点。

表 6.1: 互斥算法的比较

	每次进/出需要的消息数	进入前的延迟(以消息数计)	存在问题
集中式	3	2	协作者崩溃
非集中式	$3mk, k = 1, 2, ...$	$2m$	饥饿,效率低
分布式	$2(n-1)$	$2(n-1)$	任意进程崩溃
令牌环	0 to ∞	0 to $(n-1)$	令牌丢失,进程崩溃

6.2.6 如何选举协调者

比较四种算法后,发现集中式算法是反而是最好的。在集中式算法中,如何选举协调者是一个问题。

欺凌算法

欺凌算法是一种简单的选举算法,当任何进程发现协作者不再响应请求时,就发起一次选举。进程 P 按如下过程主持一次选举:

(1) P 向所有编号比它大的进程发送一个 ELECTION 消息;

(2) 如果无人响应,P 获胜成为协作者;

(3) 若有编号比它大的进程响应,则响应者接管选举工作。P 的工作完成。

获胜者向所有进程发送 COORDINATOR 消息,成为新的协作者。这样,总能保证进程号最大的进程成为协作者。欺凌算法的一次选举过程如图 ?? 所示。

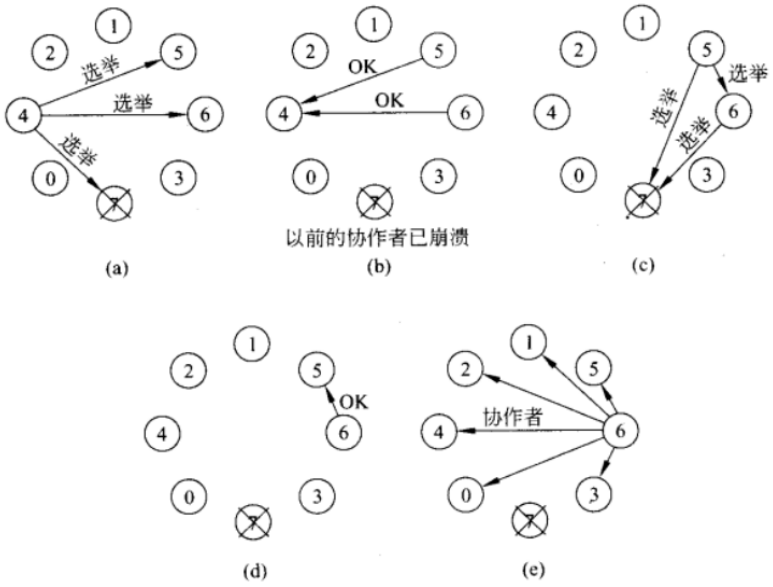


图 6.6: 欺凌算法

基于环的算法

基于环的算法是另一种选举算法，它的思想是将进程排成环形，然后按顺序选举。最初，每个进程都设置为非参与者，任何进程发现协作者不工作时都可以开始一次选举。它将自己设置为参与者，并将自己的进程号放入选举消息中，然后将消息顺时针发送给它的邻居。当进程收到选举消息时，它将消息中的进程号与自己的进程号进行比较：如果消息中的进程号较大，就将消息转发给它的邻居；如果消息中的进程号较小，且接收进程不是参与者，就将消息中的进程号替换为自己的，并转发消息；如果接收进程是参与者，就不转发消息。转发消息时，进程将自己设置为参与者。如果收到了自己的进程号，就说明自己的进程号最大，进程成为协作者；新协作者向所有进程通知谁是协作者以及新环中的成员都有谁。随后每个进程都恢复原来的工作。

例题集 6.3: 互斥算法

问题 1 True or False?

(1) Lamport's distributed mutual exclusion algorithm will fail if one of its participants fails.

答：True

(2) Bob has designed a device that allows a computer to obtain the "true time" with infinite accuracy. Ricart & Agrawala's distributed mutual exclusion algorithm still works even if we replace the logical Lamport clock timestamps with the true timestamps.

答：True. Yes, with true timestamps each component sees the same total order.

(3) Ring-based distributed mutual exclusion is a great solution for a latency-sensitive application running on 10,000 nodes.

答：False.

问题 2 How many messages are required for a node to gain access to the critical section (CS) in a system with n nodes that uses the Ricart & Agrawala algorithm for distributed mutual exclusion? Explain how you arrived at your answer.

答：$2(n-1)$ messages. The requesting node must send a request message to all other nodes and receive a grant message from all other nodes in order to enter the CS.

问题 3 List and describe the three characteristics that a solution to a synchronization (mutual exclusion) problem must have.

答：1) mutual exclusion — only one process in critical section at a time 2) progress — don't wait for an available resource 3) bounded waiting — can't wait forever

问题 4 Describe two disadvantages of a ring-based algorithm (one that passes a token around a ring) for implementing a distributed mutex?

答：There could be a very long delay for a large ring. Also quite unreliable because a process could die and it would take a very long time to generate another token. Not necessarily fair.

问题 5 For each of the following distributed mutual exclusion solutions, which properties does that solution have? Circle all that apply.

- Central mutex server: fair reliable long synchronization delay
- Ring-based algorithm: fair reliable long synchronization delay
- Shared priority queue: fair reliable long synchronization delay
- Majority rules: fair reliable long synchronization delay
- Maekawa voting: fair reliable long synchronization delay

答：待补充

问题 6 In 2027, Foodle is the world's leading Internet company with ten datacenters around the globe. This future version of the Internet has neither packet loss nor network partition nor sudden machine crashes, but the speed of light remains as a major constraint. A critical piece of Foodle's algorithms require mutual exclusion

across all servers in all data centers.

Foodle's design head, Hepp Dean, proposes a new hierarchical distributed mutex. Each datacenter has a unique coordinator, M. Across the ten datacenters, the ten coordinators uses Ricart & Agrawala's algorithm, which has the following thread-safe API: `BroadcastRequest()`, `GotGlobalLock()`, `BroadcastRelease()`.

Within each datacenter, the coordinator manages datacenter-local mutual exclusion among the local client machines. Below is Hepp's pseudo code for coordinators.

(1) Across different datacenters and in the absence of failures, explain briefly why Hepp's algorithm safely implements a mutex?

答:Ricart & Agrawala's algorithm enables only a single coordinator to hold the global lock. The coordinator gives a global lock to its clients within the datacenter only if it has a global lock. So, only one client in the datacenter gets a mutual exclusion and no two clients in different datacenter get the mutual exclusion.

(+2 for "Ricart & Agrawala's algorithm" (or "global lock", explanation of Ricart & Agrawala); +2 for only one "coordinator" (also can be "datacenter", "leader", "M") gets "global lock" (also can be "global mutual exclusion", "global mutex") or, gets into "critical section" etc.; +2 for Walk through the code to explain one of the points above. Can cite line numbers or function calls and explain what they do.)

(2) Within a single datacenter and in the absence of failures, explain in less than three sentences why Hepp's algorithm safely implements a mutex?

答:Within a datacenter, the coordinator performs centralized control over the lock among clients. `M.Lock` RPC returns (terminates) only if the coordinator has global mutual exclusion (line 26, following `M.GotGlobalLock` callback), this is the only client with `id` in mutex-protected queue (`M.Queue`, line 26). `M.Unlock` pops current client's `id` from the queue, wakes up other client's RPC using `M.Cvar`, and passes the local mutual exclusion to other client in the same datacenter.

(+2 for "centralized control" by "single coordinator"(also can be "single machine", "M", "single leader") within a datacenter. Or, can say one "queue" in datacenter but should say it's mutex, cvar protected queue — just saying queue does not count. +2 for "only one "client" (also can be "machine", "requestor", or similar, but not "M". "leader", "coordinator") can get "lock", "mutual exclusion", "mutex" or can get into "critical section". +2 for coordinator's mutex, cvar protects/gurantees mutual exclusion. +2 for Walk through the code to explain one of the points above. Can cite line numbers or function calls and explain what to do.)

(3) Assume that many client machines simultaneously seek to acquire the mutex. In what order will they acquire it? Describe the order for both cases below.

(3.a) All client machines in the same datacenter:

答:Within the same datacenter, each client will be given in First-come, First-served order of getting `M.Mutex`. Note that, there could be cases where client machine A calls `M.Lock` earlier than client machine B but client B's `M.Lock` gets `M.Mutex` first.

(+1 for any answers implying "First-in-first-out", "First-come-first-served". Also allowing: "requested order", "timestamp of calling `M.Lock`" that can implicitly mentions FCFS order. Do not have to catch the case noted in the solution (the order of getting `M.Mutex`).)

(3.b) Client machines in different datacenters:

答: Client machines in different datacenters get mutual exclusion in the order their coordinator gets global mutual exclusion by Ricart & Agrawala algorithm.

(+1 for any answers implying the order of Ricart & Agrawala's algotihm. such as "the order their coordinator gets lock", "TO Lamport timestamp order", etc. Do not allow just "timestamp order" because it is ambiguous whether it means "Lamport clock" or "physical clock".)

(4) State an advantage of Hepp's hierarchical distributed mutex, assuming that there are tens of thousands of

servers in each datacenter?

答:Efficiency. Comparing to non-hierarchical distributed mutex where all client machines across datacenters use one distributed mutual exclusion algorithm such as Ricart & Agrawala, Hepp's algorithm is more efficient. Hepp's algorithm reduces the number of requests that should be sent through relatively poor inter-datacenter links, keeping a significant number of messages within datacenters. Also, it reduces the total number of messages.

(+3 for any answers implying "efficiency", such as: "performance". Or, a descriptive answer that implies any part of the solution, such as "less number of message", "less inter-datacenter communication", etc.)

(5) State a disadvantage of Hepp's hierarchical distributed mutex, in the context of the mutex design goals discussed in class? How would you resolve it?

答:Unfairness. From line 37, the coordinator releases its global lock only if there is no more local client in the queue. So, if the local clients of a coordinator are too demanding to get the lock, one coordinator will hold the lock forever. Clients in other datacenters will starve. The way to solve this problem could be setting timeouts for the time a coordinator can hold global mutual exclusion or setting the number of times a coordinator can give local mutual exclusion, etc.

(+1.5 for any answers implying "unfairness". Any descriptive answer implying unfairness such as: "one datacenter can hold the lock forever". +1.5 for any reasonable resolution.)

问题 7 Consider a variation on the central coordinator algorithm for mutual exclusion. Instead of one coordinator, suppose we have two coordinators, such that the algorithm will continue to operate even if at most one coordinator fails. The coordinators are connected with a synchronous channel with maximum delay of D seconds. Communication between the coordinators and client processes is asynchronous. When a process needs to enter the critical section, it sends a request to both the coordinators. The process may enter the critical section when it receives an OK message from either of the two coordinators.

Help define the protocol the coordinators use to communicate between each other.

Name the coordinators P and S. P is the primary and S is the secondary. For naming purposes, assume they maintain a lock-owner variable to remember who has the lock, if held, which is nil otherwise.

Assume that P is required to send a heartbeat message to S every T seconds.

Ensure that your protocol works properly if the primary had granted a lock, and then died, that the secondary still knows that the lock is held and by whom.

Define your protocol by answering the following four questions. If any of your operations involve sending a message from P to S or S to P, please define in that answer what the receiver should do with the message they receive.

(1) When P receives a lock request from client C, what is the sequence of operations it performs?

答:

(1) P checks that the lock is available, otherwise it enqueues the request and returns (this line should be exectued atomically);

(2) P sends a note to S saying "lock granted to C";

(3) P sets lock-owner=C;

(4) P replies "lock-granted" C.

Note that it's important for P to notify S before replying to C, otherwise it might happen that P dies inbetween messaging C and messaging S. Then S could grant the lock to a different client, which would violate mutual exclusion.

(2) When S receives a heartbeat message from P, what does it do?

答:Record last_heartbeat=current_time().

(3) How does S conclude that P has died?

答:If it has not heard from S within $D+T$ seconds, it can conclude that P has died.

Note that because D is only an upper bound on the network delay, we cannot conclude anything after T seconds: for all we know, it could be that a packet took 0 seconds to go from P to S, and the next packet (T seconds later) took D seconds.

(4) When S receives a lock request from client C, what does it do?

答:If S believes the primary is alive, it puts the request in a queue of pending requests Q, and waits.

If S believes the primary has died, it puts the request at the end of its queue and handles requests in its queue.

Handling a request operates as normal: If the lock is already held it leaves it in the queue, stops working, and waits for a lock release. Otherwise, grant the lock to C (update the variable and send a message).

问题 8 The year is 1850. You and a group of colleagues are collaborating on a programming assignment for Charles Babbage's Analytical Engine, an early mechanical computer. However, you are all in different locations and can only communicate by postal mail. This makes just figuring out how to collaborate a challenge of its own! Assume that letters are never lost in the mail but may take a few days to arrive.

(1) You first try having a single paper copy of the assignment. Whenever someone has the single copy, they make any changes they want, then mail it to the next group member (in some fixed ordering of group members, with the last person then mailing it to the first and repeating the process).

(1.a) What distributed mutual exclusion algorithm does this correspond to?

答:Token ring. (The paper copy is the token.)

(1.b) Describe one disadvantage of this approach.

答:Possible answers: lost time if lots of people are not actually working during their turn; someone may forget to do it and then the token is lost forever.

(2) How would you instead implement centralized mutual exclusion in this setting?

答:Everyone mails requests to acquire the copy to a chosen coordinator, who mails it out to a winner of their choosing.

(3) Over time, the assignment becomes a thick manuscript that is expensive to mail. Your group decides that to save money, everyone will keep a replica (copy) of the current assignment and only mail descriptions of their changes. Given that everyone has their own copy that they can edit, is a mail-based distributed mutual exclusion algorithm still necessary? Why or why not?

答:Yes: they need to avoid making conflicting changes. If multiple people make changes simultaneously, the replicas may go out of sync and no longer have the same values, or they may violate some invariant.

6.3 分布式死锁检测

真不想看。

7 一致性和复制

分布式系统的一个重要问题就是数据的**复制**,复制的目的包括:使得数据与用户在地理上接近(从而减少延迟);即使系统的一部分出现故障,系统也能继续工作(从而提高可用性);扩展可以接受读请求的机器数量(从而提高读取吞吐量)。数据复制的主要难题是保持各个副本的**一致性**,亦即保证所有冲突操作(包括读写冲突和写写冲突)在所有副本上以相同的顺序执行。冲突处理的代价很高,因此我们需要降低一致性要求,避免全局同步复制。

7.1 一致性模型

应用程序能够容忍多弱的一致性?我们可以将分布式存储划分为若干**一致性单元**(Consistency Unit, conit),并将单元的不一致性定义为一个三维向量:(数值偏差, 新旧偏差, 顺序偏差)。这样来看,一致性就不是有或无的二元属性,而是一个连续的谱,这就是所谓的**连续的一致性**(Continuous Consistency)。划分一致性单元的关键在于找到一个合适的折中点:如果划分过粗(例如整个系统),那么一致性要求就过于严格,所有的系统都会被划分为不一致的;如果一致性单元过细,那么要管理的单元就过多。

连续的一致性说的很好听,但实际上没什么用。下面来看看真正的一致性模型。Jepson 的官网将各种一致性模型的强弱关系排列在图 ?? [1] 中,并根据容忍网络分区的能力分为三类:

- 粉色表示**不可用**级,无法容忍网络分区,即 CAP 中的 CP 类系统;
- 橙色表示**粘性可用**级,可以容忍网络分区,前提是每个客户端只与固定的服务器通信,不能切换到别的;
- 蓝色表示**完全可用**级,可以容忍网络分区。

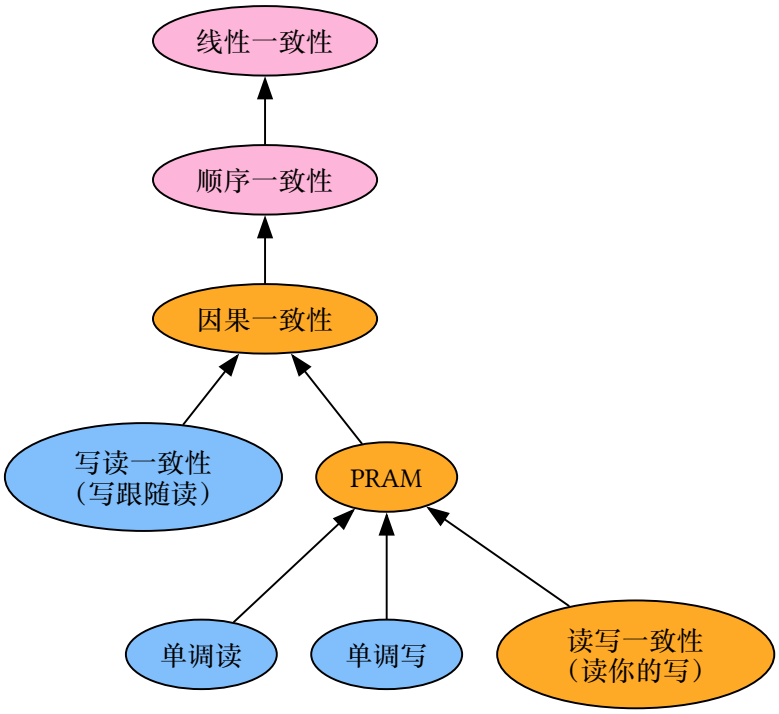
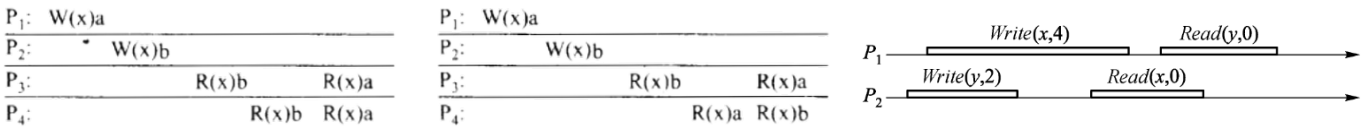


图 7.1: 一致性模型

我们用 write(x, a) 表示将 a 值写入变量 x,read(x, a) 表示读取变量 x 返回 a。最强的一致性是**严格可串行化**(在单变量场景下,就是**线性一致性**),这一模型存在一个隐式的全局时钟,在这一时钟下每一次读操作都能读到最近的写操作(有重叠则可以任意排序)。分布式环境中没有全局时钟,因此这种一致性很难实现。

Lamport 使用逻辑时钟代替全局时钟,得到的就是**顺序一致性**。顺序一致性要求:从全局视角来看,所有的操作都按照某个顺序执行,每一次读操作都能读到这个顺序中最近的写操作。这个全局顺序不一定是真实发生的顺序(由于网络延迟,不同进程的时间可能是跳跃行进的),但至少所有进程看到的顺序是一样的,且进程内部的操作顺序不变(是全局顺序的子序列)。

[1] 只截取了一部分

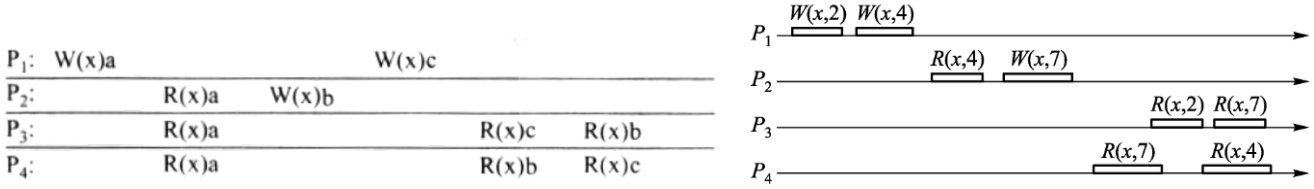


(a) 例 1 (b) 例 2

图 7.2: 顺序一致性的例子

下面举两个例子。图 ?? 左图中,尽管 write(x,a) 发生在 write(x,b) 之前,read(x,a) 却发生在 read(x,b) 之后,但这依然满足顺序一致性:存在 write(x,b) → read(x,b) → write(x,a) → read(x,a) 的全局顺序。相应的,图 ?? 右图就不满足顺序一致性。图 ?? 是另一个不满足顺序一致性的例子:read$(x,0)$ 必然发生在 write$(x,4)$ 之前(因为读到了更早的值),write$(y,2)$ 必然发生在 read$(x,0)$ 之前(因为在同一个进程中),read$(y,0)$ 必然发生在 write$(x,4)$ 之后(因为在同一个进程中),因此,read$(y,0)$ 必然发生在 write$(y,2)$ 之后,这与读到的值矛盾。

对于分布式系统而言,顺序一致性还是太强了,以至于无法容忍网络分区。更宽松的一致性是**因果一致性**。顺序一致性中,所有的操作都是按序执行的;因果一致性将这一限制放宽到所有具有**因果关系**的操作都是按序执行的。因果关系是以下两规则的传递闭包:同一进程的操作具有因果关系;如果一个读操作读到了一个写操作的结果,那么这两个操作具有因果关系。比较常见的一种因果关系就是某一计算进程读取 a 的值,再将计算结果写入 b,例如 write$(x,\text{Lunch?}) \Rightarrow$ read$(x,\text{Lunch?}) \Rightarrow$ write$(y,\text{Yes.})$。因果关系保证了任意进程不会先看到答案,再看到问题;微信朋友圈就有这种需求。



(a) 例 1 (b) 例 2

图 7.3: 因果一致性的例子

图 ?? 给出了一个满足因果一致性, 不满足顺序一致性的例子: 存在的因果链是 write$(x,a) \Rightarrow$ read$(x,a) \Rightarrow$ write(x,b) 和 write$(x,a) \Rightarrow$ write(x,c),因此必然有 read$(x,a) \Rightarrow$ read(x,b) 和 read$(x,a) \Rightarrow$ read(x,c),但 read(x,b) 和 read(x,c) 的顺序是不确定的。图 ?? 则给出了一个不满足因果一致性的例子: 存在的因果链是 write$(x,2) \Rightarrow$ write$(x,4) \Rightarrow$ write$(x,7)$,那么任何读到了 $x=4$ 的操作就都读不到 $x=2$,读到 $x=7$ 就读不到 $x=4$。

前面所属的一致性模型都是**以数据为中心的一致性**,会以副本角度考虑多个客户端的状态;另一类一致性是**以客户端为中心的一致性**,主要聚焦单个客户端观察到的状态,包括:

- **单调读一致性模型**:如果客户端读到了变量 $x=a$,则之后的读操作都会读到 $x=a$ 或更新的值,即保证了客户端不会读到更旧的值。例如,刷新邮件列表只会在现有邮件的基础上刷新,不会出现新邮件消失的情况。
- **单调写一致性模型**:同一个客户端的写操作在所有副本上都以相同的顺序执行,即保证了客户端的写操作是串行的。例如,客户端先执行写操作 write$(x,0)$ 再执行写操作 write$(x,1)$,另一个客户端不停地读 x 的值,那么会读到 x 的值先为 0 再为 1,不会先读到 1 再读到 0。
- **读写一致性模型**(读你的写):如果客户端执行了写操作 write(x,a),那么同一客户端再读,读到的就是 a 或更新的值(其他客户端可以延迟一会)。例如,更新头像后确保自己立刻看到新头像,好友可以延迟一会。
- **写读一致性模型**(写跟随读):如果某一客户端先执行了读操作 read(x,a),再执行写操作 write(x,b),那么这一写操作一定晚于这一读操作,也晚于之前的 write(x,a)。例如,看到答案的前提是看到问题。这种一致性看起来很像因果一致性,实际上也很像因果一致性,只不过以单个客户端为视角,因此又叫**会话因果一致性**(Session Causality)。

读写一致性、写读一致性、单调读、单调写都能算作**最终一致性**。最终一致性指的是,如果很长一段时间内没有新的写操作,那么所有的副本会逐渐成为一致的,最终所有的读操作都会返回相同的值。这是一种很弱的一致性保证,前述的所有一致性模型都比最终一致性强。除了这些较为通用的一致性模型,还有一些基于同步指令的一致性模型,例如基于锁的**入口一致性**。

例题集 7.1: 一致性

问题 1 True or false?

(1) In the context of data stores, having eventual consistency does not imply that the data store is also sequentially consistent.

答:True. Eventually consistency is the most relaxed form of consistency, while sequential consistency is much stronger, requiring the same sequential order on all nodes.

(2) Causal consistency is a weaker notion of consistency than sequential consistency.

答:True. Sequential consistency has stricter ordering requirements (all nodes must agree on ordering of events).

7.2 复制管理

7.2.1 副本服务器的放置

真有人研究这东西啊 `Bad \emo{grinning-face-with-sweat}`

7.2.2 内容复制和放置

从逻辑上,副本分为**永久副本**、**服务器启动的副本**和**客户启动的副本**。复制的顺序是永久副本 → 服务器启动的副本 → 客户启动的副本 → 客户。

永久副本有两种:其一是将原始数据存储于处于同一数据中心的多台服务器中(分布式不一定要地理上分布很广,集群也是分布式的),做一个比较集中的负载均衡;其二是静态配置的**镜像**,地理上散布于互联网的各处。永久副本一般起到备份的作用。**服务器启动的副本**是应对压力动态、自动化创建的:如果某个地区的用户突然增多,就会自动在该地区创建(只读的)副本。这件事主要是 CDN 在做。**客户启动的副本**其实就是客户端缓存。

内容是如何复制的呢?第一个设计决策是传播什么信息。第一种可能是传播**通知**,即通知这部分数据是无效的,不需要再复制了。在读写比很低、压倒性的写入密集型应用中,无效的中间状态很多,这种方法很有效。缓存系统也需要传递失效信息触发缓存更新,当然这种方法必然要和其他方法结合使用。第二种可能是**传播数据**本身,传播数据、传播更新还是传播日志都算。第三种可能是**主动复制**,即将更新操作发送给副本服务器。例如,将所有副本抽象为状态机,然后传播状态转移序列。实际上,有一种复制方案就是在加入限制后传播 SQL 语句。

第二个设计决策是谁发起复制。第一种可能是**推式复制**,即服务器主动分发数据,一般用于永久副本和服务器启动的副本。这种方式对源站并不透明,只对用户一侧单向透明,大型网站会考虑使用。例如,双十一之前,淘宝就在你的手机中预载了广告,这就是一种推式复制。第二种可能是**拉式复制**,即客户请求 + 无数据/数据过期时才触发复制,一般用于高速缓存。这种方式可以做到完全的双向透明,是小型站点的首选。二者的比较如表 ?? 所示。

表 7.1: 复制方法的比较

	推式复制	拉式复制
服务器需存储的状态	副本列表	无
传递的消息	更新(以及以后可能获取的更新)	轮询和更新
客户响应时间	立即(或获取更新的时间)	获取更新的时间

第三个设计决策是**单播**和**多播**,拉式一般配合单播,推式一般配合多播。

7.3 一致性在副本中的实现

根据副本可以接受操作的不同,分布式存储主要有三种架构:单主复制、多主复制和无主复制。

单主复制是最常见的架构,其中一个副本被指定为**主副本**,其他副本被指定为**从副本**。所有的写操作都发送到主副本,主副本在本地执行写操作的同时,也会将数据变更发送给所有的从副本;读操作则可以发送到任意副本。单主复制的一个重要细节是复制是阻塞的还是非阻塞的,就是主副本需不需要等待从副本的确认信息。阻塞式复制的优点是能够保

证所有副本都与主副本保持同步,且从崩溃中恢复的能力更强,但任何一个从副本宕机都会导致整个系统的故障,性能和可用性都很差;非阻塞式复制的一致性稍差,且主库发生无法恢复的故障时会丢失数据,但其实已经广泛使用了。还有一种“半阻塞式”的配置:在主副本和一个从副本上同步,其他从副本异步,这样至少能保证有两个副本的信息是最新的。

单主复制的一种变体是**本地写复制**,这种方法也分主副本和从副本,只不过主副本会频繁切换。客户端可以在任意副本上执行读写操作;当执行写操作时,当前的主副本就切换到执行写操作的副本上。(这 b 书不区分复制和分片 Bad \emo{grinning-face-with-sweat})这种方法与非阻塞式复制相绑定,能在本地执行多个连续的写操作,还能用于离线操作。另一种变体是主副本可以临时允许从副本进行写操作。

多主复制和**无主复制**都能在任意副本上执行读写操作。二者的区别在于前者需要严格将写入转发给其他副本,后者则较为宽松。**多主复制**需要解决冲突,让所有主副本收敛到一致的状态。Lamport 全序多播是一个办法,但是可用性太差了。另一种办法是避免冲突:先将所有操作转发到一个定序器,再由定序器转发给所有副本;这本质上还是单主复制。**无主复制**则完全抛弃了让副本达成一致的要求,可以容忍不一致的状态。这种方案中,每次读写都要发送给所有副本;有足够数量的副本确认写入/返回结果就算成功,剩下的副本就无所谓了;所有的数据都添加时间戳,读取的结果是所有副本中最新的结果。假定有 N 个副本,每个写入至少有 w 个节点确认才能认为是成功的,每次读取至少有 r 个节点返回结果才能认为是成功的,则 r 和 w 需满足两个条件:$r+w>N$ 且 $w>N/2$。一种常见的设置是设置奇数个 N,然后令 $r=w=\frac{1}{2}(N+1)$;读密集型应用可以设置 $w=N, r=1$,优点是读取速度更快,缺点是单个节点故障会导致写入失败。无主复制还可以通过读修复和反熵过程修复少数派副本的数据。

最后提一下高速缓存。要达到缓存和源站的一致,可以配置**主动失效**(源站通知缓存失效)或者**被动失效**(TTL 到期);主动失效又可以选择只发送失效通知还是发送新数据。缓存也分为**只读缓存**和**读写缓存**,读写缓存又分为**直写式**(写入缓存后立即写入源站)和**写回式**(写入缓存后过段时间再写入源站)。

8 容错

构建分布式系统绝非易事;分布式系统中不存在理想化的模型,只有冰冷混乱的现实。相比于集中式系统,分布式系统存在**全局时钟问题**(不存在全局时钟)、**可变延迟问题**和**部分失效问题**。集中式系统要么正常工作,要么彻底出错,但分布式系统中经常会出现**部分失效**:系统的某些部分可能会以某种不可预知的方式出错,而其他部分则正常工作。再加上网络的可变延迟,你甚至连有没有出错都无法确定。因此,在设计之初就应将出错的可能性纳入考量。

8.1 容错性

容错性包含以下四种需求:

- **可靠性**(Reliability):系统可以在一个相对较长的时间内持续工作而不被中断。可以用**平均失效时间 MTTF**、**平均修复时间 MTTR** 和**平均无故障时间 MTBF**(MTBF = MTTF + MTTR)来衡量。
- **可用性**(Availability):系统在任何给定的时刻都能及时地工作。$A =$ xfracmathfam5

MTTF-NoValue-MTBF□□□□□$Safety$□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- • **可维护性**(Maintainability):系统发生故障后恢复的难易程度。

可用性和可靠性的区别在与计算的是出错的总时长还是连续不出错的时长。如果一个系统每小时崩溃 1 毫秒,那么它的可用性超过 99.9999%,但它的可靠性很低;如果一个系统每年停机两周,那么它的可用性只有 96%,但它的可靠性很高。

当然,要想准确的评估系统的可用性和可靠性,我们必须准确的定义何为"出错"。关于这点,分布式系统中有三个相似的名词:**失败**指系统不能兑现它的承诺,即不能为用户提供服务;**错误**是系统状态的一部分,可能导致失败;**故障**指导致错误的原因,分为暂时故障、间歇故障和持久故障。故障根据严重程度的分类如表 ?? 所示。

表 8.1: 故障的类别

故障类型	说明
崩溃性故障	服务器停机,但是在停机之前工作正常
遗漏性故障	服务器不能响应到来的请求
接收故障	服务器不能接收到来的消息
发送故障	服务器不能发送消息
定时故障	服务器的响应在指定的时间间隔之外
响应故障	服务器的响应不正确
值故障	响应的值错误
状态转换故障	服务器偏离了正确的控制流
随意性故障	服务器可能在随意的时间产生随意的响应

处理故障的方式包括预测故障、防止故障、除去故障和容忍故障,最重要的是**容忍故障**(就是**容错**),即在故障发生时仍能提供服务。容错主要通过**冗余**来实现,冗余又分为**信息冗余**(添加额外信息)、**时间冗余**(重复执行)和**物理冗余**(加装额外硬件或运行额外进程)。

8.2 分布式共识

分布式系统中,进程间达成一致是很多应用程序的基本需求。进程常常需要相互协调,相互交换信息,彼此协商,最终在进行下一步行动前形成共识。**共识问题**的形式化表述如下:假设一个分布式系统有 N 个进程,每个进程都有一个初始值,进程之间相互可以通信共识问题的目的是找到一种算法,使得即使有 f 个进程出错,其他进程之间仍能够在有限步骤内协商出一个最终决定值(所有正确的进程都会认同某一个值,且认同的是一个相同的值);若所有正确进程的初始值相同,那么这个最终决定值必须等于这个初始值。共识问题、**拜占庭协定问题**(只有一个源进程有初始值,源进程出错则可以协商为任意一个值)和**交互一致性问题**(每个进程一个值,协商出一个数组)是等价的,任意一个的解都能成为其他两个的解。共识问题在各种设定下的结果列表如表 ?? 所示。

表 8.2: 共识问题的结果概览

故障模式	同步系统	异步系统
无故障	可以形成共识	可以形成共识
崩溃故障	可以形成共识 $f < N$ $\Omega(f+1)$ 轮循环	不可能形成共识
拜占庭故障	可以形成共识 $f \leqslant \lfloor \frac{N-1}{3} \rfloor$ $\Omega(f+1)$ 轮循环	不可能形成共识

其中,**同步系统**假设网络延迟,进程暂停时间和时钟误差都是有上限的:所有的消息都在有限时间内到达,每个进程的时钟漂移有一个已知的上界,每个进程的每一步都在一个已知的时间范围内完成。**异步系统**则不做任何限制。共识问题的一个变体是**带有签名的拜占庭将军问题**,此时只需 $f \leqslant n-2$,在 $\Omega(f+1)$ 轮循环后即可达成共识。

FLP 定理表明,异步系统中只要有节点存在故障就不可能形成共识。当然,异步系统是一个限制性很强的模型;如果允许算法使用超时来识别可疑的崩溃节点(即使这种怀疑有时是错误的),或者允许使用随机算法,就能够在异步系统中形成共识。现实中的分布式系统通常是可以形成共识的。

前一章的单主复制很像是共识问题,目的也是使所有副本存储的值一致,为什么我们不担心单主复制中达不到共识呢?答案是单主复制有人工干预,不需要保证有限步骤:如果主节点发生了问题,那就人工选一个新的主节点。

下一节将讲述一些达成分布式共识的实用算法。

8.3 分布式提交和 Paxos

分布式提交有点像单主复制,只不过加入了一些额外的步骤,以保证提交的原子性——亦即所有节点要么提交成功,要么提交失败,没有部分成功部分失败的说法(也就是共识)。显而易见,由于网络延迟和节点故障,直接将提交命令广播给所有节点,收到提交命令的节点就提交的一阶段提交是无法保证原子性的,可能会导致不一致的情况。

8.3.1 两阶段提交 2PC

第一个解决方案是**两阶段提交**(Two-Phase Commit,2PC)。2PC 由一个协调者和多个参与者组成,应用准备提交时,分以下两个阶段进行:

(阶段 1) 这一阶段称为**准备阶段**或者**投票阶段**,协调者询问事务的所有参与者是否准备好提交。如果参与者准备好了就回复 Prepared,否则回复 Non-Prepared。这里的准备并不是指“热身完毕”的意思,而是指已经完成数据处理,数据已经写入存储器,距离真正提交只剩一个确定的意思(不过还能回滚)。

(阶段 2) 这一阶段称为**提交阶段**或者**执行阶段**,协调者根据参与者的回复情况决定是提交还是回滚。如果所有参与者都回复 Prepared,则协调者在本地执行提交操作,然后向所有参与者发送 Commit 指令,所有参与者立即执行提交操作;否则,任意一个参与者回复了 Non-Prepared 或超时未回复,协调者向所有参与者发送 Abort 指令,协调者和所有参与者都立即执行回滚操作。

可以看出,2PC 对分布式系统做了一些假设:

- 网络不会传递错误的信息(没有拜占庭故障);
- 网络从回复 Prepared 到收到 Commit 的短时间内是可靠的,也就是参与者都能收到 Commit;
- 失联的节点最终能够恢复,不会永久性地处于失联状态(这样即使收到 Commit 后未能提交,也能从阶段 1 的日志中恢复);

2PC 的缺点有二:一是**单点故障**,协调者是一个单点,一旦协调者挂掉,整个系统就会进入不确定状态,此时唯一的解决办法是等待协调者恢复或者人工介入。二是性能问题,可以看出阶段 2 的提交是 Fast Path,是轻量级的;回滚是 Slow Path,是重量级的;换言之,2PC 已经尽力优化了大部分节点正常工作场景下的性能。然而,2PC 需要等待所有节点回复 Prepared,这个过程仍然需要等待最慢的节点,因此性能不好。为解决这两个问题提出了**三阶段提交**。

8.3.2 Paxos

第三种解决方案是 **Paxos 算法**,它将分布式系统中的节点分为三类:**提案节点**(Proposer)、**决策节点**(Acceptor)和**记录节点**(Learner)。Paxos 有三个阶段:

(阶段 1a) 提案节点收到来自客户端的请求后,选择一个最新的提案编号 n,向超过半数的决策节点广播 Prepare 消息,请求决策节点对提案编号进行投票。决策节点收到 Prepare 消息后,如果提案编号 n 大于该节点已经批准的最大提案编号,则该节点批准提案编号 n,并向提案节点发送批准消息;否则,该节点拒绝提案编号 n,并向提案节点发送拒绝消息。

(阶段 1b) 决策节点收到提案节点的提案编号 n 后,如果提案编号 n 大于该节点已经批准的最大提案编号,则返回 Promise 消息。如果决策节点之前已经批准了某个提案,那么 Promise 消息还应将前一次提案的编号和对应的值一起发送给提案节点,否则回复 null. 发送 Promise 信息后,决策节点承诺:不再接受提案编号小于等于当前请求的 Prepare 请求;不再接受提案编号小于当前请求的 Propose 请求。

(阶段 2a) 提案节点收到超过半数的决策节点的 Promise 消息后,向多数派的决策节点发起 Accept 请求,带上提案编号和提案值。关于提案的值的选择,如果之前决策节点的 Promise 消息有返回已接受的值,那么使用提案编号最大的已接受值作为提案值;如果没有返回任何已接受值,那么提案节点可以自由决定提案值。

(阶段 2b) 决策节点收到 Accept 请求后,若没有承诺过提案编号比 n 更大的提案,则决策节点接受该提案,更新承诺的提案编号,保存已接受的提案,并向提案节点发送 Accepted 消息;否则,决策节点拒绝该提案,向提案节点发送 Rejected 消息。

(阶段 3) 提案节点收到超过半数的决策节点的 Accepted 消息后,将提案值作为决策结果提交给所有记录节点。否则,延迟一段时间并重启整个过程。

Paxos 能在部分同步系统和崩溃故障下达成共识,只需 $N \geqslant 2f+1$,但不能容忍拜占庭故障。**PBFT 算法**(Practical Byzantine Fault Tolerance)是拜占庭容错算法的改进,使用了加密算法,能在 $N \geqslant 3f+1$ 的条件下容忍拜占庭故障,且通信复杂度为 $O(N^2)$。

例题集 8.1: 容错与分布式共识

问题 1 True or False?

(1) The non-blocking variant of remote-write primary backup guarantees sequential consistency.

答:False. Because it is non-blocking, clients apply writes before the server they contacted hears back from the primary, possibly causing clients to apply writes in a different order than the primary.

(2) Using Paxos, suppose a majority of participants replies "Accept-OK" of some proposed value V_1. A (possibly different) majority of participants might later reply "Accept-OK" of a different proposed value V_2.

答:False.

(3) Paxos ensures liveness in the face of $(n/2)-1$ node failures out of n total nodes.

答: True. Only a strict majority of nodes need to be up. (Also accepted False: Paxos does not guarantee liveness.)

(4) Even if the proposers in Paxos do not wait for a majority of acceptors to respond to the Prepare() and Accept() messages, the Paxos guarantees still hold.

答:False. This could lead to multiple values being chosen (violating safety).

(5) In the basic Paxos algorithm, livelocks are impossible.

答:False. The nodes may repeatedly fail to reach majority.

问题 2 You're building a storage array for videos of cats riding Roombas (扫地机器人). You've decided to use really cheap disks with a Mean-Time-To-Failure (MTTF) of 10 000 h ($\approx$ 1.1 y). The drives have the following performance characteristics:

- MTTF: 10 000 h
- Capacity: 500 GB
- Sequential read/write speed: 10 000 MB/s
- Seek time: 10 ms

(1) Assume you have built a RAID 1 ("mirrored" – each disk has a copy of all data) system using two disks. Without any humans around to replace dead drives, what is the expected mean time to data loss of this simple RAID 1 system? (Keep your answer simple, using the most simple model of MTTF we discussed).

答: $\frac{1}{2}\text{MTTF} + 1\text{MTTF} = 1.5\text{MTTF} = 1.5 \times 10\,000\,\text{h} = 15\,000\,\text{h}$.

(2) If one disk fails, how long (in seconds) will it take to rebuild the array by copying all data onto a new disk?

答: $\frac{500\,\text{GB}}{50\,\text{MB/s}} = \frac{500\,000\,\text{MB}}{50\,\text{MB/s}} = 10\,000\,\text{s}$.

(3) Again, using the simplest model, what are the chances of the second disk dying during this rebuild?

答: $\frac{10\,000\,\text{s}}{10\,000\,\text{h MTTF}} = \frac{1}{3600}$.

(4) In practice, the chances are probably higher than the simple calculation above would suggest. Give two reasons why this is the case.

答:

(1) The load on the disks is higher during repair, which can increase the chances of a failure.

(2) Failures are often correlated because they share the same cause — vibration, power fluctuations, etc.

(3) Disks do not have a flat lifetime curve. If the disks are approaching the end of their lifetime, the "bathtub curve" will start ramping up sharply, so both disks are much more likely to die than a simple mean would indicate.

(5) You decide instead to build your array using Flash-memory based solid state drives. These drives are fast and silent, but Flash memory behaves differently from disks: It suffers "wear-out." Each time you write to flash memory, it loses a small amount of its lifetime. As soon as you exceed its rated number of writes, the flash is roughly guaranteed to die. However, Flash is often more reliable than disk is before the end of its lifetime. What problem does this cause for your RAID 1 array?

答:When one disk dies, the other is extremely likely to die soon.

(6) You decide to fix this problem by giving new instructions to the system administrator for he or she should replace or repair disks in the RAID array. What do you tell them to do in order to drastically reduce the chances of having the whole system fail?

答:

(1) You could tell them to pre-emptively replace the SSDs, one at a time, before their lifetime ends.

(2) You could tell them to create the RAID array using staggered generations of disks, so that neither disk is ever totally at the end of its lifetime.

(3) Something else clever here...

问题 3 Dawn Bovik is designing a new primary-backup system. They want to assign two backups to the primary, marked as backup-A and backup-B. Their replication mechanism works as follows:

- If the primary fails, backup-A is promoted to be the new primary.
- If backup-A dies or is promoted, backup-B is marked as backup-A.

To make sure the system works with slow and unreliable networks, Bovik decides to replicate only a single copy of each operation as follows:

- The primary will forward each operation at first to just backup-A.
- If the RPC fails, then only in that case does the primary forward to backup-B.
- If that RPC fails, the primary tries backup-A again, and so on, until the RPC to one of them succeeds.

(1) Can this system tolerate 2 server failures?

答:No. In the general case there may be only one copy of each operation so it may get lost. We also wanted you to mention that the primary cannot backup if there are neither of the backups available, so will keep trying.

(2) Can this system tolerate disconnection of backup-A from the network (in the absence of any other failures)?

答: Yes. This protocol allows the primary to continue quickly in the face of slow/unreliable connectivity to

backup B.

(3) Bovik realizes this design has a serious consistency flaw. Provide an example of scenario where this system will return an inconsistent result under failures that a Paxos system should be able to handle (i.e., one of the three nodes fails).

答: The client sends put(x,99) to the primary; the primary forwards the put only to backup-B; the primary fails and backup-A becomes the new primary; now a get(x) won't return 99. You had to mention two aspects to get full credit:

- Backup A and Backup B don't have the same contents so they are inconsistent.
- Also mention that the Backup-A be becomes primary on the failure of the primary, then the system will return an inconsistent result.

问题 4 Consider a distributed transaction, T, operating under the two-phase commit (2PC) protocol. Let N_0 denote the coordinator node, and $\{N_1, N_2, N_3\}$ denote the set of participant nodes. The following five messages have been exchanged between the nodes:

(1) N_0 to N_1: "Phase 1: Prepare"

(2) N_0 to N_2: "Phase 1: Prepare"

(3) N_1 to N_0: "OK"

(4) N_0 to N_3: "Phase 1: Prepare"

(5) N_2 to N_0: "OK"

(1) Assuming no messages have been dropped in the exchange so far, who should send a message at time 6? Who is the intended recipient of this message?

答: N_3 should send a response to N_0.

(2) Suppose that N_0 never receives the "OK" message from N_2 due to a network failure (i.e. the message is dropped). Instead, N_0 "times out" after waiting for an extended period of time. What should happen under the 2PC protocol in this scenario? Briefly explain why.

答: After the timeout, N_0 will assume that N_2 has failed and it will mark the transaction as aborted. 2PC requires all participants to respond with "OK".

(3) Suppose that, at some point, N_0 enters Phase 2 and sends out a "Phase 2: Commit" message to all of the participants. However, N_1 crashes before it receives this message. What is the status of the transaction T when N_1 comes back on-line? Briefly explains why.

答: Once the coordinator receives the "OK" message from all participants, the transaction is deemed to be committed even if a node crashes during the second phase. In this example, N_1 would restore T when it comes back on-line.

问题 5 For each of the following scenarios, indicate which replication scheme you would use: either primary-backup or Paxos.

(1) A lock server for a distributed filesystem that supports buying and selling stocks on the new york stock exchange.

答: Paxos. The service must be extremely available and consistent, and it's very lightweight, so replication is reasonably cheap.

(2) Storing data on data nodes in a distributed filesystem that stores your movie collection.

答: Primary-backup. The data is large, so doing too many replicas is expensive. It's also not super-critical data nor does it need consistency.

(3) A login server (accepts username/password, tells whether OK, and allows users to change passwords) used to support both of the above services, and many others.

答: Paxos. The service is lightweight, so replication isn't too expensive, but is used by real-time and important

services that can't go down.

问题 6 Recall that Paxos — an algorithm for fault tolerant replication under non-byzantine failures — requires $2f+1$ replicas to handle f failures. BFT, on the other hand, requires $3f+1$.

(1) Why is it sufficient in Paxos to use a majority vote among $2f+1$ nodes to ensure consistency? (In other words, what property of a majority are we relying on?)

答:The key property is that any two majorities intersect in at least one node. Thus, any decision reached by one majority must be observed by any other majority, ensuring that the value chosen will propagate even if the other nodes didn't see it.

(2) Explain why Paxos cannot tolerate f failures with less than $2f+1$ nodes.

答:Assume we try to tolerate f faults with only $2f$ nodes. If f nodes fail, a leader cannot hear back from a majority of acceptors ($f+1$ nodes), and a value will never be chosen, or it is possible that two separate groups would reach their own separate "consensus".

(3) Why can BFT can succeed with using $3f+1$ replicas and requiring a consistent answer from $2f+1$ of the nodes? (Short answer! One sentence.)

答:In $3f+1$ nodes, every majority of $2f+1$ nodes must contain a majority of "good" nodes ($f+1$, since there can only be f bad nodes). Thus, the good guys overrule the bad guys.

(4) Explain why Paxos need proposal numbers.

答:Proposal numbers are necessary to ensure that there is always a strict priority of proposals. Without them, it would be impossible to know which one to accept when two different proposal are received.

(5) Explain why Paxos need both Prepare and Accept.

答:In the Prepare stage, we are seeing if our proposal numbers are out of date and gauging whether we need to update the proposed value. In the Accept stage, we are actually broadcasting the value that we want other people (nodes) to accept.

(6) Explain why livelock is possible in Paxos.

答: If two "dueling proposers" are competing with each other, then they could act in a way where they are constantly proposing and interfering with the other's proposals.

问题 7 You want to design a file system that can tolerate up to f node crash failures. For each of the following scenarios, how many file servers do you need at least? State the values and explain in one sentence.

(1) The file system is using Primary-backup to replicate the files.

答:$f+1$. In Primary-backup, we maintain n replicas (1 primary and $n-1$ backups). Each backup can take over if the primary node fails.

(2) The file servers are using Basic Paxos to elect a leader among themselves.

答:$2f+1$. Paxos requires the majority of the acceptors for a successful transaction commit.

问题 8 Srini decides to outsource his distributed system nodes to NeverNeverLand. Help him understand the reliability/availability tradeoffs of his system design.

(1) The LostBoys Corporation runs Srini's 61 server nodes. While the LostBoys run a safe system (i.e. nodes are never corrupted or run faulty code), they may fail (i.e. fail-stop) occasionally. Srini decides to use two-phase commit in his distributed application. If Srini wants to provide 100% availability to his customers, what guarantee does he need from LostBoys (i.e. how many LostBoy server nodes can fail)? Give a 1-2 sentence explanation.

答:All nodes must be reliable — i.e. no failures.

(2) The LostBoys Corporation runs Srini's 61 server nodes. Srini decides to upgrade to using Paxos in his distributed application. If Srini wants to provide 100% availability to his customers, what guarantee does he need from LostBoys (i.e. how many LostBoy server nodes can fail)? Give a 1-2 sentence explanation.

答:Up to 30 nodes can fail — Paxos uses majority voting.

(3) It turns out that the LostBoys occasionally outsource the operation of some of the server nodes to HookCo. Unfortunately, HookCo is notoriously untrustworthy and their servers suffer from all types of failures and are often running corrupted code. The LostBoys promise that despite the use of HookCo nodes only a limited fraction of Srini's 61 total nodes will fail at any time (no more than your answer to part b). If Srini continues to run Paxos on his nodes, will his system work fine? Why or why not?

答: Paxos may produce inconsistent results since it is not designed to deal with nodes that don't follow the protocol correctly.

(4) On the set of 61 nodes (some run by LostBoys and some run by HookCo), Srini decides to use the PBFT (Practical Byzantine Fault Tolerance) protocol in his distributed application. If Srini wants to provide 100% availability and correct results to his customers, what guarantee does he need from the set of 61 total server nodes? Give a 1-2 sentence explanation.

答: PBFT requires $3f + 1$ nodes — therefore $f = 20$ — no more than 20 nodes can fail/not follow protocol at any time. This probably means that less than 20 HookCo nodes and at most $20 - \#HookCo$ nodes can fail-stop at any time.

8.4 可靠的 RPC

在很多情况下,分布式系统中的容错关注的是故障进程,但是我们也需要考虑通信故障。前面讨论过的大多数故障模型(例如崩溃性故障、遗漏性故障、定时故障和随意性故障)在信道上也是适用的。本节我们会讨论 RPC 可能出现的失败以及解决方案,这些失败使得 RPC 终究不同于 IPC 或者本地过程调用。

第一类失败是**客户可能不能定位适当的服务器**,比方说所有服务器都挂了,也可能服务器接口升级了,不认识客户的请求了。这种情况下只需在客户端抛出异常即可。

第二类失败是**请求消息丢失**,只需让客户端超时重传即可,服务端也需要合适的方法来处理重传的消息。但其实这件事并不简单,我们将在下下节继续讨论。

第三类失败是**请求到达了服务器,但是服务器回复之前就崩溃了**。麻烦的部分在于区分服务器是执行之后崩溃的还是在执行之前崩溃的。服务器可以对数据处理的次数做出保证:第一种方法是**最多一次语义**,保证每个请求最多被执行一次。第二种方法是**至少一次语义**,保证每个请求至少被执行一次。**恰好一次语义**是最理想的,但是基本上无法实现。客户端则可以在**从不重发、总是重发、收不到请求确认时重发、收不到应答时重发**之间做出选择。

第四类失败是**应答消息丢失**,这种情况下客户端也需要超时重传。问题在于无法确定是请求丢失、应答丢失还是单纯的网络延迟。解决方法是将所有请求变为**幂等**的,即多次执行不会产生副作用。

第五类失败是**服务器回复之前客户端崩溃**,这种情况下服务器还在计算,但已经没有人接受计算的结果了,这种不需要的计算称为**孤儿**。孤儿会引起多种问题,比如浪费 CPU 周期、锁定文件或占用有价值的资源、造成结果的混淆等。处理孤儿有四种方法:

- **孤儿消灭**(孤儿灭绝,extermination):在客户端发送 RPC 消息前进行日志记录说明要做什么,然后在客户端重新启动之后对日志进行检查然后明确地杀死孤儿。
- **再生**(转生,reincarnation):当客户重启时,就向所有的机器广播一个消息说明一个新时期的开始,当这样的广播到达时,所有与那个客户有关的远程计算都被杀死。
- **优雅再生**(温柔转生,gentle reincarnation):这种方法是前一种方法的变种。当时期广播到达时,每台机器都进行检查来查看是否存在远程计算,如果有,那么就尝试定位它的拥有者,只有当不能找到拥有者时才杀死该计算。
- **到期**(expiration):每个 RPC 都被给定一个标准的时间量 T 来进行工作,如果到时不能结束,那么就必须显式地请求另外的时间量。

实践中这些方法都不令人满意,杀死一个孤儿可能带来无法预料的结果。例如,孤儿可能持有锁,还可能设置了任务来在将来某个时候启动其他进程,这样即使杀死孤儿也不能删除它的所有痕迹。

例题集 8.2: RPC 语义

问题 1 BergerNet is a new social network started by the TAs and Professors of 15440 providing almost the same functionality as social networks do today. As a new engineering lead, it is your job to make some crucial design decisions that are either going to make or break the BergerNet.

(1) BergerNet uses RPCs. Identify the easiest-to-implement RPC semantic (at-least-once, at-most-once, exactly-once) for each of the following features.

(1.a) Transferring money to your friend on BergerNet.

答:At-most-once. Easiest to implement and can always retry sending on failure.

(1.b) Posting a message in the BergerNet messenger.

答:At-least-once or at-most-once. At-least-once is easier to implement, but at-most-once is more efficient.

(1.c) Deleting a friend on BergerNet.

答:At-least-once. Delete is idempotent. At-most-once does not work because it is not the easiest to implement.

(2) Next, BergerNet users want to upload group collages of their friends and colleagues. Privacy is important. So before a user can upload and publish a collage, the user needs to get approval from all their friends and can only upload the collage if all of them approve. Even if one friend rejects the proposal, the collage cannot be uploaded.

(2.a) Name a protocol from lectures, with the least number of messages, to implement the collage feature.

答:2PC. Token ring might be accepted based on reasoning below.

(2.b) Describe each stage of this protocol with a bullet point in the given context.

答:

- Voting phase — uploader asks and waits for votes.
- Commit Phase — uploader decides whether to commit or not and sends decision.

(2.c) If the computer of the user who uploads fails during this protocol what could be one negative outcome other than the upload failing?

答:Blocking. Consistency is not an issue because 2PC guarantees consistency by blocking.

(2.d) Is there a way of solving this problem? If yes, name a protocol that resolves the issue, if no, explain in one sentence why.

答:3PC.

(3) BergerNet is very popular these days mainly because of how little it crashes.

(3.a) Define reliability and availability (one sentence each).

答:Reliability is the frequency of errors in a system (how long can a system run before encountering an error). Availability is the ratio of the uptime to (uptime + downtime) (slide definition also accepted).

(3.b) Let's take the previous year as an example. BergerNet crashed once every hour and recovers within a second. Would you say that BergerNet is highly available and/or highly reliable or neither?

答:Availability: $1 - 1/3600 = 0.9997222$. So, highly available. Reliability: time between crashes is less than an hour — that's bad. So not reliable.

A P2P

根据网络拓扑的不同,可以将 P2P 网络分为 4 类:集中式拓扑、非结构化全分布式拓扑、半分布式拓扑和结构化全分布式拓扑。

A.1 Napster 和集中式 P2P

Napster 是最早的 P2P 系统之一。严格来说,Napster 并不算 P2P 系统,只是实现了文件查询和文件传输的分离。Napster 将文件名和文件所在的 IP 地址存储在中央服务器上,用户通过中央服务器查询文件名,然后直接从文件所在的 IP 地址下载文件,所有的信息使用 TCP 明文传输。Napster 的缺点有以下几点:

- 中央服务器的带宽消耗大,容易成为系统瓶颈。
- 中央服务器一旦出现问题,整个系统就会瘫痪。
- 没有安全性保证,所有信息和密码都是明文传输。
- 集中式服务无法规避法律风险。

这些缺点导致 Napster 无法长期运行,但是 Napster 的成功也为后来的 P2P 系统提供了借鉴。

A.2 Gnutella 和非结构化全分布式 P2P

Gnutella 吸取了 Napster 的教训,采用了全分布式的拓扑结构,完全没有中央服务器。Gnutella 使用无向图连接节点,节点可以与邻居相互通信。用户节点可以向邻居节点发送搜索请求,邻居若有符合条件的文件则返回给用户,否则将搜索请求转发给邻居的邻居;这样,搜索请求就可以遍历整个 Gnutella 网络。找到文件后,用户节点与拥有文件的节点通信,找到最佳的节点下载文件。Gnutella 还会使用 Ping 消息和 Pong 消息来维护邻居列表。Gnutella 已经尽可能减少了搜索时的冗余网络流量:每个节点只转发一次搜索请求、只转发一次搜索结果、丢弃已经见过的搜索请求、丢弃没有见过请求的搜索结果、丢弃 TTL 到期的信息。然而,这些举措难以与洪泛式搜索中指数级增长的网络流量相抗衡,且 Ping 消息和 Pong 消息的数量也会随着用户规模的增长而增长,最终导致网络过载,可扩展性差。另一个问题在于大部分(70%)用户都是白嫖者,只有少部分用户愿意分享文件。

A.3 FastTrack 和半分布式 P2P

FastTrack 是一个半分布式的 P2P 系统,其中节点分为两类:超级节点和普通节点。超级节点是 FastTrack 网络的骨干,负责维护整个网络的拓扑结构,彼此之间直接连接;普通节点则必须通过超级节点才能连接到 FastTrack 网络。搜索时,普通节点向超级节点发送搜索请求,超级节点在自己的子节点中搜索,同时也将搜索请求转发给其他超级节点;找到后,超级节点将搜索结果返回给普通节点,普通节点再与拥有文件的节点通信。超级节点还能记录普通节点的贡献量,将贡献量高的普通节点升级为超级节点。这种分层结构介于 Napster 的集中式结构和 Gnutella 的分散式结构之间,较好的解决了单点失效和可扩展性问题之间的矛盾。

目前最为常用的 P2P 系统 BitTorrent 也是半分布式的。BT 系统中,文件被分为小块,用户优先下载稀缺的小块,然后上传给其他用户。用户加入前必须已知两种信息:一是 Tracker 地址,Tracker 是一类记录节点信息,帮助节点找到其他节点的服务器;二是种子文件,其中包含了 Tracker 的地址、块长度、块哈希值等信息。加入后,用户就可以通过 Tracker 找到其他用户,然后与其他用户通信。BT 系统能够善用已经下载的小块文件,内置了鼓励分享的机制,对于热门文件来说是非常高效的;缺点在于冷门文件可能就死种了。

A.4 Chord 和结构化全分布式 P2P

Gnutella 的效率不高:对于有 N 个节点的网络,Gnutella 的存储开销、查找开销和带宽开销都是 $O(N)$。Chord 系统在保持全分布式的前提下将它们三个都降为了 $O(\log N)$。Chord 系统已经在 **??** 节中介绍过了,这里讲述如何使用 Chord 实现 P2P 文件共享系统。Chord 将文件名使用相同的 Hash 函数映射到为一个键,然后将文件存储于大于等于这个键的最邻近的节点上。Chord 启发了很多其他形式的 DHT,如 Pastry、Kelips 等,BitTorrent 也加入了 DHT 支持。

B 谷歌云计算原理

谷歌业务范围广泛,数据量巨大,又要为全球用户提供实时服务,因此谷歌研发出了一组分布式系统技术,用于支撑谷歌的各项业务。本节将介绍 GFS、MapReduce、Chubby、Bigtable 等谷歌云计算平台的核心技术。

B.1 GFS:谷歌的文件系统

GFS 是谷歌开发的分布式文件系统,用于存储谷歌的海量数据。GFS 的系统架构如图 ?? 所示。

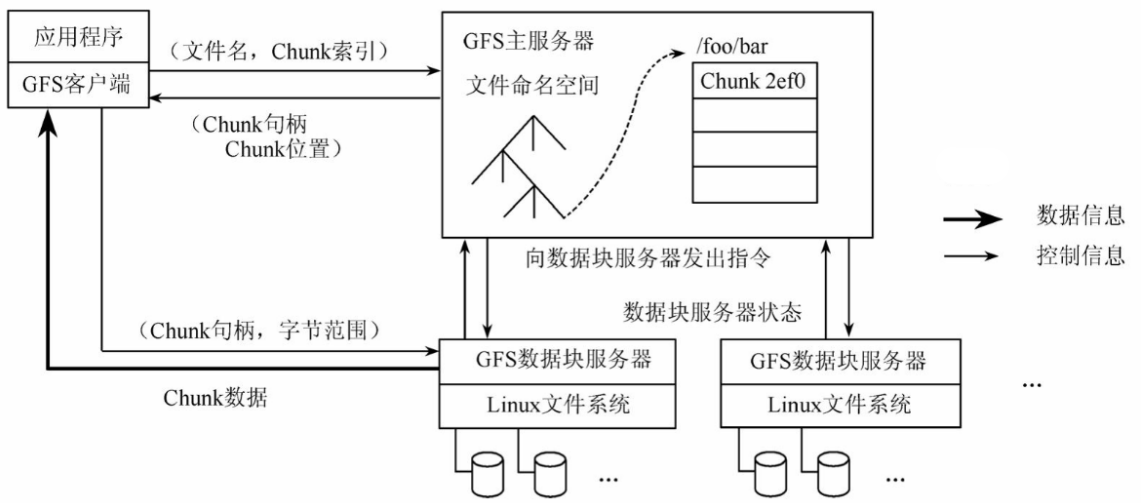


图 B.1: GFS 系统架构

客户端访问 GFS 时,首先访问主节点,获取与之进行交互的块节点信息,然后直接访问块节点,完成数据存取工作。这样就实现了控制流和数据流的分离。客户端与主节点之间只有控制流,而无数据流,极大地降低了主节点的负载。客户端与块节点之间直接传输数据流,同时由于文件被分成多个块进行分布式存储,客户端可以同时访问多个块节点,从而使得整个系统的 I/O 高度并行,系统整体性能得到提高。

GFS 的目标和基本假设如下:

(1) **硬件失败是常态**。GFS 建立在大量廉价商用设备之上,即使每一部分的故障概率都很低,连接在一起使用总的故障概率也将大幅提高。因此,快速地检测故障并从故障中恢复,是系统的核心设计目标之一。

(2) **主要负载是流式读写**。GFS 面向的应用以大规模流式处理为主,而非一般文件系统中常用的交互式处理。因此,GFS 注重存取时的高吞吐量,而非低延时。换言之,GFS 更注重顺序读取、追加写入而非随机读写。

(3) **大数据集**。GFS 应当能够用来存取规模巨大的文件,文件大小从几百 GiB 到数千 TiB 不等,数量则以千万计。因此,GFS 必须能够管理成千上万的可扩展数据节点,并且具有高数据传输带宽。

因此,GFS 做出了以下设计:

(1) **将文件划分为固定大小的块**。GFS 将文件划分为固定大小的块(chunk),每个块的大小为 64 MiB,并且每个块都有一个全局唯一的 64 位块索引号。块是 GFS 的最小存储单元,GFS 以块为单位进行存储和传输。64 MiB 是一个非常粗的粒度,作为对比,本地文件系统和 NFS 的块大小都是 KiB 级别。这样做是为了减少块的数量(相应地,减少元数据的数量),以及减少客户端与主节点、与块节点之间的通信次数(提供高效的顺序读和追加写)。

(2) **每个块都有多个副本**。一般是三个副本,存储在不同的块节点上。块的分布综合考虑了多种因素,如网络的拓扑、机架的分布、磁盘的利用率等。如果相关的副本出现丢失或不可恢复等情况,主节点自动将该副本复制到其他块节点。写入时,必须将所有的副本全部写入成功,才视为成功写入;读取时,块节点会将块与存储的校验和(每块 32 b)进行比对,如果比对失败则会返回错误,指引客户端读取其他副本。

(3) **使用主节点**。主节点负责管理所有的元数据,包括文件系统的命名空间(文件系统的目录结构)、文件名到块的映射、块到块节点的映射等。引入主节点简化了设计,实现简单,消除了元数据的一致性问题,注册新的块节点、实现负载均衡也很容易。当然,主节点也是性能瓶颈和单点故障的来源,GFS 采用多种措施(分离数据流和控制流、备份主节点、客户端缓存元数据、压缩元数据)来缓解这些问题。

(4) **不缓存数据**。GFS 不缓存数据,因为大部分的访问都是顺序读写,而且块节点(的本地文件系统)和客户端都会缓存数据,因此缓存数据对性能的提升不大。另一方面,缓存引入了复杂的一致性问题,得不偿失。

(5) **在用户态实现**。GFS 在用户态实现,而不是内核态实现,这样做的好处是提高了系统的可移植性和稳定性,同时便于调试,便于升级。

(6) **只提供专用接口**。GFS 只提供专用接口,而不是 POSIX 接口,这样做的好处是降低了实现的难度,为特定需求提供了更好的支持。

B.2 MapReduce:谷歌的分布式计算模型

MapReduce 是谷歌提出的一种分布式计算模型,用于大规模数据集(通常大于 1 TiB)的并行运算。MapReduce 封装了并行处理、容错处理、本地化计算、负载均衡等细节,提供了简单而强大的接口,大量不同的问题都能基于这一接口来解决。

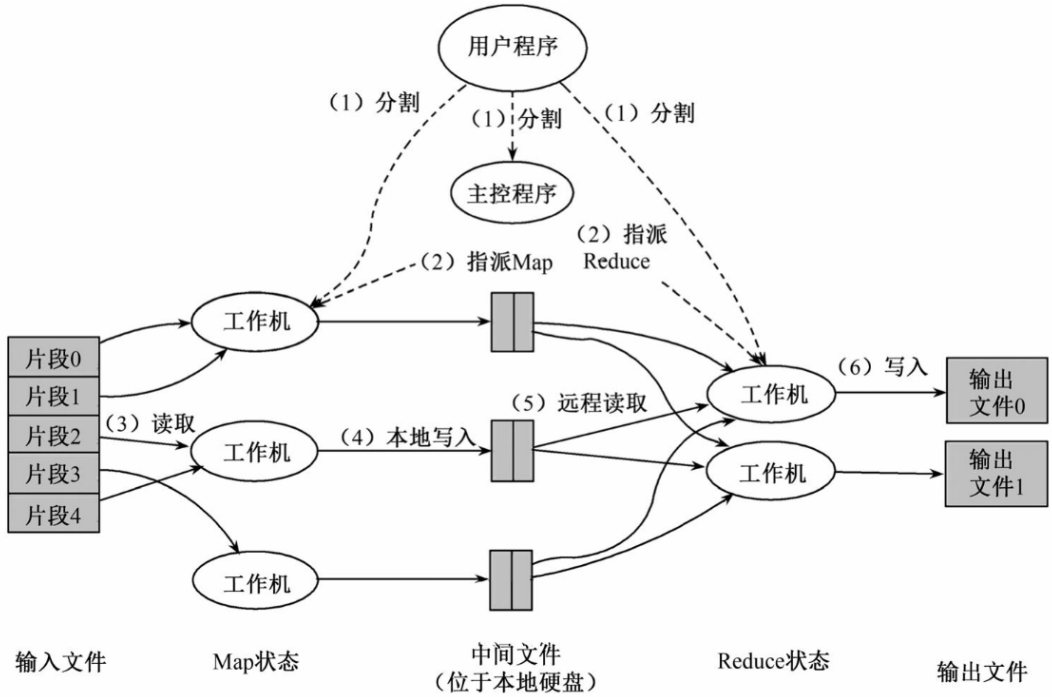


图 B.2: MapReduce 系统架构

MapReduce 的执行流程如图 **??** 所示。用户需要编写 map 和 reduce 两个函数,map 函数的输入是一小块原始数据(或者单个键值对),输出是一个键值对表;reduce 函数的输入是一个键和对应的值列表 $(k, [v_1, v_2, \ldots, v_m])$,输出这个键和处理的结果 (k, v_{final})。具体的工作流程如下:

(1) MapReduce 首先将输入文件划分成 M 块,每块大约 16 MiB – 64 MiB,接着在集群的机器上执行分派处理程序。分派处理程序分为一个 Master 和若干 Worker,一共有 M 个 map 任务和 R 个 reduce 任务需要分派,Master 选择空闲的 Worker 来分配这些任务。

(2) map Worker 读取并处理相关的输入块,并将分析出的键值对传递给用户定义的 map 函数。map 函数产生的键值对缓冲到内存,并定时写到本地硬盘。

(3) Master 将中间结果的位置信息传送给 reduce Worker,reduce Worker 读取键值对并进行排序。

(4) reduce Worker 遍历排序后的键值对,并将键和相关的值列表传递给用户定义的 reduce 函数。reduce 函数的结果写到一个最终的输出文件。

(5) 当所有的 map 任务和 reduce 任务都完成的时候,Master 返回。

MapReduce 需要在成千上百台机器上处理数据,因此容错机制必不可少。MapReduce 的容错机制主要包括两个方面:Master 失效一般较少,通过检查点和恢复机制来处理;Worker 失效较多,通过周期性的 ping 命令来检测,然后重新分配任务来解决。

B.3 其他

谷歌还开发了一些其他的分布式系统,如 Chubby、BigTable 等。Chubby 是谷歌的分布式锁服务,同时也是一个只能存储非常小文件的分布式文件系统。BigTable 是谷歌的分布式数据库,用于存储结构化数据。

例题集 B.1: P2P 和谷歌云计算原理

问题 1 True or False?

(1) In Google File System (GFS), when a client needs to access data, it contacts the master and gets the data from the master.

答:False.

(2) The Map step of MapReduce provides a way to store and retrieve items according to their keys.

答:False.

问题 2 Which of the following are advantages of using content-based naming for p2p file transfers?

A. The receiver can download a chunk using parallel TCP streams.

B. The receiver can route around failed links.

C. The receiver can download a chunk of data from any source, not just the original seed.

D. The receiver can verify that it received the correct data.

答:C, D.

问题 3 Which of the following design assumptions apply to HDFS/GFS? (Circle all that apply)

A. Only amazingly powerful servers can handle the size and complexity of the workload.

B. Any use of a centralized server would cause too much of a bottleneck.

C. Component failures are expected.

D. Files are huge by traditional standards.

E. File contents are often updated and overwritten.

答:C, D.

问题 4 Peer to peer (p2p) networks are a popular way to share files.

(1) In a centralized p2p network (such as the old Napster), how many indices must be searched if a client wants to locate a particular file?

答:1.

(2) Name one major disadvantage of the centralized p2p system (from a distributed principles point of view)

答:Single point of failure, server processes everything, server must keep track of a potentially very large number of clients, more?

(3) Query flooding is an alternative design that solves some of the problems of centralized p2p and eliminates the central server. However, it changes the mechanics of peer interactions significantly. Explain (1 sentence each) how a newly-joining node publishes the files they wish to make available, in...

(3.a) A centralized p2p network:

答:They send their list of files and metadata to the server.

(3.b) A query flooding p2p network:

答:They don't do anything - queries come to them.

(4) One popular improvement upon query flooding is to move to a "supernode" flooding architecture. Using N as the number of nodes in the network and S as the number of supernodes ($S \ll N$), explain the benefit of moving to this supernode architecture.

答:Queries require now $O(S)$ messages instead of $O(N)$. In addition, if the nodes used as supernodes are more stable or have higher capacity, can further improve the performance or stability of the network.

问题 5 MapReduce has proved an extremely popular framework for distributed computation on large clusters, because it masks many of the painful parts of ganging together thousands of nodes to accomplish a task.

(1) In general high-performance computing (HPC), programmed using message passing, what is the technique used to provide fault tolerance? (Very short answer)

答:Checkpoint/restore.

(2) Identify two important limitations that MapReduce places upon the Map functions so that the framework can hide failures from the programmer.

答:They must be side-effect free, deterministic, and idempotent.

(3) What advantage does this give MapReduce over MPI for failure handling and recovery? (There are several—list the one(s) you think is most important)

答:Recovery can be done by only executing the work that was being handled on the failed machine, not the entire cluster.

(4) How does MapReduce handle "straggler" tasks that take longer than all of the others (e.g., perhaps they're running on a slower machine or one with buggy hardware)?

答:It executes them in duplicate on another machine towards the end of execution.

(5) MapReduce and the Google File System (GFS) were designed to work well together. What important optimization in MapReduce is enabled by having GFS expose block replica locations via an API?

答: The MapReduce scheduler can arrange for Map tasks to execute on the same node that stores the data, avoiding a copy across the network.

问题 6 We studied the MapReduce programming paradigm for large scale data analysis on clusters, and compared and contrasted that to the more traditional Bulk Synchronous Programming (BSP) methods.

(1) Explain what the problem with "stragglers" is when using MapReduce and how is it addressed?

答: Stragglers mean that some Map/Reduce computations may take much longer than others, delaying the entire phase. Solution is to rerun the tasks on nodes that have finished first and then take the result for that task from whoever finishes first.

(2) Under the MapReduce programming model, how does one deal with a single master node failure? (Assume there is a single master node).

答: 1. Master writes periodic checkpoints such that a new copy can be started from the last checkpointed state when master fails. 2. If only a single master, that particular MapReduce task is aborted and has to be restarted with a new master.