02 A High Scalability Web Server

1. 特点：
    - 高请求量
    - 大量数据
    - transparent scale（隐藏用户规模）
2. 处理每个请求的所耗的时间是不一样的（ai 时代中），推理所耗的时间会长
3. LAMP 架构不能 scale，只能在单台服务器上
4. scalability 1：解耦，把不同的功能分开，不同的功能用不同的服务器来处理，比如数据库服务器，文件服务器，web 服务器
5. scalability 2：缓存，把一些数据缓存起来，比如 page cache，memcached（专门的缓存服务器，用内存来存）memecahed 可以支持在多台服务器上，可以 scale。一个经典的方法就是把 key hash 到不同的服务器上。但是原始的 hash 在 scale 的时候，会造成很多 miss。解决方法是 consistent hashing
6. scalability 3：更多的 app server，比如用负载均衡器来分发请求
7. scalability 4：数据库分库和分表
8. scalability 5：文件系统的扩展，比如一个很大的文件、可以存在不同的服务器上，然后用一个文件系统来管理这些文件。eg. GFS，NFS，HDFS
9. scalability 6：用 CDN，把静态文件放到 CDN 上，CDN 会根据用户的位置，选择最近的服务器来提供服务
10. scalability 7：分开不同的 application 和业务+分布式计算。
11. 分布式系统：A distributed system is a collection of independent computers that appears to its users as a single coherent system.
12. 数据中心的规模。rack -> row -> data center
13. 出错：用户体验到的错误。随着 scale，出错的概率会增加。出错的原因：硬件故障，软件 bug，网络故障，人为错误
14. 例子：unreliable network 的很多原因，见 ppt

03 & 04 From single to distributed: inode-based File System & CAP

1. 分布式系统要给人一种一直在线的感觉，即使有一台服务器挂了，用户也不会感觉到。
2. Availabity: 高可用 （x nines）
    - 3-nines = 99.9% = 8.76h/year
    - 5-nines = 99.999% = 5.26m/year
    - 7-nines = 99.99999% = 3.15s/year
3. reliability: 高可靠性
    - 1. MTTF (Mean Time To Failure): 平均故障间隔时间
    - 2. MTTR (Mean Time To Repair): 平均修复时间
    - 3. MTBF (Mean Time Between Failure): 平均故障间隔时间
    - 4. MTTF + MTTR = MTBF
4. 达成高可用：

4. 达成高可用：
    - Replication: 相同的副本，比如多个文件服务器
        - 挑战：一致性
    - Handling failures via retry: 重启系统，重新发送请求
        - 挑战：What about consistency? Must made trade-off
5. CAP theorem: 一个分布式系统，最多只能满足三个特性中的两个
    - Consistency: 所有的节点看到的数据是一样的
    - Availability: 每个请求都能得到一个响应
    - Partition tolerance: 系统在网络分区的情况下，仍然能够工作
6. single-node inode-based file system
    - inode: 一个文件的元数据，包括文件名，文件大小，文件权限，文件的地址等
    - block: 文件的内容
    - 一个文件的 inode 会指向多个 block，一个 block 会指向下一个 block，形成一个链表
    - 一个目录的 inode 会指向多个文件的 inode
7. 文件是持久化的、也是有名字的
8. driver 和应用对文件的读写的 api 是不一样的。应用需要更高层的抽象。driver 只可以抽象出 block 的读写。系统层需要在基于硬盘的 block 读写的 api 上，提供文件的读写的 api。
9. naive file system：用 sector index 和 append 和 reallocate 操作。但是这样会有很多问题，比如文件的大小会有限制，文件的读写会很慢，文件的删除会很慢，文件名的查找不可行，文件的权限控制不可行和安全不可行。
10. 所以有了系统层对文件系统的抽象、比如 unix 文件系统的 api
11. 文件系统有很多层。
12. L1 : Block Layer:把 sector 变成 block，一个文件由很多 block 组成。能保证文件系统可以在不同大小的 sector 的硬盘上工作，保证 block 大小相同就行。
13. 如何选择合适大小的 block?
    - 太小：会有很多 block，会有很多 seek，会很慢
    - 太大：会有很多内部碎片，会浪费空间
    - tradeoff = 4KB
14. boot block: 保存文件系统的引导信息
15. super block（每个文件系统一个）: 保存文件系统的元数据，比如 block 的大小，block 的数量，inode 的数量，inode 的大小
16. bitmap: 保存 block 的使用情况
    - 1byte = 8bit，一个 bit 代表一个 block 的使用情况

```
struct inode
   integer size
   integer block_nums[N]
   integer type
   integer refcnt
```

17. L2 :file layer:
    - inode，一个文件的元数据 (index node),记录了哪个 blocks 属于哪个文件

    - 我们把 inode 存在哪里？因为每个 block 都有大小限制，所以可以把一个父的 inode block 中的几个 index 指向另外几个 indirect block，这样就可以存更多的 block 了
    - 和页表的区别：inode 头上的几个 block 可以只访问一次就可以访问到。但是页表的话，要跳转几次才能访问到。而且文件头上的数据的访问比较常见。
18. L3 : inode Number Layer：Mapping inode number -> inode
    - inode table: 保存 inode number
    - 可以再存一个 inode table 的 table，提高资源利用率
19. L4 :file name layer：Mapping file name -> inode number
    - directory entry: 保存文件名和 inode number 的对应关系
    - mapping table is saved in directory
    - 但是需要遍历
    - 当前运行的目录同样是一个文件
    - 目录的厚度由存储的文件的文件名的长度决定
    - 目录里面记录的是一个字符串到一个 inode number 的映射
20. L5 :所以有了 path name layer：Mapping path name -> inode number
21. links：读一次 path 就可以读到文件
    - 存在多个引用的情况、所以用 refcnt 来记录引用次数，保证正常增删，文件的 refcnt 为 0 的时候就删掉
    - 但是如果文件夹结果有环的话，就会有问题
    - 不允许有环的话、就不允许 link 到 directory（除了.和..）
22. 文件重命名
    - UNLINK(to_name)
    - LINK(from_name, to_name)
    - UNLINK(from_name)
    - 需要原子操作、否则如果在中间出错、就会有问题
23. ppt 中、每一行机器代码都是一个 ext4_dir_entry
24. L6 : Absolution Path Layer
    - 因为会有多个用户，所以需要绝对路径
    - 引入了绝对路径，根目录是 /
    - 根目录下的/.和/..都是指向自己的/
25. 例子：找到"/programs/pong.c"
    - 从 super block 找到 inode table 的 block
    - 找到根目录的 inode：根目录的 inode number 是 1，是固定的、因为不能通过递归的方式找到根目录
    - inode table 中的 number 存的就是 block 的 number
    - block 中文件的 number 存的是 inode table 的 number
    - 最终找到文件在 61 的 block 中
    - 一个 block 可以是一个 directory 或者是一个 file
26. 所以相当于、在 inode table 里存的东西可以是一个目录 directory、也可以是一个文件的所有 block number。

27. 也可以创建硬连接 LINK，但是 naive 的方法、没法在多个文件系统中使用，比如一台电脑上插了两个硬盘
28. L7 : Symbolic Link Layer
    - 可以跨文件系统，即跨磁盘
    - ppt 中的例子：即使/tmp/abc 还不存在、但是可以创建一个指向/tmp/abc 的 symbolic link
    - 但是 cat s-link 会报错，因为/tmp/abc 不存在
    - 创建了之后、再 cat s-link 就可以查看/tmp/abc 中的内容了
    - ls 中的 8 代表一个 symbolic link 的大小（就是保存了/tmp/abc 的路径长度）
29. 所以有了两种连接：hard link 和 symbolic link（就是 soft link）
    - hard link：文件名和 inode number 的映射（通过 inode number 找到文件）
    - symbolic link：文件名和路径的映射
30. Context Change
    - /CSE-web -> /Scholarly/programs/www
    - cd /CSE-web
    - cd ..
    - 这样会回到根目录 / 这是一个 bash 的优化（feature）
    - 只有 cd -P .. 才会回到/Scholarly/programs
31. 文件名是不是文件的一部分?
    - 一个文件的文件名不属于这个文件
    - 因为这个字符串不保存在这个文件的 block 中
    - 一个文件的文件名是保存在他的目录中的
    - name is data of a directory， and metadata of a file system
    - 元数据是什么？就是文件的属性，比如文件的大小、文件的权限、文件的创建时间等
32. 每一个 hard link 都是等价的
33. directory size 都是很小的（见上原因）
34. 文件系统要实现的 API，包装之后实现为 system call
35. open 和 fopen 的区别（后续都是基于 linux 的 open）：
    - open 是系统调用，fopen 是库函数
    - open 返回的是文件描述符，fopen 返回的是文件指针
    - open 是低级 I/O，fopen 是高级 I/O
    - open 是 POSIX 标准，fopen 是 C 标准
    - open 只能用在 Unix/Linux 系统，fopen 可以用在大多数系统
    - fopen 会有缓冲区，open 不会，所以 fopen 会有更好的性能
36. 真实的 inode 还需要加一点内容

```
struct inode
    integer size
    integer block_nums[N]
    integer type
    integer refcnt

    integer userid
    integer groupid
    integer mode
    integer atime // last access time （by read）
    integer mtime // last modify time （by write）
    integer ctime // last change time （by LINK）
```

- 为什么要有ctime呢？要记录文件权限的变化，所以要有 ctime（比如说 chmod的时候）

36. fd : 文件描述符
    - 0: stdin
    - 1: stdout
    - 2: stderr
37. 为什么要这么设计 fd 呢?
    - 选项 1：OS 可以返回一个 inode pointer
    - 选项 2：OS 可以返回文件的所有 block numbers
    - 但是为了 Security（用户永远不能访问到系统的 data structure），和 Non-bypassability（不能绕过系统的安全机制），所以用 fd
    - 又是一层抽象，能控制一个进程没法访问其他的文件
38. file cursor：每个 fd 都有一个 cursor，记录文件的读写位置。进程和他的子进程会共享这个 cursor（即从同一个位置读写）
    - 为什么要这么设计？什么时候合理？要写的时候是合理的、如果不共享一个 cursor，就会覆盖
    - 写东西的时候最好只有一个写者，single writer principle
39. fd_table：记录了所有的 fd,里面有 fd 和 index，index 指向 file table 中的一条
40. file table：记录了所有的 file。其中有 inode num，file cursor，file refcnt 等。
41. 关掉一个 fd 之后、新的 fd 是当前可用的 fd 中最小的那一个（导致不能做并行）
42. ppt 上有一个 disk 的例子图片，还有一个 open read 的时序图的例子
43. 会默认加一个参数 -noatime。
44. 例子 1
    - read 里面的 write、是要写 inode 的 atime
45. 例子 2
    - create 里面、read write inode bitmap 是因为要创建一个文件的写
    - bar inode 里面第一个 read 是因为要先读再写（粒度是 4K）
    - foo 的 inode 第二个写：是因为自己也有 atime，然后更新目录的大小
    - 写里面为什么要读 databitmap，要先读、找到一个空的 block，然后写
    - bar inode 的第二次写、更新 metadata

46. 三种顺序哪种好?

- Update block bitmap, write new data, update inode (size and pointer)
- 第一种的顺序稍好、无非是浪费磁盘空间，而且可以监测出来（先扫一遍 inode，再和 bitmap 对比一下，如果一个 block 没有被任何 inode 指向，但是 bitmap 上是 1，就是浪费的，可以回收）（问题不是那么严重）
- （第二种）要避免数据泄漏：如果新的指针给到了上一次删除的数据，但是在 write data 之前断电了，就会泄漏数据。
- （第三种）两个 inode 可能指向了同一块 bitmap，动态数据泄漏，也很不安全

47. SYNC

- 数据不落盘，好处是快、坏处是断电了之后、数据就没了
- 所以需要 sync，把数据落盘

48. delete after open but before close

一个进程已打开文件：
当一个进程打开文件时，操作系统为该文件创建一个文件描述符，并增加其引用计数。

另一个进程删除文件：
在Unix/Linux等类Unix系统中，删除文件实际上是从目录中移除该文件的名称，而不是立即删除文件的内容。此时，文件的引用计数会减少。

通过移除指向文件的最后一个名称：
如果删除了文件的最后一个链接（例如，文件名），引用计数会变为0。

引用计数现在为0：
这意味着没有任何进程通过文件名再访问这个文件，但文件的实际数据仍然存在，因为某个进程仍然打开着它。

inode在第一个进程调用CLOSE之前不会被释放：
在Unix/Linux系统中，文件的inode（包含文件元数据的信息）不会被释放，直到所有打开该文件的进程都关闭它。这是为了保持数据的完整性。

在Windows上，可能会禁止删除打开的文件：
不同于Unix/Linux，Windows系统通常不允许删除正在被任何进程打开的文件。这样做是为了防止数据损坏或不一致性。

在打开后但在关闭之前删除：
即使在Unix/Linux系统中，文件可以被删除，但只要有进程持有它的文件描述符，文件的数据仍然可以被访问，直到所有相关的文件描述符被关闭。

48. renaming

- 原来的先删除的方法、不行、因为如果断电、那么 a.txt 就没了
- 所以要用原子性的操作
- mv 的操作、只改目录的内容、不改文件的内容（inode 不变）

49. 复杂系统的 M.A.L.H 原则

- M: Modularity 模块化
- A: Abstraction 抽象
- L: Layering 分层（每一层只和相邻的层交互）

    - H: Hierarchy 层次化（组合成一个整体）
50. 04 P10 例子，如果改成了找/CSE2020.txt，其中一个 inode block 1024 byte，读一次可以读 512 byte，文件大小为 128 byte

05 Remote Procedure Call

1. 开始的例子：一个占用多的硬盘，性能可能不如占用少的硬盘（inode）
2. 在不改变应用程序的情况下（open，read，write），把文件系统放到远程服务器上，通过网络访问，节省本地的硬盘空间（即 without coding the details for the remote interaction）
3. 例子：本地调用服务器上的 GET_TIME() 函数，CALL(GET_TIME())，返回结果。call 里面的参数基本用宏，因为要快。
4. 因为不知道用的是大端还是小端，所以传参之前需要先 convert2external。network byte order 是大端。（逻辑如此、但是这个实现改了代码）
5. 发现这些为每个函数处理这个 rpc 的实现都是差不多的、所以可以根据宏来生成代码。
6. 所以有了 stub。保证了应用层代码的一致性。Stub: hide communication details from up-level code, so that up-level code does not change.
7. Client stub
    - Put the arguments into a request
    - Send the request to the server
    - Wait for a response
8. Service stub
    - Wait for a message
    - Get the parameters from the request
    - Call a procedure according to the parameters (e.g. GET_TIME)
    - Put the result into a response
    - Send the response to the client
9. inside a message:
    - Service ID(function ID)
    - Service parameters(function parameters)
    - using marshal/unmarshal to convert data to/from network byte order (Marshal 和 Unmarshal 本身是很费时的，重点在于内存的拷贝)

“marshal” 和 “unmarshal” 是计算机科学中常用的术语，特别是在数据序列化和反序列化的上下文中。以下是这两个词的含义及其在网络字节顺序转换中的应用：

Marshal

- **定义**：将数据结构（如对象或数据类型）转换为一种适合存储或传输的格式。这种格式通常是二进制或文本格式，以便在网络上传输或保存到文件中。
- **用途**：在网络编程中，marshal 通常是指将数据转换为网络字节顺序（通常是大端字节序），以确保在不同平台之间的一致性。

Unmarshal

- **定义**：与 marshal 相对，将已序列化的数据转换回原始的数据结构。这个过程通常涉及从网络字节顺序转换回主机字节顺序（可能是小端字节序）。
- **用途**：当接收到数据时，需要 unmarshal 将接收到的字节流转换为易于处理的数据结构。

10. RPC request （in 1984）：
    - Xid （transaction ID，比如读内存，发包）（client reply dispath uses xid, client remembers the xid of each all）
    - call/reply （它是一个 call 还是一个 reply 的 request）
    - rpc version （新旧版本的兼容性）
    - program # （program number：即二进制文件的号码，# 表示 number）
    - program version （即二进制文件的版本号）
    - procedure # （即调用二进制文件的哪个函数）
    - auth stuff
    - arguments
11. RPC reply message:
    - Xid
    - call/reply
    - accepted? (Yes, or No due to bad RPC version, auth failure, etc.) （是否由于 RPC 版本不对、auth 失败等原因）
    - auth stuff
    - success? (Yes, or No due to bad prog/proc #, etc.)（是否由于 prog/proc # 不对等原因）
    - results
12. 通过 binding sever and client，建立连接。核心是要找到 function ID，然后调用。
13. 接下来就是怎么传递数据和参数了（无论是参数还是返回值）。一开始有 pass by value 和 pass by reference 的区别。在本地都是可以的。
14. 怎么传一个变量的指针呢，而且要求是对的？让不同机器上的虚拟地址空间不重叠。然后当发生 page fault 的时候，发出一个远端的 handler。但是这样会有很多问题，比如读完写同步的情况。所以有指针会复杂。但是可以在多个显示器上有用，只有一个显示器能写，其他只能读。访问一台机器上的内存和硬盘（DSM）（distributed shared memory）（但是如果一块内存要在很多机器上共享，就会有很多问题，比如 cache 一致性问题，所以应用不多）
15. 还要考虑大端和小端的问题。以及服务器 32 位和 64 位的问题，如果客户端的位数和服务器的位数不一样，比如 int 的位数就不一样。还有 float 的表达也不一样。还有比如数据类型对齐的要求不一样。
16. 还有更多的挑战。比如兼容性的问题，backward compatibility 和 forward compatibility。
17. 然后是数据格式。比如 json 有一定缺陷，比如传的数字会有二义性；或者传二进制字符串的时候会有问题。所以要用 xml 存更多的 type 信息。
18. binary formats 性能更好（工业）
    - IDL： interface definition language
19. each field has:
    - type annotation
    - type field

    - a length indication(optional for string, list etc.)
    - data
20. 可以做压缩、比如压缩 type 和 tag 和 length
21. rpc 也会使用到不同的数据传输协议
22. rpc 遇到错误的时候需要处理
23. How RPC handles failures?
    - Depends on the semantic: at-most-once, at-least-once & exactly-once
    - What makes failure handling simpler? Idempotence（幂等性）
    - (幂等性：一个函数调用多次和调用一次的效果是一样的)

06 Distributed File System （DFS）：NFS & GFS

1. NFS, GFS 即可以在本地像访问本地文件一样访问远程文件
2. upload & download 的好处：简单
    - 问题 1：客户端只需要其中小部分的数据，但是要下载整个文件
    - 问题 2：如果文件很大，本地客户端的硬盘可能不够用
    - 问题 3：一致性问题，比如一个文件被多个客户端同时修改
3. 一种实现方式：Remote access model （通过 RPC 的方式，访问远程文件）
    - 问题：Possible server and network problem (e.g., congestion)
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly
4. NFS: network file system 设计目标
   Any machine can be a client or a server
   Support diskless workstations
   Support Heterogeneous deployment (支持不同的硬件、操作系统、底层文件系统)
   Different HW, OS, underlying file system
   Access transparency
   Use remote access model
   Recovery from failure
   Stateless, UDP, client retries
   High performance
   Use caching and read-ahead
5. Sun 的实现：挑选了部分的 api 使用 RPC （有些做了修改）
6. 比如 OPEN 和 CLOSE 被删掉了
7. 还有，和之前的 fd 不同，这里引入了 fh（file handler）
8. 然后多了一个 offset
9. NFS client 在本地。
10. 这里 READ（fh，0，n 的时候）不需要传 buffer、因为 buffer 是一个指针。
11. 为什么不提供 open、而是提供 lookup?
    - 因为如果用了 open、那么 RPC 就是 stateful 了。就要考虑 performant 和 scalable 的问题了。

    - 比如如果 server 挂了、再重启，那么 open 就做不了了。（因为 open 的 context 是存在内存里的）
    - 如果有多个 client 连到了同一个 server？（所以就要维护很多的 context）
    - 因此 stateless 的 RPC 的可扩展性好
12. 所以要带 offset、这样 read 就是 stateless 的 api 了（否则要维护 cursor）
13. stateless handle failure 的时候、也不能简单地只使用 at least once 策略、因为 如果有多个 client 的话、遇到网络问题的时候、会冲突。
14. 什么是一个 file handler？是指明要访问服务器上的哪个文件
    - 我们不能用 fd、因为 server 没有 file table
    - 我们能用 path name 吗？其实可以、但是 NFS 没有用，因为会有一个问题： 用 path name 的时候，如果两个 client rename after open 的情况。（会读到另外的文件夹里）
    - 能不能用 inode number 呢？也不行。会有一个 delete After Open 的问题会读到被删除的文件里（单机不会出问题、因为多机没有维护额外信息）（有一种补救方法，让 program2 知道这个文件已经被删掉了）
    - 所以未了避免这种情况范数、引入了一个 generation number。每个 inode 号除了自己的 inode 号之外、还有一个 generation number。
    - Generation number: a number incremented upon assigning an inode number to a new file
    - If read's fh does not match the current generation, then abort
    - 所以最终设计出来的 fh（file handler） 就包括了 inode number（for server to locate the file ） 和 generation number （for server to maintain consistency of a file）
    - 如果 read 的 fh 和当前的 generation number 不一样，就会返回一个错误（abort）
15. 还有一个问题，就是性能问题
    - 分布式文件系统不一定比单机的慢（看网速和磁盘速度，目前网速会大于磁盘速度）
    - 一种优化：cache at Client
16. cache 最大的难点、coherence 一致性问题
    - close to open consistency:解决方法是 close 的时候，把 cache 数据写回服务器
    - read/write consistency：server 和 client 存 timestamp、定期询问并更新（30s）
17. 还有一种可以增加 read performance 的优化就是 transfer data in large chunks（8KB default）
18. 还有就是 read ahead（prefetch，提取读未来可能用到的数据）
19. nfs 的缺陷
    - 它只能利用单台服务器上的磁盘资源，有磁盘数上限
    - 如果这台服务器挂了、那么所有的文件都不能访问了
    - 这个性能会被限制在单个文件和单个网络带宽上
20. 进一步扩张机器数（重新设计文件层）： L1：Distributed block layer
    - 把 block_id 延申为<mac_id,block_id>
    - 一个 client 如何知道哪台机器有 free block?
        - 第一种方法，随机直到找到
        - 用一台 master server 去记录哪台机器有 free block，然后所有的 分配和释放都通过 master server 进行
21. L2：File Layer 不用改

22. L3： distributed inode number Layer
    - 把 inode table 存在 master 上面
23. L4： file name Layer 也不用改
24. L5： path name Layer 也不用改
25. 这个 naive design 目前还有问题（只能解决 capacity 的问题）
    - 比如性能：现在需要多个 RTT 找
    - 可靠性：如果挂了、还是访问不了
    - 正确性：如果发送异常，会对整个系统产生影响
26. 解决方法：
    - 可以用 data replication 解决 performance 的问题（不过会影响正确性）
27. 例子：GFS（Google File System）
    - 设计初衷：传统的文件系统没法满足业务了
    - 设计场景：文件很大、而且很多文件被 appended，随机写的情况很少
    - 要解决：scalability，Fault-tolerant，performance
28. Interface（其中要关注 append，因为保证 append 的一致性比 write 简单）
29. 文件系统就是一个大的 byte array 然后把文件分成很多的 chunk
30. 每个 chunk 在不同的 chunk server 上（备份）（chunks size 64MB 很大）
31. chunk 很大、减少和 master 的通信次数。Chunk size = 64 MB (default) 32-bit checksum with each chunk
32. chunk 变大、也能减少 tcp 连接的连接数（重新建立 tcp 连接很耗时，每个 linux 系统对支持的 tcp 连接的个数有上限限制）
33. chunk 变大、master 需要存储的文件的元数据就会变少，就能存更多文件 Master can store all metadata in memory
34. 总结：GFS 和 naive 的对比
    - Master stores current locations of chunks (blocks)
    - Data are stored in large chunks
    - Chunks are replicated for fault tolerance & high performance
35. 为什么 GFS 只用一个 master?
    - 简单，在大部分时间够用
    - All metadata stored in master’s memory
36. GFS 用了 GFS adopts a relaxed consistency model 来保证写一致
37. name-to-chunk mapping 是存在 memory 又持久化在 an operation log 中的
    - Name-to-chunk maps (e.g., using an in-memory tree)
    - Stored in memory
    - Also persist in an operation log on the disk (talk about in later chapters)
38. GFS 没有目录结构
39. Stores chunk ID-to-chunk location maps in memory (and no need for log)
    - This is queried from all the chunkservers at startup
    - Can keep up-to-date: master controls all the management

- Benefits: simpler for consistency management

07 Google file system & from file system to key-value store

1. client 如何和 GFS 交互（另一种模型 Client-GFS interaction model ）
   - GFS client code linked into each app
   - Interacts with master for metadata-related ops
   - Interacts directly with chunkservers for data
   - 就可以让 master 不是一个瓶颈了
   - no caching data at client nor server(Why? Large files offer little opportunity for caching , 因为数据很大、所以 cache 的机会不大)Except for the system buffer cache
   - cache metadata at client（比如说 location of a file's chunks）
2. 读的情况类似，但是写不一样。
3. 写的时候会带来不一致的问题、一个读没有办法读到写（读了不同的 chunk？）
4. GFS 的解决结果最终实现了 chunk 最终内容都是一致的、但是读不一定保证读到最新的（relaxed consistency model）
5. master 会判断哪个 chunkserver 是 primary 的
   - primary can request extensions of the lease 因为可能 primary 会挂
6. write in GFS
   - 阶段 1：传输数据，通过链传递的方式在 chunkserver 之间传递数据（数据流）
   - 阶段 2 ： 写入数据，写入数据到 chunkserver，先写 primary，然后写 secondary（广播）（控制流）
   - data flow makes use of the network bandwidth, the single primary server will not be the bottolneck of transmitting data
   - control flow ensures the consistency of the chunk among all replicas
7. 判断 chunk 的复制是不是旧的，通过 chunk version number
8. 并发写的问题在 GFS 中问题并不大、因为多数文件是 append 的（并发写不会被覆盖掉）
9. HDFS : Hadoop Distributed FS(架构和 GFS 一样)
10. NFS cant scale，is not fault-tolerant，is not high performance
11. Key-value store 的 workload 和 file system 的 workload 有不同，kv 更多是小数据、file system 更多是大数据
12. naive 用文件系统存储 kv store 的问题
    - 会造成多余的系统调用
    - 空间利用率不高
13. 为什么 Why KV builds upon the file system?
    - We still need a system to interact with the disk hardware!
14. Idea: using one or few file to store key-value store data
    - Different key-values can pack into the same disk block
    - Reduce system call overhead: open file once, serve all the requests then
15. How to implement the update? Two strategies:

    - directly modify the file content
    - append the new updates to the last of the file
16. - cpu <-> memory : 100ns
    - memory <-> disk : 10ms
17. - HDD performance (for reference)
    - Sequential: 100 – 200 MB/s
    - Random: 0.5 – 2 MB
    - SSD performance (fore reference)
    - Sequential: 2,000 – 3,500 MB
    - Random: 200 – 300 MB
18. update 和 insertion 变成对日志里的 append 顺序写（因为随机写会很慢）
19. delete 使用 null entry、然后定期清理
20. 所以叫 log-structured file, 是一种 append-only 的
21. 这里需要使用 index 的目的主要是为了加速查找
22. index：使用 offset 来记录
23. 但是存在内存里的 kv 和磁盘里的 kv 是不一样的
24. cuckoo hashing 避免了 naive hash 的 list 过长的问题，保证了 get 在 O(1)时间内完成，但是 put 的时候会慢
25. 解决 cuckoo hash 不够的方法：不够的时候扩充或者 rehash，循环踢
26. 解决这个文件永远增长的问题：compaction
    - 或者 compaction with segmentation（让 compact 变快）（Break a file into fixed-sized segments 固定文件大小）
    - 然后再 merge，否则一个文件的浪费的空间就会变多（因为 compaction 让一个文件变小了）
27. 为了做 operation efficiency，会有 Hash Index (in-memory & on disk)，在不同条件下结合 log 会有不同的性能特性
28. 为了支持范围查找、会引入 B（+）tree
    - 每个节点都是固定大小的
    - 数据存在叶子节点
    - 支持范围查找（因为有序）
    - 不需要 log-structured，因为 B（+）tree 本身就是有序的，而且可以把值存在叶子节点里面
    - 问题：对于 insertion-intensive workload，B（+）tree 性能不够好，因为会有很多的随机 io
    - 对于小的数据、index 的大小可能会和数据一样大，也是一个问题
    - Get(K), Insert(K,V) and Update(K,V) are slow
    - O(log(n)) random disk accesses
    - Typically, is small, since B-Tree indexes uses large node size. But is still a killer for performance
29. SSTables（sorted string table）：把数据按照 key 排序，然后存到磁盘上
    - 优点 1：找 key 很方便（二分查找）。只需要 sparse index （不需要 hash index）（sparse index：只存一部分 key，比如最大最小的 key）

    - 优点 2：支持范围查找
    - 优点 3：merge 的时候很方便（因为是有序的，和归并排序类似）
30. 怎么保证 SSTable 是有序的呢?
    - 用一个 memtable，把数据先存到 memtable 里，然后再存到 SSTable 里
    - 但是 memtable 会有一个大小限制，如果超过了，就会存到 SSTable 里
31. Reads：先读 memtable，然后读 SSTable，如果 memtable 里没有，就读 SSTable，如果 SSTable 里没有，就读下一个 SSTable
32. 对于单一层的 SSTable，仍然会有缺陷：
    - 找旧的数据会很慢
    - deduplication 会很慢（deduplication：去重）
    - 范围查找还是比较低效
33. 所以引入了 SSTable Hierarchy
    - 除了 L0，其他的都是 compacted SSTable，都是有序的
    - insert 满了后，会和下一层的 SSTable merge
    - Hierarchy can speed up old value lookup
    - 找旧值变快是因为 Each layer has only one file that store the key (except L0)
    - So we can search only one file per-layer (not per-file search)
    - Hierarchy can speed up range query
    - Store a [min, max] per file ，然后用二分查找做每一层的范围查询
34. 如何处理 crash?
    - 再维护一个独立的 log、记录 Memtable 的修改
    - Before inserting to the MemTable, adding the KV to the log first (also a sequential write); reply if the log is successful
    - 重启的时候，会把 log 里的内容 replay 一遍
35. LSM good when
    - massive dataset
    - rapid updates
    - fast single key lookups for recent data
36. LSM 并不完美
    - 因为写的时候会触发 compaction，会慢（Write stall ）（只能用 ssd 来减缓，或者加速合并的逻辑）
    - 查不存在的 key 会很慢（但是可以用 bloom filter 来解决）

08 Consistency model (linearzability)

1. KVS 应该怎么部署?
    - 方法 1：保存 KVS 在一个 centralized server 上，All operations: execute an RPC at the server
    - 但是很慢，因为会有很多的网络延迟
    - What are the drawbacks?
    - Inefficiency! Must wait for server ACKs

    - Cannot work with offline!
2. 方法 2：centralized server + at each device
    - 是很多聊天软件的做法
    - Naïve solution
    - Read: return the latest copy on the local KVS
    - Write: update the local KVS, sync with other KVS, then return to client
3. 对于方法 2 中 sync 同步的 naive 方法：
    - 问题 1：不高效（对于 write）（read 还行）（就是写的延迟会很高）
    - 问题 2：不能容忍网络连接性断连 Under network disconnection, the sender will be blocked
    - 改进方法 3：写的时候不等待，直接返回，在后台同步（sync with the others in background & return）
4. 方法 3 会高效很多、但是会有这个数据更新和丢失的问题（ppt 例子）
5. 为了解决这个问题，引入了 consistency model（规定什么情况可以出现、什么情况不能出现）
    - How a data storage system behavior under concurrency, distribution and failure
    - E.g., in GFS, the consistency model is that all the chunk will eventually be the same
    - A strong consistency model will give more precise on what can happen & what cannot
6. 有强和弱的一致性模型，但不一定存在对的和错的一致性模型（因为不同的应用场景需要不同的一致性模型，易于开发和性能的 balance）
    - Note all concurrent execution (exe.) can be deduced to a serial execution
7. 主要关注强的一致性模型
8. 强的一致性模型，符合单线程的读写模型
    - 例子：所有操作都是原子的，所有操作都是线性的
    - 即使一个并发例子是逆时序的，也可能转化成线性的模型（ppt 例子）
9. 最强一致性：strict consistency
    - 根据 global issuing order，所有的操作都是线性的
    - 就是假设有一个全局的统一的时钟、结果完全相当于在单机上面执行的结果
    - 好处就是和单线程一样完全串行
    - 坏处就是难实现，实现这个模型会有挑战、因为如果 p0 先发送请求、p1 后发送请求，但是 p1 先到达 server，那么 server 会先处理 p1 的请求，这样就不是 strict consistency 了
10. 退而求其次：有了 sequential consistency
    - 保证了对于单台设备、单个 process，所有的操作都是线性的
    - 但是不保证 global issuing order
    - 只要保证 process 内的操作是线性的就可以了
    - 但是会有问题、就是先更新 x、但是在另外的设备上没法读到这个更新的值（因为读的时候是读到了旧的值）
11. 所以还有了 linearizability
    - 即可以根据开始-结束时间进行排序
    - 比 sequential 多了一条规则：即如果在全局时钟下、一个操作在另一个操作发出之前完成，那么后续的操作一定要能看到这个写完的状态

- 比 sequential 强
- 实际中多使用 linearizability，因为避免读不到写完的内容

12. 所以后面开始讲 linearizability 的实现
    - local property（不考）
13. 下面都是讲如何去实现 linearizability
14. 最简单的实现方式：通过 centralized KVS
    - RPC 有了返回结果即认为完成了
    - 但是(More suitable for the chat app)
      Each device has a replicated KVS on its local machine
15. 有多个 KVS 的时候，方法一：Primary-backup model
    - 一个是 primary，其他的是 backup
    - 但是有问题：读的时候会有额外的 RTT（round trip time）和 primary 通信。读的时候不能读 backup，要读 primary（ppt 例子）
    - 写的时候还需要和 backup 通信。
    - scalability 会由于 primary 的问题而成为瓶颈
    - 解决 primary 的瓶颈：通过 partitioning（让不同的 object 有不同的 primary）
16. Impl. Primary-backup model
    - Each read and write must be handled by a designated device
    - Write must wait for all the replicas to acknowledge
17. 方法 2：使用 seq number 来保证 linearizability
    - 延迟更新、直到前面的操作都做完了
    - ppt 上的那个例子是一个范例、就是说能不能做一个优化、不在 primary 上读、这个反例就不符合 linearizability，所以都需要在 primary 上读
    - 如果这样的话、就会有一个环（读到一个中间状态）
18. what are the drawbacks of primary-backup?
    - Performance issues
    - Fallback to the centralized KVS case for non-primary devices
    - E.g., Extra roundtrip, Primary becomes the bottleneck, etc.
    - Reliability issues
    - If primary crashes, others cannot work
    - 解决方法、在不同的 KVS 上面指定不同的内容作为不同的 primary（partitioning）

09 eventual consistency

17. 对于微信这种聊天场景、使用 eventual consistency 来保证性能的提升、而对于一致性的要求不高
    - 因为 linearizability Not practical for our chat applications
    - Performance issue: adding a sentence requires syncing all my devices
    - Availability issue: what if some device (e.g., iPad) is offline?
    - 就是很慢、然后难以提供 fault tolerance
18. All various read/write rule implementations for a better performance

    - Read: return the latest local copies of the data
    - Write: write locally (and directly returns), propagate the writes to all the servers in background (e.g., upon sync), which is the optimal
    - 这是一个比较 weak 的 model
19. GFS 用了 primary 去做 eventual，但是这个只适用于数据中心的场景，需要一个新的适合 chat app 的一致性模型
20. 如果采用向最近的 server 进行写的话、如果同时对不同的 server 有两个写操作、那么会有 write-write conflict
21. eventual consistency 认为 conflict write 是比较少见的、所以就先写、然后再做一致化，能提升性能
    - 相当于对于写冲突，这个是 optimistic 的、而 linearizability 是 pessimistic 的（因为 linearizability 认为写冲突是常见的）
    - 最终的目标都是要让最后的 data 是一致的
22. eventual consistency 还是有两个需要保证的
    - 写写冲突
    - 因果性 causality
23. 如果要做一致化、那么这个类似 chat 的就不能做成 KVS 中那些简单的键值的覆盖、而应该做一个 update 的 function
    - 在这个场景下、应该是类似于 append 的语义
    - 同时、受到了一个 function 之后、不能直接 update、需要先写到 log 里面（log 可能是不一样的、但是没有关系，因为还没有到数据库里面）（可以先 sync log，log 一样之后再 apply）
    - 应该怎么排序呢?
    - 然后排序 log 的 entry、然后再写到数据库里面（怎么排序：还是根据时间戳）（<time T, node ID>，中、还是先根据 time 比较，如果 time 一样、就根据 node id 比较）
    - 一个问题是：什么时候去 apply 呢？如果同步间隔就的话、一致性差异就大。所以可以先 apply，然后 sync log 之后 rollback
    - 但是这个 replay 的 sync 会比较慢、而且会导致比较大的 log 大小
    - Question#1: can we use <node ID, time T> as the ID?
    - Sort the updates: no problem
    - User-experience (or causality preserviung): not so good. For example, I will always see my iphone's data before my Ipad's data, even I posted the sentence earlier
24. 如何选择这个独特的 ID 是比较重要的
    - <time T, node ID>如果用这个的话、还是会依靠于这个 nodes 的物理时钟，不同的 server 的物理时钟也是不同步的，比如 Srv1 went first in wall-clock time with <10, Srv1>，Srv2 could still generate update ID <9, Srv2>
    - 这种不同步的物理时钟、甚至会导致不同的 server 之间会 crash（会 rerun delete before add）
25. causality 因果性 causal ordering
    - 比如在一台机器上面、X 先、Y 后、就是 X->Y
    - 或者在不同的机器上、一个操作是由于另外一个操作发生的 X->Y（X 加、Y 删）
26. 所以一种方法是 X 接受到时间戳之后、更新本地的时间戳到更大的时间戳

27. 解决方法 Lamport Clock
    - Lamport clock: a logical clock used to assign timestamps to events at each node
    - 每个 server 维护一个 time T
    - 随着真实时间的增加，T 也会增加（比如 1s 增加一个）
    - 如果看到了一个其他 server 的时间戳 T'，把本地的时间戳和 T' 比较，令 T = max(T, T' + 1)
    - 这就解决了因果性的问题
28. Can lamport clock give a total order to all the events?
    - i.e., given e1 & e2, T(e1) < T(e2) or T(e1) > T(e2)
    - No. There are ties （可能会有相同的情况）
    - Yet, we can trivially break the tie by adding an additional order:
    - E.g., <Logical time, node ID> （所以也可以引入 nodeid 加入排序）
    - The total ordering preserves the causality （这个排序就能保证因果性）
    - Why total ordering is ideal?
    - If replicas apply writes according to total ordering, then replicas converge
29. 扩展：实现 partial order： vector clock （缺陷：会比较大）
    - Each entry corresponds to a local lamport clock on the server （每个条目代表了每个 server 上的 lamport clock 的值
    - Increments local T as real time passes, e.g., one second per second
    - Clocks are represented as a vector (one entry per server)
    - 然后做类似的更新，如果看到了一个其他服务器的更大的 timestamp 的操作、就更新 Ti = Max(Ti, Ti'+1)
30. In comparison:
    - Lamport clock: total order
    - Vector clock: partial order
31. For many scenarios, Lamport clock is sufficient
    - Sufficient to order events （足够去排序事件了）
    - Save space for storing the clock（节省空间）
32. 我们需要对 log 进行进一步优化（truncate log）
    - 因为每个 write 都是立即 write，所以是 unstable 和 tentative 的 （需要做区分）
    - 为了减少 log 大小
    - 为了减少每次回滚的开销
    - 一开始的方法会回滚所有的记录、但是会浪费时间
    - 我们期望能只回滚不 stable 的记录，就是回滚 tentative 的记录
33. 方法 1：可以做一个 De-centralized 的方法
    - 把某条 log 的 entry 作为一个 checkpoint，如果所有的 server 的时间戳都大于这个时间戳，那么就可以把这个 log entry 设为 stable
    - An update (W) is stable iff no entries will have a lamport timestamp < W
34. 但是有一个问题 Problem: If any node is offline, the stable portion of all logs stops growing，如果有设备离线、还是会有很多次回滚，所以引入中心化方法

35. 方法 2：centralized approach
    - 选一个服务器作为 primary
    - 引入<CSN, local-TS, SrvID>的 timestamp，对于每次写操作、都有一个 CSN
    - CSN: commit sequence number
    - CSNs are exchanged between servers
    - CSNs define a total order for committed update
    - Advantage: as long as the primary is up, writes can be committed and stabilized
    - 这种方法可能不能保证所有情况下的因果关系、需要做额外的处理来解决这个问题、比如一次拿很多个的操作?
36. 可能会出现一个问题、就是一个 local-ts 大的 server 会先把这个 log 同步到 primary、然后 local-ts 小的会在后面向 primary 做同步，然后最后同步到另一台机器时、会做 reorder updates、导致用户看到的更新顺序不一致
37. 意思是，一旦某些事务已经提交并且被记录下来，通过 CSN 可以确认它们的状态，之前的日志条目就不再需要保留。Result: No need to keep years of log data
38. lamport clock 会考
39. If TS(event #1) < TS(event #2), what does it say about event #1 (create on node #1) and event #2 (created on node #2)?
    - Event #1 occurred at a physical time earlier than event #2 NO
    - Node #1 must have communicated with Node #2 NO （可以是任意的时钟）
    - If event #1 has been synced to node #2, then event #1 must have occurred at a physical time earlier than event #2 NO （就是在同步之前、这个 node2 的 lamport clock 已经比 node1 的大了）
40. Read may return （可能会有读的问题 eventual consistency）
    - Tentative data
    - Outdated data
    - Trade read/write consistency for high performance
41. 这个 consistency anomalies（异常现象）重要嘛?
    - 会根据发生的频率和重要性来决定
    - 100 万次读可能会有一次
    - 对于 chat app 来说、可能不是很重要

10 Consistency under crash: All-or-nothing atomicity

1. 保证事务 all or nothing 的性质
    - 因为 crash 会导致数据的原子性的问题
2. 原始方法：shadow copy，先复制一份 bank_temp、然后再修改，改完再改名字到原来的名字
    - 原来的文件永远是正确的。
    - 如果 rename 的时候挂了、Two names point to fnew's inode, but refcount is 1， which reference is the correct one?（rename 的时候要改 3 个 data blocks）
    - Naïve solution
    - Ask application to remove the unnecessary inodes

- i.e., the application knows that "bank_temp" should be removed
- Typically, not a good idea
- Problem #1. The filesystem consistency depends on the application. What if the application is buggy or malicious?
- Problem #2. What about more complex applications?

3. 所以有了一个改进的方法：journaling
   - 应用的恢复策略应该和文件系统的恢复策略解耦
   - 通过 append 日志来记录操作 （但是貌似是 sector、不是文件）（而且这里的日志貌似是记录文件系统操作的日志）
   - 先写日志，然后再写数据
   - 如果写日志的时候挂了，可以根据日志里的信息做回滚或者恢复
   - 应该 naive 的方法是写在一个 sector 里面
4. Observation: （journaling 的改进，可以只保护关键的 metadata 的写）
   - Not everything in file system has equal importance
   - Usually, the metadata is more important
     E.g., which blocks belong to the inode
   - Mitigation:
   - Only protect metadata via Journaling
     Data is written only once
   - What if data is also important?
   - Ext4 options: data=journal/ordered/writeback
     Application can also handle the write itself, e.g., wait for fsync to flush the data synchronously before proceed to the next
5. journaling 的缺点
   - 数据要写两次，一次是在 journal 里面、一次是在原位置里
   - 当文件很大的时候会有问题
6. crassh during commit journal?
   - 假设在写磁盘的一个 sector 的时候、是 all or nothing 的、那么就不会有问题
   - 但是当 log data 比 sector size 大的时候，就会有问题（可以用多个 sector 来存 log，动态大小，现在的 log 是在文件里、就可以解决这个大小的问题、抽象层上再加一层抽象）
7. shadow copy 的缺点
   - 当多个 client 共享同一个文件的时候，没法做并行（需要强制安排顺序）
   - 很难对多个文件做 copy （会有多级文件和目录的情况，需要对所有的子目录都 copy）
   - 当文件很大的时候，会耗时
8. 所以有了 Logging：
   - Key idea： 和 journaling 很像，Avoid updating the disk states until we can recovery it after failure
   - 分为 log file 和 log entries
   - Log file: a file only contains the updated results
   - Log entries: contain the updated values of an atomic unit (e.g., transfer)

    - 有了 transaction 的概念，需要保证原子性的操作就是 transaction（begin，commit，abort）
    - Before we write the disk, write the log to the disk synchronously
    - 通过 append 日志来记录操作（这里是更新文件）
    - 如何确保一个 log entry 是完整的？（通过一个特殊的标记 checksum（算一遍或者特殊的约定））
    - After the logging succeed, we can update the disk states
9. 每一次操作完的时候、其实不需要直接做 fsync
    - Not necessary, because we can do the recovery
10. 如果 crash 了，可以再执行一遍 log，然后再执行 commit （redo log）
    - Travel from start to end
    - Re-apply the updates recorded in a complete log entry
11. 为什么 commit point 重要？需要确保 all-or-nothing atomicity
12. redo-only 的好处
    - commit 的速度块、只需要 append 操作到日志
13. 但是 redo-only 也有缺点
    - 会浪费磁盘 IO 的时间，就是磁盘很多时候都是空闲的，all disk operations must happen at the commit point
    - 内存可能会不够用（因为直到这个事务提交、都不会落盘）
    - 这个 log file 会不断地变大
14. 为了解决这个问题，引入了 undo-redo logging
    - 因为 We allow the transaction directly writing uncommitted values to the disk
    - Before the commit point to free-up memory space & utilize disk I/O
    - 记录了所有 update actions
    - 回滚到 old state
15. redo-only 和 undo-redo 的区别，Log entry vs. log record
    - redo-only 只 append 了 log entry，Containing all the updates of the transaction
    - undo-redo 会 append log records（会出现多个线程的 operation 记录交织的情况）Containing the updates of a single operation
    - undo-redo 会 At commit point, append a commit record to the log last
    - 相当于这个 redo-only 是一个一个事务整个地写 log 的、undo-redo 是会 interleave 的
16. log record 的格式
    - transaction ID
    - operation ID
    - pointer to the previous record in the transaction
    - (values)
17. 根据 log record 进行恢复
    - 从最后一个 log record 开始
    - Travel from end to start

- Mark all transaction's log record w/o CMT log and append ABORT log
- UNDO ABORT logs from end to start
- REDO CMT logs from start to end

11 Before-or-after atomicity and Serializability

1. 为了防止这个 log 不断地变大、需要引入一个 checkpoint 机制，即定期地将 log 中的内容进行决定、决定哪些可以扔掉、然后扔掉可以扔掉的
   - Checkpoint: Determining which parts of the log can be discarded, then discarded them
2. naive 的方法：全部跑一遍 recovery，但是太慢了、
   - For redo logging, we only need to flush the page caches so we can discard all the logs of committed TXs
   - For undo logging, we only need to wait for TXs to finish to discard all its log entries
3. 做 basic checkpoint 的方法
   - 等到当前没有 transaction 在运行的时候，就可以做 checkpoint 了
   - 然后 flush page cache，然后把所有的 log 都写到 disk 上
   - 然后把 logs 都扔掉
   - 不过会有问题：就是一个 transaction 运行了很长时间，会没法做 checkpoint
4. 改进的 checkpoint 方法
   - 等到当前没有任何 action 操作进行的时候（如果正在写的话、会有正确性的问题）（注意这里才是 action、不是 transaction）
   - 写一个 CKPT 记号到 log 中
     - 包含了当前所有正在运行的 transaction 的 ID 和他们的 log（就是运行这个 transaction 的 log 在 CKPT 里面 ）
   - 然后 flush 一下 page cache
   - 然后把所有 log 都删了、除了 CKPT 中正在运行的 transaction 的 log
5. 对于 ppt 上的例子
   - T1 就不需要做任何事情了
   - T2 需要做 redo、从 checkpoint 开始
   - T3 需要做 redo、从它自己的 log entry 开始
   - T4 因为没有 commit，所以需要做 undo、从它自己的 log entry 开始（和 CKP 也要做）
   - T5 因为没有 commit，所以需要做 undo、从它自己的 log entry 开始
6. Question:
   - Which one is faster during execution? Redo-only logging
   - Which one is faster during recovery? Redo-only logging
7. Redo-only logging 在正常执行的时候的性能更好（下面就是回答了上面的两个问题）
   - Less disk operations compared with undo-redo logging
   - Only need one scan of the entire log file
8. Redo-only logging is typically preferred except for TXs with large in-memory states （除了对于大的事务，一般都用 redo-only log、因为可以自动保证原子性）
9. undo-only logging 没有那么常用、因为性能不好、除了恢复的时候

10. Systematic methods to support all-or-nothing atomicity
    - Shadow copy (Single-file updates)
    - Journaling (Filesystem API)
11. Logging (General-purpose approaches)
    - REDO-only logging (commit logging)
    - UNDO-REDO logging (write-ahead logging)
    - UNDO-only logging
12. Logging 也是一种保证 durability 的方法
    - Durability: Committed Action’s data on durable storage
13. 下面开始讲 before-or-after atomicity
14. 计算机需要将实际上并行进行的操作、整理成一个逻辑上的串行操作
15. 因为这个操作会有 race condition，然后这个 race condition 是很难控制的
16. before-or-after 和 linearizability 是不同的
17. Before or after atomicity 定义：
    - Concurrent actions have the before-or-after property if their effect from the point of view of their invokers is as if the actions occurred either completely before or completely after one another （并发的操作有 before-or-after 的性质、如果他们的效果、从调用者的角度来看、就好像这些操作要么完全在前面、要么完全在后面）
18. before-or-after 要解决 race condition 的问题， need a group of reads/writes to be atomic
    - E.g., cannot see/overwrites the intermediate states of a concurrent action
19. 对于 before-or-after 的实现、需要用到锁
    - naive 的实现是对每个事务都上锁，共用一把大锁
20. 如果用全局的大锁、那么每次操作都要拿锁和释放锁、性能会很差（粒度太粗了）
21. 可以做 fine-grained locking，比如做一个 lock table、对数组里的每个数据项都有一个锁
22. 但是 ppt 上的有一个例子显示了、就是在多个函数中的时候、用这个 fine-grained locking 还是会出现 race condition 的问题
23. 但是 fine-grained lock 对于每个数字都有一个锁、太浪费内存了
24. 所以引入了一个 two-phase locking protocol
    - 2PL can guarantee before or after atomicity with serializability（可串行化）
    - Run actions T1, T2, .., TN concurrently, and have it "appears" as if they ran sequentially (we will prove it later)
    - 就是基于 fine-grained locking 的实现、需要操作哪些数据的时候就获取锁、但是只有当所有的操作都执行完之后、才会释放锁
25. 会有情况是最终结果是对的、但是会有读写冲突之类的
26. 什么是线性的操作顺序？貌似和另外一门课上的概念一样
27. 有时候最终结果是对的、但是并不是完全线性的
28. 所以有了不同的 serializability 的分类
    - final state serializability
        - A schedule is final-state serializable if its final written state is equivalent to that of some serial schedule

    - conflict serializability （最广泛使用的）（条件最强）
    - view serializability
29. Two operations conflict if: （读写冲突、写读冲突、写写冲突）
    - they operate on the same data object, and
    - at least one of them is write, and
    - they belong to different transactions
30. conflict serializability:
    - A schedule is conflict serializable if the order of its conflicts (the order in which the conflicting operations occur) is the same as the order of conflicts in some sequential schedule (可以从一个串行的 schedule 中找到一个和当前 schedule 的冲突顺序一样的 schedule)
31. 对于存在冲突的并发事务调度 S'，在保证冲突操作的次序不变的前提下，通过仅交换不冲突的操作而转化为等价的串行调度 S 时，称调度 S'和调度 S 冲突等价，调度 S'为冲突可串行化调度。
32. PPT 上有 conflict 的例子
    - 这样就能构建一个 conflict graph
    - 如果这个 graph 是 acyclic 的、那么就是 conflict serializable 的
    - （按照执行顺序、可以从 T1 到 T2 画边，或者 T2 到 T1 画边）
    - 那就只需要检查这个图就可以了（如果有环、就不是 conflict serializable 的）
33. 但是又有一个例子、就是结果可以符合线性的，但是上面有环、引入了 view serializability.不过貌似这个第一个例子是 final-state serializabile 的，还有一个是 view serializable 的（这个操作会被另一个线程擦除）
34. view serializability:
    - A schedule is view serializable if the final written state as well as intermediate reads are the same as in some serial schedule
    - ppt 上也有详细定义
35. 视图可串行化调度包含的调度范围比冲突可串行化调度更广，但是判断难度也更大
36. 冲突可串行化调度一定是视图可串行化，但视图可串行化调度不一定是冲突可串行化
37. 一个调度是视图可串行化，但不是冲突可串行化，一般由盲写造成
38. final-state > view > conflict （包含）
39. 2pl 是能很简单实现 conflict serializability 的
    - 即证明这个操作没有环，用反证法，如果有环
    - 因为 T1 必须要在执行完之后再放锁
40. 但是有时候 2pl 也不能保证实现 conflict serializability
    - 主要是因为这个 update 需要扫一遍 list
    - 会出现幻读
    - 解决方法：1. 用 predicate lock 2. 用 index lock（在 B+ tree 上加 range locks）3. 有时候忽略这个问题（性能）
    - 什么时候能保证：所有数据 conflict 都被一把锁给识别出来了。

12 Serializability, OCC & Transaction

1. 这个 2pl 会导致死锁，解决方法：

- 在一个 pre-defined order 中获取锁 （Prevention）
  - Not support general TX: TX must know the read/write sets before execution
- 用 conflict graph 来检测死锁，如果有环、就死锁了、然后可以 abort 一个 transaction（这个方法不太好，成环监测代价太大）（Detection）
- using heuriestics to pre-abort the TXs that are likely to cause deadlocks（heuriestics 即启发式算法，比如说用一个 timeout 的时间来检测死锁，超过就 abort）（Retry）
  - May have false positive, or live locks

2. 产生死锁的原因：
   - 因为 2pl 是悲观的，来避免 race condition
3. 所以能不能以乐观的方式去处理这个问题呢？即 Optimistic Concurrency Control，即 OCC
4. 乐观地去拿锁、然后 commit 的时候再检查一下
5. OCC 执行 transaction 的三个阶段：
   - 阶段 1：Concurrent local processing:
     - Read data into a read set
     - Write data into a write set
     - 写的时候，如果这个变量已经被读过了，也要写到 read set 里面
     - 读的时候、如果这个变量已经被读过了、也要先从 read set 里面读
   - 阶段 2：Validation serializability in critical section:
     - Check whether serializability is guaranteed
     - 看一看这个数据是不是被改过了，看一看 read set 里面的值和 commit 时候的值是不是一样（类似地、会有一个 ABA 的问题，即他的值是一样的、但是其实是被改过了、所以在记录数据的时候可以再记录一个版本号，比较的时候比较版本号就可以了）
   - 阶段 3：Commit or abort:
     - If validation is successful, commit
     - If validation fails, abort
     - critical section 就是里面那个监测的步骤、也需要保证 before-or-after atomicity （如何保证、第 2，3 阶段里、让这个提交的函数时候加一把大锁）（因为 The phase 2 & 3 are typically short）
6. occ validate 和 commit 的时候能不能避免 2pl 的死锁呢？
   - 可以避免死锁，把这个 readset 和 writeset 都排序一下，再用 2pl （sort 的时候需要进行排序）
7. 一个 occ validate 和 commit 的时候用 2pl 的优化：
   - 可以不用拿 read set lock，只用拿 write set lock（因为有一个监测是否改动过在）
   - 但是也是不完全等价的，所以要判断这个 d 在 read set 里面有没有被上锁，如果上锁了、就判断为 abort
8. 所以 occ 相对于 2pl 的优势：
   - occ 对于读操作不需要加锁，只有在 commit 的时候才加锁
9. OCC (in the optimal case, i.e., no abort)
   - read to read the data value
   - 1 read to validate whether the value has been changed or not (as well as locked)

- 2PL
- 1 operation to acquire the lock (typically an atomic CAS)
- 1 read to read the data value
- 1 write to release the lock
- A single CPU write is atomic, no need to do the atomic CAS

10. 锁的实现：从可以从软件或者从硬件的层面进行实现
    - spin lock 的开销很大

13 OCC, MVCC, TX & Multi-site atomicity

1. Occ 会导致 false aborts 的情况、即一个事务被 abort 了、但是其实是没有冲突的
    - 特别是对于一个长时间的多读的 transaction
    - 即 occ 的问题是活锁 livelock
    - 解决的方法是设置一个 retry limit，如果 abort 次数超过这个 limit，就 delay 其他的 transaction，让这个 transaction 先执行
2. Occ 和 2pl 没有哪个绝对更好，可以混合使用
3. hardware transactional memory(HTM) 是 cpu 的一种特性、为了并发地写（保证了 before-or-after atomicity）
    - Intel 实现了一个 restricted transactional memory(RTM)
4. 怎么用 HTM（RTM）
    - XBEGIN()
    - XEND()
    - 如果监测到了冲突，就会 abort，然后 rollback 到 XBEGIN() 的地方，也可以手动 xabort
5. RTM 的好处和坏处：
    - 好处：保证原子性、然后性能更好
    - 坏处：不能保证成功（因为本质是在硬件上实现了一个 OCC）
6. RTM 的实现：
    - 复用了 CPU cache 去存储这个 transaction 的 read set 和 write set
    - 用了 cache coherence protocol 去检查冲突
    - 但是这就导致了硬件的限制、比如说 cpu cache 的大小是有限的
    - RTM 的 read wirte sets 大小依靠很多因素（ppt 上）
    - OCC is yet another classic protocol for before-or-after atomicity Whose idea has even been adopted by hardware designers
7. RTM 也会有受限制的执行时间、因为 CPU interrupt 会随机 abort 这个 transaction，导致这个事务执行越长、abort rate 越高
    - RTM 在不同的 dataset 上面有不同的效果
8. occ 和 2pl 在长时间运行读操作的事务上的时候、性能都不好（比如说淘宝上的浏览商品）
    - OCC & 2PL are bad when TXs are long running w/ many reads
    - occ 是保守的 abort
    - OCC: abort due to read validation fails
    - 2PL: read will hold the lock and block others

9. 所以就有了 MVCC(multi-versioning concurrency control)
    - 首先让这个数据有很多个版本（把数据改成一个 List、里面有值和版本号）
    - 比如说可以让读永远在一个序列化结果的版本上读，写的时候就写到一个新的版本上
    - 解决方法：用一个 global counter，用 atomic fetch-and-add（FAA）来拿到 TX 的开始和 commit time，读和写的时候会拿着这个 counter 去读或者写对应的 snapshot
    - Using atomic fetch and add (FAA) to get at the TX's begin & commit time
    - TX Begin: use FAA to get the start time
    - TX Commit: use FAA to get the commit time
    - 也会有一个更好的方法，就是原子钟
10. MVCC 的读写的 idea
    - Read
    - Only read from a consistent snapshot at a start time
    - Write
    - Install a new version of the data instead of overwriting the existing one
    - Version ~= the commit time of the TX
    - Goal: avoid race conditions on reading a snapshot
11. 下面是 Try #1: Optimize OCC w/ MV (incomplete)
12. Acquire the start time
    - Phase 1: Concurrent local processing
    - Reads data belongs to the snapshot closest to the start time
    - Buffers writes into a write set
13. Acquire the commit time
    - Phase 2: Commit the results in critical section
    - Commits: installs the write set with the commit time
14. 好处：read 不用做 validation，Compared to the OCC, no validation is need!
15. 但是会有一种情况、就是读在写后开始的时候、可能读会读不到还没写完的数据（所以需要保证 commit 之后写的原子性）
16. 所以要保证读的 snapshot 内容是需要保证写完的，所以可以在写之前加锁
17. MVCC 保证了 read 没有 race、但是 writes 会有
    - 所以还需要做简单的 validation
    - The validation is simple:
    - During commit time, check whether another TX has installed a new snapshot after the committing TX's start time (in the write set)
    - P48 示意图
18. 这个 MVCC 会有一个 Write skew anomaly 写偏差异常的情况
    - 需要用 read validation 去解决
    - The simplest way is to validate the read-set in read-write TX
    - Essentially fallbacks to OCC for read-write TX （解决了之后、就退回了 OCC）
    - But read-only TX can still enjoy the benefits from MVCC

        - Never aborts & no validations
    - Usually being ignored in practice (Snapshot isolation)
        - The MVCC without the read validation is also called snapshot isolation (SI)
19. 做了 read validation 之后就是 MV-OCC 的
20. transaction 的概念不仅在数据库里面有用、在分布式系统里面也有用

14 Multi-site atomicity & Primary-backup replication

1. 数据库对每一个修改的版本 snapshot 都存在，然后不可能全量复制，因此 snapshot 里面也存的是 incremental 的内容
2. 复习 MVCC
    - phase 1：read 读 snapshot，write 读 buffer，每个数据会写两次、一个 write 是 cache、第二个是 commit
    - phase 2：validation
    - phase 3：commit
3. ppt 里的 write 都是写到数据库里
4. 如果我们简单地把 A-M 和 N-Z 的数据存在不同的数据库里、那么会遇到 multi-site atomicity 的问题
    - 即如果一个 server commit 了但是另外一个 abort 了
5. 第一个 Multi site 场景：high-layer tx 和 low-layer tx （compose multiple single-site TXs ）
6. 第二个 Multi site 场景：跨机器数据的事务 （a TX access data across sites ）
7. 需要保证：所有 low-layer tx 都是原子的
8. 所以有了 Two-phase commit
    - phase 1：准备/投票
        - 延迟这个 low level tx 的 commit
        - low layer 要么 abort，要么进入了 tentaively committed 的状态
        - high layer 的决定 low layer 的状态（通过投票）
    - phase 2：commitment
        - high layer 来决定 low layer 是否需要 commit 或者 abort
        - 会协调 low layer 的 tx 状态，所有人都提交了、再提交
9. 实现：high_begin 和 high_commit，遍历所有的 low layer tx，然后投票
    - 先会发一个 prepare message，然后等待所有的 low layer tx 的回复
    - 如果都回复了、就发一个 commit message，然后等待所有的 low layer tx 的回复
10. 不能直接在 low tx 里面用 commit logging，因为 The high-level transaction can abort
11. 修改：需要把 low layer 的 commit log entry 变成 tentatively commit log entry（PrePared）
    - 还会包含一个 reference 到 higher tx 的 ip，如果可以提交、就变成 commit 状态
12. 分布式下会有一个 partial 的问题
    - 就是 low level 的 tx 不知道其他的 low level tx 的是否 commit 了，只有 high level 的知道，会造成不一致性。
    - 通过 coordinator 来统一解决这个问题，即 worker 不会动、直到 coordinator 说可以动
13. 流程：为了让这个所有 low level tx 要么都提交、要么都不提交

- 首先、这个 low level 的里面的 log 会记录一个 prepared 的状态（原来是 commit）
- 然后、high level tx 要 log 一下它是否 commit 了
- 所以根据 coordinator 的 log、来向所有 low level 的 tx 发信息、等到接收到所有 low level tx 处理完 prepare 之后的返回结果之后、再在 high 上面记录一个 commit、然后再发信息给所有 low level tx，让 low level 的写 commit log 在自己的 log 里面
- phase 1 是协调 prepare、phase 2 是从 high 的 ok 了，开始 commit 开始提交

14. 出错处理：
    1. 一
    - prepare 阶段（还没 ok）
    - 如果 prepare 发过去的时候出错：timeout retry
    - 如果 prepare 的 ACK 丢了：timeout retry
    - worker failure：abort（给所有 live worker 发一个 abort message）
    2. 二
    - commit 阶段（coordinator 告诉 client ok 了）
    - 如果 commit 发过去的时候出错：timeout retry（low level server 发 tx？）
    - 如果 commit 的 ACK 丢了：timeout retry
    - worker failure：如果 worker restart、会自己重新问 coordinator，然后根据 coordinator 的 log 来决定
    3. 三
    - 如果 coordinator failure during prepare（在有 commit log 之前）：全部 abort（因为这个时候还没有 commit log、根据 ppt 上的图）
    - 如果 coordinator failure during commit：重新 commit 所有的 low level tx（因为这个时候已经有了 commit log）（重复 commit 不会影响正确性、因为如果之前有 commit 过、不做就行了）
15. 为什么 prepare 完之后就能在 high level 里面记录 commit log 了？
    - 因为这个时候所有的 low level tx 都是 prepared 的状态，所以可以 commit 了
16. 总结：
    - tx 是否被 commit/aborted 是必须被记录的
    - prepare 不需要被 log，因为可以恢复
    - 我们需要最小化 logging、这对于性能很重要
    - 怎么做 checkpoint？对 low level 很好做、但是对于 high level 需要额外操作（因为这个 worker 很可能再来问这个 commit entry log 还在不在，所以在实现的时候、high level 会在后台接受所有的 low level 的 commit 确认、然后如果一个 commit 的所有 tx 都提交了、就标记这个 commit entry log 可以被做 checkpoint 了）
17. 在 2pc 里的 2pl 和 occ
    - 2PL: each low-layer TX cannot release its lock until the high-layer TX decides to commit
    - OCC: the validation & commit phases are done by the coordinator
18. Two-phase commit allows us to achieve multi-site atomicity: transaction remains atomic even when they require communication with multiple machines

19. In two-phase commit, failures prior to the commit point can be aborted. If workers (or the coordinator) fail after the commit point, they recover into the PREPARED state, and complete the transaction
20. 问题在于 coordinator 挂了，就不可用了，所以需要 replication 保证高可用
    - 不能依靠重启去做恢复、因为太慢了
21. 2PC only guarantees consistency! in CAP（Consistency， Availability，Partition tolerance(the system continues to operate despite arbitrary message loss or failure of part of the system)）
22. replication 的好处
    - Higher throughput: replicas can serve concurrently
    - Lower latency: cache is also a form of replication
    - Maintain availability even if some replicas fail
23. 分为 optimistic replication 和 pessimistic replication
24. Optimistic Replication (e.g., eventual consistency)
    - Tolerate inconsistency, and fix things up later
    - Works well when out-of-sync replicas are acceptable
25. Pessimistic Replication (e.g., linearizability)
    - Ensure strong consistency between replicas
    - Needed when out-of-sync replicas can cause serious problems
26. Some applications may prefer not to tolerate inconsistency
    - E.g., a replicated lock server, or replicated coordinator for 2PC Better not give out the same lock twice
    - E.g., Better have a consistent decision about whether transaction commits
27. Trade-off: stronger consistency with pessimistic replication means:
    - Lower availability than what you might get with optimistic replication
    - Performance overhead for waiting syncing w/ other replicas
28. RSM (Replicated State Machine) 能 ensure single-copy consistency
    - Start with the same initial state on each server
    - Provide each replica with the same input operations, in the same order
    - Ensure all operations are deterministic （确定的、即不受随机干扰）
        - E.g., no randomness, no reading of current time, etc.
        - 并发和随机数会导致输入一样的情况下、输出不一样
29. 但是如果不做协议的话、那么可能会导致每个 replica 受到 request 的顺序不一样（因为广播速度受到距离的影响）
30. 所以可以做 RSM + primary backup replication
    - 一个 primary replica 会接受所有的 request，然后广播给所有的 backup replica
    - backup replica 会等待 primary replica 的确认、然后再执行
    - backup 也有助于状态恢复

15 RSM & PAXOS Consistency across multiple machines

1. 有时候这个有 race 或者不一致没有关系、比如商品的人气数据
2. RSMs provide single-copy consistency
    - Operations complete as if there is a single copy of the data
    - Though internally there are replicas
3. RSMs can use a primary-backup mechanism for replication
    - Using view server to ensure only one replica acts as the primary（分化出了一个 server 叫做 view server，用来告诉别人谁是 primary）
    - It can also recruit new backups after servers fail
4. Primary does important stuff
    - Ensures that it sends all updates to the backup before ACKing the coordinator
    - Chooses an ordering for all operations, so that the primary and backup agree (i.e., one writer)（相当于只有一个写者了）
    - Decides all non-deterministic values (e.g., random(), time())
5. 原来、coordinator c 会认为 S1 是 primary、S2 是 backup，如果连不到了 S1，就会把 S2 变成 primary
6. 如果发生了一个 network partition，就相当于 C2 连不到 S1 了、就会把 S2 变成 primary，S1 变成 C2 的 backup
7. 所以引入了一个 view server，来决定谁是 primary，决定完之后、在 view server 上面只需要记录一个 key-value pair，即 primary 和 backup。实际操作的时候、每次 coordinator 会先去 view server 上面问一下、然后再直接根据 view server 返回的 primary 去直接访问那个对应的 primary、不会经过 view server 进行转发、就会导致 view server 的压力很小，不会成为系统中的瓶颈。（view server 只需要听各个 server 的心跳，而且所以因此只需要引入一个 view server）
8. 然后 view server 怎么保证可用性呢？所以就需要 paxos 来维护多个 view server 之间的 kv 值是一样的
9. view server To discover failures
    - Replicas ping to the view server （replica 会给 view server 发心跳包）
    - If view server misses N pings in a row, it deems a server to be dead （对于几次 ping 失败的情况、view server 会认为这个 server 挂了）
10. Basic failure (actual worker crash):
    - Primary fails; pings cease
    - View server lets S2 know it's primary, and it handles any client requests
        - Before S2 knowing it's the primary, it will simply reject requests that come directly from the coordinator(在 s2 确定自己是 primary 之前、S2 会拒绝所有的请求)
    - View server will eventually recruit a new idle server to act as backup
11. Rules when Facing Network Partitions
    - Primary must wait for backup to accept each request
    - Non-primary must reject direct coordinator requests

        - That's what happened in the earlier failure, in the interim between the failure and S2 hearing that it was primary
    - Primary must reject forwarded requests
        - i.e., it won't accept an update from the backup
    - Primary in view i must have been primary or backup in view i-1
12. 下面是 paxos 的内容
13. 有 3 个角色：
    - proposer: 提出一个 proposal
    - acceptor: 接受一个 proposal
    - learner: 学习一个 proposal（接受上面达成一致的 proposal）
14. 任何一台机器、都可以是 proposer 又是 acceptor
15. 目标：所有的 acceptor 都接受一个 proposal 的值
16. Quorum = any majority of Acceptors（是一个多数派的概念）
17. 大致流程：
    - One proposer decides to be the leader (optional)
    - Leader proposes a value and solicits acceptance from acceptors (majority)
    - Leader announces result or try again
18. 但是最难的问题在于处理以下这些问题：
    - What if >1 proposers become leaders simultaneously?
    - What if there is a network partition?
    - What if a leader crashes in the middle of solicitation?
    - What if a leader crashes after deciding but before announcing results?
19. Paxos has rounds; each round has a unique ID (N)
20. Rounds are asynchronous
    - Time synchronization not required
    - If you are in round j and hear a message from round j+1, abort everything and move over to round j+1
    - Use timeouts; may be pessimistic（即认为不投票的人都是反对的）
21. Each round itself broken into phases
    - Phases are also asynchronous
22. Paxos in Action:
    - Phase 0：Client sends a request to a proposer（客户端发请求给 proposer）
    - Phase 1a (Prepare)：Leader creates a proposal N and send to quorum（发 prepare 请求）
        - N is greater than any previous proposal number seen by this proposer
    - Phase 1b (Prepare)： if proposal ID > any previous proposal
        - reply with the highest past proposal number and value（这时候 server 会返回一个值）
        - promise to ignore all IDs < N
        - else ignore (proposal is rejected)

    - Phase 2a (Accept)（leader）
        - Leader: if receive enough promise
        - set a value V to the proposal V, if any accepted value returned, replace V with the returned one（为什么要设一个 propose 号最新的 value 呢、不然的话这个旧值可能不是被 agree 的）
        - send accept request to quorum with the chosen value V（这时候 proposer 发 accept 请求）
        - 即如果在所有收上来的 promise 里面的 v 都是 null、那么 leader 可以随便设一个 value、如果收上来的 promise 里面有 value、那么 leader 就要设为这个 value（继承遗产）
    - Phase 2b (Accept)（Acceptor）
        - Acceptor: if the promise still holds
        - register the value V
        - send accepted message to Proposer/Learners
        - else ignore the message
    - Phase 3 (Learn) Learner: responds to Client and/or take action on the request
23. 为什么要多个 acceptors、因为一个可能会 fail
24. Why not accepts the first proposal and rejects the rest?
    - Multiple leaders result in no majority accepting
    - Leader dies
25. 注意这个 prepare 和 accept 的对象也可以是发出这个 proposal 请求的自己
26. 抽屉原理、保证不会有两个 leader 同时看到两个 majority
27. 当 Leader receives a majority <accepted, ...>的时候、就可以确定这个 value 是确定的了（只有当这个时间点的时候、才知道有 majority 了，但实际上这个值被确定是在这个时间点之前的某个时刻发生的，但是因为节点间不会一直通信导致的）（就是当 A majority of acceptors accepts ok for the same proposal 的时候）
28. What if acceptor fails after sending promise?
    - Must remember Nh
    - What if acceptor fails after receiving accept?
        - Must remember Nh and Na Va
    - What if leader fails while sending accept?
        - Propose Mn again
29. 有很多数据需要记录在 log 硬盘上、在 ppt 第 50 页上（考虑重启的时候、单个节点从失败中恢复时）
30. 就是一轮里面、可能会有多个里面的值不同、比如 9 个里面 5 个是不同的值、其余 4 个是 null、所以可能最终的值是这个 5 个值里面的任意一个、而不保证是最新的那一个
31. paxos 可能选不出来，没有终止的保证，所以会有出现 view server 对应的 primary 和 backup 的 server 不一致的情况
32. 所以 paxos 的目标和达成了就是让所有的机器都接受同一个值
    - Each machine will have the latest state or a previous version of the state

16 Multi-Paxos, Raft & NewSQL Consistency across replicas

1. consensus 的本质是固定一个唯一的 writer（paxos 中，号最高的是 writer）
2. 如果没有 learner、本质上也可以拿到 paxos 的统一后的值，就是再发一个 proposal，然后取最新的值，但是会有性能问题、因为要拿到 value 的话、需要两轮 round trick 才能拿到，所以 learner 可以加速拿值
    - n_a 记住是为了辨别哪个 writer 是最新的 writer
    - n_p 记住是为了避免重启之后把票投给多个人
    - my proposal number 是为了 write 的时候发 proposal，如果没有这个应该也可以跑，就是要再发一轮 proposal，直到成功
3. single-decree paxos 其实还不能用来给多个 view server 做同步、因为他只能 accept 一个 value。而这个 primary kv 可能会更改、所以我们需要一个 sequence 的 value
4. 简单的解决方法：multi-paxos，就是用多个 paxos instance 去维护不同的值
    - 问题：如果一个位置上面在其他的 server 上面已经有了值，所以发起一个 accept 就能自动把这个丢掉的值给恢复起来
    - 如果一次 append 没成功、就重起一个 paxos 去做这个新值的维护
    - 但是他有一个性能问题，理论上一次网络操作就期望能存这个 append 的值，但是 basic 的可能需要很多次 append 才能写一个值（会有不同的轮数来解决前面的不匹配 conflict）（而且或者可能很多 server 没拿到 majority、会导致重新投票）
    - 解决方法：选一个 leader，只有 leader 能 propose，然后这个 prepare 请求可以涉及到多个 instance（batch）（把 3 和之后的所有都 prepare）（也可以一次里面 append 多个 entries，即 prepare 2+3）
5. 简单方法的问题：每次需要 2 个 round trick

   With multiple concurrent proposers, conflicts and restarts are likely

    1. The server may conflict on the position of the new log entry to insert
       (our previous example)
    2. Even in the same Paxos instance, different proposer may conflict
       Only the proposers with the highest number can win（冲突的情况）
6. Distinguished proposer (aka. leader)
    - The only one that issues proposals （leader 是唯一发 propose 的，为了避免 proposer conflicts）
        - i.e., no proposer conflicts in the optimal case
    - Client only sends the commands to the leader （client 只发请求给 leader）
        - Decides the value position
    - Question: is single leader necessary for multi-paxos? （本质上不需要单一维护的 leader）
        - No: if two or more servers act as proposers at the same time, the protocol still works correctly
7. 但是还是会有问题，就是 log 里面有 hole 怎么办（所以需要程序自己去推导、貌似就需要 raft 了）所以 multi paxos 好像没啥实现