计算机系统工程复习

inode-based file system

- **文件系统指令：**

  Open 的返回值是fd file descriptor

  Read 的第一个参数是fd

  Fsync 的第一个参数是fd，是同步内存中所有已修改的文件数据到储存设备的。 函数：写进去的时候要同步一下，以防有数据没写进去。

  STAT 返回文件状态，包括：文件大小、创建者、创建时间、修改时间

  CHMOD 改变文件权限 755常见、777危险、400最安全，三个数字代表user、group、other的权限，r=4，w=2，x=1。

  CHOWN 改变文件的主人，改变user的id

  RENAME 重命名

  LINK 创建一个新目录项，并且增加一个链接数refcnt。

  UNLINK 删除目录项，并且减少一个链接数。如果链接数达到0并且没有任何进程打开该文件，该文件内容才被真正删除。如果在unlilnk之前没有Close，那么依旧可以访问文件内容。

  SYMLINK 符号链接（软连接）

  MKDIR 创建文件

  CHDIR cd打开文件

  CHROOT 把自己的root修改，在指定的根目录下运行指令。下载新的gcc之后要把新的build的系统放到单独的目录，CHROOT进去，使其变成唯一

  MOUNT/UNMOUNT 挂载文件系统以外的内容，比如：键盘、鼠标、usb等

- **硬连接与软连接：**

  硬连接：

  在Linux系统中，多个文件名指向同一索引节点(Inode)是正常且允许的。一般这种链接就称为硬链接。硬链接的作用之一是允许一个文件拥有多个有效路径名，这样用户就可以建立硬链接到重要的文件，以防止“误删”源数据。不过硬链接只能在同一文件系统中的文件之间进行链接，不能对**目录**进行创建。

  软连接：

  软链接就是一个普通文件，只是数据块内容有点特殊，文件用户数据块中存放的内容是另一文件的路径名的指向，通过这个方式可以快速定位到软连接所指向的源文件实体。软链接可对文件或目录创建。

  软连接的作用：

1. 便于文件的管理，比如把一个复杂路径下的文件链接到一个简单路径下方便用户访问。

2. 节省空间解决空间不足问题，某个文件文件系统空间已经用完了，但是现在必须在该文件系统下创建一个新的目录并存储大量的文件，那么可以把另一个剩余空间较多的文件系统中的目录链接到该文件系统中。
3. 删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接就变成了死链接。

文件系统层次

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
Path name layer	Organize files into naming hierarchies.	↓
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑
File layer	Organize blocks into files.	machine-oriented names
Block layer	Identify disk blocks.	↓

- **L1 Block layer**：

  输入：整数block number

  输出：block number对应的block里的内容

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block
    return device[b]
```

  旧版一个block的大小是512个字节，新的是4K个字节

  metadata存放在superblock中，每个文件系统的第一块block。

  superblock中含有：

  block的大小、inode list大大小、free block的数量、free inode的数量、一系列free block、一系列free inode 、下一个freeblock的index、下一个free inode的index

- **L2 File layer**：

  输入：inode实体、 整数索引index

  输出：该inode实体中索引为index的block的block number

```
procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]
```

输入：整数偏移量 offset 和 inode实体

输出：该inode实体中偏移量为offset的block的内容

过程：offset/blocksize得到一个index

调用index_to_block_number的函数

得到block number之后调用block_number_to_block

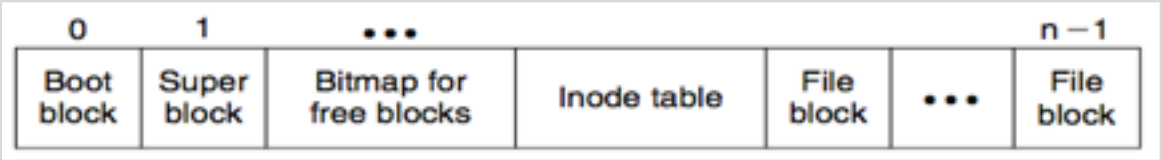
```
procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

- **L3 Inode Number layer：**

输入：整数inode number

输出：节点数为inode number的inode实体

```
procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode
    return inode_table[inode_number]
```



L1 L2 L3:

输入：整数 偏移量offset 整数 inode number

输出：节点数为inode number的inode偏移量为offset的block里面的内容

过程：inode number ➜ inode

offset/blocksize ➜ index

inode、index ➜ block number

block number ➜ block

```
1  procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
2                                                    returns block
3      inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
4      o ← offset / BLOCKSIZE
5      b ← INDEX_TO_BLOCK_NUMBER (i, o)
6      return BLOCK_NUMBER_TO_BLOCK (b)
```

- **L4 File Name layer：**

输入：文件名、目录的inode number

输出：文件的inode number

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer
    return LOOKUP (filename, dir)
```

输入：文件名、 整数dir（目录的inode number）

输出：对应该文件名的inode mumber

过程：由dir找到目录对应的inode

遍历inode的每一个block，查找是否存在该文件名

找到返回它对应的inode number，未找到返回失败

```
1    procedure LOOKUP (character string filename, integer dir) returns integer
2        block instance b
3        inode instance i ← INODE_NUMBER_TO_INODE (dir)
4        if i.type ≠ DIRECTORY then return FAILURE
5        for offset from 0 to i.size − 1 do
6            b ← INODE_NUMBER_TO_BLOCK (offset, dir)
7            if STRING_MATCH (filename, b) then
8                return INODE_NUMBER (filename, b)
9            offset ← offset + BLOCKSIZE
10       return FAILURE
```

文件名的好处：

1. 隐藏文件的元数据

- **L5 Path Name layer：**

输入：path路径名 dir（目录的 inode number）

输出：这个path对应的inode number

过程：如果是最后一层 就搜索这个层的file对应的inode number

如果不是最后一层，就层层搜索，根据上一层的path来搜索到下一层的path的位置

```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
    if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
    else
        dir ← LOOKUP (FIRST (path), dir)
        path ← REST (path)
        return PATH_TO_INODE_NUMBER (path, dir)
```

- **L6 Absolute Path Nmae Layer：**

每个user的根目录HOME

输入：输入一个path

输出：如果第一个是/，就输入path，根目录（root inode）

如果第一个不是/，就调用path，当前文件的dir的inode number

```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
    if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
    else return PATH_TO_INODE_NUMBER(path, wd)
```

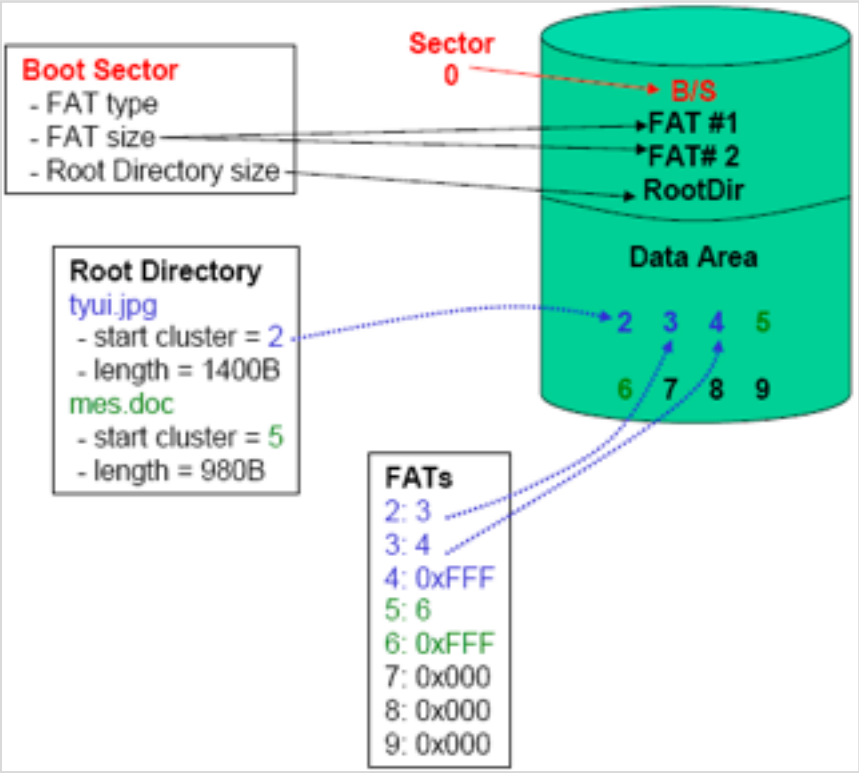
- **L7 Symbolic link layer**:
  软连接（不同文件系统之间的挂载）

FAT文件系统

用链表实现

没用的block在 FAT free list中

有一个文件名和文件编号的mapping，文件的属性保存在目录中



文件系统的API

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
    integer userid
    integer groupid
    integer mode
    integer atime
    integer mtime
    integer ctime
```

- **open**
  输入： 文件名 flags mode

过程：

文件名 ➜ inode number

若未找到inode number： 新建这个文件

inode number ➜ inode

检查用户的permission

如果有permission：把inode number加入到file table

建立fd的联系

更新last access time

输出：fd

- **read:**

输入：fd， buf ，读的数量n

过程：

fd ➜ filetable中的index

file table中的index- ➜ cursor

File table中的index ➜ inode number

m是n和文件最末尾和当前cursor间的最小值

更新last acces time

如果m=0，返回EOF

对读的每一位i： inode number ➜ block

block中的内容copy到buf中

I+block的大小或m-i

cursor+m

输出： m

- **close：**

清除fd table中的entry

在file table中减少ref cnt

如果cnt=0，在file table关闭这一reference

DISK的I/O

- **SYNC：**

SYNC是application和文件系统之间的刷新接口

SYNC确保所有的的文件都写进disk了

FLUSH是Disk driver和DIsk Cache之间的刷新借口

- **DISK的构成与接口调用：**

Disk的接口有寄存器，分别为status address data command

内部有 cpu memory 和其他芯片

版本1: 一直等待直到status不是busy

写入data 和address

写入command

等待直到昨晚做完任务

问题：轮询浪费cpu

版本2:使用interrupt而不是wait

- **interrupt：**

  操作系统提出一个request，然后处理另一个process，直到磁盘完成发送interrupt， 根绝interrupt handler进行修改

  举例：键盘，user按键的时候，键盘向处理器发送message和interrupt，下一个cycle处理器处理interrupt

  问题：

  livelock：cpu只处理大量的interrupt而无法执行用户的process

  解决：当已经出现一个interrupt时，使用一段时间polling，然后再使用interrupt

  优化：可以把很多小的interrupt合并起来，提高程序性能

- **DMA（direct memory access）：**

  传统方式：外设数据读到内部寄存器，再送到内存

  如果外设的数据比较频繁，每一次都中断会浪费大量的时间，因此采用DMA机制：

  DMA方式：外设将数据放在RAM中，然后产生中断，操作系统直接将内存中的数据传给需要数据的任务，系统减少了读取外设IO的时间开销。

  优点：减轻CPU的负担，可以运行其他程序

  减少一次数据的转移

- **设备交互的方式：**

  kernel级别的： PIO 使用in和out的命令

  user级别的： memory mapped IO （mmio）使用load 和store的命令

  举例：

  out %eax, 0x120 把eax的数据发到device 120中

  0x120是设备的name space ，不是物理内存的name space

IDE 硬盘

```
Address 0x1F0 = Data Port
Address 0x1F1 = Error
Address 0x1F2 = Sector Count
Address 0x1F3 = LBA low byte
Address 0x1F4 = LBA mid byte
Address 0x1F5 = LBA hi byte
Address 0x1F6 = 1B1D TOP4LBA: B=LBA,
D=drive
Address 0x1F7 = Command/status
```

先读取01F7等待status变成ready

在01F2 ➜ 01F6（command）中写参数：sector、logical block address、drivenumber

在01F7写read/write到command

BUS层

BUS的端口分配是不平均的，大部分接口在memory中

Process 发送指令： （memory目的地端口号，READ，进程所在的端口号）

同步数据传输：共享一个clock

异步数据传输：用ack线等线来传输

文件系统设计与优化

- **速度：**

原来block的大小为512B，只用2%的最大磁盘速度

解决方案： block大小变为1024B

由于碎片化 会造成随机访问 ➜ 文件系统使用一定时间变慢

例子： A FAST FILE SYSTEM FOR UNIX

用bitmap代替freelist

试着连续创建新的文件

剩余10%的文件space

skip-sector提高磁盘命中率

把inode放的距离文件更近一些

好处：小的文件不需要seek

可靠性增强

划分block group，每个group有自己的inode bitmap

好处：提高大小文件的局部性

坏处：小文件效率低

需要预留空间放置碎片化

问题1： 不知道文件会变成多大

问题2:由于磁盘旋转有延迟，读下一个block的时候磁盘继续旋转 ➜ miss

解决1:把文件按一定间隔正好让满足磁盘的转速

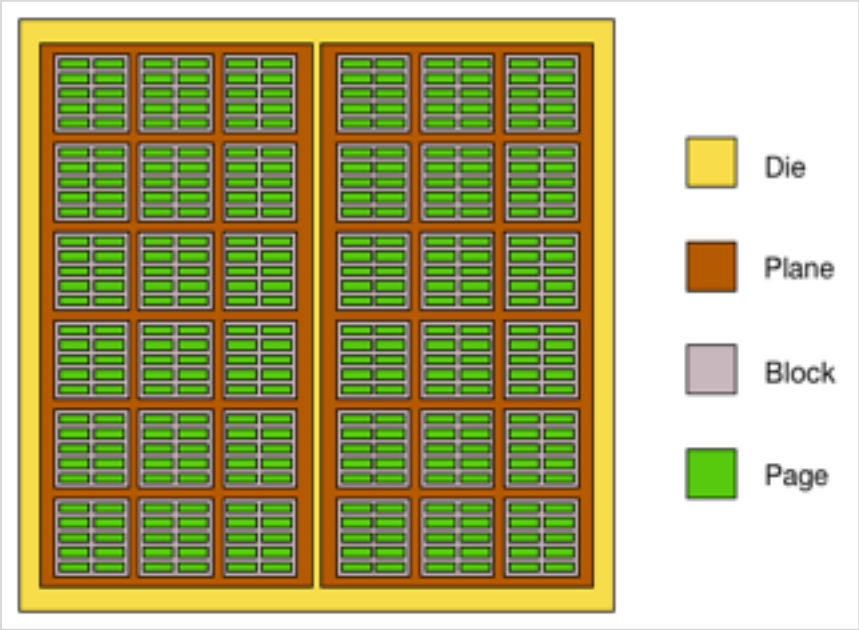
解决2:继续读下一个block

flash 与SSD

flash是介质，SSD是商品，SSD是很多flash颗粒的组合

flash disk的安排：

有1/2/4die 1/2plane n个block n个page n个cell



channel：controller可以同时交互的chip数量

写/读的单位是page 8-16KB

擦除的单位是block 4-8MB

- **读写方法：**

1. 一一映射
2. Flash translation Layer（FTL）

   在硬件里维护一张表，这个表的输入是用户给的输入，它的输出是内部维护输出。自己维护一个mapping。
3. 一直在写同一个block，为了均匀分布，改到指向第二个block。

   open channel

   硬件告诉软件自身的信息

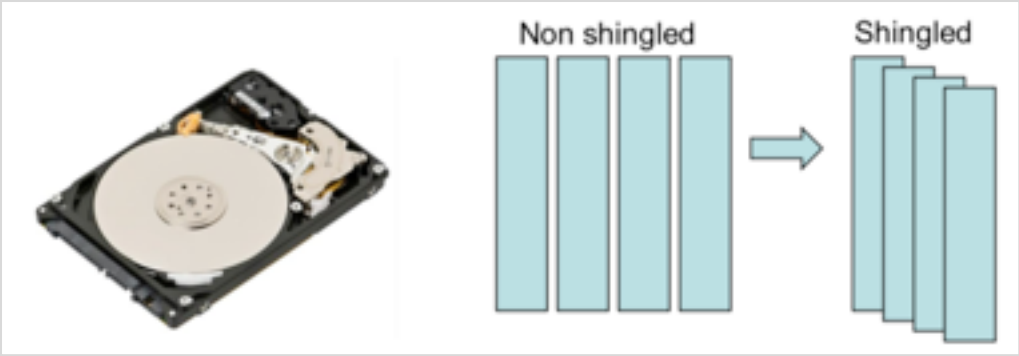
   好处：软件知道磁盘信息

   方便做数据的调度

- **flash translation layer：**

  追踪不同的sector，可以在软件也可以在硬件

- **SMR 重叠的磁性读写：**



需要按顺序写，读操作比写操作读取的面积小

- **RAM：**

1. Phase-Change Memory (PCM)：
   amorphous（无定形） high0
   crystalline（透明） low1
2. Resistive RAM (RRAM)
   用离子存信息
3. Spin-Transfer Torque RAM (STT-RAM)
   反向 high0
   桐向 low1
4. 3D-XPOINT

Crash Consistency

在更新block的数据、更新bitmap和更新ninode三者中：
如果只有一个成功了：
Db被写入： 写入却访问不到
Inode被写入： 指向了不正确的内容，且会被覆盖
bitmap被写入：再也无法使用这个bitmap
如果有两个成功了：
没写db：访问错的内容
没写inode：访问不到这个内容，不知道这个inode属于哪个
没写bitmap：会随时被覆盖

- **MTTF与availability：**

MTTF： mean time to failure
MTTR：mean time to repair
MTBF：mean time between failure
MTBF=MTTF+MTTR
availability：
3-nine： 8hour/year
5-nine：5min/year

7nine： 3sec/year

- **redundancy冗余：**

Voting： 对同一个事执行3/5次然后vote

NMR（n-modular redundancy）：

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3$$

这样的机制可以提高reliability，但MTTF却减小了：

假设一个机器的MTTF是6000，那么三台机器的MTTF是2000，2台机器的MTTF是3000，这样MTTF变成5000，比6000小了

但加入repair机制：

2个复制坏掉的概率是2/MTTF

在等待修复的时候supermodule坏掉的概率是 2MTTR/MTTF

如果修复的时间是1h，那么几率是1/3000

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}}$$

MTTF降低到原来的1/3，但是加入了repair之后变得更强

举例：如果MTTF是5 year，了爱人是10hour：

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years}$$

- **磁盘的fault tolerance：**

all_or_nothing_put：同一个磁盘分成三个区，每次写数据都是写三份，读三份 。修复repair：一旦硬盘断电了，如果前两个一样，就是前两个的值， 如果前两个不一样，那前面就应该改变。stage4会被repair成stage7.

RAID1:使用块级拆分的磁盘镜像。8块（复制了4块）。

RAID2:内存风格的纠错码组织结构，奇偶校验。（7块）。

RIAID3:位交叉的奇偶校验组织结构。（5块）。

RAID4:块交叉的奇偶校验组织结构。（5块）。

RAID5:块交叉的分布奇偶校验。（5块）。

RAID6:P+Q冗余。

- **同步的metadata更新与FSCK：**

1. 检查superblock： file system的size 要比block的数量多，如果违反这个原则就用一个copy替换。
2. 扫描所有的inode，把所有的inode里的block进行统计

扫描所有的free block，如果一个block在free里也在inode里，则把free的删除

3. 检查inode 的state
4. 检查inode的link， 扫描file system的tree，数refcnt， 根据实际的link更新refcnt
5. 如果一个inode被分配但没有目录包含它，在根目录建立一个lost and found，失物招领
6. block只能被一个inode 用，检查是否有两个inode指向一个block，进行copy
7. 检查指向out of range的block，进行remove的工作
8. 检查directory：任何目录必须有.和..两个，没有任何一个目录可以被link。 同一个目录不能有相同的文件名，如果有改名。

   缺点：太慢了

同步书写metadata的正确顺序：

create：

1. 在bitmap显示被标记了
2. 创建目录的entry

delete：

1. 清楚目录的entry
2. 在bitmap清除inode，让它变成free
3. 在bitmap清除block，变成free

- **logging 与journaling：**

  Journaling：在更新fs之前，写一个note，确保note写好了之后再更新，如果中途被打断，只需要重读note，redo一遍

  有序的journal：data与journalmetadata一起flush ➔ journal commit flush ➔ checkpoint

  无序的journal：

RPC 与NFS

- **RPC远程过程调用：**

  RPC stub：隐藏client与server交流的细节，让上层代码不变

  在一个RPC的request message中包含：

1. service ID （xid，x是transaction的省略）
2. call/reply
3. Rpc version
4. Program

在一个RPC的reply message中包含：

1. service ID （xid，x是transaction的省略）
2. call/reply
3. Accepted?
4. Success?

5. result

   RPC提供了：

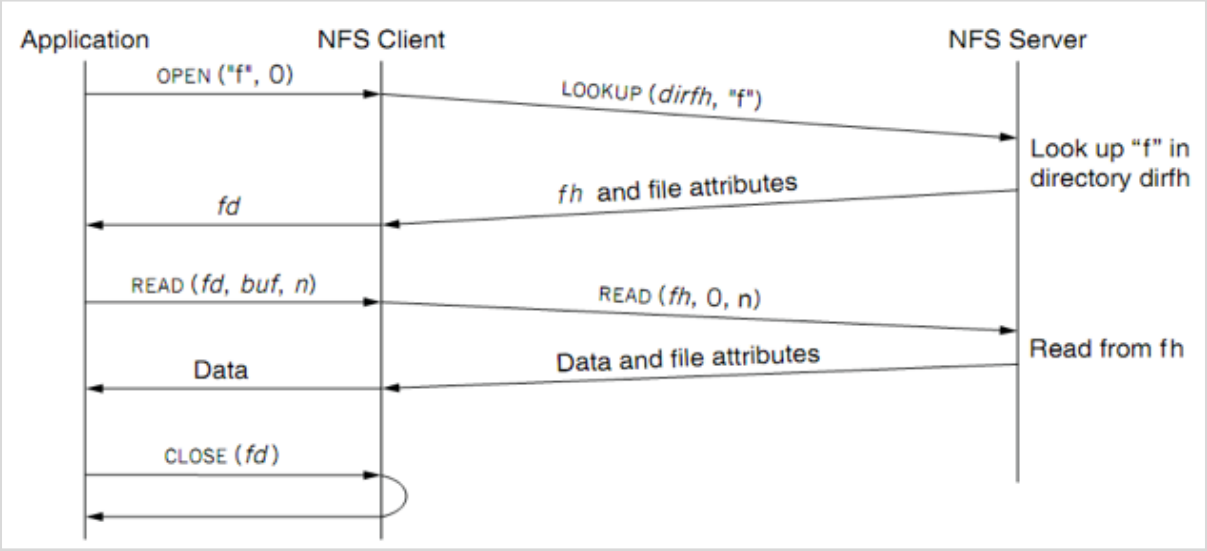
1. RPC运输数据的标准
2. 统一的marshal/unmarshal数据的库
3. 为RPC进程生成stub的generator
4. 把每个client的信息发送给server
5. 把server的repky发回去

   Server可以每一个请求建一个thread，可以有固定数量的线程池，只有一个thread

- **marshal与unmarshal：**

  把一个对象变成一系列byte并加上足够的annotation 让unmarshal能够完成

NFS（network file system）



client端的file handler：

filesystem的id

inode number 让server能找到文件所在

generation number 为了让server保持文件的consistency

传递inode number而不是path是为了防止两个client的rename等出现问题

server是无状态的，cursor在client保存

通过generation number 来确定是哪个版本的文件

- **client的cache：**

  用来存储打开过的文件的信息，来提升性能降低延迟

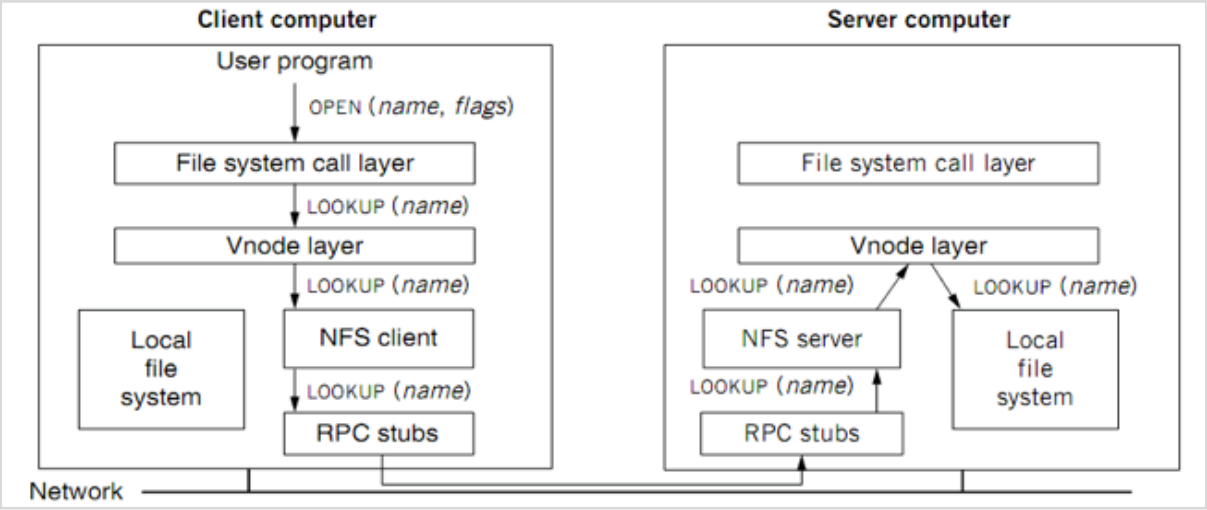
  coherence的实现：

  read/write coherence: open时用getattr获得最新的modification time

  获得最新的时间，和cache中的修改

- **vnode：**

virtual inode

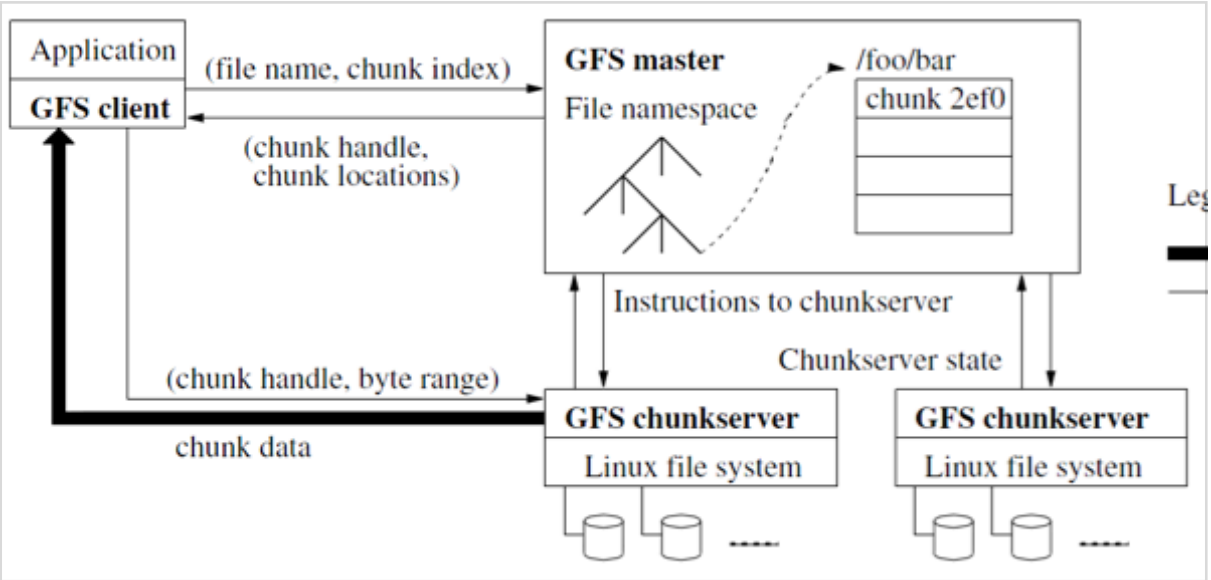


- **RPC处理failure的方式：**

1. At least once
   直到收到回复一直发送请求
2. At most once
   Client一直发送请求，但server只回复一次，idempotent（幂等）操作
3. exactly once

一些包含指针的内容需要rpc翻译成实际的object

GFS google file system



GFS不存在目录， 比如一个名称是-foo-bar，则当作一个用户名，把-foo-bar存进chunk里，这样的好处：

1.减少对chunk的访问次数

2.节约chunk的空间，提高空间利用率

一个master可以serve很多机器，不是性能的瓶颈，很少崩溃，因此master只有一个节点。

- **读操作顺序：**

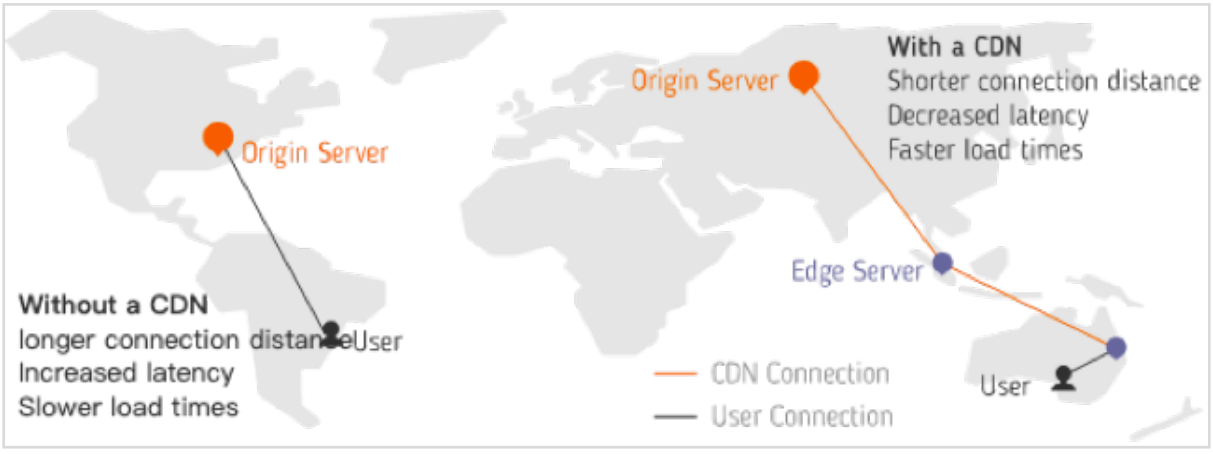
1. application发送请求
2. GFS的client把request翻译成filename、chunk index发送给master
3. master返回chunk handle和备份的地址
4. client拿到地址，发送chunkhandle 和byte range
5. chunk server 把数据发给client
6. client把数据发给application

- **写操作顺序：**

1. application发送请求
2. GFS的client把request翻译成filename、chunk index发送给master
3. master返回chunk handle和备份（所有的）的地址
4. client吧write的数据吧给所有的loctations
5. client给primary chunk server 发送写命令
6. primary执行写操作
7. primary给其他备份发送写命令
8. 其他的备份响应primary
9. primary响应client

CDN（content distribution network）与DNS

- **CDN：**



使用DNS决定投送到哪个CDN中

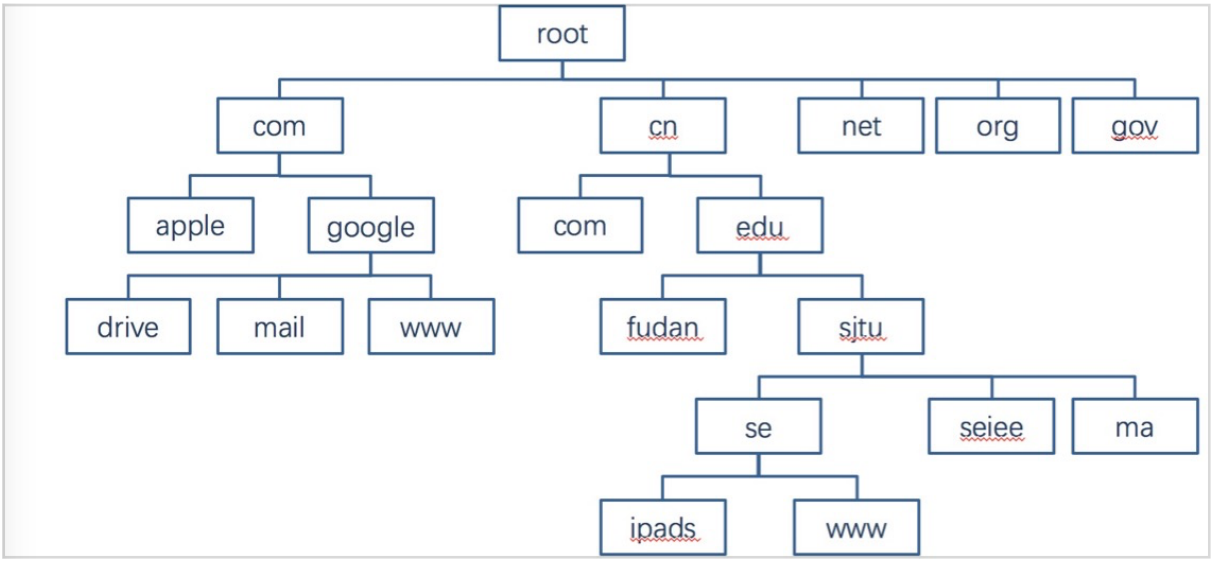
- **DNS：**

DNS

一个域名可以对应多个ip address

对找关系表被储存在不同的name server中

并且域名有不同的层级

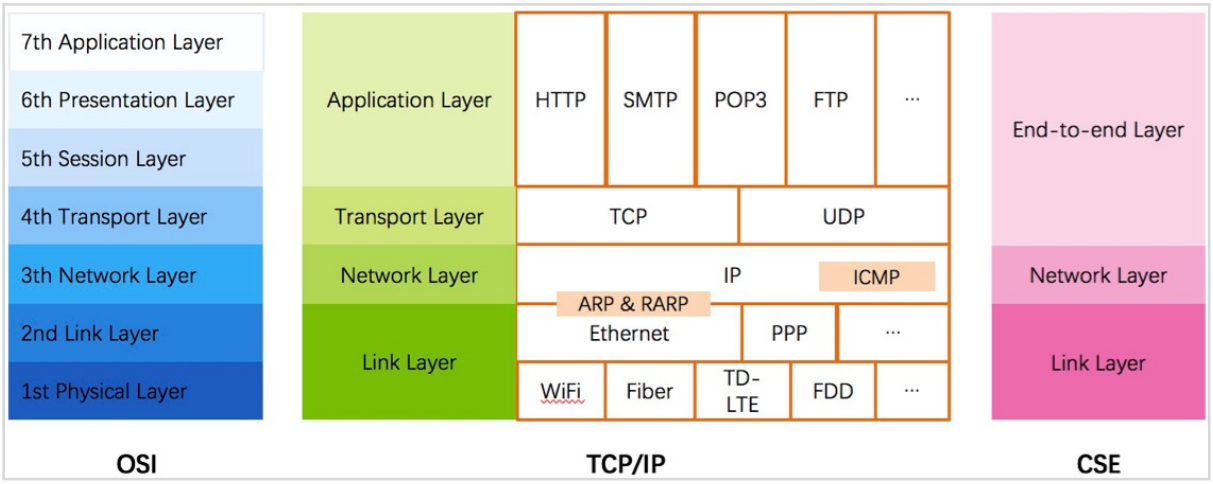


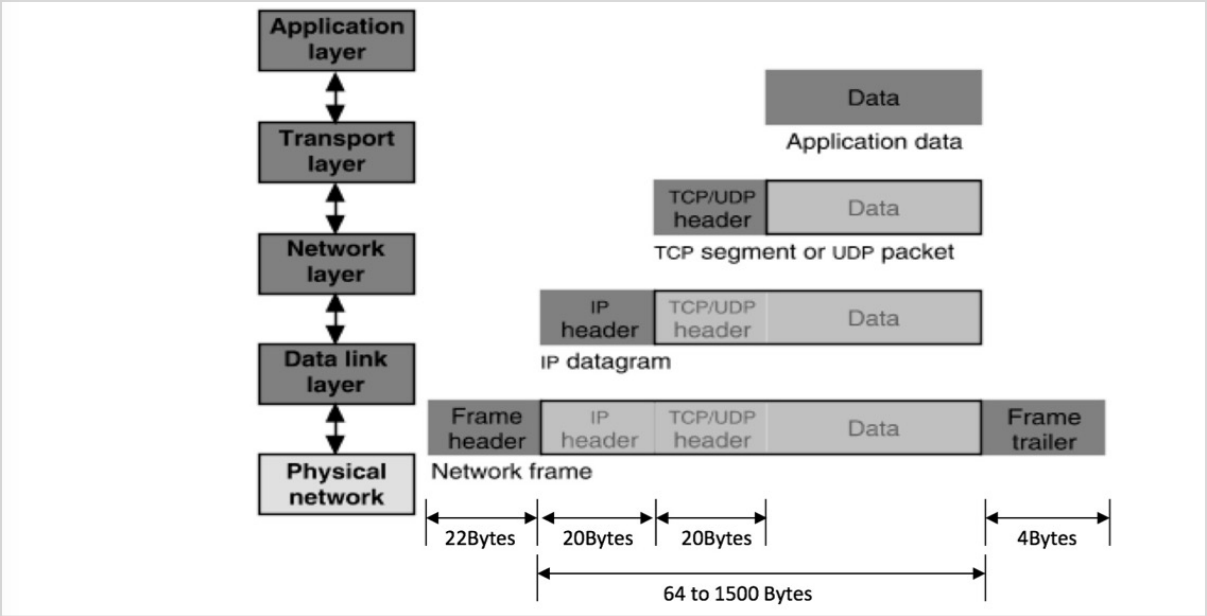
根server 返回com的ip，com返回apple的ip...以此类推

优化：

1. 可以访问不是根的server这样可以不每次都返回到根server中
2. 递归查找，name server 帮助查找，因为nameserver的网速更快
3. 做一个cache，下次就保存这个地址了

layer in network网络分层



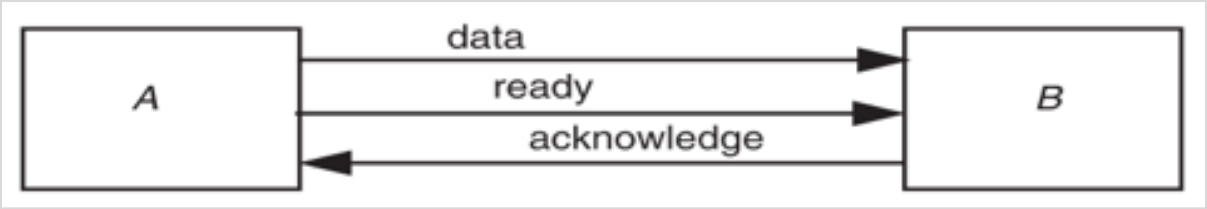


- **link layer：**

  是三个层级中最底层的

  作用：把数据从一个物理位置移向另一个物理位置

  由于不在同一个位置不能使用shared clock



因此使用ready ack线

没有clock不知道连续的一串数字到底有几个：

使用manchester编码 0：01，1:10

1. Isochronous 同步的信号传输：

   假设64 kbps each phone 45mbps each link

   如果一个包的大小是8个bit 那么每秒有8000个包

   每隔5624个bit time（125微妙）发一个包

   允许同时有703（5624/8）个call conversation
2. 包（packet）由guidance 和frame组成

   判定frame何时结束：

   可以用7个bit的1判定，如果有6个连续的1 后面自动加一个0
3. error handling

   可以在末尾添加checksum

   发现错误后可以resend，让reciever丢弃这个包
4. error correction

   第一种：2bit-3bit

   一共有四个状态，要是出错了就是另外四种：

00 ➡ 000

11 ➡ 110

10 ➡ 101

01 ➡ 011

第二种：4bit-7bit

$$P_1 = P_7 \oplus P_5 \oplus P_3$$
$$P_2 = P_7 \oplus P_6 \oplus P_3$$
$$P_4 = P_7 \oplus P_6 \oplus P_5$$

在四个中错一个可以分辨

- **network layer**：
  - NETWORK_SEND (segment_buffer, destnation, network_protocol, end_layer_protocol)
  - NETWORK_HANDLE (packet, network_protocol)

data plane是用来传输数据的

control plane是用来传递命令的

查找dst，如果认识就发送，不认识丢掉

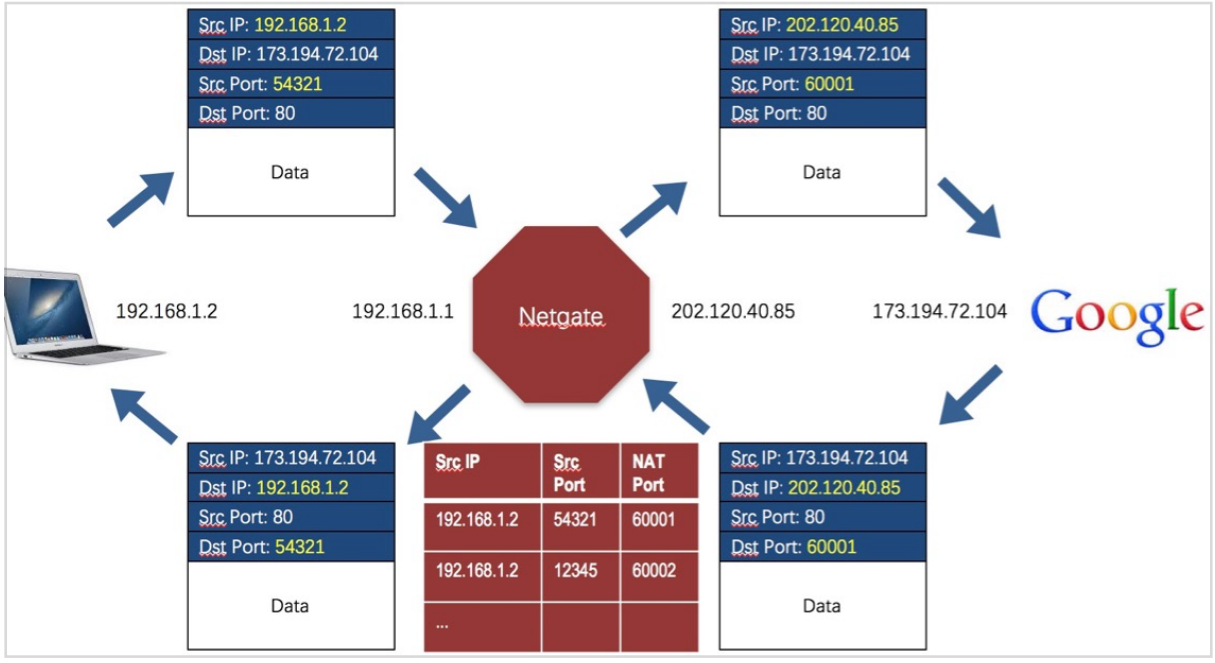
TTL（time to live）如果=0丢掉

检查checksum

把包发到output端口

把包发到link layer

NAT（network address translation）



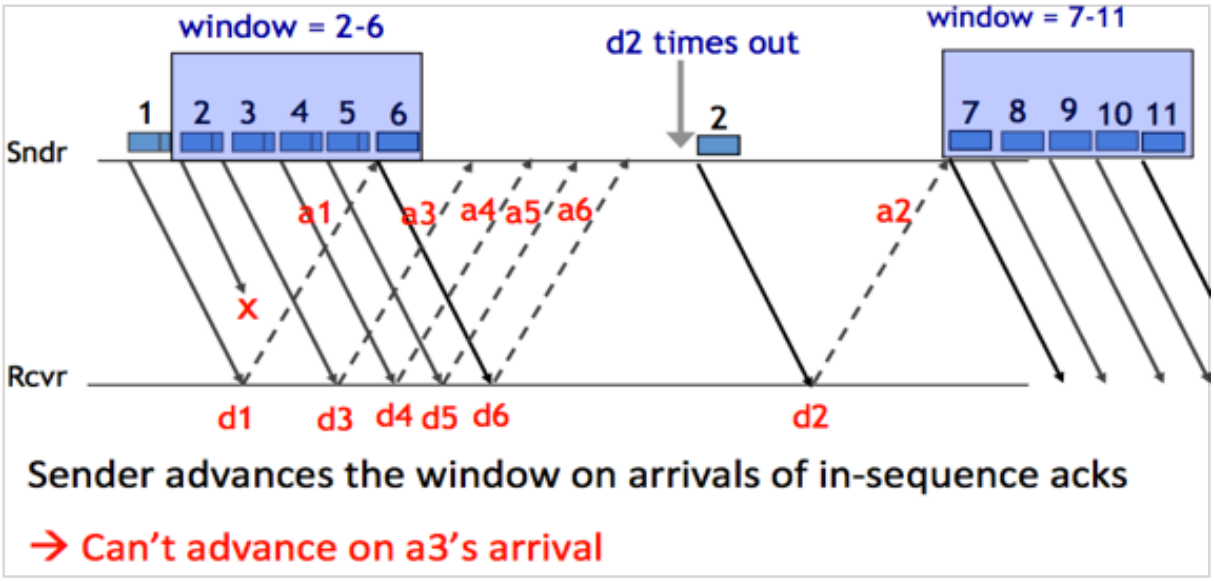
- **end-to-end layer**：

1. assurance of at least once
   RTT（round-trip time）： to_time + process_time + back_time (ack
   但不能使用fixed time 来判定是否超时：
   会造成nfs的拥堵和崩溃

   如果确定一个timeout：
   方法1:观察近期的rtt（8次，给一个平均），根据1.5倍设置一个timeout
   方法2:receiver回复还缺少什么没收到，这样就不需要一个timer了
2. assurance of at most once：
   保留一个请求的表，看是否有相同的请求 但是请求会越来越多
   只保留每一个sender发送的最厚的一次nounce
3. assurance of data integrity
   检查checksum，如果不一样就丢弃
4. 把大文件进行分段再重新整合
   收包方法：
   方法1:只按照顺序收包，提前收到后面的丢掉
   方法2:用一个buffer先收包，再排序，1-10收 再收11-20
   要用sliding window 一个滑动的窗口，收到了第一个包，就收2-11即可
5. assurance of jitter control
   看电影的时候不希望一帧一帧
   一般用缓存一段时间等一段时间再看
   Dlong是比99%时间都久的延迟
   Dshort是最短的延迟
   Dheadway是平均的延迟

$$\text{Number of segment buffers} = \frac{D_{\text{long}} - D_{\text{short}}}{D_{\text{headway}}}$$

6. assurance of Authenticity and Privacy
   public key：给reader加密
   private key：给writer加密
7. end-to-end performance
   slide window的策略进行发送



如何选择window的size：

window size ≥ round-trip time × bottleneck data rate

- Receive 500 KBps
- Sender 1 MBps
- RTT 70ms
- A segment carries 0.5 KB

- Sliding window size = 35KB (70 segment)

以太网ether net

地址格式：FE：07:66:C2:00　　(48bit)

以太网用一根总线连接在一起，上面有很多网口，所有的包所有人都听得见

Hub模式 ➡ 一条线，广播所有人都能收到通知。在hub上所有的包都是共享的，网卡的混杂模式收到一个包就收到，网络层有一个handler，判断包是不是给自己的

Switch模式 ➡ 星状 增加并发性

半双工：发的时候不能收

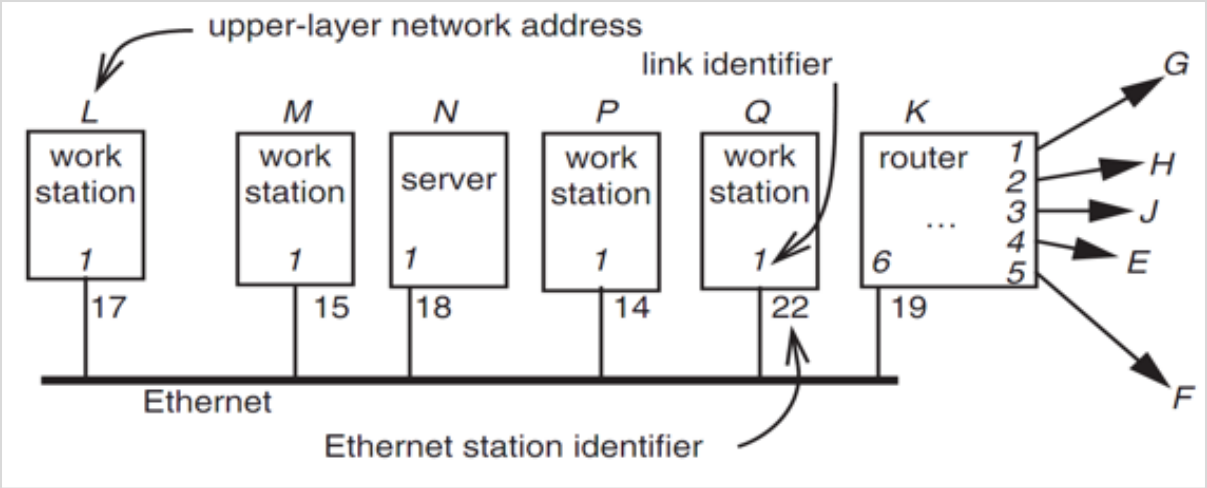
全双工：发的时候也可以收

以太网的包：

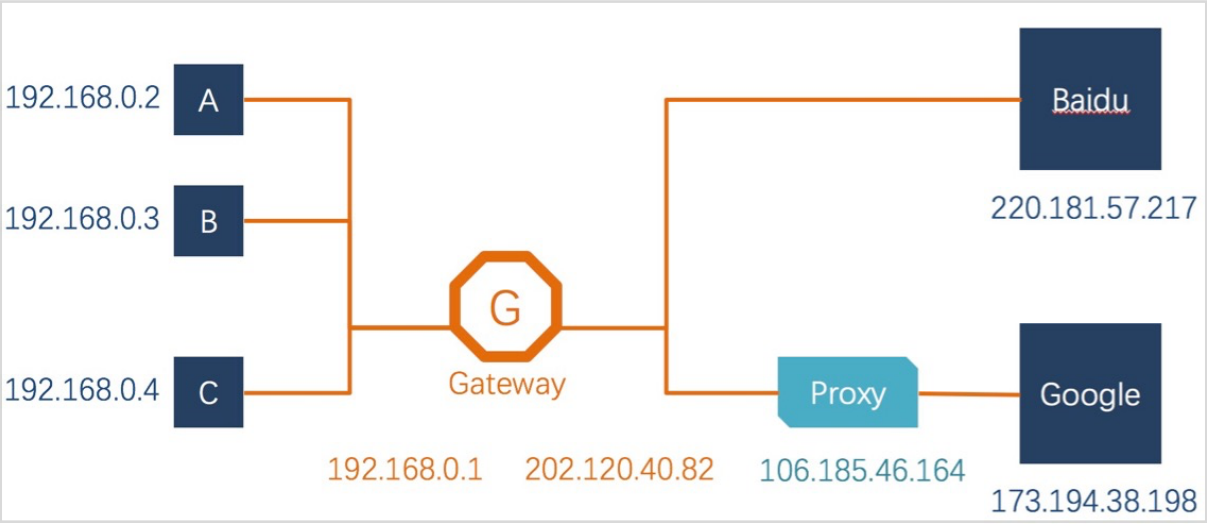
leader	destination	source	type	data	checksum
64 bits	48 bits	48 bits	16 bits	368 to 12,000 bits	32 bits

在上层还有其他的checksum，层层的checksum保证文件不会出错

Ip上有组播，形成组播的地址，组播地址(在直播很有用)



在一个以太网内用station地址，在以太网外发给router即可



app：

发送以下信息给百度，附带百度的ip，和source ip

Target MAC:	Source MAC:	Target IP: Baidu's IP	Source IP: Node-C	Data

os：

把信息发送给router（gateway），但是cource ip不变，知改变source Mac地址和目标Mac（以太网）地址

Target MAC: Gateway	Source MAC: Node-C	Target IP: Baidu's IP	Source IP: Node-C	Data

Router1（gateway）：

改变source ip， 改变source Mac 和target Mac，传给下一个router，并维护一张表

Target MAC: Router-2	Source MAC: Router-1	Target IP: Baidu's IP	Source IP: Router-1	Data

router2:

发现可以连接百度，改变 source Mac 和 target Mac和source ip，并维护一张表

Target MAC: Baidu	Source MAC: Router-2	Target IP: Baidu's IP	Source IP: Router-2	Data

ARP spoofing

一个hacker机器伪装成ip对应的地址发送到mac-pooling地址表，这样A与B混淆之后所有发送

的信息都要经过hacker
这种攻击模式叫 man in the middle

防止man in the middle的方法：
Arp协议写死，只有管理员有能力改表。
定点一段时间进行视察

router/gateway是ip层（network layer）
switch/hub的线 是以太层（link layer）

Routing

- **link state链路状态：**
  一个node对所有的node进行广播，知道整个网络的拓扑结构
- **distance vector routing距离矢量：**
  Node只需要知道下一个目的地
  每一个node对临近的node发信息
  最后计算出发送给下一个人需要给谁

  存在问题：
  loop环路和counting to infinity（计数到无穷大）
  解决方案：split horizon（从哪个接口收到的信息不能再发回去）
- **path vector routing路径矢量：**
  类似于distance，但是保存了整个路径
  避免永久的loop：
  当node更新路径的时候不会更新包含它自己的路径

  当路径变大，优化的方法是由于path过大，采用region（AS: Autonomous System ），每个region只有一个端口和外界交互

- **BGP：（Border Gateway Protocol）**
  为了利润最大化，peer之间可以互换信息，交换自己的consumer，但不交换自己的provide和别人的consumer

TCP的拥堵控制

- **window大小的选择：**
  性能优先：window size ≥ round-trip time × bottleneck data rate
  减少拥堵：window size ≤ min(RTT x bottleneck data rate, Receiver buffer)
- **发送机制：**

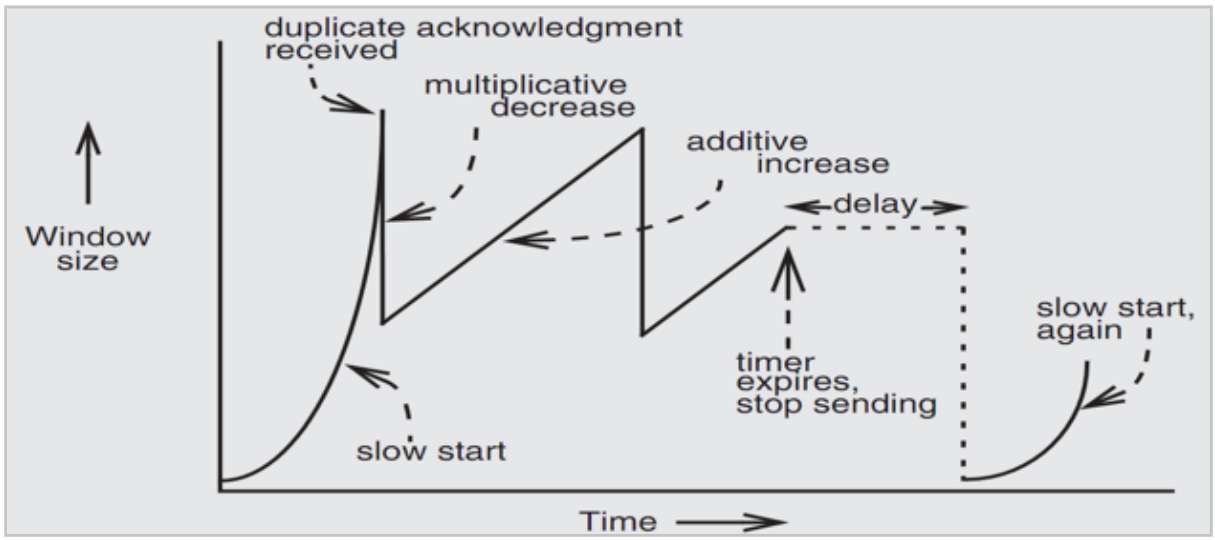
AIMD：每一个RTT

No drop: cwnd = cwnd + 1

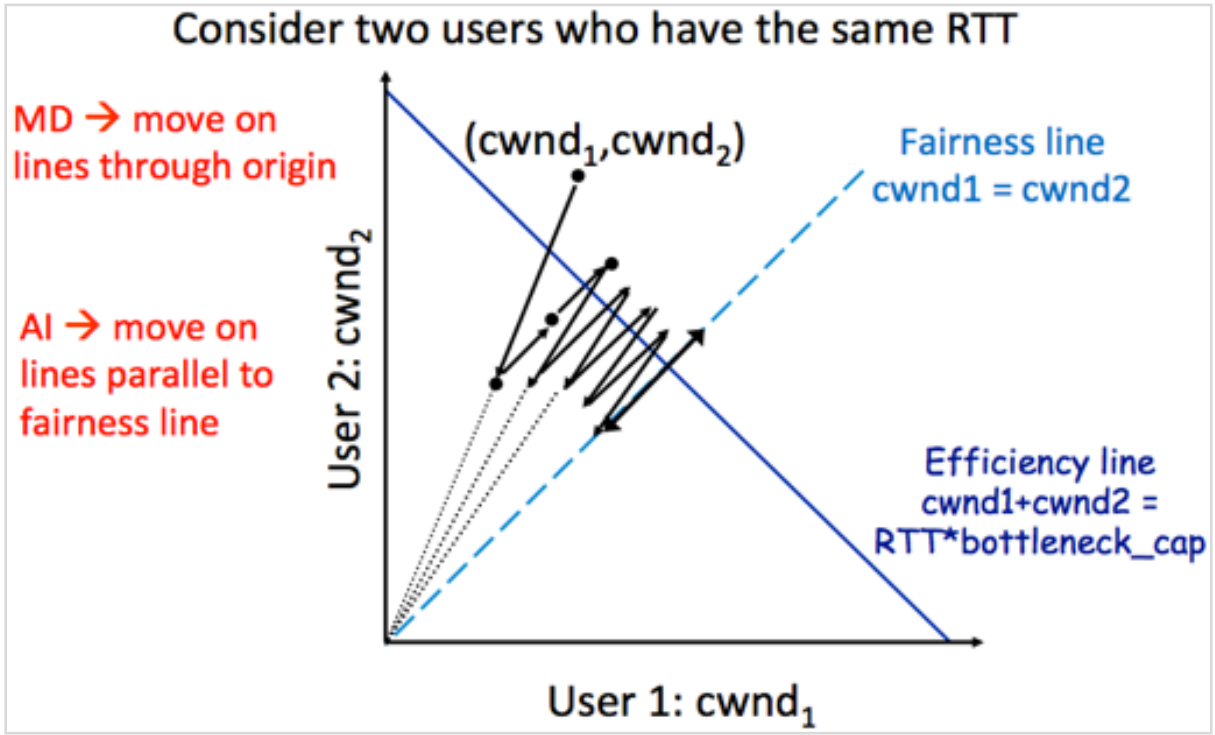
A drop: cwnd = cwnd / 2

问题：最开始加的太慢

解决方案：在第一次增长用指数增长，之后再线性增长



两个同时争夺同一个内容的点会最终逐渐趋向于平均分盆



思考问题汇总

- **rename与link问题**

  Renaming-1:（若新名字已存在）

  把a.txt命名为b.txt

  1 unlink（to_name）

  unlink b.txt

  2 link(from_name ,to_name)

给文件a添加一个名字b

3 unlink(from name)

删除原有名字a

在1与2之间如果电脑断电，则文件名to_name丢失

Rnaming-2:（若新名字不存在）

1 link（from_name, to_name）

2 unlink(from_name)

保证link之后inode 的ref_cnt+1，如果此时断电，from_name与to_name都可以访问。

- **软连接的cd..**

  系统会保存一个软连接所在位置的目录，..默认是这个目录的上级

  如果输入-p，就进入真实的目录中

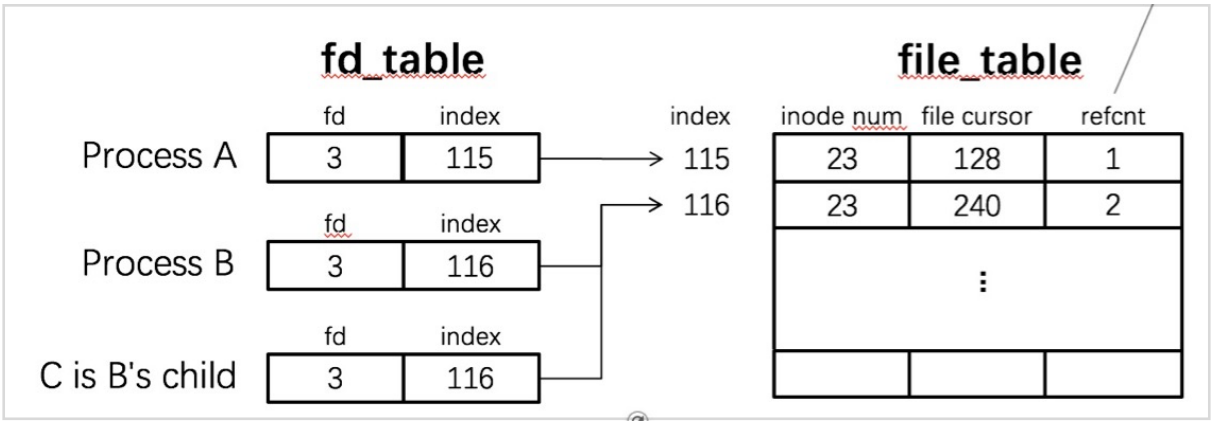
- **fopen与open**

  open返回fd

  fopen返回file*

  open比fopen更低层，但fopen由于有buffer所以更快

- **fd与file cursor**:



- **write的正确操作顺序：**

  Allocate new blocks, write new data, update size

- **使用MMIO的时候的注意事项：**

  Volatile指令：确保指令不会因为编译器的优化而被省略

- **CPU与设备、物理地址的交互：**

  与物理地址的交互：使用bus

  用物理地址标注memory的位置

与设备交互：也适用物理地址

pooling interrupt 和DMA

PIO或者MMIO

注：外设的物理地址被BIOS设置，但是键盘等常用外设的物理地址是固定的

- **为何pc不使用GFS的内容：**

  为什么不在本机上用这个方法？：

  Google的文件一旦写好，就很少再更改了，但本地不同

  Google的文件一般是大文件，本地使用chunk对空间的利用率太低

  本地有几百万的文件，google有一个只做识别分发的master，但在本机上是一条一条存在一起

  在本机上做ls更快，在flat上更慢一些

  Mvdirectory在gfs上也很慢，关于目录的操作都会变慢

- **file name和host name的异同：**

1. 都是为了user friendly
2. 都不是object的一部分
3. 一个文件名只能对应一个value

   一个域名可以对应多个ip地址

- **DNS放大攻击：**

1. 攻击者使用受损端点将带有欺骗性IP地址的UDP数据包发送到DNS recursor。数据包上的欺骗地址指向受害者的真实IP地址。
2. 每个UDP数据包都向DNS解析器发出请求，通常会传递诸如“ANY”之类的参数，以便接收可能的最大响应。
3. 在收到请求后，尝试通过响应提供帮助的DNS解析器会向欺骗的IP地址发送大量响应。
4. 目标的IP地址接收响应，周围的网络基础设施因流量泛滥而变得不堪重负，导致拒绝服务。

- **receiver 进行checksum的必要性：**

  linklayer 已经进行checksum了，但是memory读写也会出错，所以receiver进行检查也是很有必要的