

饱满的

CSE 答记

目录

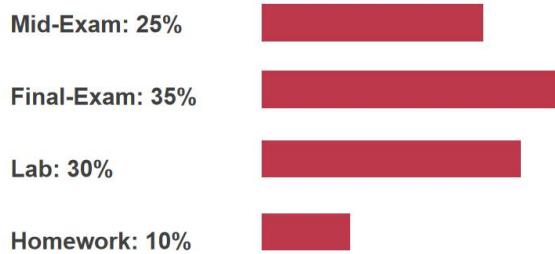
2021/9/14 (导论)	6
计算机系统的十四个特性.....	7
2021/9/16.....	12
淘宝优化架构的心路历程.....	15
MTTF、MTTR、MTBF.....	24
附录：各大模型计算方法.....	25
2021/9/23.....	26
CAP 定理.....	28
inode-based file system.....	29
2021/9/26.....	38
2021/9/28.....	49
RPC.....	50
RPC 的参数传递.....	53
RPC 遇到 Failure.....	55
2021/9/30.....	56
传输文件：上传下载模式.....	57
传输文件：远程访问模式.....	57
传输文件：NFS (network file system)	57
NFS 读取文件的流程.....	59
GFS(Google File System).....	63
2021/10/9.....	68
Batch Processing.....	69
MapReduce 的容错机制.....	76
MapReduce 提升 Locality 的方法.....	77
冗余执行 (redundant execution)	78
MapReduce 的局限性.....	78
Dryad.....	78
2021/10/12.....	80
使用 MapReduce 来计算 PageRank.....	83
Pregel 框架.....	84
Pregel 计算 PageRank.....	86
Pregel 的容错机制.....	87
2021/10/14.....	91
Key-Value Store.....	91
Log-structured File.....	92
SSTable.....	94
2021/10/19.....	96
Crash Consistency.....	96
EXT4.....	101
2021/10/21.....	103
All-or-nothing Atomicity.....	104
2021/10/26.....	112

怎么保证 ACID.....	114
Serialization.....	114
Conflict Graph.....	116
View Serializability.....	117
生成 conflict serializable 的调度.....	119
Simple fine-grained lock.....	119
2PL(Two-phase Locking).....	120
优化: read-write lock.....	121
2PL 的主要问题.....	122
2PL 是否真的能满足 serializability 呢?	122
2021/10/28.....	122
OCC (乐观并发控制)	125
RTM (restricted transactional memory)	127
Multi-version Concurrency Control.....	130
2021/11/2.....	132
分布式的 Transaction.....	133
2PC(Two-phase Commit).....	134
确定文件的新旧的前置问题: 同步时间.....	140
处理 network partition: Quorum.....	143
2021/11/4.....	143
PAXOS.....	148
2021/11/9.....	156
Paxos 算法回顾.....	156
Raft 协议.....	157
数据库.....	158
Data Model.....	161
Query language.....	163
Table schema design of relational model.....	165
2021/11/11.....	166
DBMS 中的 Page.....	169
Tuple 是怎么保存在 Page 中的.....	172
减少磁盘太慢带来的问题.....	174
总结.....	176
2021/11/16.....	176
1. Multiple buffer pool.....	179
2. Prefetching.....	180
3. Scan sharing.....	180
Buffer Pool 的 Replacement Policy.....	181
通过 query execution 来优化 buffer replacement policy.....	181
2021/11/23.....	182
2. materialization model (物质化模型)	185
3. Vectorized/batch model.....	186
Operator 的优化.....	186
Select 的优化.....	187

早期 SQL.....	189
NOSQL.....	190
Weaken transaction.....	191
3. 异步复制 (asynchronous replication)	192
NewSQL.....	193
Case Study: TiDB.....	193
2021/11/25.....	194
Layers in Network.....	194
TCP/IP 协议.....	195
Transport Layer.....	198
Network Layer.....	200
Link Layer.....	201
数据并行传输(Physical Transmission without Shared Clock).....	201
数据线性传输.....	202
一根线传数据的同步模式.....	202
Data communication network (包交换网络)	203
海明距离.....	204
2bit->3bit 做奇偶校验.....	204
4bit->7bit 自动纠一位.....	205
2021/11/30.....	205
Routing.....	207
分布式路由的三个步骤.....	208
Link-state 路由协议.....	208
Distance-Vector 路由协议.....	210
Data Plane.....	219
NAT.....	221
2021/12/2.....	222
以太网.....	223
Hub 模式.....	224
End-to-end Layer.....	227
1. 确保 At-least-once 的分发.....	228
NAK.....	229
2. 保证 At most once.....	230
3. 数据的 integrity.....	230
4. 分片和重组.....	231
5. Assurance of Jitter Control (抖动控制)	232
6. 确保权限控制和隐私.....	233
2021/12/7.....	233
1. lock step.....	234
AIMD (线性增长、指数下降)	237
DNS 的设计.....	240
DNS 查找域名和 IP 映射的算法.....	240
2021/12/9.....	244
BitTorrent.....	246

DHT (Distributed Hash Table)	247
一致性哈希的 O(N)算法.....	249
一致性哈希的 O(logN)算法.....	249
2021/12/14.....	251
安全性.....	254
Threat Model: assumption.....	255
2021/12/16.....	255
GUARD MODEL.....	256
认证.....	259
Salting.....	262
1.challenge-response 的协议.....	263
2.use passwords to authenticate the server.....	264
3.Turn offline into online attack.....	264
4. 每个网站最好有不同的密码.....	264
5. One-time password.....	264
6.我们将 application 和 request authorization 绑定在一块.....	265
7.FIDO (replace the password)	265
2021/12/21.....	266
攻击者如何偷走我们的数据.....	267
TaintTracking.....	269
Dynamic Taint Analysis.....	270
TaintDroid.....	271
TaintCheck Detection Module.....	274
No Data To Protect.....	275
安全信道.....	276
DH 密钥交换算法.....	278
RSA (非对称加密算法)	279
2021/12/23.....	280
Stack Buffer Overflow Attack.....	281
Code Reuse Attack.....	283

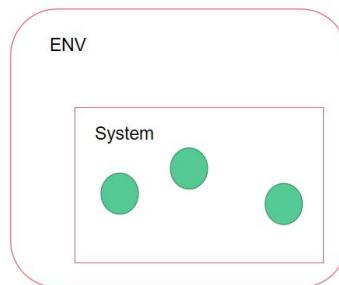
2021/9/14 (导论)



我们希望大家把重点放在 Lab 上，考试都是开卷，核心都是在 ppt 上。

MIT 神课：6.828 OS, 6.033 CSE, 6.824 分布式, 6.858

这门课的话题就是复杂性问题。当我们面对几千万台机器同时协作的时候，复杂性就出来了，很难把单体的机器的逻辑直接应用到集群中。比如最近大火的 metaverse，元宇宙。



系统就是指一堆 component，通过一些接口互相连接，和周围的 environment 有一个边界。系统的定义并没有很清晰，一个系统可能是一个大系统中的子系统。系统是可以不断增加的。我们发现了很矛盾的地方，系统可以不断增加，但是人的大脑对复杂性的理解是很有限的。如果我们需要搞定每个 component 的工作机制才能去理解，那么大脑是不能理解的。Linux kernel 几千万行。这时就要用到系统的方法，描述气体分子的属性就需要用到宏观统计量。

Compare with the Computer Systems

Programming / Data Structure

- LoC (Lines of Code): From hundreds to millions

Operating System / Architecture

- CPU cores: from one to hundreds

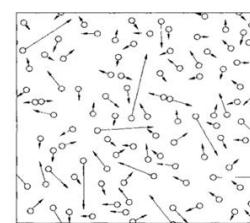
Network

- Nodes: from two to millions

Web Service

- Clients: from tens to millions

○—○



超级计算机几百万个 cpu，就会有新的特性。并发几百和几千万对系统的冲击是完全不一样的。

计算机系统的十四个特性

Correctness, Latency, Throughput, Scalability, Utilization, Performance Isolation, Energy Efficiency, Consistency, Fault Tolerance, Security, Privacy, Trust, Compatibility, Usability.

1. 正确性，大家觉得最直观的特性。怎么定义一个系统的正确的：做它该做的事情。其实在实际中是很难定义它的。很难定义是 bug 还是 feature。Eg：在 linux 中以点开头的文件是隐藏文件 需要 `ls -a`。

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return; /* len too large */
if (buf + len < buf)
    return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

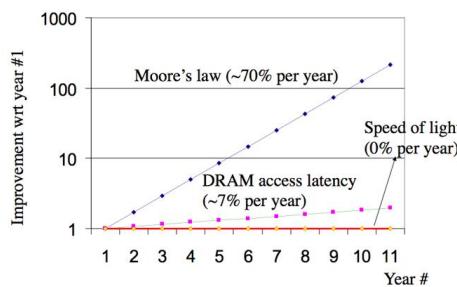
	Construct	Sufficient condition	Undefined behavior
Language	$p + x$	$p_\infty + x_\infty \notin [0, 2^n - 1]$	pointer overflow
	$*p$	$p = \text{NULL}$	null pointer dereference
	$x \text{ op}_s y$	$x_\infty \text{ op}_s y_\infty \notin [-2^{n-1}, 2^{n-1} - 1]$	signed integer overflow
	$x / y, x \% y$	$y = 0$	division by zero
	$x << y, x >> y$	$y < 0 \vee y \geq n$	oversized shift
	$a[x]$	$x < 0 \vee x \geq \text{ARRAY_SIZE}(a)$	buffer overflow
Library	$\text{abs}(x)$	$x = -2^{n-1}$	absolute value overflow
	$\text{memcpy(dst, src, len)}$	$ \text{dst} - \text{src} < \text{len}$	overlapping memory copy
	use q after $\text{free}(p)$	alias(p, q)	use after free
	use q after $p' := \text{realloc}(p, \dots)$	alias($p, q \wedge p' \neq \text{NULL}$)	use after realloc

在 C 的规范中，指针的溢出是 undefined behavior，编译器会认为这不会发生这种情况，把这段代码删除掉了。

手动操作：强制类型转换为 int，再做操作，再判断溢出，再转化回来。

2. latency，系统的时延是很麻烦的事情，因为吞吐量可以通过花钱添加服务器来增加。但是要缩短每个人访问的时延是很难的。

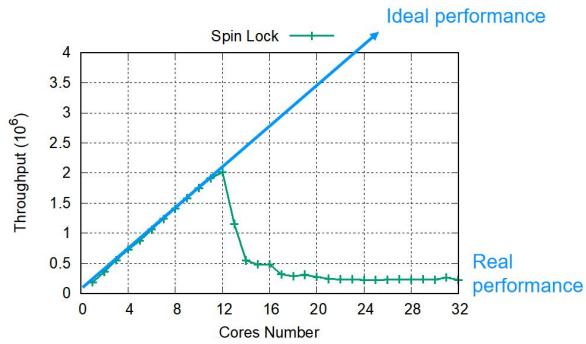
Latency is hard to optimize, it has physical limitations



在处理器上光速传播的距离造成的 latency 也是不可忽略的。

3. Throughput，一旦我们的芯片密度不能越来越高，它势必会越来越大。一旦物理达到极限以后，软件的作用就越来越大。

4. Scalability 真实的 arm 测试 spin lock，到 12 个核的时候，性能下降了



结果 32 个核的时候和 1 个核的时候差不多，这因为大家都在抢 spin lock。Spin lock 是内存中的一个 bit。随着 cpu 越多，争抢锁的消耗越大。

5. performance isolation

Eg: 一台机器跑了 benchmark，情况 1: 一个人自己跑，情况 2: 背后还跑了一个 while(1)，加上这个 noisy neighbour，影响是对于 99% 的 latency 会增加 42%，非常非常 noisy 的情况，latency 会增加 29 倍。虽然 CPU 可以运行多个程序，但是 neighbour 的情况影响是很大的。

6. Utilization

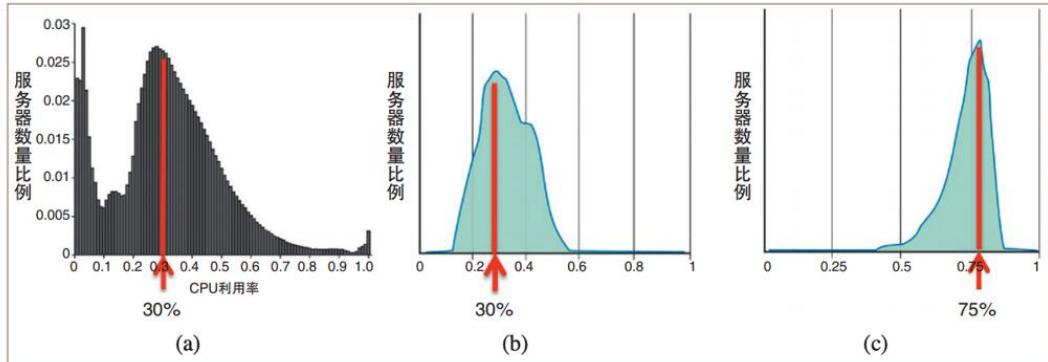


图2 谷歌数据中心CPU利用率：(a)2006年5000台在线应用服务器；(b)2013年2万台在线应用服务器；(c)2013年2万台批处理应用服务器⁴

对于云来说，Google 数据中心的 CPU 利用率是 30%。对于要求快速响应的服务器，需要预留很多资源，所以云端的服务器都开着虽然耗着电，但是利用率并没有拉满。

7. 节能 (energy efficiency)

这又是一个 trade-off，数据中心最大的开销是电费（服务器电费、空调电费），尤其现在是碳中和，对用电产生了很大的挑战。在服务质量不变的情况下，怎么压缩用电量。

8. 兼容性 (compatibility)

Intel's Itanium, 64-bit, not compatible to x86, died.



安腾处理器是 inter 从 32 转 64 的重大选择。导致之前所有的 x86 代码都需要重新编译

才能跑。

9. Usability



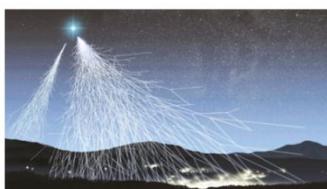
左边是 window mobile，直到 iphone 第一代出现。用户的易用性也很重要

10. Consistency

一致性非常重要。为了容错，我们需要多个备份，但是备份之间就会出现数据不一致这个问题，对于支付宝来说，不能接受数据不一致的情况。比如用不同的服务器一笔钱花了两次或者一个订单扣了两次钱。为了性能一定会有多个副本在多种 cache 中，所以一定会有这个问题。

11. Fault Tolerance

A bug of Cisco router is caused by cosmic radiation



Cisco Bug: CSCuz62750 - Incremental drops on counter DROP_FRM_CRC_ERR_SGMII0 causes traffic loss

Last Modified
Sep 20, 2016

Product
Cisco ASR 9000 Series Aggregation Services Routers

Known Affected Releases
all

Description (partial)

Symptom:
Partial data traffic loss can be observed. Below list counters need to focus to determine the issue:-

show controller np counters all loc <location> could indicate FRM_FRM or CRC drop counts.
1082 DROP_FRM_CRC_ERR_SGMII0 1423310954 0
1083 DROP_FRM_FRM_ERR_SGMII0 74455703 0
1084 DROP_FRM_CRC_ERR_SGMII1 1427728248 0
1085 DROP_FRM_FRM_ERR_SGMII1 74942820 0
1086 DROP_FRM_CRC_ERR_SGMII2 1431171380 0
1087 DROP_FRM_FRM_ERR_SGMII2 0
1088 DROP_FRM_CRC_ERR_SGMII3 0
1089 DROP_FRM_FRM_ERR_SGMII3 0

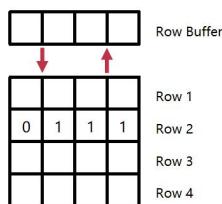
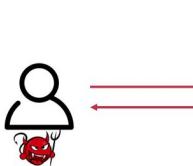
Cosmic radiation

It is also theoretically possible for data corruption to happen after the CRC check which should then be detected at the higher protocol layers.

Conditions:
Problem observed on operational network. Possible trigger is cosmic radiation causing SEU soft errors.

This issue can affect "Juggernaut" LCs (A9K-1X100GE-SE/TR, A9K-2X100GE-SE/TR)

Cisco 路由器厂商的 bug report，由于宇宙辐射造成的 bug。



CPU Architecture	Errors	Access-Rate
Intel Haswell (2013)	22.9K	12.3M/sec
Intel Ivy Bridge (2012)	20.7K	11.7M/sec
Intel Sandy Bridge (2011)	16.1K	11.6M/sec
AMD Piledriver (2012)	59	6.1M/sec

Rowhammer Attack: flip memory bits!

在我们的内存中，存储模式是先定位行再定位列，内存需要为这一行进行充电操作，在充电的时候，临近两行的电就会弱一点点。如果我们频繁地读 row1 和 row3 会导致 row2 的电量下降，下降到一定程度的时候，row2 的 1 就会变成 0

页表的后面几位可读可写可执行，user/kernel 可访问。如果我们通过这些漏洞来 hack，就可以控制一台虚拟机。DDR4 发生 row hammer 的概率远小于 DDR3。复现这个问题也很难，因为多次访问会进 cache，所以研究者会尝试每次访问内存前 flush cache。

12. Security

Cold-boot attack: physically attack that can scan memory to get any plaintext



Müller, Tilo, and Michael Spreitzenbarth. "Frost." *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2013.

以手机为例，里面有一堆关键的数据和账号。内存掉电之后，数据不会马上丢失。放在非常冷的环境下，可以延缓这个过程。手机内存芯片其实就是一个 SSD，可以直接读只是数据加密了。数据加密的密钥放在 DDM。内存里是明文，加密到硬盘里是密文。现在保护数据基本上是保护硬盘上的数据，而不保护内存上的数据。

Eg2: 装 APP 的时候，有些索要的权限很关键。有些看起来不关键的就是陀螺仪。陀螺仪可以用来判断到底是哪几位密码。

13. Privacy

当我们打开页面的时候，有一个模式是隐私模式。理论上用隐私模式访问网站之前之后本地是没有区别的。在访问网页的时候，`OSdump` 数据，当物理内存不够的时候，会把内存文件 `swap` 到硬盘上。它不会 care 上面是隐私文件还是别的文件。也会留下一些印记。

14. Trust

钱存在银行里，银行没有那么多现金，钱只是存在银行的服务器中。去中心化的货币就是在 `trust` 依赖上的进一步减少，不会依赖某一个人某一个国家每一个军队。

Conflict between these Properties

Usability VS. Privacy

Performance VS. Security

Fault Tolerance VS. Consistency

...

All these lead to more complexity

性能和安全的权衡。容错（要求多台机器）和一致性（多台机器导致的结果）

System complexity 又有一些共性。

Problem Types

Emergent properties (surprise!)

- The properties that are not considered at design time

Propagation of effects (butterfly effect)

- Small change -> big effect

Incommensurate scaling

- Design for small model may not scale

Trade-offs

- Waterbed effect

1. **Emergent properties:** 你永远不知道系统会出什么问题, 直到你 build 出来真正地 run 它。Eg: 伦敦的千禧桥垮塌

Emergent Property Example: Ethernet

All computers share single cable

Goal is reliable delivery

Listen while sending to detect collisions

- If two nodes sends data at the same time, then both cancel and wait for a random time

00011011

1101100



Max length: 1km

00011011

1101100



Max length: 1km

What if A finishes sending before data from B arrives?

- 1km at 60% speed of light = 5 ms (microseconds)
- Original Ethernet Spec: 3 Mbit/sec
 - A can send 15 bits before bit 1 arrives at B
 - A must keep sending for 2*5 ms (to detect collision when first bit from B arrives)
- Minimum packet size is $5*2^3 = 30$ bits
- The default header is 5 bytes (40 bits), so no problem for now

46

47

Eg: A 和 B 之间有一个双绞线连着两个机器。如果 A 发送的同时, 发现 B 也在发送, 那么 A 和 B 都会 cancel 掉, 等待一段时间再发送。在最早的时候这么做没什么问题。

最坏情况: A 发了 1 个 bit 到 B 的时候, B 发了一个 bit, 还需要一个 5ms 才能到 A, 让 A 知道 B 也在发。A 要发的包的最小的长度是 $5\text{ms} * 2 = 30\text{bit}$ 。

当时包头的大小为 40, 所以没这个问题。

3 Mbit/s -> 10 Mbit/s, What will Happen?

First Ethernet standard: 10 Mbit/s, 2.5 km wire

- Must send for $2*12.5 \mu\text{seconds} = 250 \text{ bits} @ 10 \text{ Mb/s}$
- Header was 14 bytes
- Needed to pad packets to at least 250 bits (~32 bytes)

Emergent property: Minimum packet size!

- The 250-bit minimum packet size is a surprise

后面包头变成了 14 个 byte, 很可能发现包在空中飘的时候, 对方也发了一个包。就没有办法知道对方也在发了。只能给包加很多 pad。这就是 minimal packet size。

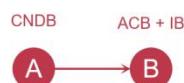
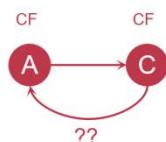
蝴蝶效应

WTO 用 DDT 杀蚊子, 导致疟疾没了鼠疫来了。

Example: No Small Changes

Phone network features

- CF: Call Forwarding
- CNDB: Call Number Delivery Blocking
 - The caller's number should be hidden
- ACB: Automatic Call Back
- IB: Itemized Billing



- A calls B, B is busy
- Once B is done, B automatically calls A
- A's (caller) number appears on B's bill

电话网络，有 call forward，打家里的电话可以 forward 到手机上。第二个功能：把你的电话藏起来，看不到主叫是谁。第三个功能：自动回拨，避免电话费。第四个功能：账单上会一行一行出现打了哪些电话。

单独来看都没什么问题，但是 A 启用的号码隐藏会因为号码隐藏功能失效。

Eg：主机要升级，买 CPU 发现针脚不支持，然后换主板、换内存，最终只能把整套东西都换掉。

不成比例地扩展：

一个系统十倍的时候是不一样的。从数学角度来看，一个维度变化的时候，不是系统中的所有组件在以相同比例进行变化。

第四点就是权衡 trade-off

在看似充满确定性的世界里充满着风险，后面就是讲怎么应对。

2021/9/16

和传统工程不一样的地方：复杂性的控制没有太多的经验，当我们考虑控制复杂性的时候，经验还是不足的。传统的方法有：constructive theory，在造建筑的时候有很大的指导意义。为了填补这些空白，我们只能从 case study 中找到一些 visible 的特点抽取出来去看现有的系统的怎么做的。很重要的一点就是控制 complexity 的维度，比如考虑第一节课的气体，如果每个气体都去考虑方向和动量，会不可控的。比如我们从代码深挖到 CPU 到流水线到电路板上的布局布线，我们考虑所有东西就会使得我们没有办法去思考。所以我们要控制住 debug 的边界，至少在大部分时候要相信编译器。

M.A.L.H

Modularity

- Split up system
- Consider separately

Abstraction

- Interface/Hiding
- Avoid propagation of effects

Layering

- Gradually build up capabilities

Hierarchy

- Reduce connections
- Divide-and-conquer

在控制复杂性的时候，有四个方法，叫做 M.A.L.H，其实核心就是 M (modularity，分而治之，系统很复杂的时候对它模块化) 同时，对模块化要有很好的抽象，有很好的 Interface。尽可能做到高内聚低耦合。Layer 是一层层的层次化，hierarchy 是范围的层级化 (学校->区域->国

家)。

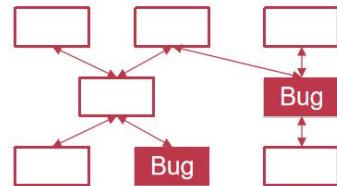
假设我们的程序中有 bug，那么通常情况下我们 debug 的时候和代码行数是正比的。

Modularity



$$\begin{aligned} \text{BugCount} &\sim N \\ \text{DebugTime} &\sim N \times \text{BugCount} \\ &\sim N^2 \end{aligned}$$

Original System



$$\begin{aligned} \text{DebugTime} &\sim \left(\frac{N}{K}\right)^2 \times K \\ &\sim \frac{N^2}{K} \end{aligned}$$

System with Modularity

在 linux 内核中，通过估算大约每一千行有一个 bug. 对于我们自己的代码，预估的是一个没有发现的东西，比较难以估算。

Debug 的时间等于代码的行数*bug 数量，所以就是 $O(\text{代码行数}^2)$ ，拆分成 K 个小模块之后，总的时间复杂度就可以除以 K。

抽象：有几个原则

1. 遵循自然的边界，比如抽象 user 的时候，user 就会有自然的行为，买东西游览网页等
2. 尽可能少的 interaction，模块之间交互越少越好。
3. 抽象要尽可能把错误控制住。

那么之前在做模块化的时候，有一个冲动把 k 变得很大，但是问题在于它们之间的交互非常地频繁和密集。

软件本身对抽象这一点有很大的缺点，边界本身不是特别自然。一些优秀程序员和新手生产力的比例甚至可以到非常大。因为软件没有物理的极限，只要去想就可以想出足够复杂的东西。软件的极限是人类大脑理解的极限，所以不同人的极限是不一样的。

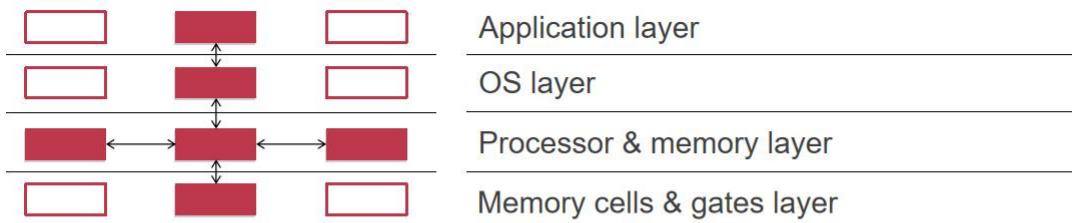
系统变得复杂以后就会有错误，我们需要充分地定义它的接口来判断出软件的行为是对还是不对。调用一个函数，第一件事情就是要 check 它的返回值（错误码）。这是在明确出错情况下的检查，如果调用一个函数返回了是看上去正常的值，怎么去判断它，这会变得很难。如果不去检查，这个错误的值就会扩散到别的模块。

像 OS 的话，接收 user 的输入的时候就会判断传入的参数是对还是不对。OS 需要用最坏的恶意去揣测应用程序是坏的。Eg：应用让 OS 把一个指针的数据（hello world）转发显示到屏幕上。OS 要先 check 这个指针是不是指向内核态的地址。如果 OS 不检查的时候就可能把内核中的信息打印到屏幕上。

EG2：把一段数据从一个地址 copy 到另一个地址，OS 也需要检查地址是否合法。而这个检查之所以能做，就是因为有很好的模块化的隔离，需要让使用接口的人判断返回值对错。

Layer 控制模块之间的 interaction，如果有 n 个点全部连接，它们的 connection 是和

N^2 成正比。



House:

- Inner layer of studs, joist, rafter (shape & strength)
- Layer of sheathing and drywall (wind out)
- Layer of siding, flooring and roof tiles (watertight)
- Cosmetic layer of paint (looks good)

Algebra:

- integer, complex number, polynomials, polynomials with polynomial coefficients

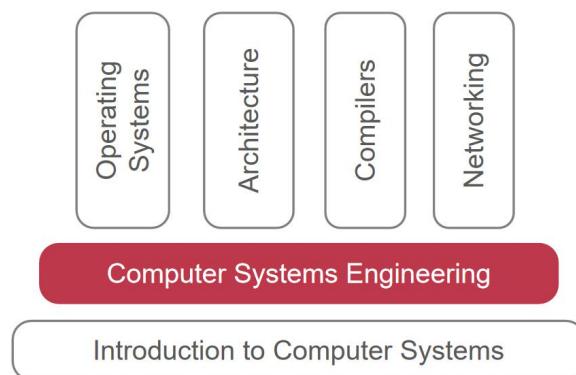
对于一个 layer，不能越过两边的层去交互，所以减少层之间的交互，debug 的时候只需要考虑周围的模块。Layer 在网络中用的非常地成熟，比如 7 层协议和 4 层协议，从而减少考虑问题的范围。

层级 (Hierarchy)

有很多模块的情况下，对外可能只是一个模块的形式存在。

比如我们要连亚马逊的服务器，中间要通过很多路由，比如先找到美国的路由器，让它再帮我们去找，否则我们要把所有的服务器和 ip 都记录下来。而这样我们就只需要记录到美国的一个地址就可以了。

但是计算机系统有一些独特的点，它软件的复杂性没有物理定律限制，原因就是一个数字世界。虽然我们讲了 M.A.L.H 四个点，我们很难找到一个正确的模块化/抽象/分层/层级方法，我们依然需要 case by case 地去分析。



我们这门课是 ICS 的上面。

第二节课 (PPT CSE-02-distributed)

当上节课我们在讲系统的复杂性的时候，从一行到几千万行，从一台到几百万台，从网站用户从一个到几十万个的时候，事情就变得复杂了。

淘宝优化架构的心路历程

我们来讲一个更具体的模型，一个 **highly-scaleable** 的网站，eg: 淘宝和支付宝网站。我们希望学习到内部的架构和核心技术是什么。

High request rate

- 100 billions of requests **daily**

Massive data

- **Facebook** has more than **1 billion** of images uploaded **weekly**, **Baidu** stores **tens of billions** of web pages

整个淘宝天猫双十一，一天 22.5 亿单，QQ10 年同时在线数到 1 亿，还有 12305。我们在做这些网站（scalable website）的时候的经验已经在世界领先了。

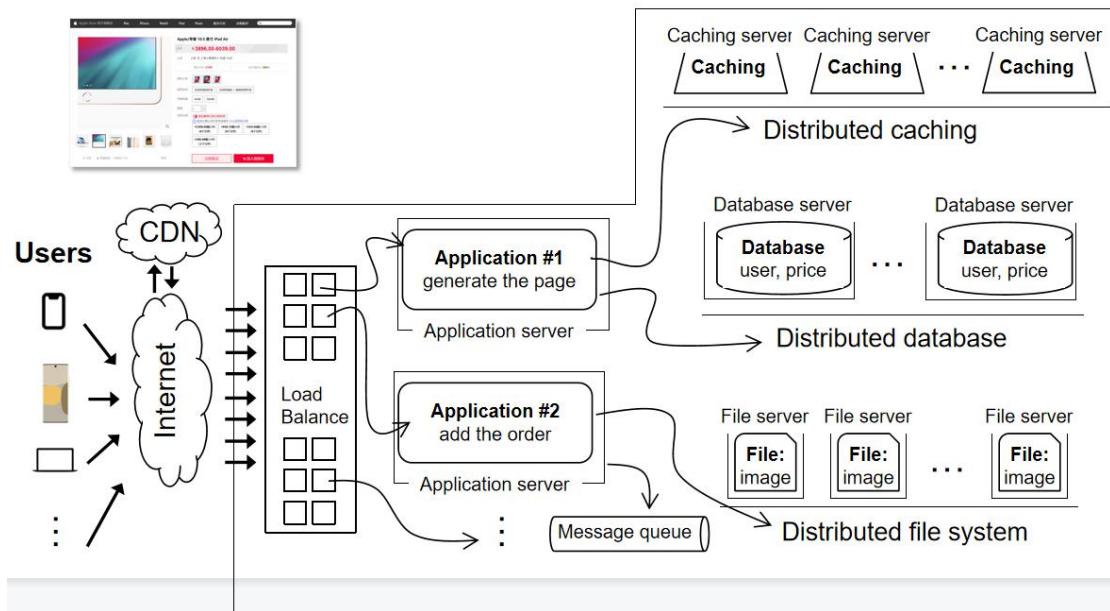


这个是天猫的界面，是一个产品的介绍图，图以 png/jpg 保存在文件系统里，这个价格放在 **database** 中，这里还有一个“收藏商品”界面，人气是根据游览量等信息推算出来的值，它就不是放在 **database** 中，而是放在{"人气":10883}，这个键值对中。**fileserver** 保存非层次化的数据，而 **database** 可以维护更好的层次信息。

为什么呢？这些数据有什么区别？人气这个值经常变来变去。价格频率小一点。有一个经常被忽视的问题，这两个数据，价格更重要一点。重要意味着这个数据不能丢，需要可持久化地维护。重要还意味着精确，每个访问的人都应该是一致的。而人气这个值就算丢了就算不一样都无所谓。所以价格需要用久经考验的数据库，而人气随便存存就行了，甚至不用管锁和同步甚至可以在内存里写。成本是不一样的。

每次点击都会导致淘宝内部数以千计的服务器的协作。

Each click needs thousands of servers to cooperate



Operating system:

- Linux (in OS class)

Serving the requests: web server

- Apache, Nginx (in ICS)

Serving the data: file system & DB

- MySQL & inode file system (in CSE)

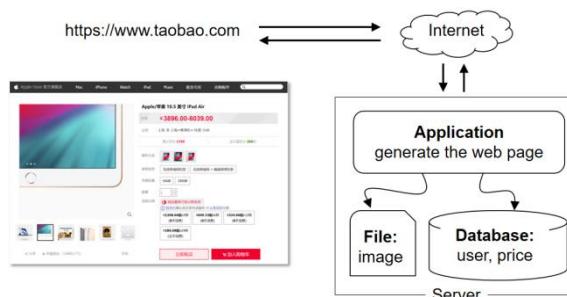
Displaying the page: HTML

- PHP (in Web class)



LAMP = Linux + Apache + MySQL + PHP

在很久很久以前，使用 LAMP 机制，是建站的首选。



访问数据的时候就 **database**，访问文件（图片等的时候）就是 **file** 系统。这个架构有一个问题，就是不可伸缩很难去扩展。内存最多到 256GB，硬盘最多到 40T，但是像 Facebook，每周有十亿张图片，显然单体是做不到的，同时 CPU 的性能也是有极限的。

当时淘宝的选择：

1. 投靠 IBM。
2. 改软件架构，要扩展就堆积便宜的服务器。

Step #1 for scalability: disaggregating application & data

Application: handles the application logic

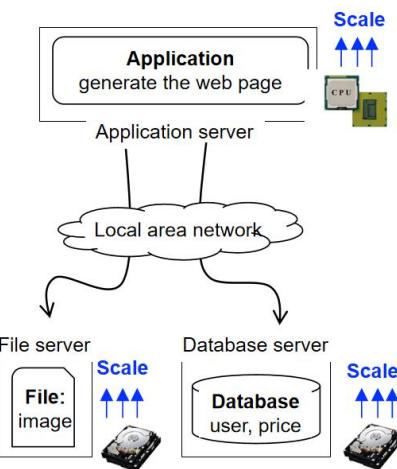
- Can be scaled with more CPUs

Database: requires reading/writing disk & cache

- Can be scaled with faster disk & larger memory

File system: store large bulk of data

- E.g., images, videos
- Can be scaled with faster disk



第一步:用三台取代一台，一台专门用 webserver，一台 fileserver，一台 database。可扩展性比原来的要好。有了这个方案之后，紧接着有了一个问题，发现走网络和走本地磁盘的时延差别非常大，另外一个是 database，因为 database 越来越大，查询变得非常慢。

Step #2 Avoid the slow data accesses? Caching

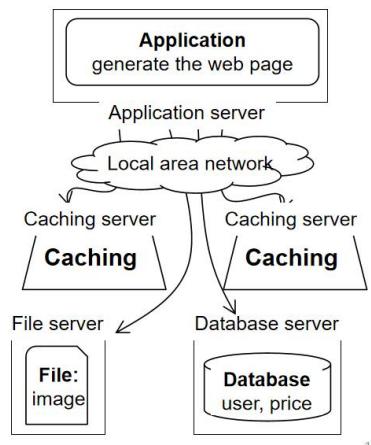
Observation: most requests are accessing **a small portion** of the data

Caching system with a single node:

- E.g., page cache
- Drawbacks: **limited** DRAM capacity

Distributed caching server

- **Benefits:** huge DRAM capacity, e.g., deploy many caching servers



想出来的办法就是加一个 cache，cache 是纯内存的，有了 cache 的时候就可以减少硬盘的操作。用什么去做 cache 呢，就是 redis。

Case study of distributed cache server: Memcached

Distributed caching server

- Benefits: huge DRAM capacity

Memcached server

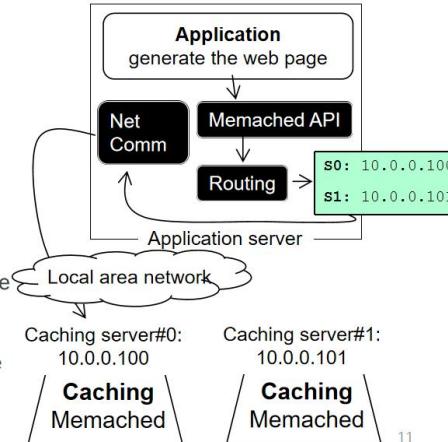
- Store all the cached data in memory
- Can scale out to use multiple servers

Memcached clients

- Check whether server has cached data
- On **cache miss**, **fallback** to database/file

Question:

- How to find which server has cached the data?



有了中间这一层，找到了就返回，没有找到就去 **database** 服务器上去找。我们怎么知道哪个服务器缓存了数据呢？下一个问题：内存和硬盘比起来量是少很多的。内存不够的时候，多弄几个机器，组成一个 **Memcached** 的集群。我们该怎么知道什么数据在谁那儿呢？可以把 **key** 算一个哈希，哈希的结果是 1 到 10 对 11 取余。

但是如果我们要加一台服务器，哈希是 1 到 3，加一台变成 1~4，有 75% 的数据要去做转移（**miss rate** 搞达 75%）。怎么样把数据重新分配的 **miss** 率降低呢，用到就是一致性哈希。最后用一致性哈希把 75% 的 **miss** 率降低到了 25%。

Application 成为瓶颈。

Step #3 for scalability: more servers

For **stateless** application servers

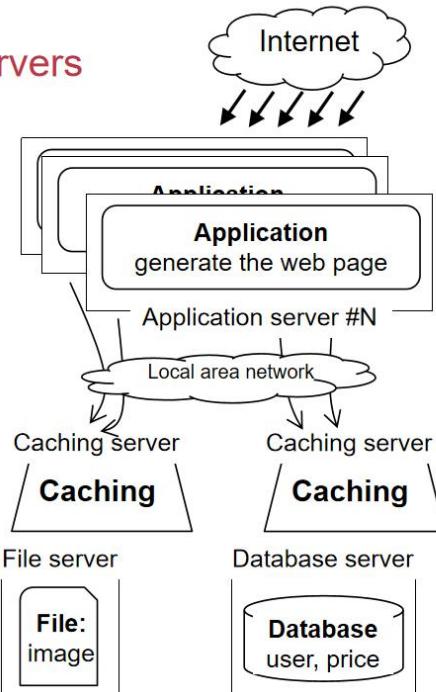
- E.g., web servers
- We can add more servers for scaling its performance

What is **stateless**?

- The server only executes the logic that only relies on input data but no long-term state

Benefits:

- Better fault-tolerance
- Better elasticity



购买更多的 applicaiton server，我们希望底下所有的 filesystem 不用在乎上面有多少台 application server，我们希望即插即用，我们就需要把 application server 做成无状态。

Webserver: 解析 url，找到文件，发回给用户，在这个过程中没有状态。所有状态都在文件服务器和数据库中。如果一个 application server 突然断电了，那么第二次就会找一台别的 application server 随时随地可以被取代，可扩展性变得非常容易。下一个问题是，有很

多台 application server, 怎么告诉用户呢?

Step #3 for scalability: How to do the load balance?

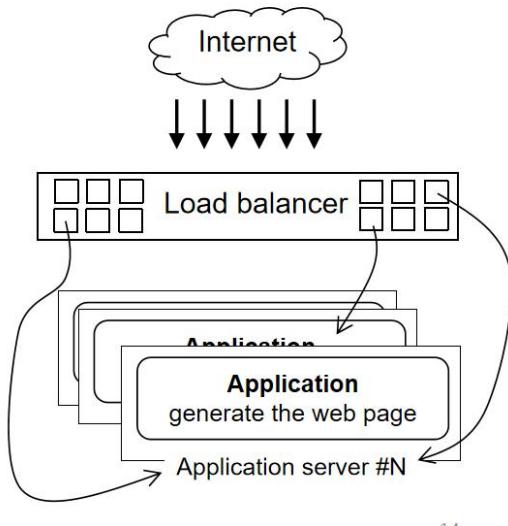
Load balance: how to route the user requests to the application servers?

Leverage the network layer

- use HTTP redirection,
- use reverse proxy,
- ...

Load balance algorithms

- round-robin
- random
- hashing
- ...



首先要有一个 load balancer, 统一进入网站的接口。一直做这样的分发呢, 谁挂了谁没挂就都可以通过定时心跳包知道。它也有很多算法, 轮训、随机、哈希, 这种策略在不同的场景中会有一些差异。

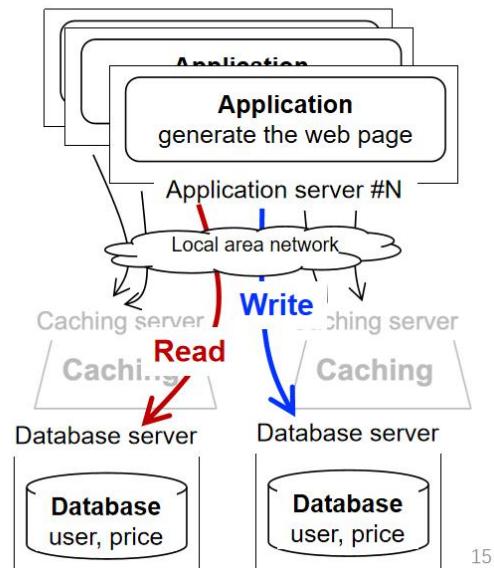
Step #4 for scalability: scaling database

#1. Separate the database for read/write

- E.g., use primary-backup replication
- The primary servers the writes and backup only serves reads

#2. Separate a single table on multiple databases

- e.g., a bank has reported that a single table has **100,000,000 rows**
- However, split a table on multiple machines **complex consistency management**



15

我们现在把 application server 变多了, 此时 database server 有了瓶颈。所以我们就要去有更多的 database server, 如果我们简单地买两台 database server 存一模一样的数据, 性能不会变好, 但是读的时候就会更好。所以想到一个办法就是读写分离, 有一个 primary database 只有一台用来写, secondary 可以有很多台用来读。写的时候只要写一台, 这个 primary 会逐渐把新的值发给很多台。所以主从分离。

但是还不够, 因为我们发现有些时候一张 table 可能非常非常大, 一个 bank 一张 table

有一亿行，一台机器存这个 table 就会有问题。有办法就是把这个 table 拆分成很多个 table，比如十台机器保存一张 table。这个拆分需要复杂的一致性管理问题。

当我们把 database 的 server 从一台变成多台，file server 也需要变成多台。

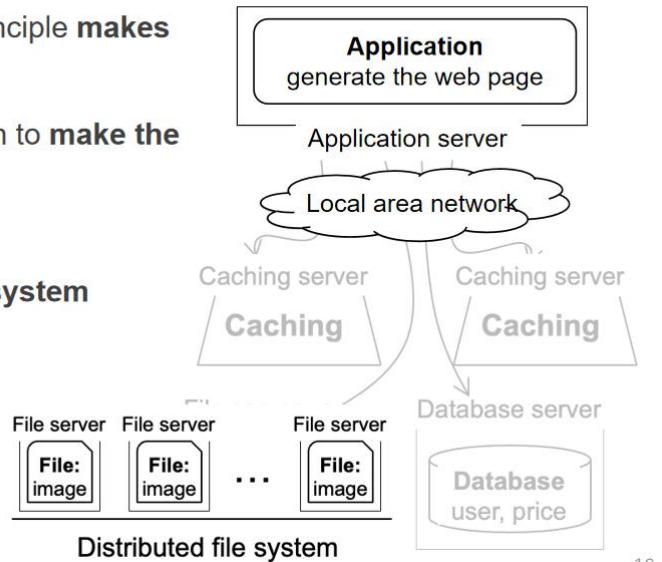
Step #5 for scalability: using distributed file system

Using multiple database in principle makes the database distributed

We can use a similar approach to make the file system distributed !

Well-known distributed file system

- NFS (Network file system)
- GFS (Google file system)
- HDFS



16

需要变成分布式的文件系统，eg:NFS,GFS,HDFS。好处文件对外的接口是统一接口，内部的文件存储形式是不可见的，1台和100台对外其他服务的暴露形式是相同的。

现在三个都有了，一部署问题又来了，服务器在杭州，从广州访问就很慢。

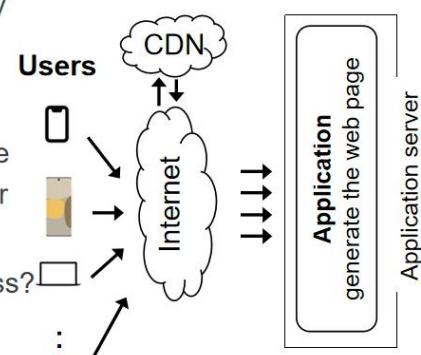
Step #6 for scalability: using CDN

Goal: return the results to user as soon as possible

- Why? Amazon has reported that 100ms latency increase will cause 1% financial loss

Core idea: caching (again)

- E.g., CDN (content delivery network) caches the content at the network providers, which is closer to users
- Challenge: how to do it without users' awareness?

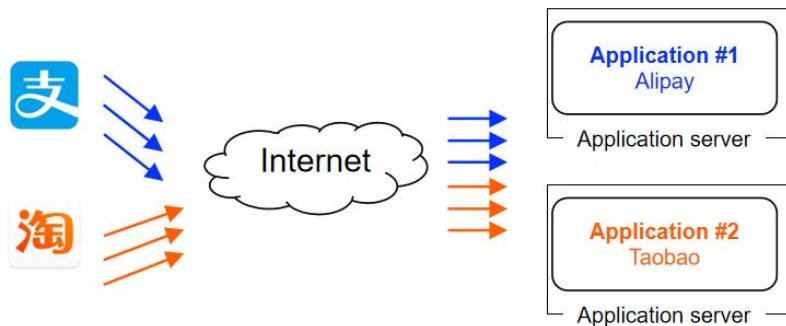


我们要在全国的一些地方租一些服务器，把照片这种数据缓存在这些服务器上，第二次广州的临近地区的访问就可以走临近的 CDN 服务器，淘宝也可以通过主动推送图片到 CDN 上。有了 CDN cache 之后，在访问 database 的时候依旧可以从淘宝的服务器上下载，在访问视频的时候就可以从 cdn 上去访问。在淘宝的 webserver 中，图片其实是 CDN 的地址，发到我们计算机上以后，再会从获取的 CDN 路径上下载图片。我们可以进一步地减少时延。

Step #7 for scalability: separate different applications

Use dedicated servers for different applications

- E.g., micro-services

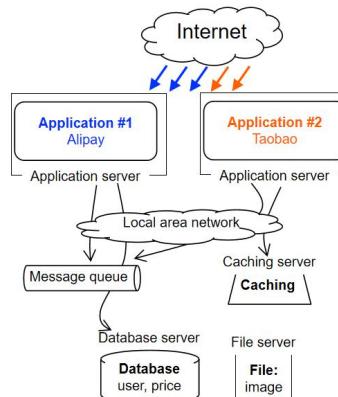


前面这套机制，所有程序要运行不太一样的应用程序的时候就会出问题，比如支付宝和淘宝用一样的 webserver 就不会更稳定。专用服务器性能会高，这个结论来自于淘宝的结论。可能的推论：可能和实际的 workload 有关。

How different applications communicate? MQ or DB

Message queue (MQ): applications send the message to the queue, and the queue can buffer the message (somewhere between RPC and database)
– E.g., Apache Kafka

Or, applications can directly use the databases, caching (e.g. KV) or file system to communicate with shared data



19

这个分开会把支付宝的代码拆分成微服务，可以发现哪些服务需要的资源多少。可以有针对性地配置不同权重的资源。可以进一步分拆，变成微服务运行。原来的交互就变成了微服务之间的交互。一类是 MQ，MQ 是很神奇的东西，是专门用来实现微服务之间的交互。MQ 专门做了一些优化使得两个微服务之间的交互更加快速和高效。也可以通过 database 或者 file system 做通信，但是效率不高。

How to handle complex requests?

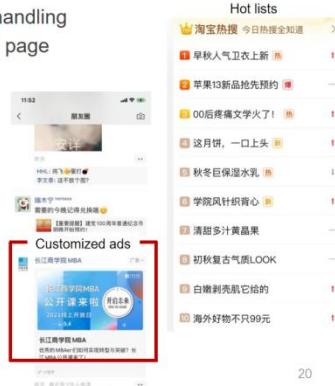
Example: after the website become larger, handling requests is not only displaying a (static) web page

Alipay (支付宝)

- E.g., fraud detection

Taobao

- Hot list (热榜)
- Dynamic product ads (千人千面)
- Recommendation (you might also like)
- ...



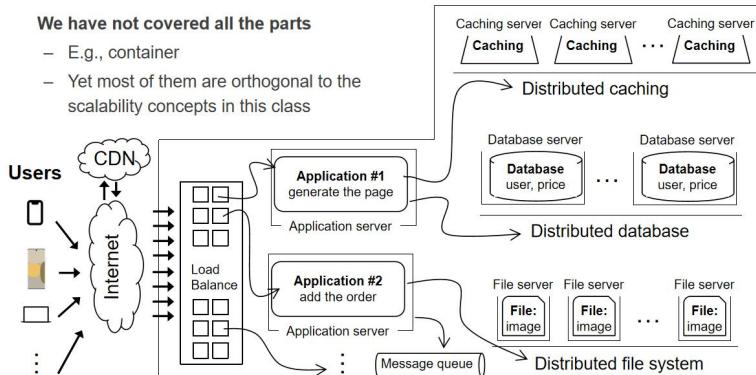
20

接下来，当网站变得更大之后，不再是从数据库或者文件库中拿一些拼在一起。比如淘宝热搜是怎么出来的。**Naive:** 搜索数据库的痕迹排序。需要用到 map-reduce 的分布式计算框架，每次用淘宝都会搜几十遍，一台机器肯定做不了，在上层 要有一个 framework 去处理，淘宝热搜的变化频率不会那么快。

还有一类是千人千面，不同用户的淘宝首页是不一样的，最简单是最近搜过的和推荐的东西。后台有一个记录你所有搜过东西的数据库，根据这个数据库动态的把推荐东西放到你主页上。推荐就是典型的离线计算过程，如果两个用户经常买的东西类似，一个买的可能是另一个人也想买的。所以这个可以提前算好，登录上来以后直接推送给它。

不一样的是，支付宝在点击支付的时候，支付宝做了非常多的工作。比如信用卡套现。当支付宝处理支付的时候，需要识别出交易是不是合法的。如果交易链形成一个圈，那就涉嫌套现了。支付宝需要预定义一些非法的 pattern，把支付行为和非法的 pattern 做一些匹配，比如冻结、打电话等。对于离线的推荐算法，就是 map-reduce 或者 spark。对于信用卡欺诈，需要用到图计算 graphlab、pregel。

A scalable website: overall picture



这个大的分成很多模块，层次偏 system。之前提到的 map-reduce 等是建立在这些上的一些框架。

分布式系统(Distributed system): a collection of independent/autonomous connected through a communication network working together hosts to perform a service

Large server and storage farms

- CPUs: > 20,000,000 cores
- Disk capacity: > 1,000,000 GB

Large-scale distributed systems: 10K – 100K servers

- Each rack: 40 servers
- Network: 10Gbps – 100Gbps in rack
- 10-100 MW of power



大的数据中心的数据，在全球不同的地方都会有数据中心。

那么这么复杂的系统组成在一起最大的挑战是什么呢？

最大的挑战就是 fault

Fault is common: fault, error, failure

Fault can be latent or active

- if active, get wrong data or control signals

Error is the results of active fault

- e.g. violation of assertion or invariant of spec
- discovery of errors is ad hoc (formal specification?)

Failure happens if an error is not detected and masked

- not producing the intended result at an interface

Fault 是错误的原因。Error 是 fault 的结果，failure 就是很大的失败。当我们的系统不出错的情况，一千万台机器都不出错是不太可能的。因为总体中有一台机器的出错率是随着机器数量指数增加的。

Fault is common especially in large distributed systems What are the causes?

Why faults are common especially in distributed systems? Scale!

- "Suppose a cluster has ultra-reliable server nodes with a stellar mean time between failures (MTBF) of 30 years (10,000 days)—a cluster of 10,000 servers will see an average of one server failure per day."

Causes:

- Operation error (human, configuration, etc.)
- Software error (e.g., bug)
- Hardware error
- Power outage
- Natural disaster



Fault:人、软件 bug、硬件问题、断电、自然灾害

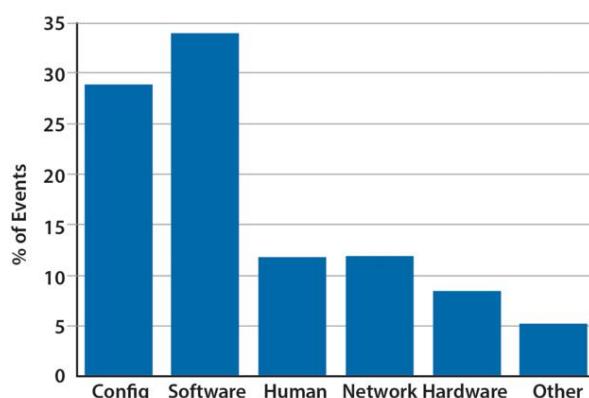


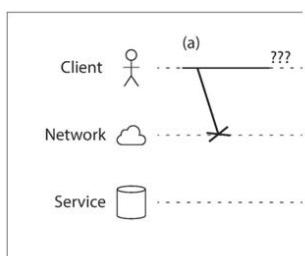
图 1 错误分布

我们不可能打造一个所有机器不出错分布式系统，我们希望用不可靠的组件组合在一起组成一个可靠的分布式系统。我们希望同时断掉 90% 机器的电，其他的机器还能 work。这就是容错能力，但是又有一堆问题。

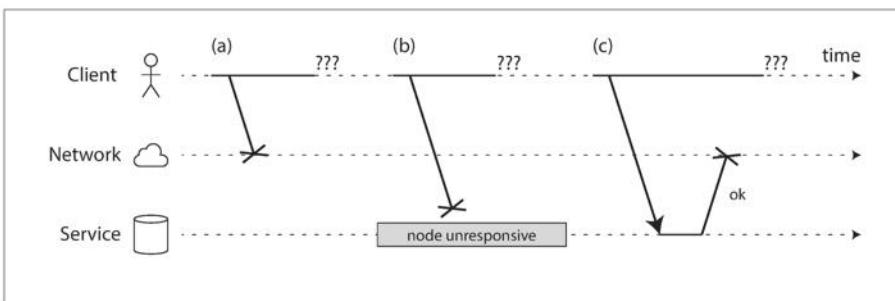
Faults and partial failures

Example: unreliable network

- A client (e.g., Smartphone),
- through network (5G, Wifi),
- sends requests to the service (e.g., Taobao).
- The client gets: “网络竟然奔溃了”, how can a network break?



36



当一个用户发送数据到 service, 原因

1. 网络丢包
2. 请求在网络上排队（路由器包满了在排队，但是你的耐心到了）
3. 远端服务器可能挂了
4. 远端服务器没挂，但是暂时停止了响应，java 的内存回收机制 jc，需要 jvm 暂停住。
5. 其实处理了操作，但是在返回的时候挂了，比如连点了两次处理了两笔转账。

Lamport 提出了很多很多分布式理论，2013 图灵奖。

MTTF、MTTR、MTBF

对于可靠性的评价方法，可以用可靠性，有几个 9，也就是 99.9%，每年允许 8 小时网站不 work。

MTTF 就是 mean time failure，连续多久不宕机。MTTF 时间非常长不意味着 availability 就会好。比如 debug 了一个 bug 一个礼拜。同样，每分钟出一次错，每次修复一毫秒。

MTTF：连续多久不宕机的平均时间。

MTTR：出错以后修复的平均时间。

MTBF：两次出错的平均间隔时间。

$MTBF = MTTF + MTTR$

附录：各大模型计算方法

摘自：[https://sars.org.uk/BOK/Applied%20R&M%20Manual%20for%20Defence%20Systems%20\(GR-77\)/p4c06.pdf](https://sars.org.uk/BOK/Applied%20R&M%20Manual%20for%20Defence%20Systems%20(GR-77)/p4c06.pdf)

首先明确原子模块的 MTTF 为 $MTTF = \int_0^{INF} R(t)dt = \frac{1}{\lambda}$, 其中 $R(t)$ 是 system reliability。

对于串联的情况：MTTF 是各项倒数和的倒数

 Series System	$MTTF = \frac{1}{\lambda_s} = \frac{1}{\sum_{i=1}^N \lambda_i}$	N Unequal Blocks
	$MTTF = \frac{1}{\lambda_s} = \frac{1}{N\lambda}$	N Equal Blocks

对于最低执行要求是 M 个，目前有 N 的组件的情况：

 Active Redundancy	$MTTF = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \frac{1}{\lambda_1 + \lambda_2}$	Unequal Blocks $M = 1, N = 2$
	$MTTF = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \frac{1}{\lambda_3} - \frac{1}{\lambda_1 + \lambda_2} - \frac{1}{\lambda_2 + \lambda_3} - \frac{1}{\lambda_3 + \lambda_1} + \frac{1}{\lambda_1 + \lambda_2 + \lambda_3}$	Unequal Blocks $M = 1, N = 3$
	$MTTF = \frac{1}{\lambda_1 + \lambda_2} + \frac{1}{\lambda_1 + \lambda_3} + \frac{1}{\lambda_2 + \lambda_3} - \frac{2}{\lambda_1 + \lambda_2 + \lambda_3}$	Unequal Blocks $M = 2, N = 3$
	$MTTF = \frac{1}{\lambda} \left(\frac{1}{N} + \frac{1}{N-1} + \frac{1}{N-2} \dots \frac{1}{M} \right)$	Equal Blocks $M & N$ general, see also Table 4 for $N \geq 6$.

接下来是 MTBF 的一张表

Reliability Block Diagram	System MTBF (m_s)	Conditions
 Series System	$m_s = \frac{1}{\sum_{i=1}^N \lambda_i}$	N Unequal Blocks
	$m_s = \frac{1}{N\lambda} = \frac{m}{N}$	N Equal Blocks, each with MTBF = m
 Active Redundancy	$m_s = \frac{A_s}{A_1 Q_2 \lambda_1 + A_2 Q_1 \lambda_2} \quad (Q = 1 - A)$	Unequal Blocks $M = 1, N = 2$
	$m_s = \frac{A_s}{Q_1 Q_2 A_3 \lambda_3 + Q_1 A_2 Q_3 \lambda_2 + A_1 Q_2 Q_3 \lambda_3}$	Unequal Blocks $M = 1, N = 3$
	$m_s = \frac{A_s}{K}$ where $K = A_1 A_2 Q_3 (\lambda_1 + \lambda_2) + A_1 A_3 Q_2 (\lambda_1 + \lambda_3) + A_2 A_3 Q_1 (\lambda_2 + \lambda_3)$	Unequal Blocks $M = 2, N = 3$
	$m_s = \frac{A_s m}{M_N C_M A^M Q^{N-M}} \quad \left({}_N C_M = \frac{N!}{(N-M)! M!} \right)$	Equal Blocks $M & N$ general, see also Table 6 for $N \geq 6$.

$$A = \frac{MTBF}{MTBF + MTTR}$$

有公式: $MTTR_{system} = \frac{MTBF_{system}(1 - A_{system})}{A_{system}}$

2021/9/23

我们回顾一下上节课介绍了当我们要构造一个淘宝、支付宝这样的网站的时候，单单一台机器是不够的，CPU,硬盘，内存都做不到线性扩展。所以我们要把系统进行拆解，变成分布式的组件，只有每一块都变成分布式的时候才会便于扩展。但是加入越多，机器越容易出错，比如谷歌 20%的机器每年会错误。

网络连接是分布式系统中独有的错误，单机中就是组件之间的内存连接，这个故障率相对比较低。所以我们要去做容错，为了衡量容错指标，我们提出了 **availability** 和 **mttf**, **mttf** 越长越好，也就是单次运行时间不出故障的平均时间。但是这不意味着可用性越高，一旦出错，修复的时间的不可用的。所以我们需要另一个指标，也就是一短时间之内的可用性的比例。这两个指标是互相正交的。

事实证明，我们可以得到高的可用性。微信、百度这种我们很少遇到出问题的情况，说明它的 **mttf** 是相当长的。在技术上怎么实现高的可用性呢？

就是 **redundancy**，也就是冗余，单台机器挂了之后其他机器可以顶上，我们不是为了构造一台永远不会出错的机器，而是一旦挂了就替换出去。接下来的一个问题就是 **consistency**，一台机器挂了以后，我们怎么保证数据是相同的。

Achieving high availability: handling failures

Redundancy

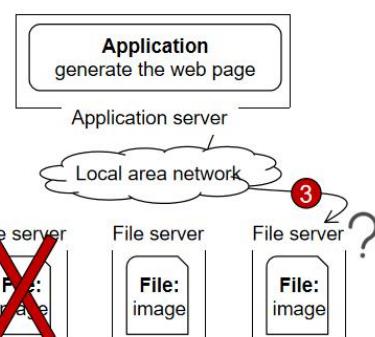
- E.g., replicas: identical multiple copies

Example: replicated file servers

- As long as one copy survives, the application is available

Challenge: consistency

1. Application put file **A** to one server
2. The server crashed
3. What happens when read **A** again?



发送文件 **A** 后，服务器就 crash 了。我们可以设计如下的一些技术。**Replicated state machine**，三台服务器永远保持一样，接收输入一样，运行逻辑一样，最后输出一样。

Techniques:

- RAID (Redundant Array of Inexpensive Disks)
- Primary-backup replication
- Replicated state machine

一旦出错，我们可以 **reconstruct** 或者重启，我们可以把一份数据备份在三台上，也有可能一台是完整的数据，而其他几台只记录下操作。这样就可以提高 **replica** 的性能。

还有一个问题是是不是一定要保证 **consistency**，这个问题真的这么严重吗？在很多

场景下，**consistency** 不一定是一个 0 和 1 的问题。Google search 同一个关键词第一次搜和第二次搜一定是相同的结果吗？可能某次会换一下顺序或者有几个没有之类的。人的认知并不是关注是不是同一个结果，对于这种情况下，我们可以进一步放松 **replica** 之间的同步，从而进一步提高并行的性能。所以我们需要对 **consistency** 有一个定义，从而明确约束。

最强的约束：一个人往一个分布式系统中写一个数据，所有其他地方也能读到相同的数据，但是这个约束太强了会影响性能。

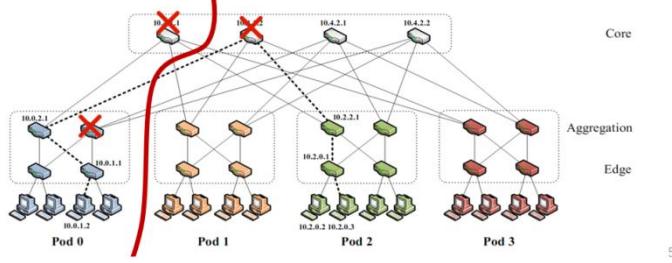
另一个 network failure，

Another common fault: network partition

A **network partition** refers to network decomposition into relatively independent subnets

- Can happen when a switch is being upgraded in a datacenter

For example, suppose the datacenter adopts a **fat-tree** topology



51

有些网络挂了，有些网络没挂。为什么会导致这些问题呢，因为这里有网络设备（路由器和交换机），这几台发生了错误，直接切断了左边部分和右边部分的连接。那么产生 **network partition** 在数据中心也是很常见的，更不用说在广域网络里，会出现更多五花八门的原因。比如海底光缆可以被鲨鱼咬坏。可能直接造成两个大陆之间的某几条网络链路中断。因为不是所有链路都中断了，所以并不影响 **availability**。

A **network partition** refers to network decomposition into relatively independent subnets

- Can happen when a switch is being upgraded in a datacenter
- Can even happen when being **attacked by sharks**



如果出现了 **partition**，就出现了若干个独立的子网，当前是没有连接的。未来修好了以后就可以重新连上，就出现了 **cap** 问题。

CAP 定理

► The CAP theorem: 2 out of 3

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response about whether it succeeded or failed)
- **Partition tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)

CAP 定理：一致性（Consistency）、可用性（Availability）、划分的容错（Partition）三者只能选二。

Partition tolerance: 就算网络被鲨鱼咬了，一个网络变两个网络了，也能正常运行。CAP 这三个同时只能满足两个。

我们可以用下面这个例子来简单的证明一下：

亚马逊的业务有两个区域：美国区和欧洲区。美国的用户连着美国区，而欧洲用户连接欧洲区。

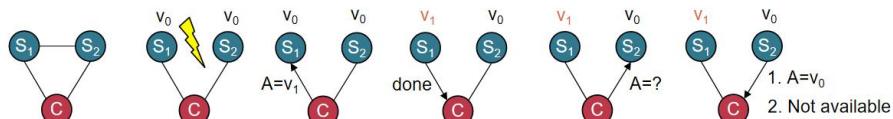
我们定义可达性和一致性：

一致性：所有用户看到的商品剩余数量是真实的。

一致：用户看到商品库存为 1，实际上已经没有了。

可达性（Availability）：所有用户可以在任何时间购买还有库存的商品。

不可达：用户可能得到提示“当前暂时不能购买商品”。



假设我现在有三台服务器，本来之间都是互相连着的，突然 S1 和 S2 不能连了。设置 $A=v_1$ ，这时候 C 再问 S2 要的时候，要么回复 $A=v_0$ ，要么回复不知道 A 等于多少。

- 如果回复 $A=v_0$ ，那么 CAP 中的 availability 是满足了，但是 consistency 被牺牲了。
- 如果回复了 not available，那么这个情况下，牺牲了 availability，保证了 consistency。

很可惜，在 CAP 中，我们能牺牲 P 吗？这个很难，因为 P 这个东西不是你可以决定的。一铲子把光纤挖断掉是不可预估的，所以 P 往往是一个前提，所以我只剩下 AP 和 CP。

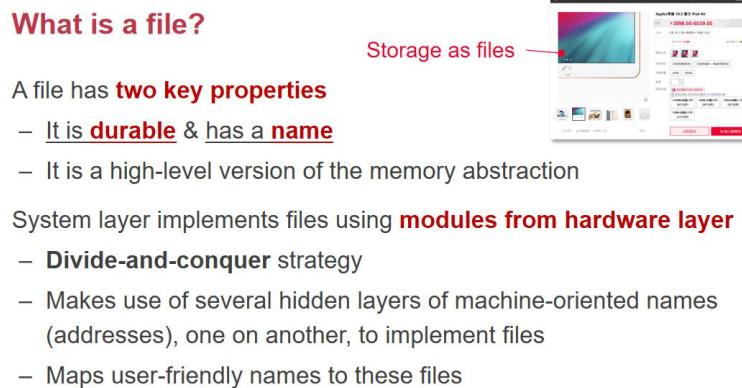
如果我们选 AP 放弃了 C，对于淘宝这种场景是可以接受的。淘宝牺牲了 C，最坏情况下是一本书卖给了两个人。支付宝选的是 CP，最后付了多少钱一定是一样的。而淘宝选的是 AP，因为要优先保证可用性，不一致的情况下相对可以接受，最多就是货量不足不发货了。

在 Amazon 上有一个功能是一键购买，也就是没有付款流程，点一下就能买好了。也就是亚马逊利用用户那一瞬间的冲动，所以亚马逊是不可能放弃 A 的。所以我们要考虑业务场景下，来选择 AP 和 CP。在 12306 一张票卖给两个人，这是不能接受的，所以需要 CP。

对于一个大的系统中，其中每个 component 都需要仔细研究。

inode-based file system

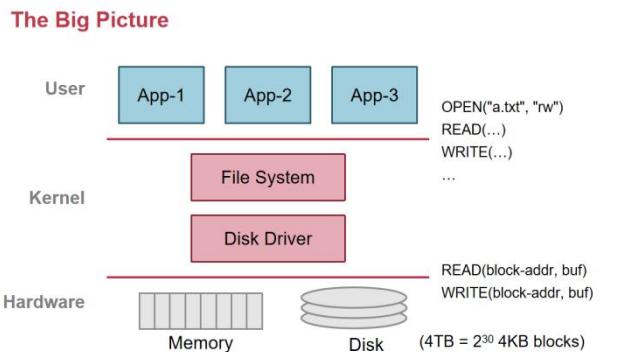
在单机上文件系统的实现是 inode-base file system，我们先对单体文件系统解剖，最后再扩展到分布式文件系统。



什么是一个文件？文件似乎已经成为了一个很难解释的基本术语。文件和数据有什么区别？

第一个文件（**durable**）：它是一段放在存储设备上的可以保存的数据。它不会因为你关机而消失。

第二个特性：文件是有名字的，我们可以去命名一个文件，我们也可以通过一个名字去指定对应的文件。



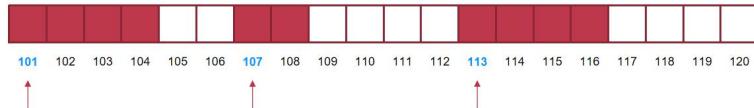
系统对于文件来说，这两个特性都是为了让程序员和用户在使用的时候更方便一点。在一个大的 **bigpicture** 里面，内核在磁盘和内存之上，抽象出文件的 **abstraction**，也就是要发明出文件的接口。把硬件提供的接口，经过 **os** 的组织和包装变成一组新的接口，用户使用起来更便捷的接口。

文件提供什么 **api** 呢？Open，读写文件，关闭文件，同步文件到磁盘，文件的元数据，可读可写可执行，改变文件拥有权，创建目录，改变目录，mount 和 unmount。

A Naive File System

Each file occupies one continuous range of blocks

- Use **block index** as file name, e.g., 107, 113
- Every file write will either **append** or **reallocate**



What are the **problems**?

我们先来看一个简单的文件系统怎么设计。一个磁盘就是一个很大的数组。每个 **block** 早期是 512 字节，后来变成了 4k。如果我现在有一个文件要放在磁盘上，既然放在磁盘上了，肯定就有 **durable** 特性了。

名字这么设置？我们可以把文件顺序的放在磁盘上，存放的第一个 **block** 存放文件的名字。这种简单的设计有什么问题呢？

1. 因为连续存放，增删后存在碎片
2. 文件 **copy** 到另一个磁盘以后，可能就不是 107 编号了。因为名字和磁盘本身耦合的太多了。

我们需要在磁盘之上，做层层软件的抽象，把它变成容易使用的。

The Naming Layers of the UNIX FS (version 6)

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.	↓

分为 3+3+1，上面三层让人来用

1. Block Layer

1. block layer，你给我一个 **block number**，我给你一个 **block data**，这是磁盘提供的。我们

要 107 号，你给我 4k 数据。但是显然不够，我们需要知道每个 block 多大，是 4k 还是 512b；哪些 block 被用了，哪些是 free 的；有用的数据从什么地方开始；

Super Block

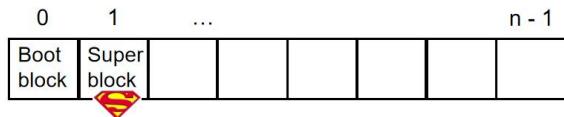
Block
num

One superblock per file system

- Kernel reads superblock when mount the FS

Superblock **contains**:

- Size of the blocks
- Number of free blocks
- A list of free blocks
- Other metadata of the file system (including inode info)



所以我们就需要在 block 这一层放一个特殊的 superblock，它一般来说第一个是用来启动的。Superblock 中包含了 block 的大小，free block 的数量，所以 superblock 中记录了很多原信息。它怎么记录一个 block 是用了还是没用呢？它会预留一块用来记录 bitmap，这就是一串 010101010,0 表示 free，1 表示 used，每个 01 就是针对了一个后面的 block 是用了还是没用。Block size 是一个 trade-off，软件可以修改 block size，如果这个 block size 太小了。这意味着磁盘分的太碎了。太大的话就容易浪费掉。需要合理设置长度。

L1: Block Layer

B
n

Block size: a trade-off

- Neither too small or too big

Question

- What will happen if the block size is too small? What if too big?
- How to efficiently track free blocks?

Use a bitmap



接下来就是记录哪些是 free 的。如果我们每个 block 前面加了一位，我们的 block 是 4095 的 size，导致读取到的数据是不对齐的。所以我们在这个文件系统内我们是使用 bitmap 来存储这个信息的。4k 的 block 可以存放 $8 \times 4096 = 32K$ 个 block 的是否使用过。所以一个 4K 的 bitmap 就可以对应到 $32K \times 4K = 128M$ 的磁盘空间，效率还是很高的。有了这些以后，我们就有了第一层。

2.File Layer

第二层：一旦有了 block 之后，如果我们保证每个文件都小于 4k，我们就可以直接使用 block_id 作为文件的 name，但是文件必然比 4k 大，一个文件占 block 的数量就会超过 1.

planA:文件连续存放

L2: File Layer

File
(inode)

File requirements

- Store items that are larger than one block
- May grow or shrink over time
- A file is a linear array of bytes of arbitrary length
- Record which blocks belong to each file

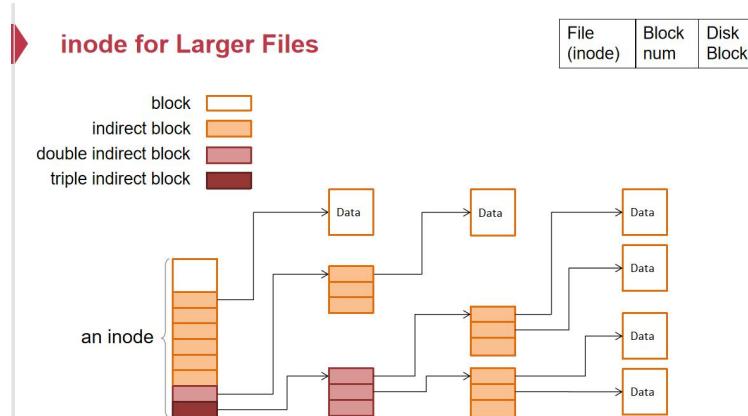
inode (index node)

- A container for **metadata** about the file

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

planB：文件不连续存放，这个效率更高。但是怎么记录不连续的 block number 呢？我们就引入了 index node。假设文件包含了 8 个 block，如果是连续的就记录第一个 id+长度，如果不连续，我们把 8 个 block id 记录在 index node 里。这里面包含了 block number，这个 N 的大小很有讲究。如果这个 size 记录的是 block 的数量，文件的大小必须是 4k 的整数倍，肯定有问题。所以这个 size 是文件的 byte 数量，最后一个 block 可能没用完。

如果我们的 N 是一个定值，那么 Inode 支持的文件的最大大小是多少呢？怎么提高呢？如果 N 是 1000，文件大小最多才 $1000 * 4K$ ，显然是不够的。我们不能用线性的数组的记录这个 block number。如果我们想支持 4G 大小的文件，那么 $4G / 4KB * 64B$ （磁盘的 block_id 的长度）=8M，为了支持一个 4G 的文件，光是存它的 index 就要用到 8M，如果一个 Inode 8M，如果我们用一半的磁盘存这个 Inode，比如这个磁盘 8T，用 4T 来存 Inode， $4T / 8M = 50$ 万个文件。我们计算机中的文件毫无疑问是百万级别，最大支持的文件才 4G，这些都满足不了我们的需求。



如果我们的文件没那么大，我们预留了一个 8M 的 inode，就本末倒置了。我们要预留成可伸缩的结构。我们把数据的前面几位设置成指向 block 的。后面几位指向一个二级的

block。其中每一项都是指向一个 block_index。二级的 block 是 4k，每一项是 8 个 byte，可以指向 512 个 block。这个和页表做的事情是很像的。如果二级也用满了，我们可以再加一级。那么就是 512×512 个指向 block 的 block_index。这样对于小文件，就只需要用到前面几个。对于大文件就可以一点点增长出来。

对于 layer2，其中可以作为 double indirect block 和 triple indirect block 对于第一层来说是无所谓的。

L2: File Layer

File (inode)	Block num	Disk Block
-----------------	--------------	---------------

Given an inode, can map a block index number (of a file) to a block number (of a disk)

- Index number: e.g., the 3rd block of a file is number 78

```
procedure INODE_TO_BLOCK(integer offset, inode i) -> block
  o <- offset / BLOCKSIZE
  b = INDEX_TO_BLOCK_NUMBER(i, o)
  return BLOCK_NUMBER_TO_BLOCK(b)

procedure INDEX_TO_BLOCK_NUMBER(inode i, integer
  index) -> integer
  return i.block_nums[index]
```



有了第二层之后，就有了 inode。页表的话所有统一是 4 级页表。要解决的本质问题其实也就是映射。有了第二层之后，只要告诉我们 inode 在哪，和需要数据在文件中的 offset，我们就可以找到对应 block 在哪，返回给用户。

3. Inode Number Layer

下面一个问题就是 inode 存哪呢？Inode 紧跟着 bitmap 存在后面，也就是 inode table。Inode table 的起点就存在 super block 中。有了 inode table 后，我们就有了一个 inode table number，它本质上就是一个数组，根据下标就可以得到 inode 的值。

Put Layers so far Together

Inode num	File (inode)	Blk nu
--------------	-----------------	-----------

```
procedure INODE_NUMBER_TO_BLOCK(integer offset,
  integer inode_number) -> block
  inode i = INODE_NUMBER_TO_INODE(inode_number)
  o <- offset / BLOCKSIZE
  b <- INDEX_TO_BLOCK_NUMBER(i, o)
  return BLOCK_NUMBER_TO_BLOCK(b)
```

inode number is sufficient to operate a file. However,

- inode numbers are convenient names only for computer
- inode numbers change on different storage device

A file needs a more **user-friendly name!**

Inode 也需要去记录哪些 inode 被使用了，哪些 inode 没有被使用。可以再记录一个 bitmap，或者在每个 inode 前加一个 bit，这也是可行的；也可以记录一个 free 的 inode 的链表串起来。

所以给了一个 Inode number 就可以得到文件的所有信息。这对获取文件是足够的，但

是它对于人类是比较难的。而且 copy 的时候，inode-number 是会变动的。

4.File Name Layer

所以第四层，我们需要一个 user-friendly 的 name，怎么把字符串和 inode-number 做一个映射呢，最简单数据结构就是一个 dict 映射。

File name

- Hide metadata of file management
- Files and I/O devices

Mapping

- Mapping table is saved in directory
- Default context: **current working directory**
 - Context reference is an inode number
 - The current working directory **is also a file**

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

Overview of inode content

File name	inode num
helloworld.txt	12
cse2021.md	73

```
procedure name_to_inode(string filename, integer dir) -> integer
    return LOOKUP(dir, filename)
```

- Max length of a file name is **14 bytes** in UNIX version 6 (what does it mean?)

这个表最后在硬盘上就是这么存的，它存在一个文件里，它是文件系统规定的格式，这个文件就叫做目录。也就是里面文件的名字是什么，文件的 inode-number 是多少。Foulder 这个词是很误导的，这个文件夹只是包含了其中文件的 inode-numer，而不是包含了所有文件内容。所以我们倾向用目录这个词，更加准确。

访问文件方式：给一个文件名，给一个目录，去目录里找这个文件名，找到对应的 inode-number，就可以再找到文件了。

这样就产生了 lookup 的方式：

LOOKUP in a Directory

File name	Inode num	File (inode)	Block num	Disk Bloc
-----------	-----------	--------------	-----------	-----------

```
procedure LOOKUP(string filename, integer dir) -> integer
    block b
    Inode i = INODE_NUMBER_TO_INODE(dir)
    If i.type != DIRECTORY then return FAILURE
    for offset from 0 to i.size - 1 do
        b <- INODE_NUMBER_TO_BLOCK(offset, dir)
        if STRING_MATCH(filename, b) then
            return INODE_NUMBER(filename, b)
        offset <- offset + BLOCKSIZE
    return FAILURE
```

Name comparing method: **STRING_MATCH**

- LOOKUP("cse2021", dir) will return 73

Next problem:

- **What if there are too many files?**

Overview of inode content

File name	inode num
helloworld.txt	12
cse2021.md	73

21

到这一步为止，整个文件系统都完备了，每个文件都有一个字符串的文件名。但是还有缺点，我们现在只有一个目录，这个目录下有很多很多文件（几百万个），ls 有好几个屏幕，都不能重名的。所以我们应该有一个 path name 层，如果目录也当做一个文件，那么目录就可以被包含在另一个文件里面。目录也有一个 Inode-number，目录和文件在 inode 这一层是没有区别的。就像 inode 和 data 在下层都是 block 一样。

5.Path Name Layer

L5: Path Name Layer

Path name	File name	Inode num	File (inode)	Block num	Disk Block
-----------	-----------	-----------	--------------	-----------	------------

Hierarchy of directories and files

- Structured naming: E.g. "projects/paper"

```
procedure PATH_TO_INODE_NUMBER(string path, integer dir) -> 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)
```

Context: the working directory **dir**

既然不区分，我们就可以有了 **path name**，我们可以创建 **projects/paper**，我们先去当前目录下找 **projects**，找到以后再以 **projects** 为目录找 **paper**。

接下来一个问题就是目录名太深了，要打好几层目录。在我们前面说的目录结构里面，并没有任何地方组织我们把一个文件名对应的 **inode-number** 设置成同一个。也就是不同文件名对应到同一个 **inode-number**。如果我们为一个 **inode** 创建多个 **name**，我们就可以为一个很深目录的文件名创建一个很短的文件名，最后对应的 **inode** 是一个。说明这个文件只有一个。

假设一个文件有两个名字，当我们删掉一个名字的时候，用另一个去访问是能访问还是不能访问呢？

什么是删文件：本质操作只是删除目录中的 **helloworld.txt** -> **12** 删除掉，并不是删除 **block** 中的数据，因为可能别的地方还引用了。所以我们要在 **inode** 中维护一个 **reference counter**，如果删除完 **reference counter == 0** 就可以删除了。同时我们还需要加一个 **type** 来标识是一个普通文件还是目录。

Links

Path name	File name	Inode num	File (inode)	Block num	Disk Block
-----------	-----------	-----------	--------------	-----------	------------

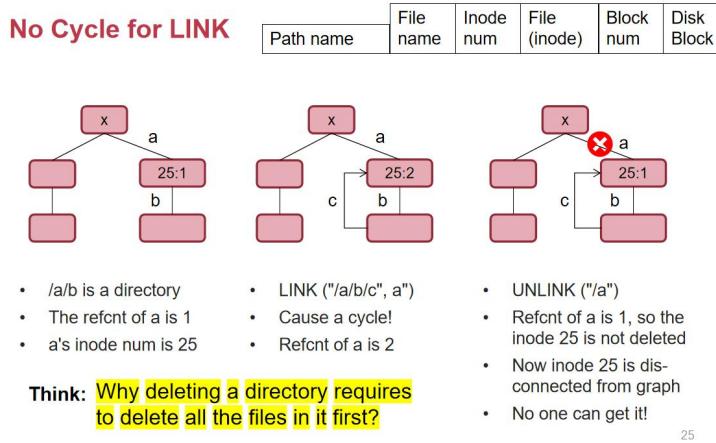
Reference count

- An inode can bind multiple file names
- +1 when **LINK**, -1 when **UNLINK**
- A file will be deleted when reference count is 0
- **No cycle allowed**
 - Except for '.' and '..'
 - Naming current and parent directory with no need to know their names

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

在 **unix** 的 **v6** 版本文件系统里面，有一个特点，虽然 **inode** 有 **ref counter**，但是目录不能有多个 **name**。点和点点的实现就是在表里加一个.**和..**，对应当前目录和上一级目录的

inode-number。我们能不能建立一个指向目录的 link 呢？也就是一个目录能不能在不同地方有多个 name 呢？



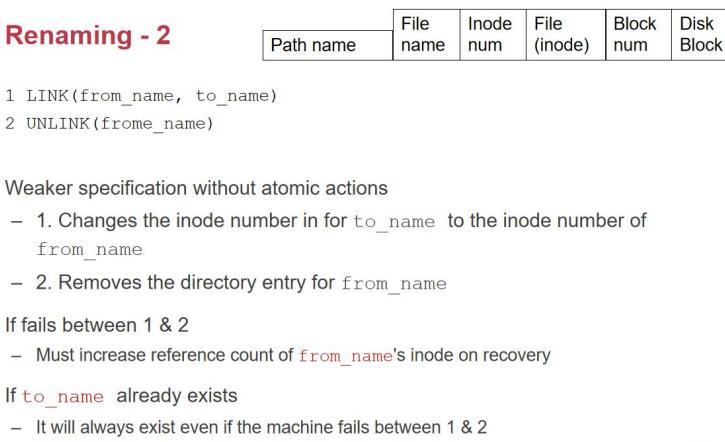
假设 inode 有个 x，记录了目录 a，目录 a 中记录了目录 b。假设我们现在来 link 把/a/b/c 对应到/a，也就是在 a 的子目录下建立了一个指向 a 的 link。a 现在有两个人指向它，也就是 /a 和 /a/b/c，它的 ref count 是 2。现在我们尝试删除/a，它的 ref 是 1，但是从 x 再也找不到 a 了。因为它唯一的名字被删除了。它既没有被删掉，又不能被另外的地方访问到。所以就会成为内存垃圾。所以禁止指向目录。

当我们删除一个非空目录的时候，其实我们是递归删除的，在真实的 linux 里面，我们已经考虑了这个问题了，但这个 rm -r 在 unix 第 6 版是不存在的。

当我们去 rename 的时候，改文件名是文件系统中非常复杂的事情。假设我们要去做改文件名的操作，比如 mv (from,to)，这一个指令牵涉到 3 件事情。

1. 把 to 删了，to 的 ref count 变成 0.
2. 把 from 加一个 to 的名字，from 的 ref count 变成 2.
3. 再把 from 的 ref count 变成 1

在这个过程中一旦发生了一次 crash。比如在 1 和 2 之间发生了断电，我们发现少了 to 这个文件。所以这就是我们 atomic (原子性) 的问题。原子性就是不管中间断电的什么情况，回复到正常情况以后，要么就是全做，要么就是没做，不存在中间的情况。

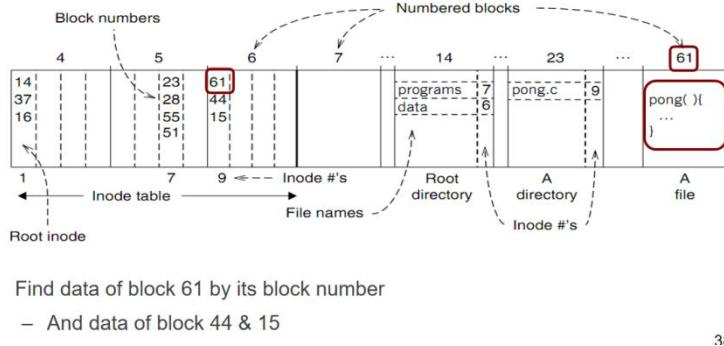


6. Absolute Path Name Layer

有了这个之后，我们现在有了目录了，一台机器有很多人要登录，我们看到我的文件，

你看到你的文件，文件之间是没有任何文件渠道去沟通的。为了解决这个问题，就引入了根目录，有了根目录之后，一台机器上的文件都可以根据根目录开始索引。根目录最后就定义为 /，根目录的 . 和 .. 都指向它自己。

An Example: Find Blocks of "/programs/pong.c"



这是一个磁盘，我们要找 /programs/pong.c

现在我们是 os 文件系统，我们先看根目录的位置。这是所有人都公认的地址。根目录首先是一个文件，有对应 inode，根目录的 inode 是 1，super block 记录了 inode table 的地址。我们就去看 1 所对应的 block。14 是一级的 block number。然后我们去读 block = 14 的内容，我们发现了 programs，找到 7 是 programs 目录的 inode-number，我们又回到 inode-table 中找到第 7 个。（一个 block 通常可以放 4 个 inode 左右），7 号里的是 block number，我们就去找 block number = 7，我们找到了 pong.c = 9，9 是 inode-number，其中保存的 61 又是 block number，我们找到 block = 61 即可。

Directly Dump a Directory

```
$ ls -ai temp
7536909 . 7530417 .. 7536939 a 7536940 b 7536941 c 7536942 d

$ echo "obase=16;7536909;7530417;7536939;7536940;7536941;7536942" | bc
73010D 72E7B1 73012B 73012C 73012D 73012E

$ sudo /sbin/debugfs /dev/sdal
debugfs 1.43.4 (31-Jan-2017)
debugfs: dump temp temp.out
debugfs: quit

$ xxd temp.out
0000000: 0d01 7300 0c00 0102 2e00 0000 b1e7 7200 ..s.....r.
00000010: 0c00 0202 2e2e 0000 2b01 7300 0c00 0101 .....+s....
00000020: 6100 0000 2c01 7300 0c00 0101 6200 0000 a...,s....b...
00000030: 2d01 7300 0c00 0101 6300 0000 2e01 7300 -.s....c....s.
00000040: c40f 0101 6400 0000 0000 0000 0000 0000 ....d.....
00000050: ...
```

我们可以列出 temp 目录的 inode 号是多少。我们变成 16 进制以后就是这个。我们可以用 debug 的方式打出这个文件。这就是目录在磁盘上的数据。这些十六进制的数是什么意思呢？

Directly Dump a Directory

```
struct ext4_dir_entry {
    uint32_t inode_number;
    uint16_t dir_entry_length;
    uint8_t file_name_length;
    uint8_t file_type;
    char name[EXT4_NAME_LEN];
}

File Type
0x0: Unknown
0x1: Regular file
0x2: Directory
0x3: Character device file
0x4: Block device file
0x5: FIFO
0x6: Socket
0x7: Symbolic link
```

0d01	7300	0c00	0102	2e00	0000
b1e7	7200	0c00	0202	2e2e	0000
2b01	7300	0c00	0101	6100	0000
2c01	7300	0c00	0101	6200	0000
2d01	7300	0c00	0101	6300	0000
2e01	7300	c40f	0101	6400	0000

↓

0d01	7300	0c00	0102	2e00	0000
------	------	------	------	------	------

0d01 7300: inode number
0c00: entry length is 12 bytes
01: file name length is 1 byte
01: file type is regular file
2e00 0000: file name (2e -> ".")

我们重新排序后就可以看出，每一行对应的就是文件名。2e 就是..，2e2e 就是...。0102 就是目录的意思。0c00 就是长度。c40f 就是最后一项到头了。前面红色的部分就是 inode-number。我们就通过这个实验看到了这个真的目录长这个样子。

2021/9/26

今天我们接着上一节课。上节课讲的是 **inode-based file system**，当一个大的分布式的系统需要支持成百上千的容量的话，需要把大部分文件变成分布式的。在讲分布式的文件系统，我们需要先介绍文件的分布式系统。上节课我们讲了 7 层中的 6 层，从 **block layer**，**Block layer** 就是给我一个 **block number** 给一个数据。**File layer**: 文件大小会超过 **block** 大小，需要用 **inode** 记录 **block number list**。有了 **inode** 之后就包含了文件的所有信息。**Inode table** 之后就可以给 **inode** 命名，用 **inode-number** 指代每一个具体的 **inode**。有了下面三层之后，整个文件系统已经成立了，但是光用 **inode-number** 做名字不是很友好。所以需要用 **inode** 和文件名对应，我们记录在单独的一个文件里。这个文件不是我们随意可以通过 **read-write** 修改的，因为这个文件是一个目录文件。它对于文件系统来说是元数据。对于 **Inode** 来说，目录所有的数据都保存在数据区，和一般的文件没什么区别。但是从用户的角度来看，目录就是一个元数据，不能通过 **open/read/write** 来修改目录，只能通过 **ls**。然后有了目录之后，再往上递归地树状组织结构。接下来还有 **link** 的概念，**link** 其实就是文件名。

```

$ touch a.txt

$ ls -il
40975357 -rw-r--r-- 1 xiayubin wheel 0 9 26 08:15 a.txt

$ ln a.txt a.ln
$ ls -il
total 0
40975357 -rw-r--r-- 2 xiayubin wheel 0 9 26 08:15 a.ln
40975357 -rw-r--r-- 2 xiayubin wheel 0 9 26 08:15 a.txt

What does "2" means? Reference count

$ echo hello > a.txt
$ ls -il
total 16
40975357 -rw-r--r-- 2 xiayubin wheel 6 9 26 08:17 a.ln
40975357 -rw-r--r-- 2 xiayubin wheel 6 9 26 08:17 a.txt

What does "6" means? Size

```

我们创建一个指向 a.txt 的 a.ln link。我们发现现在多了一个文件 a.ln。我们发现它们的时间、大小一起变了。所以上述的 2 是 reference count，当 ref cnt 变为 0 的时候，inode 和 data 就会被 free 掉。6 就是文件 size 的大小。

当我们知道 link 的时候，如果我们现在插了一个 u 盘到了电脑中。如果我们想在主机中创建一个指向 u 盘的 link，这本身是两个文件系统，inode-number、space 都是不一样的。我们不能直接使用 u 盘文件系统中的 Inode-number。所以不能创建跨文件系统的 link。

7. Symbolic Link Layer

而第七层 symbolic link layer，我们在文件系统中创建了一个新的文件，内容是一个字符串，指向需要链接的文件路径和文件名。它不会仅仅停留在打开 symbolic link 本身，它会读取出数据，并且指向进一步的操作。所以在 inode_type 中要添加 symbolic link。

```

$ ln -s "/tmp/abc" s-link

$ ls -l s-link
7536945 lwxrwxrwx 1 xiayubin 8 Sep 20 08:01 s-link -> /tmp/abc

$ readlink s-link
/tmp/abc

What does "8" means? File size

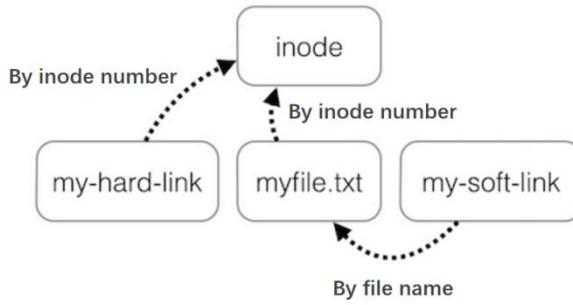
$ cat s-link
cat: slink: No such file or directory

$ echo "hello, world" > /tmp/abc

$ cat s-link
hello, world

```

我们创建 s-link 的信息的时候，我们发现是指向/tmp/abc 的。我们可以通过命令 readlink 来读出 symbolic link 文件的内容。8 是/tmp/abc 的大小，它其实不需要去存/0。如果我们尝试使用 cat 来读取数据的时候，会报错。当我们把 hello, world 重定向到/tmp/abc 的时候，才会读到 hello, world。所以这个告诉我们哪怕原先的路径的文件不存在的话，我们也可以为那个不存在的文件创建一个 symbolic link，但是对于 hard link 来说，我们不能为不存在的文件创建一个 hard link，因为 inode 都不存在。



我们可以通过上图来审视一下 hard link 和 symbolic link。Hard link 是通过 inode-number 绑定, 而 symbolic link(soft link)是通过路径和文件名来绑定。Symbolic link 是可以指向目录的, 而 hard link 除了.和..是不能指向目录的。

Sidebar: Notice the Context Change

Another interesting behavior of soft link

- There is a directory: "/Scholarly/programs/www"
- The root directory contains a soft link
 - "/CSE-web" -> "/Scholarly/programs/www"
- Run following commands
 - cd CSE-web
 - cd ..
- What is the current directory? Why?
 - ".." is resolved in a new default context: by bash, not file system

有一个/Scholarly/programs/www, 我们觉得太长了, 我们创建一个 symbolic link /CSE-web。我们使用 cd CSE-web; cd ..; 我们因为答案是/Scholarly/programs。事实上, 答案是根目录。

Sidebar: Notice the Context Change

The bash tries to be "**human-friendly**"

- When you cd /into/a/symlink/, the shell remembers the old location (in \$OLDPWD) and will use that directory when you cd .. under the assumption that you want to return to the directory you were just in

If you want to use the real .., then you must also use "**cd -P ..**"

The -P option says to use the physical directory structure instead of following symbolic links (see also the -P option to the set builtin command); the -L option forces symbolic links to be followed.

```

$ cd
$ cd a/b/symlink
$ cd -P ..
$ pwd -P
/home/sarnold
$
```

这是 bash 自己做的优化, 它认为这样跟符合直觉。

我们简单做一个总结, 前面 7 层里面有几个和直觉不符合的地方。

1. 文件名和文件是没有关系的。文件相关的数据分为元数据（存在 inode 里）和数据（inode 指向 block 中）。而文件名是存在目录里的。所以文件的文件名和文件并没有直接的关系, 这也是我们可以通过 hardlink 给一个文件赋予多个文件名。所以任何一个指向 Inode 的文件名是等价的, 新创建的 hard link 和原先的文件索引是等价的。

2. 以及目录的大小是很小的, 只和文件名的长短有关系。

讲完文件的结构以后, 我们已经知道了磁盘中静态文件是怎么组织的。我们接下来看文件是怎么被读写、访问、同步的。

这些操作都是通过 system call 的方式提供给用户的应用程序的。而用户通常还要通过 lib

的封装来调用。

我们之前学过 C, C 是不是一个跨平台的语言呢？我们 include 不同头文件的时候会有不同的环境。我们用 C 的 lib 是 fopen，在 linux 下是对应的 open 的 syscall。Fopen 是返回一个 file*，而 open 是返回一个 int 的 syscall。

File Meta-data

Owner ID

- User ID and group ID that own this inode (can be changed by chown)

Types of permission

- Owner, group, other
- Read, write, execute

Time stamps

- Last access (by READ)
- Last modification (by WRITE)
- Last change of inode (by LINK)

```
struct inode
    integer block_nums[N]
    integer size
    integer type
    integer refcnt
    integer userid
    integer groupid
    integer mode
    integer atime
    integer mtime
    integer ctime
```

userid 和 groupid 做访问控制。然后是可读可写可执行，还要三个 time, last access, last modification, last change of inode。

Mtime 说的是 data 改动，ctime 说的是 inode 改动（link 改动 refcount，修改权限）。

OPEN a File

Check user's permission

Update last access time

Return a short name for a file

- File descriptor (**fd**)
- fd is used by **READ**, **WRITE**, **CLOSE**, etc.

Open 文件的时候，先要看文件是归属于哪个用户的。匹配 userid 和 groupid，如果权限不匹配也不会让你打开，然后更新 last access time。然后返回一个 fd（文件被打开之后就会有 fd）。

File Descriptor

Each process starts with three **default open files**

- Standard in(stdin): **fd = 0**, standard out(stdout): **fd = 1**, standard error(stderr): **fd = 2**

Can also use fd to name opened **devices**

- Keyboard, display, etc.
- Allow a designer not to worry about input/output
 - Just read from **fd 0** and write to **fd 1**

Each process has its **own fd name space**

Fd 同样可以被命名打开的设备。比如我们打开一个键盘。读的话就是得到键盘正在输入的字母。每个 **process** 都有我们自己的 fd namesapce。现在提一个问题。为什么我们要用 fd 来作为返回的文件呢？为什么不返回 **Inode** 给用户呢？

Why File Descriptor?

Other options

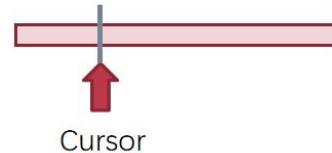
- Option-1: OS returns an inode pointer
- Option-2: OS returns all the block numbers of the file

Reasons and considerations

- Security: user can never access kernel's data structure
- Non-bypassability: all file operations are done by the kernel
- (aka., complete mediation)

我们希望用户不能访问内核的数据结构，如果我们直接返回 **inode** 的指针，那么就把内核数据直接暴露给用户了。Non-bypassability: **inode** 指针可能绕过 **open**，因为 **open** 的时候拿到的是一个文件名，之后再返回 **fd**。但是如果我们返回的是 **inode**，第二次之后的 **Open** 就可以随时随地拿第一次的 **Inode** 返回的指针去用了，可以绕过 OS 在 **open** 的时候的权限检查。这样应用程序就没有任何部分绕过权限检查。

File Cursor



File cursor

- Keep track of operation position within a file
- Can be changed by the **SEEK** operation

Case-1: Sharing file cursor

- Parent passes its fd to its child
 - In UNIX, child inherits all open fds from its parent
- Allow parent and child to share an output file

Case-2: Not sharing file cursor

- Two processes open the same file

Fd 还记录了 cursor，也就是当前的文件。

fd_table & file_table

One **file_table** for the whole system

- Records information for opened files
- inode num, file cursor, ref_count of opening processes
- Children can share the cursor with their parent

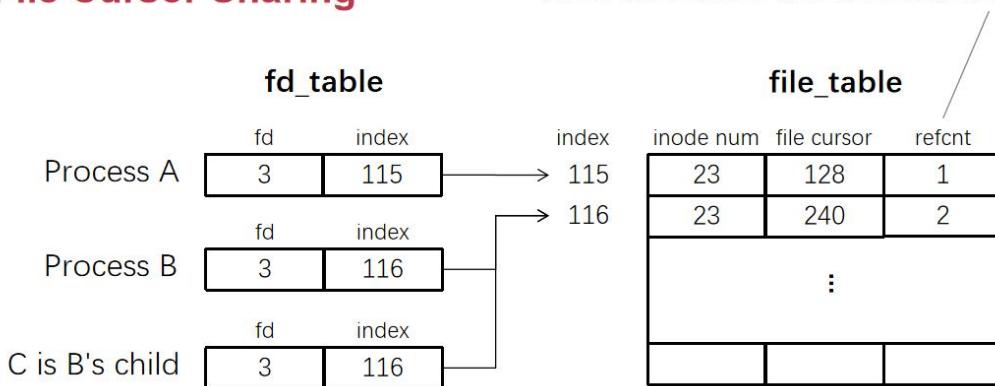
One **fd_table** for each process

- Records mapping of **fd** to index of the **file_table**

File_table 是整个系统一张表，而 fd_table 是一个进程一张表。

File Cursor Sharing

Note: this refcnt is not the refcnt of inode!



Process A, B and C all open just one file with inode number 23

Process A and B open the same file, not share file cursor

Process B and C share the file cursor

不同进程中的 fd_table，哪怕 fd 相同，指向的 file_table 中的位置是不一样的。C 是 B fork 出来的，继承了所有数据。换句话说，两个进程既可以共享 fd_table (file cursor 相同)，也可以不共享 (独立 file_cursor)。

实际上我们在 print 的时候，就会遇到共享 fd_table 的情况，如果不共享，child process 可能就把 parent process 的输出覆盖掉。

注意 file_table 里的 ref count 是表示多有多少个 fd 指向它。

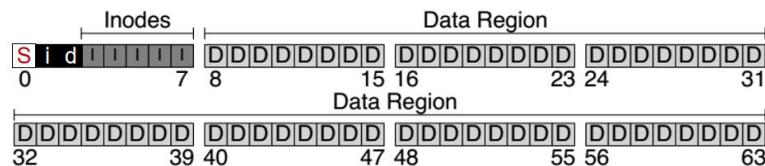
▶ OPEN Implementation

```
procedure OPEN(string filename, integer flags, integer mode) -> integer
0 inode_number <- PATH_TO_INODE_NUMBER(filename, wd)
1. if inode_number = FAILURE and flags = O_CREATE then // Create the file?
2.   inode_number <- CREATE(filename, mode) // Yes, create it.
3. if inode_number = FAILURE then
4.   return FAILURE
5. inode <- INODE_NUMBER_TO_INODE(inode_number)
6. if PERMITTED(inode, flags) then // Does this user have the required permissions?
7.   file_index <- INSERT(file_table, inode_number)
8.   fd <- FIND_UNUSED_ENTRY(fd_table) // Find entry in file descriptor table
9.   fd_table[fd] <- file_index // Record file index for file descriptor
10.  return fd // Return fd
11. else return FAILURE // No, return a failure
```

注意 FIND_UNUSED_ENTRY，是选择当前没有用过的 fd 中最小的一个。

接下来我们看实际的磁盘上的结构：

At the Head of a Disk Partition



i: inode free block bitmap

d: data free block bitmap

S: super-block

- How many inodes: 80
- How many data blocks: 56
- Where the inode table begins: block 3
- ...
- A magic number to identify the file system type

The super-block is used when
the file system is mounted

在磁盘的头部我们放一个:free inode bitmap，还有一个 data bitmap，最头上是一个 Superblock，其中记录了一些元数据。

File Open & Read Timeline

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)						read		read		
						read		read		
						read		read		
read()						read		write	read	
						read		read		
read()						read		write	read	
						read		read		
read()						read		read		
						write		read		

Why "write" on bar inode in a read operation? Why no "write" on foo inodes?

当我们去打开一个文件/foo/bar 的时候，我们应该先去读 root_inode (约定俗成的地方，第一个 inode 的地方)，我们就去读根目录对应的 block number 的 block，读到了根目录下的文件，找到了 foo 的 inode，找到 foo 的数据，找到了 bar 的 inode。读的时候，找到 bar 的 block 读，读完了以后修改一下 inode 的 access time。

我们发现在 read 的时候，我们不得不去 write 一次磁盘为了更新 access time。在 linux 的时候，加载文件系统启动的时候默认会添加 noatime 选项。我们弱化 atime 的更新，也就是当我们 close 的时候才会更新 atime，而不要读一次更新一次，尽可能减少磁盘的写操作。

File Creation Timeline

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)						read		read		
						read		read		
						read		write		
						write		read		
write()						read				
						read				
						write				
write()						read				
						write				
write()						read				
						write				
write()						read				
						write				

当我们要创建/foo/bar 的时候，先要找到/foo 对应的 inode 和 data，然后我们要读 Inode-bitmap，找到一个空闲的 inode_node 然后写到 foo 目录的数据中。然后写 block number

写到 `bar_inode` 里，然后更新 `foo` 的 `inode(size)`。为什么 `create` 中为什么要做 `read` 操作呢？其实是因为越过了 `Inode` 的层次到达了磁盘的层次。我们为了写 $1/4$ 个 `block`，必须 `read before write`。由于磁盘数据要么 $4K$ 全上来，要么 $4K$ 全下去，没有办法做到写其中的个别 Byte。所以要读上来再改其中的 $1/4$ 。

接下来我们就可以来写了，从 `data_bit_map` 中找一个空的写回去。然后写回对应的 `block` 后就再把 `block number` 写回 `inode`。

WRITE, APPEND & CLOSE

WRITE is similar to READ

- Allocate new block if necessary
- Update inode's size and mtime

APPEND

- Similar to write, directly write to the end of the file

CLOSE

- Free the entry in the `fd_table`
- Decrease the reference counter in file table
- Free the entry in file table if counter is 0

Failures in the middle may cause inconsistency!

这里我们发现在前面的过程中，最难的事情也就是一旦发送了 `failure` 就会导致严重的问题。

我们直接回到刚才这个 `write` 操作，一次 `write` 需要写 3 次磁盘（`data_bit_map`, `inode`, `data`）。那么问题来了，当我们在做这个操作的时候断电了，会发生 2^3 情况，也就是 `000,001,010, ..., 111`,

1. `data` 写了，而 `data_bit_map, inode` 没写。元数据啥都没更新，属于 `nothing` 的情况。
2. `Data_bit_map` 写了，而其他两个没写。引入了垃圾，没有任何文件的 `inode` 指向这块区域。我们可以通过把整个 `inode` 扫描一遍得到 `list`，然后做匹配，更新一下 `bit map`。
3. `inode` 更新了，而 `block` 和 `data_bitmap` 没更新。我们发现数据是错的，有可能访问到权限文件引发安全问题，可用性也有问题。这个问题是最大的。
4. `Data` 没写，会导致安全性问题。
5. `Data_bitmap` 没写，会被另一个文件共用一个 `block`。
6. `Inode` 没写，产生浪费。

如果让我们排序的话，我们会先写 `data`，再写 `data_bit_map`，再写 `inode`。因为 `inode` 非常重要。

Questions

When writing, which **order** is preferred?

- Allocate new blocks, write new data, update size
- Allocate new blocks, update size, write new data
- Update size, allocate new blocks, write new data

Order 没办法保证，因为我们发送操作的数据可能是按照顺序的，但是 cache 的情况我们不可控。

SYNC

Block cache

- Cache of recently used disk blocks
- Read from disk if cache miss
- Delay the writes for batching
- Improve performance
- **Problem:** may cause **inconsistency** if fail before write

SYNC

- Ensure all changes to a file have been written to the storage device

为了避免这个问题，这时候我们就要用到 SYNC，把内存中的所有东西 flush 到盘上。

如果我们把一个打开的文件删掉了，如果 file_table 中的 ref_cnt 不为 0，需要等待 inode 在进程 close 的时候把它删掉。

Other Choices instead of inode

Method-1: use continue blocks

- Re-allocate if the file expands
- E.g., data in memory
- Why not?

Method-2: use a linked list

- Each block links to its next block
- Use special one as EOF (End of File)
- E.g., FAT32
- Why not?

How to integrate different FS?

- vnode (will discuss later)
- Interface is similar with inode

除了 inode，我们还可以怎么组织数据呢？为什么我们不适用链表呢？为什么的 FAS32 就是一个链表。

FAT (File Allocation Table) File System

File is a collection of disk blocks

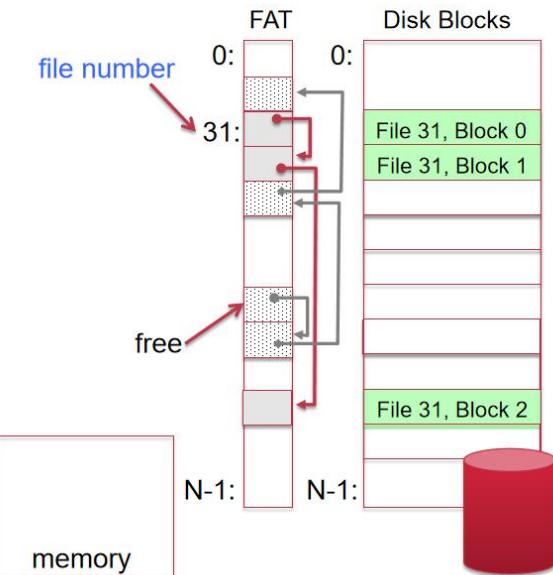
FAT is a linked list 1-1 with blocks

File Number is index of root
of block list for the file

File offset ($o = < B, x >$)

Follow list to get block #

Unused blocks \Leftrightarrow FAT free list



FAT32 使用 file allocation table 来记录文件的第一个 block-number。它没有 Inode number。FAT 表其实就是一个 64byte 的表。当我们把这个表串起来，我们就可以把对应数据串起来。freelist 一路上把所有 freeblock 串在一起。我们要新加一个 block 的时候，我们就把 free_list 指向它的 next，多出来一个 free，然后改 file32 的最后一个指向 free block。

那么这么一种设计有什么特点呢？对于随机读写，需要遍历一遍这个 linked_list，而我们的 Inode 只需要读一遍三级索引的链。如果 linked_list 有一个地方坏了，那么全部都坏了。三级间接索引坏了也会导致大量数据没有，但是概率小一点。

NAT 适合顺序读写。所以适合用来做数码相机的存储介质。

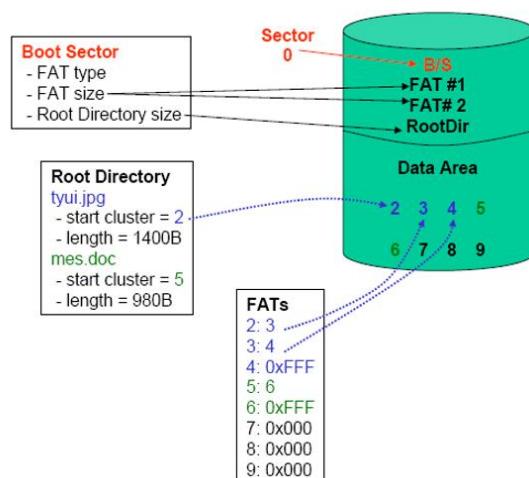
FAT File System

File allocation table (FAT)

- Organize files as linked lists

No inode

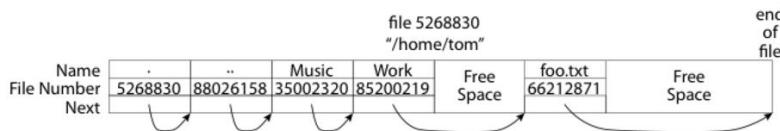
- File metadata: name & size
- Metadata are saved in dirs



由于 fat 表没有 inode，所以我们还是要记录一些元数据。是记录在 directory entry (目录项) 中。inode 的结构是 name: inode-number。而 FAT 中是 name :第一个 block 的 number, size,，各种元数据的信息。就算有 ref count，我们也不知道在哪，需要扫描一遍所有的目录。所以 FAT 不支持 hard link 的。核心就是 name 是文件的元数据的一部分，于是我们就

不能让两个 name 指向一个数据。

What about the Directory in FAT?



Directory: essentially a file containing <file_name: file_number> mappings

- Free space for new entries
- File attributes (metadata) are kept in directory
- Each directory is a linked list of entries

Question: Where to find root directory ("/")?

- Root dir at sector 0

如果我们把 U 盘格式化成 FAT，最大支持的大小是 4G，这个 4G 是由什么决定的？是因为元数据中的 size 只有 32 位，最大只能表示 4G 的文件。而 Inode-based file system 是由有多少级 indirect-block 决定的。如果我们在一个文件系统里面，我们现在要存大量的文件，但每个文件小于 1k，平均 256byte。而一个 i-node 就有 1k，于是我们为了这个每个文件的存储分配一个 4k 的 block 和 1k 的 inode。事实上我们今天的磁盘中，有很多 4k 的数据没有存完，这个地方是由 size 决定的。最后会留下一些空隙的。我们有什么办法把利用率提高呢？一个简单的办法是直接把数据放在 inode 中。为了避免文件系统误解为 blocknumber，再加上几个 flag 就行了。这样文件利用率就提高了

场景：战地记者带手机，进过一个区域，进出都有一个哨兵。这个哨兵非常懂文件系统，记者不能让文件系统有任何的变化，就可以用文件系统中每个文件的最后一个 block 多余的空间。这个方法不仅战地记者也会用，黑客也会用，病毒如果放在这个区域就扫描不出来了。动态地修改 inode 中的 size，就可以在想用的时候调用到病毒的代码。

2021/9/28

我们希望把 file system 从单个扩展到多个。这个在接近 40 年前就有人做了。SUN 公司希望开发没有硬盘的计算机，数据是不是都能全部来自网络。对用户是透明的，最底层的核心技术就是 RPC。RPC 不仅仅是用来提供 Open,Write,Read 接口，RPC 还能实现跨机器的函数调用。

传统的，如果我们不用 RPC，大家要调用网络的时候，就要使用 socket，打开一个 socket，服务端要 listen，然后可以 accept 得到一个 fd，然后在双方之间做交互，我们需要写很多 glue code，不是很方便，我们希望有更加简单的操作，client 的代码可以直接写一个 fool() 调用的是服务器端的函数，返回值就是对应的返回值。换句话说我们希望 RPC 和 PC (procedure call) 是一样的，我们不需要 care 它是 local 还是 remote 的。除了在 framework 级别，在语言级别甚至都包含了 RPC 相似的机制，比如 Java 中的 RMI (remote method invocation)。说明 RPC 是一个非常非常基础的作用。

RPC

Example of RPC

```
1  procedure MEASURE (func)          1  procedure GET_TIME (units)
2    start ← GET_TIME (SECONDS)      2    time ← CLOCK
3    func () // invoke the function 3    time ← CONVERT_TO_UNITS (time, units)
4    end ← GET_TIME (SECONDS)        4    return time
5    return end - start
```

The implementation of GET_TIME.

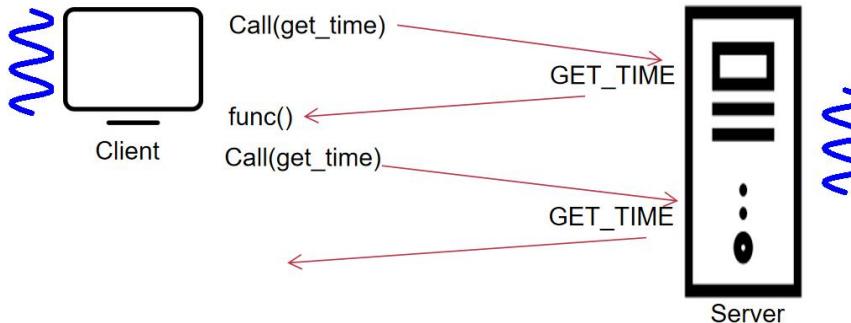
Suppose we want to measure the execution time of *func()*

Assumption:

- Only the server has the implementation of GET_TIME

How can the client call server's GET_TIME?

这个 measure 函数是用来测量 func 的运行时间。Get_time 就是根据传入的单位返回对应单位的时间。如果我们把 GET_TIME 放到服务器上，比如我们本地的时钟不正确，该怎么去做呢？



9

网络延迟该怎么办？我们之后会讨论对时的算法。

最简单的就是多 ping 几次，取均值，我们假设了网络时延基本上不变，并且假设了过去的时间等于回来的时间。一旦在谈到分布式的时候，我们会发现时钟是看起来简单但是背后是一个非常大的问题，因为没有两个人的时钟是一样的。

Client program

RPC

```
1  procedure MEASURE (func)
2    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
3    response ← RECEIVE_MESSAGE (NameForClient)
4    start ← CONVERT2INTERNAL (response)
5    func () // invoke the function
6    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
7    response ← RECEIVE_MESSAGE (NameForClient)
8    end ← CONVERT2INTERNAL (response)
9    return end - start
```

我们需要把 second 参数 convert 一下，因为网络上的数据是 big-endian，然后传一个 Get time 打包成一个 Message 发送给服务器，得到 response。然后再 convert 回来。

```

10  procedure TIME_SERVICE ()
11  do forever
12      request ← RECEIVE_MESSAGE (NameForTimeService)
13      opcode ← GET_OPCODE (request)
14      unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
15      if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
16          time ← CONVERT_TO_UNITS (CLOCK, unit)
17          response ← {"OK", CONVERT2EXTERNAL (time)}
18      else
19          response ← {"Bad request"}
20      SEND_MESSAGE (NameForClient, response)

```

得到 request 后，对其做解析得到 opcode 和 unit。得到数据以后，返回一个 response (OK 和 bad request 两种可能)。

我们看到它来来回回做了很多别的事情，很多内容是可以模块化的。

RPC simplifies the implementation of remote calls

Abstracts away the common parts with stub

Provided in RPC's stub

```

10  procedure TIME_SERVICE ()
11  do forever
12      request ← RECEIVE_MESSAGE (NameForTimeService)
13      opcode ← GET_OPCODE (request)
14      unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
15      if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
16          time ← CONVERT_TO_UNITS (CLOCK, unit)
17          response ← {"OK", CONVERT2EXTERNAL (time)}
18      else
19          response ← {"Bad request"}
20      SEND_MESSAGE (NameForClient, response)

```

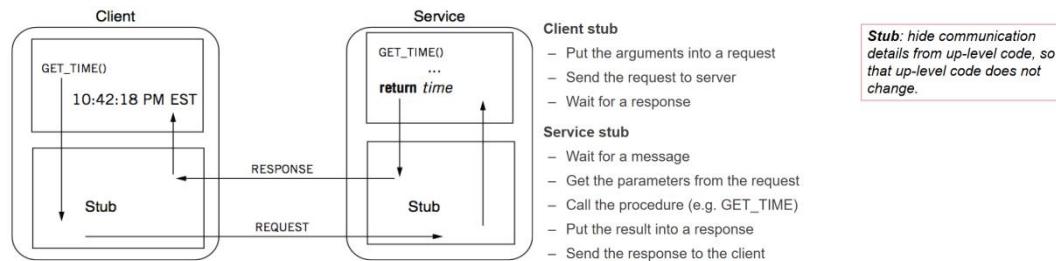
```

Client program
1  procedure MEASURE (func)
2      SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
3      response ← RECEIVE_MESSAGE (NameForClient)
4      start ← CONVERT2INTERNAL (response)
5      func () // invoke the function
6      SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
7      response ← RECEIVE_MESSAGE (NameForClient)
8      end ← CONVERT2INTERNAL (response)
9      return end - start

```

12

这些都是可以放在 RPC 的 stub (桩代码，不用程序员管的) 中，全部用库的方式去封装起来。



在 stub 中会实现 GET_TIME 并且发送给远端得到 response。Stub 就很好地封装了底层网络通讯的细节。对于 server 来说一样，我们实现了底下的一层。

这是非常经典的解耦方式，程序员不用考虑函数是在本地还是在远端，当然这是在不出异常的情况下。

我们需要把数据用合理的方式 marshal 排队发到网上去 (serialize)。我们来详细看一下，在 1984 这篇论文中提出的 RPC 包含了什么内容：

1. Xid, X 是 transaction 的缩写，它是用告诉 server 请求是发过了还是没发过，在出错的时候非常有用

2. Call/reply

3. RPC version, 意味着可以支持多个 version, 一台单机的应用程序和库通常是合在一起的, 但是 client 和 server 的版本可能不相同, 要考虑到兼容性
4. Program, program version, procedure, 可能调用一个可执行文件所提供的函数
5. Auth stuff, 来做验证
6. Arguments, 参数

同样在 reply 的时候也有几个不同的参数

1. Accepted

2. Success, accepted 和 success 的区别是什么呢? Accepted 是和 RPC 相关的, 而 Success 是和我们调用的 service 相关的。换句话说如果 accepted 是 false, 说明 RPC 阶段就错了, 没有调用到程序, 而 success 是说明 RPC 之间是可以正常对话, 但是具体执行的时候出错了

3. Results

4. Auth stuff

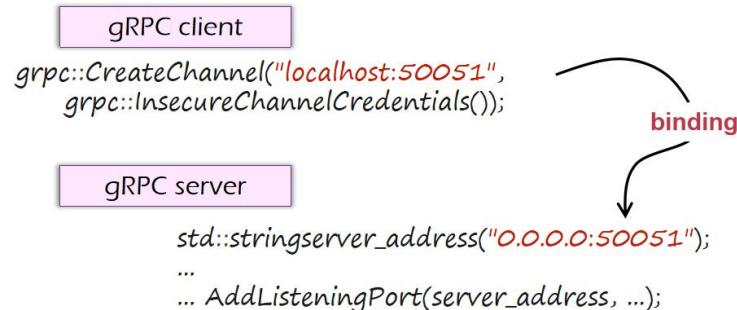
怎么让 client 和 server 互相绑定呢?

Binding: find the server

Can implement with other network name services

- E.g., 192.168.10.233:8888 + function ID

Example: gRPC



Client 该怎么知道这个端口呢? 所以在系统中, main discovery 服务注册端口, 可以列出所有 service 的端口。所谓的服务发现也是在分布式系统是很重要的组成部分。我们需要把数据发送给 server, 这件事情看似简单还挺复杂的, 在现代的 rpc 系统中由于数据在不同的格式中转换, 其时间损失要占据 20%以上。我们在传 paramater 的时候, 必须 pass by value 而不是 pass by reference。

如果实现 pass by reference 我们只需要让在 page fault 的时候, 从另一台电脑的对应虚拟地址 copy 过来就可以了。这个就是 DSM (distributed shared memory), 这个技术在体育馆中的电视墙用的最多。在屏幕矩阵上, 用 DSM 来共享内存对应一块块屏幕的显存即可。

RPC 的参数传递

Process

- Client converts data structure into **pointerless** representation
- Client transmits data to the server
- Server reconstructs structure with local pointers

Server 重新在自己的程序中创建 pointer。那么传参到底有什么挑战呢？核心就是每台机器对数字的理解是不一样的，比如 IEEE 754 的浮点解释标准……etc。是 64 位还是 32 位还是更大，包括对齐方式等，否则我们的数据就会发生丢失，为了解决这些问题，就必须有一些标准，大家统一满足什么规范。一旦 server 更新了 message field，而 client 没有更新，那么就会出现不一致。

所以，我们需要考虑 **backward compatibility**（新代码可以去读旧代码写的数据），**forward compatibility**（旧代码可以去读新代码写的数据），我们就要用 **encoding** 的方式，最早的 SUN 公司提出了 XDR，今天我们用的很多的是 JSON，还有一些 google protocol buffers。为什么不用 **language specific format** 呢？比如 java 提供了原生的 **Serializable** 接口，一旦继承了它，就可以变成序列化，Python 也有 **pickle**，但是缺点就是把我们自己绑死在某个语言上了，并且兼容性也不好。

Logic client request format	JSON representation
– Xid	{ "xid" : 12, "call": true,
– call/reply	"rpc_version": 73,
– rpc version	...
– program #	
– program version	}
– procedure #	
– auth stuff	
– arguments	

这种格式的好处：**human-readable, easy to debug.**

缺点：

1. 关键的数据结构的 **encoding** 会出现二义性，比如 12 这个数字我们不知道 **type** 是什么，是 **signed** 还是 **unsigned** 之类的。
2. 就是要支持 **binary** 的文件该怎么办，我们不得不转换成基于 **base64** 的格式，但是这又是手动要做这样的一个转换。
3. 冗余

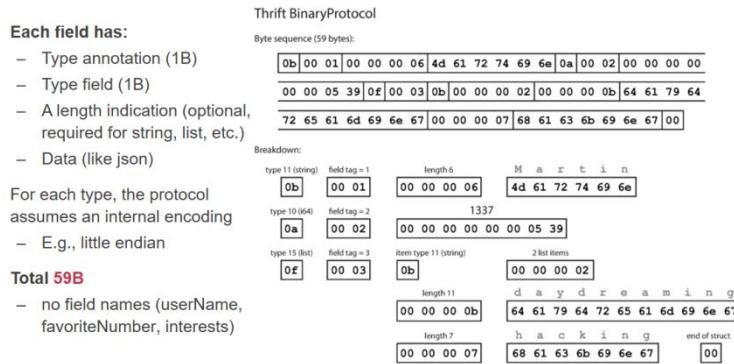
在 **linux** 里有个原则，不希望构造复杂的程序而希望构造出一系列小程序，通过管道的形式去拼在一起。在管道传输的时候必须使用 **asc2** 来做传输。

上节课我们讲了为什么要用 **rpc**，当我们把两个 **procedure** 拆分成分布式的时候，我们把拆的方式细化到了函数的粒度，把不同服务器的函数串在一起。这是我们以后要学的 **function as a service**，函数可以在服务器上任意的来回部署或迁移，就可以提升资源利用率，函数怎么拆是一个关键，拆的不好就会把交互频繁的两个函数从 **function call** 拆成 **rpc** 降低

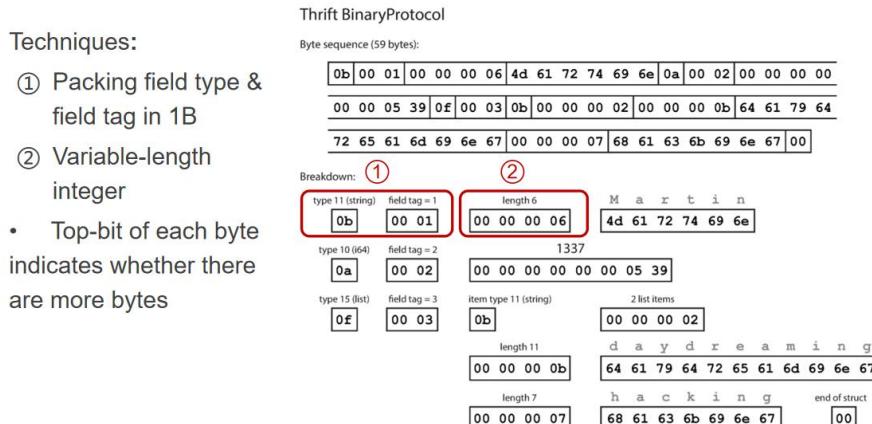
性能。

在传输数据的时候，不同机器对数据的解释是不相同的，所以我们要对数据重新去做序列化。与此相对的就是 **binary format**，好处就是很容易压缩、很紧凑，很快就可以变成内存中的数据结构；缺点就是人类要读懂就很难。

The BinaryProtocol of Thrift

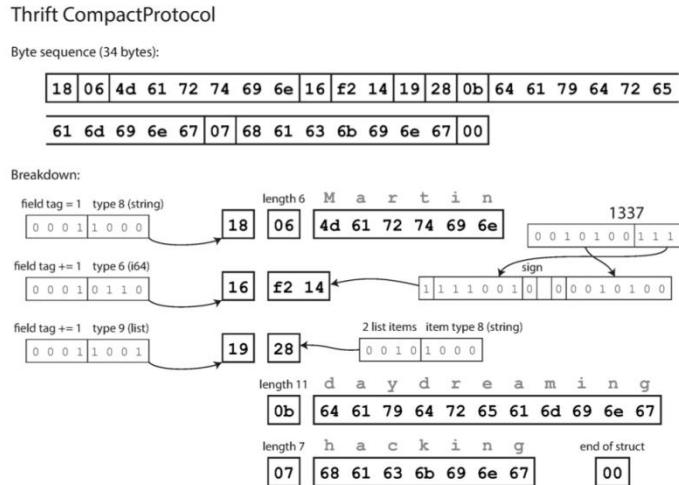


我们可以进一步去压缩，为什么要把 type 放在最头上呢，这个顺序是很关键的，当我们在做一个 server 收到一个 request，一开始什么都不知道，读到的第一个 byte 就很关键，告诉我们后面是一个 string，这个 string 是一个 1 号（username），接下来串一个 length=6，后来是传 6 个 byte。然后是 i64 类型……，真正有用的数据都是 data，前面这些数据都是 metadata，为了组织这些数据，metadata 是必需的。



1. 把 3 个 byte 变成一个 byte, 因为 tag 不会太大
 2. Length 很浪费, 我们可以用一个变长的 8 位 list, 也就是每 8 位如果第一位为 1, 说明剩余 7 位中有数字。

最后结果如下：



通过这种方式可以把 tag 和 type 压缩成一个 byte，最后就是 34 个 byte 就是这么来的。

Forward compatibility: 我们只想要保证 field tag 和 field type 是兼容性即可，跳过不认识的字段。

我们简单来看一下参数传递的小节：human-readable 和效率的权衡、最短越好、保证兼容性。整个序列化的过程是自动的。用户只需要提供 IDL 去写这个结构。在最底下还要考虑是 TCP 还是 UDP，不容易丢包的情况下就可以使用 UDP。

Transport protocol of RPC

Stubs also include implementations of sending/receiving messages

- **TCP or UDP**, which one to use?
- What about new networking features, e.g., RDMA?

Hide the transport protocol from the user

- **Benefits**: e.g., transparent migrate to a more advanced network

Most support several

- Allow programmer (or end user) to choose at runtime

甚至可以使用 RDMA (direct memory access)。当用户使用的时候，我们不用关系底层的网络实现。

RPC 遇到 Failure

当 RPC 遇到 FAILURE 怎么办？对于一个 local 的 procedure call 的情况下，failure 的情况下是很少的，如果调用函数的情况都出错了说明机器已经出了很大的问题。Call 就是把地址压栈跳转到对应的函数地址去执行，如果这种情况下都出错了，说明这个计算机有很大的问题了，这个程序基本上会被 kill 掉，一般来说我们不去考虑 call 带来的问题。那么如果变成了 RPC，遇到问题的概率远远大于本地的调用。序列化出错、RPC 版本出错、对方 server crash、网络出错，当一个 RPC 调用没有得到任何返回的时候，有以下情况

1. 网络问题，包没有到 server。
 2. Server 的 response 没有回到 client。
- 我们没有办法区分出上面两者。
3. 远程 crash
 4. Request 队列
 5. 远程执行了，但是你超时了

6. 执行了，还在 delay 等待

RPC 调用以后等多久？这个时间就很 *tricky*，只能根据经验来设置。对于 RPC 来说，会 return 一个错误 status。RPC 的期望结果，*exactly once*，我们会遇到：

- 0 次，server 没执行
- 1 次，正常
- 1 次或多次，client 可能等不及了再发一次。

所以 RPC 一般提供 *at least once*（没收到 response 就重试，直到 ok）和 *at most once*（只发送一次）

幂等性：我们希望实现 *at most once*，我们能不能让一个操作执行一次和执行两次是一样的。比如按电梯：按三次和按一次是一样的，但是存钱不是幂等的。如果实现不了幂等性，怎么实现 *at most once* 呢，我们要有别的机制去记录。Server 可以把执行成功的 xid 记录下来，如果发现已经做过了那就返回 OK。

Ideal RPC Semantics: exactly-once

Like single-machine function call

Implement exactly-once semantics:

- Server remember the requests it has seen and replies to executed RPCs (across reboots)
- Detect duplicates, reqs need unique IDs (XIDs)

Assumption: failures are *eventually* repaired, and client retries **forever**

- How to correctly recover from failure? See next lectures

RPC 有很多 component，在网络上传输的格式要有标准，我们通过 lib 来实现底层细节 stub，定义完了 IDL 后，我们就可以自动产生一些 stub code，然后去做序列化，server 就是去 reply，server 的 framework 会把 message dispatch 到不同的 server 中。

超时方案：要设置一个超时时间。

2021/9/30

今天讨论分布式文件系统，先来讲一下基本的 file system。RPC 可以把本地调用变为远程调用，上节课讲的就是如何在程序员几乎不可知的情况下，把本地的 PC 变为 RPC。在失败的情况下，程序员需要知道，要分为 *at least once*（直到收到响应），*at most once*（不断重试的情况下还要保证 *at most once*->操作需要是幂等的）

Transaction 存储在服务器，占据空间，什么时候删除呢？这就变成了一个问题。我们尽量希望做到幂等，但是做不到幂等的情况下，我们就记录一个 Transaction id，但是不能记录在内存中，重启的情况下，需要记录在持久化的介质中，而且还要记录相当长的一段时间。所以 *exactly once* 是很难实现的，所以大家现在都在想拆成幂等的服务。

传输文件：上传下载模式

传输文件有很多种方法，有 `ftp`, `telnet`, `ssh`, 它基本上这种方式有一些缺点，我们需要显式的把一个文件拖到本地，这是一个用户的方式。要用的时候拖下来，要传的时候拖上去，这对用户来说不是透明的，需要手动去操作，这就是“上传下载模式（`upload/download mode`）”，比如现在我们使用的网盘，网盘就是第三方的上传下载同步工具。总的来说，这种方式比较简单，缺点就是非常浪费（比如 `client` 只需要很少的数据，作为网盘的来说，比如说 `100M` 的表格，我们只想看第一页，我们做不到用多少下载多少，必须打开整个 `excel` 文件），`client` 端没有足够的空间（比如磁盘只剩了几百兆想下电影，但是实时看的时候理论上电影应该很少），一致性（多个节点同时想修改节点，我们同时编辑的时候发现文件会冲突，一旦下载到某个节点上更新了以后，除非手动上传，否则就会不一致）。

传输文件：远程访问模式

`remote-access mode`, 文件存储在远端，本地要用到的东西下载下来，客户端在用的时候去按需的下载，`on-demand`，一旦写完了我们就同步回远端，这样就可以避免没有必要数据传输，尽可能保证多个 `client` 看到的时候同步的，eg: 石墨文档的同步情况，当然它暴露的是网页的接口而不是文件的接口。缺点就是：过分依赖网络，每次访问都要通过网络访问，性能会差一点；刚访问过的数据同步回去，下次再访问的时候又要通过网络访问。这是中心化的 `remote-access mode`

传输文件：NFS（network file system）

NFS（`network file system`）：1980 年代 sun 公司提出的，它开发的目的是支持无盘工作站，为了节约成本。在网络中，各种各样的机子的操作系统和版本不同，不同的 `hardware`, `OS` 等是不一样，需要在各大硬件、`OS` 上架起一层大家都能使用的网络文件系统；`transparency`: 很有挑战性的目标，可以增加实用性，为了使用网络文件系统所有东西重写我们是受不了的；性能也不能太差，每次过网络是不显示的，为了实现 `transparency`，我们实现的是 `remote access mode`。在 NFS 中，把文件系统的调用实现成了 `RPC` 的方式。

RPC used in NFS

Table 4.1 NFS Remote Procedure Calls	
Remote Procedure Call	Returns
NULL ()	Do nothing.
LOOKUP (dirfh, name)	fh and file attributes
CREATE (dirfh, name, attr)	fh and file attributes
REMOVE (dirfh, name)	status
GETATTR (fh)	file attributes
SETATTR (fh, attr)	file attributes
READ (fh, offset, count)	file attributes and data
WRITE (fh, offset, count, data)	file attributes
RENAME (dirfh, name, tofh, toname)	status
LINK (dirfh, name, tofh, toname)	status
SYMLINK (dirfh, name, string)	status
READLINK (fh)	string
MKDIR (dirfh, name, attr)	fh and file attributes
RMDIR (dirfh, name)	status
REaddir (dirfh, offset, count)	directory entries
STATFS (fh)	file system information

Where is OPEN
and CLOSE?

14

这个表格中，多了 lookup，少了 open 和 close，为什么没有，我们后面再讲。

NFS Protocols: Mount

Protocol:

- Request access to exported directory tree
 - Requests **permission** to access contents

Client: parses **pathname**
contacts server for file **handle**

- Server returns **file handle (fh)**

Client: create in-memory VFS **inode** (vnode) at
mount point internally points to remote files
(client keeps state, not the server)

Static mounting

- mount request contacts server

Server: add list of shared directories to `/etc/exports`

Client: `mount -t nfs 192.168.1.100:/users/paul /home/paul`

Mount: client 把目录转换成 server 端的 file handler, client 依然是通过 inode 的形式, 创建 VFS-inode。我们假设 client 端没有硬盘, 那么此时的 inode 没有 block number 可以记录, 记录什么呢?

Server 有一组目录, 暴露出去作为可挂载的目录, 而 client 就可以通过 mount 目录, 直接映射到无盘的主机上, 直接打开这个文件。

Lookup 返回文件的 handler 和文件的相关数据 (attribute), lookup 和 open 的区别就是 lookup 是没有状态的, 返回的 file handler 是服务器端维护的元数据, 并没有 open。Open 的时候, 文件系统有 fd table 和 file table, 但是 lookup 时, server 端没有做任何的 open, 也就

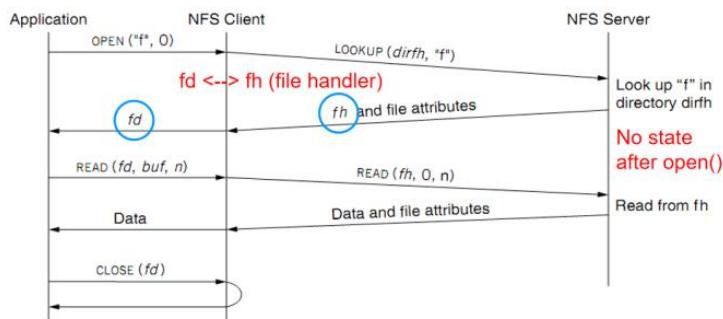
是 `lookup` 操作是无状态的。后面要去做读的时候，我们就通过 `read(handle, offset, count)`。

Q: `handle` 是什么东西？我们把 `fd` 换成了 `file handle`，这不是改变了 API 吗？

Q: 我们在执行 `read` 的时候，原先的 `read` 参数是 `fd`，读哪儿，读多长。为什么现在 `read` 的 API 要这样设计呢？

NFS 读取文件的流程

Read a file of NFS



我们前面所说的 RPC 操作都是 NFS Client 去调用，所谓新的 RPC 都是我们的 NFS Client 去实现的，而 NFS Client 的位置在 kernel。以前 app 调用 `open` 直接去 File System 返回，现在 app 调用 `open`，现在是到了 NFS Client 这里，它接收到来自用户的传统的文件系统 API (`open`, `close`, `etc`)，然后走网络调用 `Lookup` 到 server，所以真正的 RPC 发生在 NFS Client 这一步。调用 `LOOKUP` 时，里面有一个参数 `dirfh`（远端文件的目录），我们要告诉 NFS Server 在哪个目录下去查一个叫做 “f” 的文件，然后返回 `file handler` 和 `file attribute`（文件属性）。于是 NFS Client 就得到了 “f” 文件得到的 `file handler`，同时返回用户一个 `fd`。然后当 application 去调用 `read` 的时候，参数是 `read(fd, buf, n)`。然后 NFS Client 把 `fd` 转换成一个 `fh`，然后转化为 `offset` 是 0，大小是 `n`，把这个 RPC 操作发到 server 端，server 端根据这个 `fh` 把文件读出来，然后把数据传输给 NFS Client，然后由 NFS Client 把数据写到 buffer 中，用户读完后，`CLOSE` 直接返回（因为没有 Close RPC）。

我们来分析第一点，当 application 调用 `open` 之后，application 拿到 `fd` 了，但是在 NFS Server 这一段，打开前后是没有任何区别的，当第二次 app 想要 `read` 文件的时候，NFS Server 调用 `open` 和 `read` 和 `close`，换句话说，app 每次 `read`，nfs server 会像从来没有遇到过这个请求一样做完一整套打开、读取、关闭文件的操作，然后返回数据。换句话说，这样无状态的文件读取操作，如果 NFS Server 中间重启了一次，Client 是不知道的。

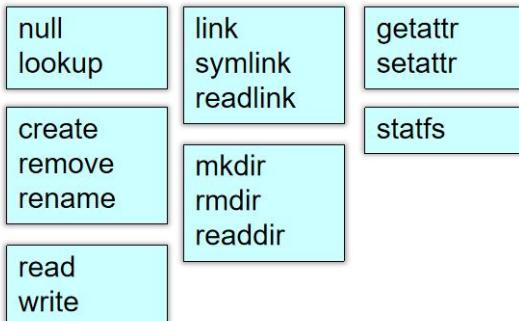
这个操作是 *at most once* 吗？是的，因为 `offset` 和 `count` 同一个请求多次发送是无状态的，所以 `read` RPC 是幂等的。但是 Client 端的 `read` 不是幂等的，因为其中的 `cursor` 维护了状态。所以这个状态是由 NFS Client 来维护的 `cursor` 的位置。

NFS Client 把传统的文件操作是有状态的转化为了无状态的 RPC 操作。`Close` 与否对 NFS server 端没有影响，在 `close` 的时候，在 NFS Client 这一端应该做什么事情呢？

通过 `vnode` 就可以把远端的状态记录下来。

NFS Protocols: Lookup/READ/WRITE . . .

NFS has 16 functions (version 2)



Remote Procedure Call	Returns
NULL ()	Do nothing.
LOOKUP (dirfh, name)	fh and file attributes
CREATE (dirfh, name, attr)	fh and file attributes
REMOVE (dirfh, name)	status
GETATTR (fh)	file attributes
SETATTR (fh, attr)	file attributes
READ (fh, offset, count)	file attributes and data
WRITE (fh, offset, count, data)	file attributes
RENAME (dirfh, name, tofh, toname)	status
LINK (dirfh, name, tofh, toname)	status
SYMLINK (dirfh, name, string)	status
READLINK (fh)	string
mkdir (dirfh, name, attr)	fh and file attributes
rmdir (dirfh, name)	status
readdir (dirfh, offset, count)	directory entries
statfs (fh)	file system information

File handler 应该包含哪些信息？

File Handler for a Client

File handler contains three parts

- File system identifier: for server to identify the file system
- inode number: for server to locate the file
- Generation number: for server to maintain consistency of a file

Can still work across server failures

- E.g., server reboot

Q: Why not put path name in the handle?

从 server 得到了 inode 之后，当 server 删掉 inode-number 去复用别的文件的时候，generation number 就为 4。如果不这样，我们就认为请求就过期了。

Q: 存储 inode-number 还是 path?

A: 似乎都存在版本问题，当前获取到的数据不一定是下个时刻的 path 或 inode。最后还选择基于 inode-number，注意它是不能脱离 file system 的 id。inode-number 是基于一个特定的文件系统的 file system。

Case 1: Rename After Open

Program 1 on client 1 <pre> 1 CHDIR ("dir1") 2 fd ← OPEN ("f", READONLY) 3 4 5 READ (fd, buf, n) </pre>	Program 2 on client 2 <pre> RENAMEx ("dir1", "dir2") RENAMEx ("dir3", "dir1") </pre>
---	---

↓
Time

UNIX Spec:

- Program 1 should read "dir2/f"
- NFS should keep the spec

上例中，如果我们保存了 Path，我们就会 confused

因为 NFS Server 不知道哪些文件正在被打开，打开后被删除的情况很常见。

Case 2: Delete After Open



UNIX spec:

- On local FS, program 2 will read the old file

How to avoid program 2 reading new file?

- Generation number
- "stale file handler"

Not the same as UNIX spec! It's a tradeoff...

Program2 打开文件得到一个 inode，当我们去读的时候还是可以得到旧的 f，如果旧的 f 和新的 f 共用一个 inode，那么需要根据 generation number 来判断是否是打开的文件被删除了。（unlink()会删除参数 pathname 指定的文件）

如果发生在本机，我们打开一个 fd，读取两次后，fd 不会过期，但是在 Server 这边，因为没有维护任何状态，读取第三次的时候就会告诉你 fh 是过期了，这就会导致 NFS 和本地的 unix 文件系统 spec 是不相同的，所以写程序的时候我们不是完全 transparency，需要根据新的情况去做处理。

下一个问题是，如果每一次 Client 都通过 RPC 去调用远程服务的话，走网络会比较慢。

NFS performance

Usually **slower** than local

- Faster example: server store file in memory, and the network is super fast

Improve by **caching** at client

- **Goal:** reduce number of remote ops
- Caching: read, readlink, getattr, lookup, readdir
 1. Cache file data at client (buffer cache)
 2. Cache file attribute information at client
 3. Cache pathname bindings for faster lookup

Server side

- Caching is “automatic” via buffer cache
- All NFS writes are write-through to disk

NFS 的写操作必须写穿到磁盘，避免重启的情况。

一旦有了 cache，我们就要考虑一致性的问题。

1. 读写一致性

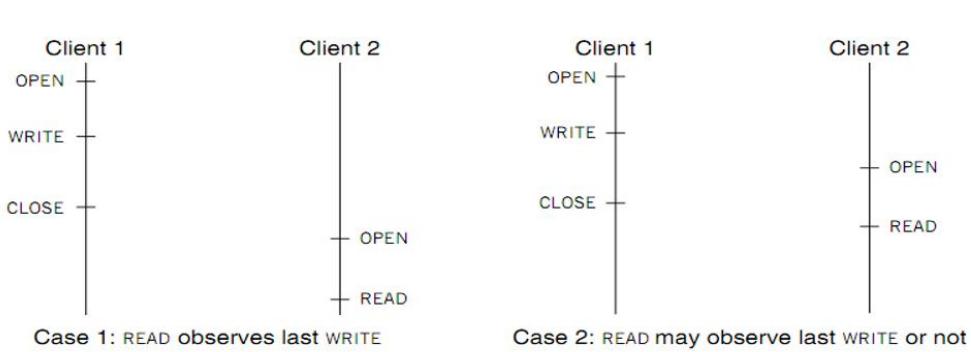
对于本地来说，每次都是最新的数据。对于 NFS，如果我们读到了本地的旧数据，而服务器端是新的该怎么办？NFS 能保证对于每个来说都是最新的。

2. Close-to-open consistency

在 close 和 open 之间保证是一致的，一旦 close 了就把写操作发到远端。

Coherence

Two cases of close-to-open consistency



More contents of consistency in chapter 9 and 10

Case1: client1 CLOSE 在 Client2 的 open 前，我们可以保证 client2 的 read 能读到 client1 的 write。也就是 client 在 close 的时候会把 cache write 发到服务器，服务器拿到后就可以写穿 flush 到硬盘。

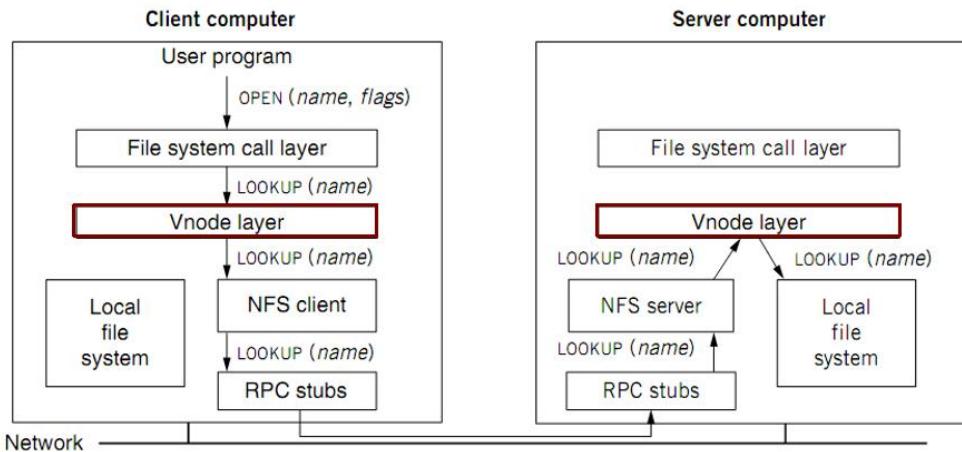
而 Case2: Client2 有可能读取到旧的数据，因为 client1 写到本地 cache 上，flush 到远端的时间的不固定的。

如果我们要做到读写一致性，本地没有 cache 相对简单一些。本地有 cache 的时候，该怎么实现更强的读写一致性呢？修改的时候，Server 告诉所有的 Client，cache 过期了，重新拉取一遍数据。所以多核中，中央节点需要记录哪个核的 cache 数据有没有被改掉，是不是要 invalid 掉，还是这就变成有状态的情况了。

那么有了这个之后，在 NFS Client 要做到 read-write consistency 的情况下，就是锁的情况下。保证只有一个人可以写，避免两个人同时写而出现不一致的问题。

怎么扩展普通文件系统扩展成 NFS 呢？普通文件系统有一层 v-node。它可以支持底层的不同的文件系统，如果是本地文件系统，vnode 就直接映射成 inode；如果是远端文件系统，vnode 就映射成 fh。

VFS: Extend the inode-based FS to support NFS



Client 打开一个 vnode，NFS Client 就调用 RPC，到 NFS Server 可以运行在用户态，拿到这个 name 之后也得去找 Vnode（因为它也要去 open 这个文件，最后是存在服务器的 inode-based system 上），所以 NFS Server 和 Local file system 之间依然是做了这个 vnode 去连接。这个我们就是实现了异构性，任何一台服务器只要实现了 Vnode 这一层，就可以向外提供 NFS Server，把异构性给封装掉。

在真实的 NFS 文件系统中，解决 cache 问题确实使用了 invalid 访问。

Improving Read Performance

Transfer data in large chunks

- 8KB default

Read-ahead

- Optimize for sequential file access
- Send requests to read disk blocks before they are requested by the applications

当我们要的时候，数据就在本地了。

NFS 也不是没有问题的，一致性和性能的权衡一直是很难解决的问题，我们需要全局的时钟、全局的 lock manager，包括无状态导致的 delete after open 的情况。

NFS 也在改进，RPC 版本也在改进，所以还要保证前向和后向兼容。

GFS(Google File System)

我们来看另一个系统 GFS (google file system)。这篇文章是 2003 上的 SOSP 上发表的，最底层的是 GFS，上层是 Big Table，再上层是 map-reduce。对我们来说，这是一个基础性的系统。GFS 开始的想法就是，我们就这么多的机器，能不能用一个完整的 FS 让所有的机器去存储。规模巨大的数据密集型的情况，NFS 不能做到很好的性能和容错。Google 的特色是大量的小机器，NFS 是一台服务器提供很多 client，server 本身怎么多台机器组成一个 server

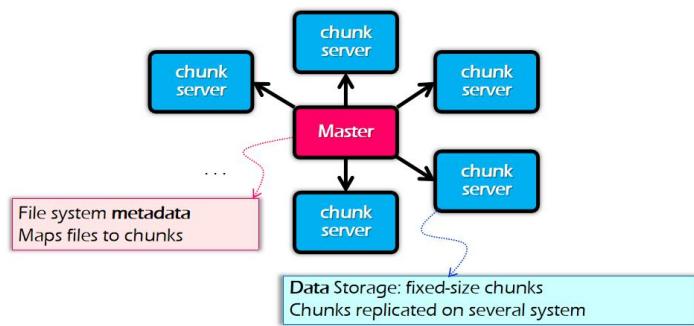
是做的不够的。

传统的假设：大部分文件都很小，和谷歌的要求是不一样的。在 Google 中，大部分文件都很大，每个网页包含的所有内容都很大，传统文件的 lifetime 都比较短，Google 场景中没有什么临时文件，最常见的操作是 append 而不是 delete。这样我们才能查到 Google 中网页在不同时间段的缓存。还有一个要求就是 failure，11 个月以内，几乎所有的服务器都会重启一次，所以对容错性要求很高，我们要设计一个文件系统，可以少支持 random write，但是要支持 sequential 的 append、大量的从头读到尾、append 很大的数据、大量的进程会同时访问和 append 一个文件。如果没有做好并发写，就会导致不同进程的 append 互相覆盖。

我们直接来看 Google 的 api，google 不去遵守 unix api。它可以支持 create/open/close/read/write/snapshot/append，没有 link 和 symbolic link。

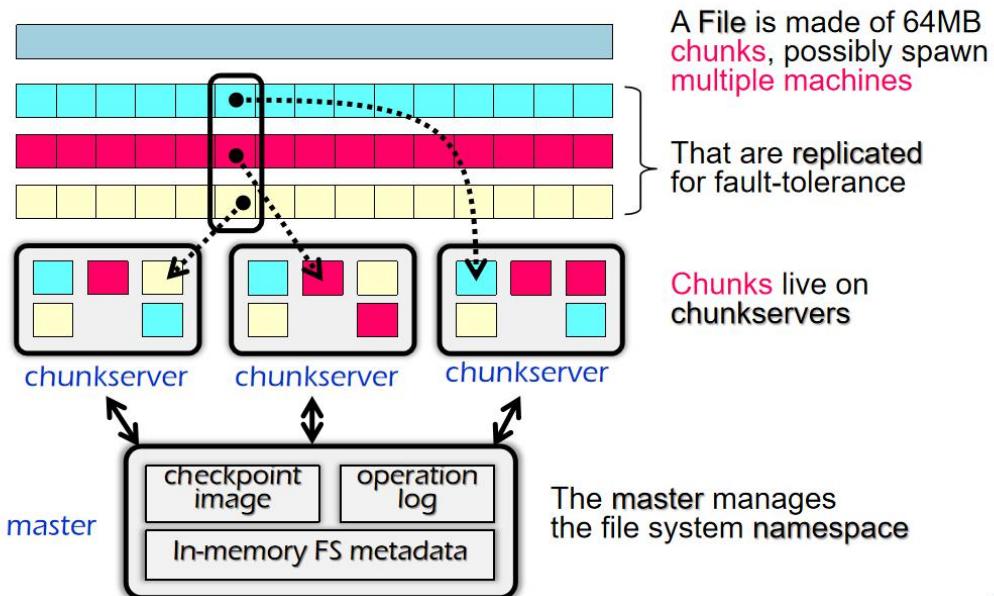
GFS architecture

A GFS cluster = 1 Master + N Chunkservers



Google 把所有计算机分成 master 和 chunk server。这个 GFS 的扩展主要是通过 chunkserver 的扩展，我们可以简单的把 master 认为是元数据，而 chunk server 可以认为是数据。然后在每一个 chunk server 只存 data，以 fixed-sized chunk 作为最小的数据，一个大小 64M，每个 chunk 都在多个系统中做三备份 replica，对于经常访问的可以做更多的备份。Master 还把文件映射成 chunks。

GFS files



43

一个文件存在不同的 chunk server 上，在 GFS 中 inode 应该叫做 chunk-index。Master 除了记录这个之外，还需要记录 checkpoint 和 operation log。GFS 只有加数据，所以 snapshot 是比较容易的，只需要存一堆指针就可以了。再有一个新的指针指向新的数据就可以了。

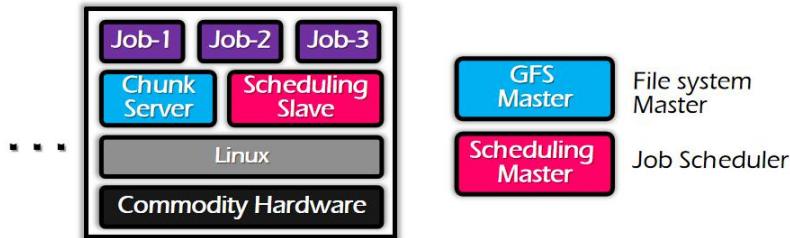
In-memory NFS data。Crash 怎么办，我们还要 operation log。

GFS 的特点：

GFS in Google cluster

Google cluster environment

- Core services: GFS + cluster scheduling system
- Typically, 100s to 1000s of active jobs
- 200+ clusters, many with 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems, 40GB/s read/write loads



Chunk size = **64 MB** (default)

- 32-bit checksum with each chunk

Chunk **handle**

- Globally unique 64-bit number
- Assigned by the master when creation

Chunks are stored on local disk as Linux files (aka. file system overlay)

Each chunk is **replicated** on multiple nodes

- Three replicas (default)
- More replicas for popular files to avoid **hotspots**

每一个 chunk 有一个 handle，就是 block number。当 master 在创建的时候，就会分配到每个 chunk server 上。每一个 chunk (64M) 就是一个文件。在 GFS，我们是用 64b 的 handle 去索引它们。

当我们要去读文件的时候

1. client 联系 master，得到文件的 metadata，知道每个 chunk handle 保存在哪个 chunk server 上。
2. 选择网络距离较近的备份，如果挂了就去找另外的两个 replica。

Writing a File in GFS

Less frequent than reading

- But is more complex, e.g., what about consistency?
- GFS adopts a **relaxed consistency model**
- E.g., may have inconsistency state, but work well for their apps
- Benefits: simple & efficient to implement

Master grants a **chunk lease** to one of the replicas

- This replica will be the **primary** chunkserver
 - The only one that can modify the chunk
- Primary can request extensions (of lease), if needed
 - Master increases the chunk version number and informs replicas

写本身不是那么频繁，GFS 使用了 **relax consistency model**（搞不定强 consistency，就用了一个 weak）。

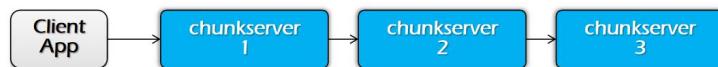
1. 如果任何一个人都可以往三备份中的任何一个备份中写数据，就会乱掉。所以我们只能保证只有一个人去写，三个里面谁去写呢？Master 就会发一个 lease（租约，有时间限制的令牌），master 就会给 replica 中的一个 lease，当前你就是这个 chunk 对应的 primary，只有这个 primary 才能改这个 chunk，还有两个 replica 怎么办？就由这个 primary 告诉它们怎么写，如果超时了 primary 还可以续租。

Writing a File in GFS: Two-phases

Phase 1: send data

Deliver data but **don't write** to the file

- A client is given a list of replicas
 - Identifying the primary and secondaries
- Client writes to the closest replica
 - Pipeline forwarding
- Chunkservers store this data in a cache (in memory)

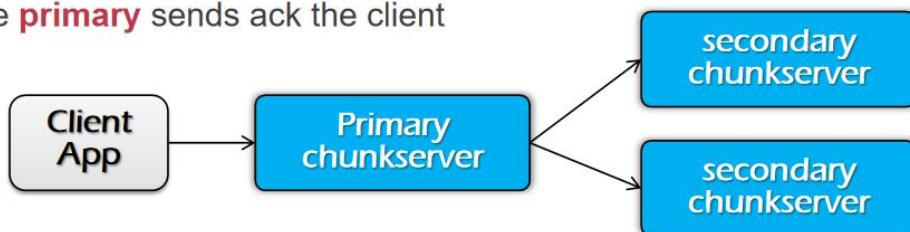


阶段 1（传数据阶段）：client 去发数据的时候一个个发是比较慢的，它可以发给一个最近的 server，让这个 server 把数据发给其他的 server，因为 server 之间的传输是相对比较快的。如果在传数据的时候发送 crash，充其量删了就行。

Phase 2: write data

Add the data to the file (commit)

- Client waits for replicas' ack of receiving data
- Send a **write** request to the **primary**
- The **primary** is responsible for serialization of writes (applying then forwarding)
- Once all acks have been received
 - The **primary** sends ack the client



阶段 2：发一个 write request 发到 primary，先应用到自己的 chunk 上，然后再告诉其他两个 chunk server，当三个都结束了一个，才会告诉 client 写完了。

为什么要分成两个阶段呢？我们想把 **control flow** 和 **data flow** 拆开，**data flow** 的先后是无所谓，我们先做慢的，**order** 不重要。到第二阶段的时候，primary 发给所有的 secondary。我们考虑大量的 append，真正关键的是谁在前谁在后，不要互相覆盖掉。所以这时候就有 primary 去负责排序 **order**。

Writing a File in GFS: Two-phases

Data flow (phase 1) is different from **control flow** (phase 2)

Data flow

- Client → chunkserver → chunkserver → ...
- Order does not matter

Control flow

- Client → primary → all secondaries
- Order maintained (also for concurrent writes from multiple clients)

Chunk version numbers are used to detect if any replica has stale data

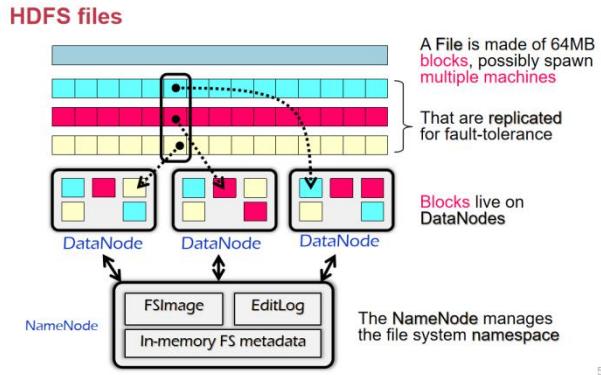
- Is maintained by the primary chunkserver
- If a replica has stale data, it shall be replaced

Primary 如果发现哪个 secondary 版本过时了，就把它踢掉。

在 GFS 中没有 **directory** 的概念，GFS 的目录就是一个 字符串，最后映射到 **metadata** 上，其实就是一个 **pathname to metadata**，其实就是一个 **key-value store**。这都是在 **master** 上去记录的。为什么我们的 PC 上不用这种方式呢？有什么问题？什么操作会很慢？比如搜索操作、

改目录名的操作，需要把全部包含这个目录名的文件找到，然后一个个去改，因为它没有去做层次化。

有了 GFS 之后，就有开源的 HDFS（Hadoop Distributed FS）。它允许在大数据集上使用集群的分布式处理。它是一个 GFS 的开源设计，可以在廉价的服务器上做集群、有高容错和大规模的吞吐和部署能力。



它分为 namenode 和 datanode，其实和 GFS 的逻辑差不多。

分布式的文件系统设计不简单，开源通过单个节点+RPC，需要对于 client 是透明的，NFS Client 的接口可以不一样，第二个 cache 怎么保证 consistency，还有一个 NFS 和 GFS 的相似的和不同的地。

2021/10/9

RPC:本地可以调用远端的，这样本地和远端就可以解耦，本地安装一个 NFS 的 client，实际上是通过 RPC 访问远端的服务器。GFS 是 Google 提出的文件系统，希望支持大量的、大体积的、大部分操作都是只读、很少一部分写、修改少量但是 append 很多的这样的应用场景。GFS 主要的特点是 meta-data 和 data 的分离。文件名、文件大小、文件类型文件存在哪些服务器上，这些都是元数据，存放在 master server 中。而真正的数据以 64M 的一个个 chunk 的形式存放在 chunk server 上。当一个 client 要访问一个文件的时候，它就去联系存放 meta-data 数据的 master server，然后 master 会告诉 client 你需要的文件存放在哪些 chunk server 上，然后 client 会通过 GFS 的 library 去对应的 chunk server 那里读取数据。

通过 Google 的 GFS 我们有一个发现，当我们想把一个文件系统从单体变成分布式的时候，并不是横向地把单体的代码一份份复制出来，而是我们要做一个横向的拆分，我们要把重要的数据放在一台机器里面，而把那些容量大的数据放在多台数据里面。

这样拆分的好处是因为 meta-data 一定要保证 consistency。一旦发生了 failure，meta-data 和实际的数据对不上，这就是 inconsistency。机器一多、元数据一多，机器之间 replica 的一致性就会很麻烦。为了解决这个问题，我们并不能无脑地把 meta-data 也做进 replica 里。

那么一台机器来存放 meta-data 够不够呢？GFS 的实践告诉我们是够的，原因还是在使用场景。如果我们需要对数据做大量的 update 操作的话，其实是很难做到 meta-data 存放进一台机器里面的。但是在 GFS 的使用场景中，

1. 由于都是大文件，那么打开大文件的频率相对来说要低
2. 由于操作基本上都是 append-only 的，我们很少需要去新增文件及其 meta-data，我们只需要修改已有文件的 size 即可。所以对 meta-data 的修改也相对少。

所以，当我们有大量的小文件要操作的时候，使用 GFS 可能就不是一个最好的选择。比如我们要编译一下 Linux Kernel，里面一个几十 M 的压缩包可能有几十万个小文件。那么我们解压 tar 包的时候，要创建多个目录以及目录中的目录，并且在新建几十万个文件，我们可以想象磁盘的磁头要在 inode map 和 block bitmap 中不断地移动。解压 tar 包也是测量文件系统性能的一个通常的方法，它可以在短时间内创建很多文件。

如果在这种情况下，我们还使用 GFS 去存，那么就要在 master 中建立很多 key-value store，就会变得非常慢。这也是我们应用系统设计的原则，我们很难做到 one fit all.

这节课我们主要讲 application,

1.render 页面，把页面渲染到不同地方。

2.当我们在做支付的时候，计算的背后还会用到一下 fraud detection，避免信用卡套现。

这些都需要用到一些更加复杂的技术，比如买方和卖方构成一个环了，那么就涉及到套现。

3.搜索热榜，当前搜索最热的词排序。

因为反欺诈是每次交易的时候都需要实时地算一下的，如果用户等的太久下次可能就不再用支付宝交易了。而热榜中热词的显示，其实我们并不关心一秒内其排名的变化。所以反欺诈的计算是需要 online 的，而热词的变化可以 offline 计算。

Batch Processing

Spark, MapReduce, Hadoop 等框架解决的是一个当时非常棘手的问题：用户量上去以后，我们怎么样保证自己的 scalability。但是很多 task 可能就需要几个小时到几天，并且这些任务（比如热词搜索）是一直要更新的，所以需要一个方法维持很好的扩展性和性能，并且能很好地容错。

我们来举个例子，现在有一个 webserver，它会把每个访问的请求都记录在 log 里，这是典型的 GFS 面临的场景。这个 log 会不断地在后面增加新的内容。这是 web 最常用到的情况。我们来看一下这个日志，它里面有一个对应关系：用户（IP 地址）在什么时候发起了一个什么 request，向着什么网站请求了什么文件，后面是一些浏览器的信息。

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000]
"GET /css/typography.css HTTP/1.1" 200 3377
"http://martin.kleppmann.com/"
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115
Safari/537.36"
```

上图就是日志中一条的信息，现在我们怎么找到最受欢迎（被访问次数最多）的五个页面呢？假如我们现在有一个大日志文件，每一行中我们先要抽取出访问的页面名字（上图黄色部分），然后因为没有现成的计数工具，我们可以先做一次 sort，然后使用 unique 计数，再做一次 sort 就可以得到结果。

如果对 Linux 下脚本命令熟悉的话，其实就是这 6 步然后用管道串在一起。

Find top five most popular pages

```
cat /var/log/nginx/access.log | ①
  awk '{print $7}' | ②
    sort | ③
      uniq -c
        sort -r -n | ⑤
          head -n 5 | ⑥
```

① Read log file

② Filter the line, pick the 7th token,
→.U., /css/typography.css

③ Sort so that the same requests
come together

④ Aggregate, with a counter for each line

⑤ Sort again (using the counter)

⑥ Print 1-5 items

10

`awk` 是一个很著名的命令行的工具, `$7` 是默认以空格作为分隔符出现的第 7 个单词是什么。找到之后就只打印出了对应的访问路径。`uniq` 命令如果不出现`-c`, 就是从上往下但凡有一样的, 就全部变 1 行。加了`-c` 就会告诉你压了多少行。这样操作完, 文件内容就变成了:

```
8 /css/typography.css
13 /index.html
1 /....
...
```

因为这样出来的文件还是乱序的, 所以我们 `sort -r -n`, 就是逆序地(从高到低)以 `number` 进行排序。`head -n 5` 就是只打印出前 5 行。

任何一个程序员都可以在很快的时间内写出这个脚本, 但是问题来了, 当我们想要把这个命令扩展一下, 我们想要让它运行在 1000 台机器上的时候。

第一种方法, 我们可以做一个 `remote shell`。这个 `remote shell`, 它的作用就是打开一个 `remote shell server`, 1000 台打开 1000 个 `shell server`, 然后我们运行一个指令, 1000 个 `remote shell` 会同时运行这个指令, 并且把结果返回。

Command line tools are mostly **single-threaded**, and **single-machine**

- ① : scalability is restricted by the disk capacity
- ② ~ ⑥ : restricted by single-thread computing power
- ② ~ ⑥ : also restricted by the machine's DRAM capacity

这样的话, 我们就可以按照之前的逻辑, 把命令发送给 1000 个服务器, 然后再把结果收回回来。但是这个方法有个问题。通常命令行是单线程的, 我们只能一台台机器去运行, 并且运行的时候是单线程的。这就会导致 1 受限于一台机器的磁盘大小, 而 2 到 6 受限于单线程的算力以及机器内存 (DRAM) 的容量。

所以我们考虑使用分布式来解决问题, 以汽车总动员为例, 它平均需要 12 小时才能渲染出 1 帧, 如果使用单核来渲染需要 200 年。再以 Google 为例, 它平均一天有十亿请求, 需要多少 250 亿的网站做索引。这就告诉我们, 如果我们有 1000 台服务器, 我们希望的是 1000x 的速度提升, 而不是 100x 或更少。

那么在整个分布式系统里面, 首先会把数据分成大小相等的 `chunk`, 每个 `process` 在 `chunk` 上同时运行。我们通常假设每个 `process` 的运算能力是相同的, 因为在互联网企业中它们最喜欢做的事情就是买一堆一样的机器。我们可以让硬件一样, 一旦有别的需求, 我们就给每台机器烧不同的 `FPGA`。这也是 `FPGA` 在云端很受欢迎的理由。因为它使得每台机器很方便

受管理。Google 中有很多同样计算能力的机器，所以每个数据划分大小都差不多。

有了 equal-size 的数据，我们再回到这个应用中。其中最麻烦的事情就是我们要找到可并行的部分。

Accelerate batch processing with distributed computing

Distributed computing

- Split data into **equal-size** chunks
 - Each process can work on a chunk parallelly
- Most significant challenge: identify concurrency**
- It should be noted that not all tasks can be parallelized
 - E.g., dependencies

```
cat /var/log/nginx/access.log | ①
awk '{print $7}' | ②
sort | ③
uniq -c | ④
sort -r -n | ⑤
head -n ⑥
```

④ depends on ③

并不是所有的任务都可以并行的，存在 **dependency** 的任务就不能并行，比如 2 和 2 之间可以并行，而 3 和 4 存在依赖关系。

挑战

1. 数据怎么收，1 台机器管理 1000 台机器
2. 节点之间的协作，谁来做协调工作
3. 节点挂了怎么办（crash, network 做不到 at least/most once）
4. 分布式系统性能很差，大部分时间都是在做网络储存，而没有在做计算，就要考虑优化局部性，要尽可能减少数据搬动的次数。

这几个问题对于所有分布式系统是都存在的，我们来看一下 map-reduce 是怎么实现这件事情的。

Map-reduce 是 2004 年 Google 的一篇 paper，发现 Google 的程序员一直都在写 socket 发送、接收、出错处理等，所以它们都在找一种方法使得程序员以简单的方式实现想要的功能，最终它们选定了 map 和 reduce。

MapReduce: a distributed batch processing system

Created by **Google** (OSDI'04)

- Jeffrey Dean and Sanjay Ghemawat

Inspired by LISP (function language)

- **Map** (function, set of values)
 - Applies function to each value in the set

```
(map #'length '((a) (a b) (a b c))) → (0 1 2 3)
```

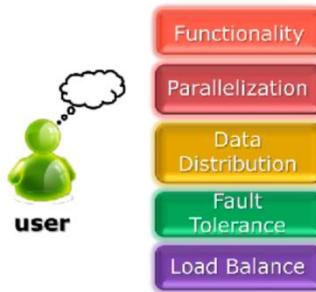
- **Reduce** (function, set of values)
 - Combines all the values using a function (e.g., +)

```
(reduce #'+ '(1 2 3 4 5)) → 15
```

Reduce 就是把多个东西 reduce，而 map 并没有对 list 的数量做任何的改动。

任何做了 map, reduce 是提供给程序员的框架，程序员可以利用这个框架去写 map 和

reduce，不用担心数据是怎么分布的、怎么同步的、怎么分发的。



MapReduce Programming Model

E.g., to find the most popular keywords (热词)



Two Primitive:

Map (*input*)
for each *word* in *input*
emit (*word*, 1)

Reduce (*key*, *values*)
int *sum* = 0;
for each *value* in *values*
 sum += *value*;
emit (*word*, *sum*)

寻找热词，map：对于每个词，变成(词,1)，把 key 的中间表示都加起来，也就是数数。这样做了 word count，这样输入是一个文件 shard。

MapReduce Programming Interface

Map: (input shard) → intermediate (k/v pairs)

- Partition the input data into **M** “shards”
- Group all intermediate values associated with the same key
- Pass them to the **Reduce** function



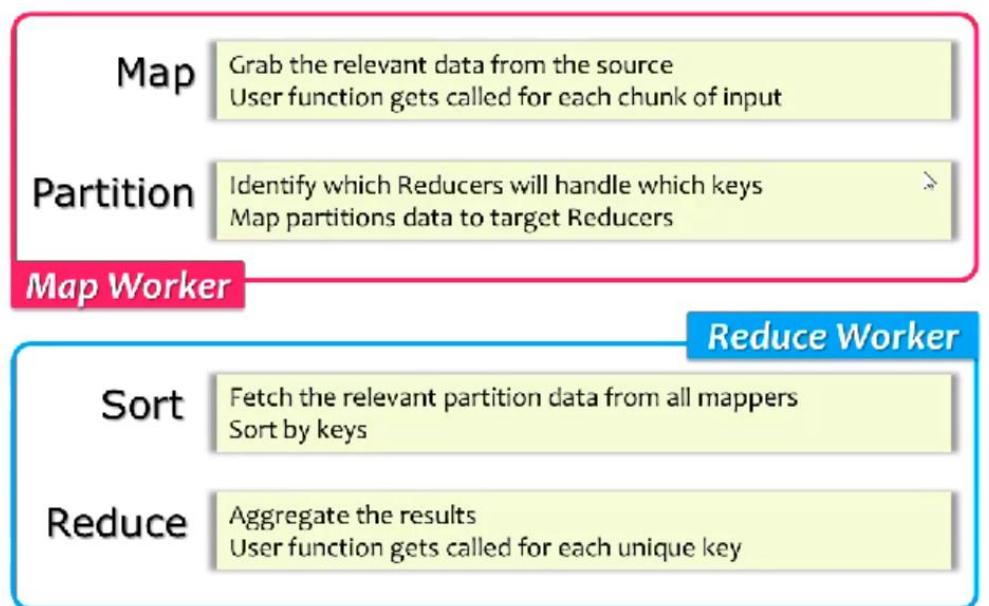
Reduce: intermediate (k/v pairs) → (results)

- Partition the key space into **R** “pieces” using a **Partition** function
 - E.g., $\text{hash}(\text{key}) \bmod R$
- Accept a key & a set of values
- Merge these values to form result of the key

比如我们的 100 个 map 和 reduce，我们就要把大文件切成 100 个 shard，map 就把中间结果发给 reduce。

Mapper 要做的事情是收到数据、得到新数据，发送给 reducer，而 reducer 就是识别、排序和做出相应的动作。

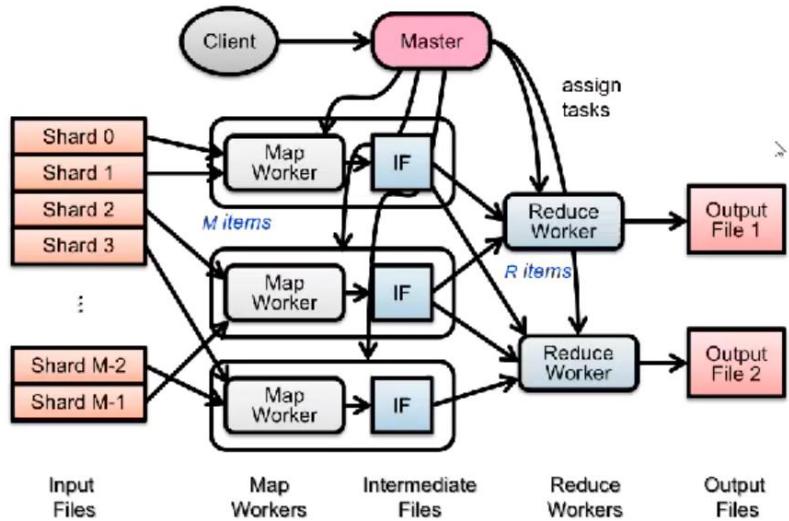
MapReduce Execution Flow



30

用户就是 client，当他要起一个 map-reduce 任务的时候，它就会在一台服务器上起一个 master，然后它就会联系多台服务器创建 map-worker 和 reduce-worker。创建完了之后，

MapReduce: The Complete Picture



文件和中间文件都是存在 GMS，然后做完以后得到生成的文件也生成到 GMS 上。

1. 切成 64K

MapReduce: The Complete Picture

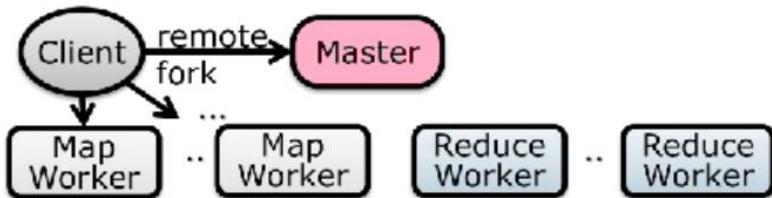
Step2: fork processes

Start up many copies of the program on cluster

- 1 master: scheduler & coordinator
- Lots of workers

Idle workers are assigned either

- Map tasks (each works on a shard)
- Reduce tasks (each works on intermediate files)



2. 很多个 worker,
3. 读到 shard
4. 通过 local write 放到本地，然后生成 intermediate file，去做一个 partition

MapReduce example : Word count

Input file	After Map	After Sort	After Reduce
	[Intermediate]	[In-memory]	
<p>It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and streetstalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in case at least, take the highly-piggledy whale statements, however authentic, in these extracts, for veritable gospel celology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously ...</p>	<p>... it 1 will 1 be 1 seen 1 that 1 this 1 more 1 pains-taking burrower 1 grub-worm 1 of 1 a 1 poor 1 devil 1 of 1 a 1 sub-sub 1 ...</p>	<p>... a 1 a 1 aback 1 aback 1 abaft 1 abaft 1 abandon 1 abandon 1 abandoned 1 abandoned 1 abandoned 1 abandoned 1 abandoned 1 abased 1 abased 1 ...</p>	<p>a 1736 aback 2 abaft 2 abandon 3 abandoned 7 abased 2 abasement 1 abashed 2 abase 1 abated 3 abatement 1 abating 2 abbreviate 1 abbreviation 1 abeam 1 abed 2 abednego 1 abol 1 abhorred 3 ...</p>

这是一段文本，在运行 map 后就变成了<WORD, 1>，做完之后就要对文件做一个排序，为什么要做排序呢，因为同一个词要发给同一个 reducer。然后就可以把最终的结果发在一起。

大家会不会觉得这是一个低效的实现，每个词后面为什么都是 1 呢，这是 map-reduce 为了通用性考虑，可以映射成其他东西，不一定是 word。

Other examples of MapReduce

Distributed grep

- [Search for words in lots of documents](#)
- **Map:** emit a line if it matches a given pattern
- **Reduce:** just output the intermediate data

Count URL access frequency

- [Find the frequency of each URL in web logs](#)
- **Map:** process logs of web page access;
output <URL, 1>
- **Reduce:** add all values for the same URL

Inverted index

- [Find what documents contain a specific word](#)
- **Map:** parse document, emit <word, doc-ID>
- **Reduce:** sort the doc-ID for each word, and output <word, list(doc-ID)>

Reverse web-link graph

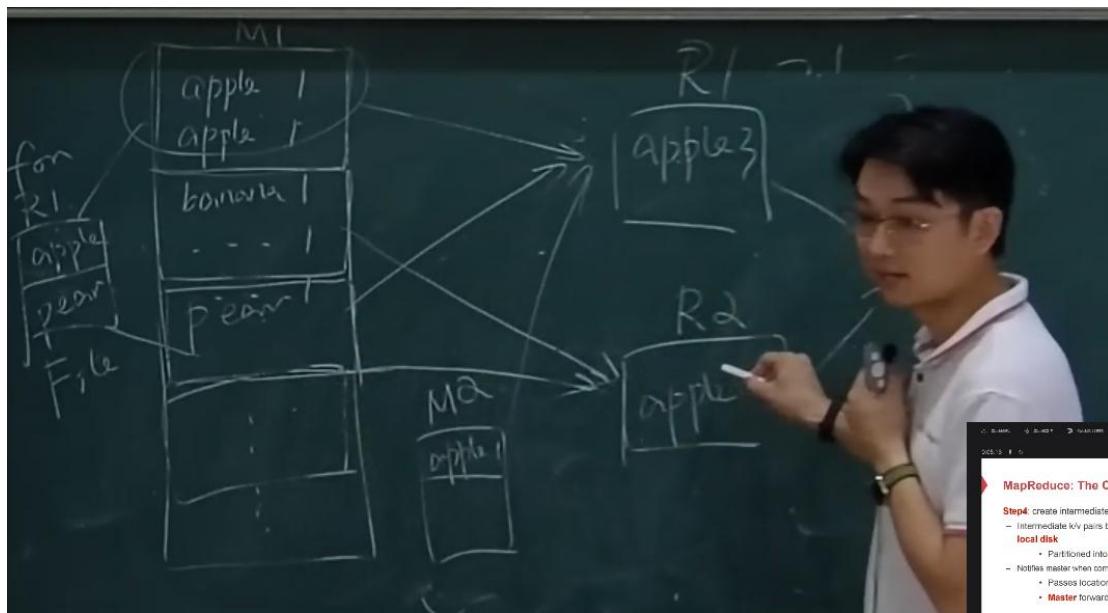
- [Find where page links come from](#)
- **Map:** output <target, source> for each link to target in a page source
- **Reduce:** concatenate all source for each target, output <target, list(source)>

比如我们可以去做分布式 grep，reduce 什么都不用管，而 map 就找到对应的 pattern

反向索引，map 会产生<word, doc-ID>告诉你这个单词出现在哪个文档里，也可以理解为网页。记录下来就可以用来搜索，知道这个关键词出现在哪些网页里。Reduce 就可以知道了<word, list<doc-ID>>。

反向的 web-link 图，可以知道对于某个网页，有多少人指向你。

一些实现的细节，我们必须保证同一个单词发给同一个 reducer，否则不同的 apple 发给不同的 reducer，那么最后 reducer 之间还需要再加起来，这会导致实现很麻烦，我们通过哈希就可以把同一个单词发给同一个 reducer。



mapper 会根据 hash 的值，把喂给同一个 Reducer 的 word 生成一个文件，然后对应的 reducer 就可以通过 RPC/GFS 等方法去把要的文件拿到。Mapper 中没有 partition 的数据在内存里，而 partition 的数据是在本地文件系统（home disk）中。

Map-reduce 提出之后，非常受欢迎，甚至用 map-reduce 实现了 machine learning。那么 map-reduce 到底是怎么解决分布式系统遇到的问题呢？

前两者相对比较简单，map-reduce 的框架帮我们写好了。

MapReduce 的容错机制

Machine **failures are common** in datacenters

- A MapReduce job can possibly run on thousands of machines

MapReduce simplifies fault tolerance due to the following two choices:

1. Programming model simplifies fault recovery
 - e.g., **no side-effect**: a map or a reduce can simply re-execute the computation to recover from failures
2. Builds on a **reliable service** (i.e., GFS)

Map-reduce 的任务很有可能直接挂掉了，有两个选择，

1. 比如 map 和 reduce 可以重新执行，因为在过程中没有全局状态（在本地做的事情对全局没有影响），所以没有 **side-effect**。
2. 需要保存的全局状态就需要 **reliable server**，所以可以使用 GFS 当备份。

Worker failure

- Master pings each worker periodically via heartbeat
- If no response is received within a certain time (**timeout**), the worker is marked as failed
- Map or reduce tasks given to this worker are reset back to the initial state and rescheduled for other worker (**re-execution**)
- Robust: lost 1,600 of 1,800 machines once, but finished fine

Master 不断地 ping，如果 timeout 了，它就会重新指定一个 worker 去执行，无非就是从硬盘上切，如果挂了就直接重新再做一次。

Master 节点很重要，所以需要同步在 GFS 上，一旦发生了 crash，就从 GFS 中恢复出来。信息有：执行状态、中间文件在哪些地方。

Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
- Best solution is to debug & fix, but not always possible
- On segmentation fault:
 - Send UDP packet to master from signal handler
 - Include sequence number of record being processed
- If master sees two failures for same record:
 - Next worker is told to skip the record

Effect: Can work around bugs in third-party libraries ↴

一旦出错之后，发现给多个 mapper 做了任务都崩了，可能不是 mapper 的问题，那就跳过这个任务，可能最后给一个 report 告诉你跳过了哪些记录，这就可以绕过第三方库的一些 bug。这是去对硬件和软件都容错的方法。软件出错被认为是长久的，这时候我们只能绕过去。

MapReduce 提升 Locality 的方法

Locality

Problem: the bandwidth of datacenter network (at that time) is scarce

- If Map worker reads all the data from the network, it would be slow

Google: Input and Output files are on GFS

- Google File System (SOSP'03), see lecture 0x
- Each chunk is replicated on multiple servers (3)

MapReduce runs on **GFS chunkservers**

- Collocate computation and storage

Master tries to **schedule** map worker on one of the nodes that has a copy of the input chunk it needs ↴

因为在那个时候，数据中心的访问带宽是有限的。跟本地的磁盘比起来肯定是很慢的，如果一个 mapper 要把所有数据从网络中来读的话会变的非常的慢。于是谷歌就想出，既然我们已经有了 GFS(一个 master + N 个 chunk server)，我们能不能够把 map-reduce 的 worker 运行在 chunk server 上呢？因为 chunk server 除了做一些网络服务之外，CPU 占用也不高，绝大部分时间都在做网络 IO，CPU 与其空闲，还不如做 MAP-reduce 的任务。这样就把计算和存储合二为一了，不过在逻辑上是完全分开的。Master 会尽可能把 map worker 调度到正好有 mapper 需要的数据的 chunk server。注意我们要区分开上层的 MapReduce 的 master 和底层 GFS 的 master。

底层的 GFS 的 master 和 worker 都对应了一台机器。而上层的 MapReduce 的 Master 和 worker (mapper 和 reducer)，注意这里是每个任务有一个 master 和数个 worker，都是我们启动 MapReduce 任务的时候 fork 出来的进程。

我们之前讲整合的时候，是在 MapReduce 的 Master 里有一个 GFS 的 library，我们就可以知道数据存在哪个 chunk server 上。

冗余执行 (redundant execution)

经常会出现“straggle”的问题，因为在 MapReduce 的早期版本中 reduce 需要等 map 执行完才执行，100 个 map 中 99 个好了，唯独有一个特别慢，那么所有 reducer 都在等着它。Google 工程师发现慢的机器各有各的原因，可能是机器磁盘坏了、还有其他任务占用、CPU 的 cache 出错了。

怎么办呢？比如让闲着的 worker 帮最慢的机器做这件事情，也就是把最后几个任务撒出去做 backup，多做的浪费就浪费了。“饱和式救援”，这样大大提高了速度。这篇论文的意义就是在几千台容易出错的机器之上，让这个系统正确地执行。

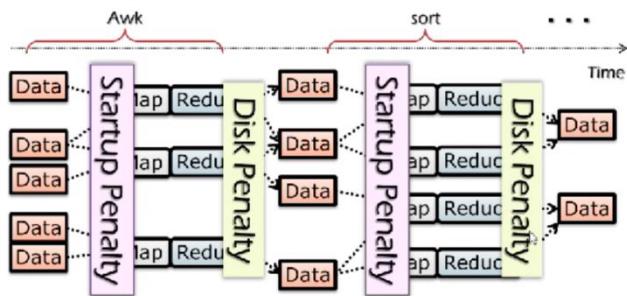
MapReduce 的局限性

Mapreduce 并不适合做所有事情，并不适合做 PageRank。对于图这种结构也不适合。

MapReduce is not optimized for multiple-sages execution

MapReduce runtime is not optimized for iteration

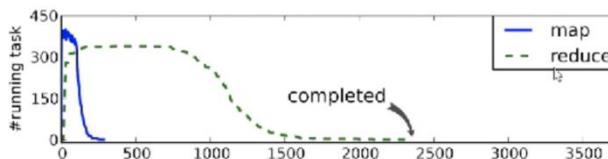
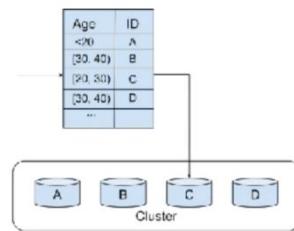
- Persistent I/O (e.g., GFS)



Sharding is critical

Example: Weibo's user data

- Sharding by age?
- Most users in [20,30)
- Node-C becomes the straggler
- How to shard may depend on the application semantics



Dryad

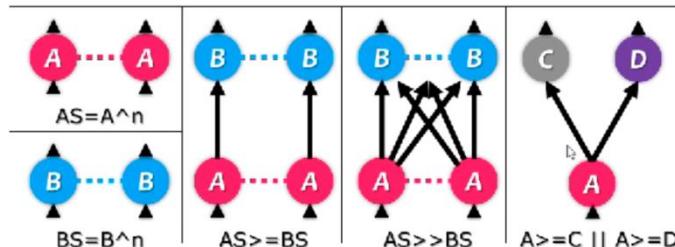
一个解决 MapReduce 的思路就是微软 07 年提出的 Dryad。我们提出一种新的数据流图的概念，也就是我们之前看到的 case，无非就是一个节点算 awk，算完以后给一堆节点算下一个操作。整个就是一个数据流，模式就和 MapReduce 很不一样了。把计算和数据拆分，

在一个个节点上计算，数据在节点之间流动。

Programming model of Dryad

Computations are expressed as a graph

- Vertices are computations
- Edges are communication channels
- Each vertex has several input and output edges



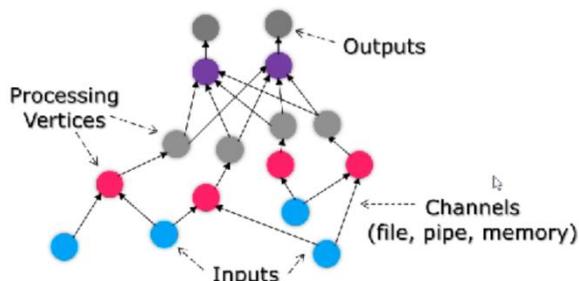
一个节点有多个输入和多个输出。这样我们整个程序的写法就变掉了，当我们使用数据流图的时候，就会产生一个有向无环图。

Dryad uses a dataflow graph

Why?

- Many programs can be represented as a distributed dataflow graph

Dryad Job = Directed Acyclic Graph (DAG)



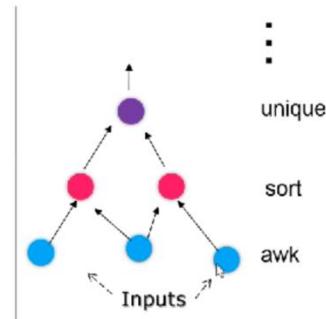
当结束的时候，最终我们就得到了一个 output。

Dryad uses a dataflow graph

Dryad Job = Directed Acyclic Graph (DAG)

Back to our log analysis example

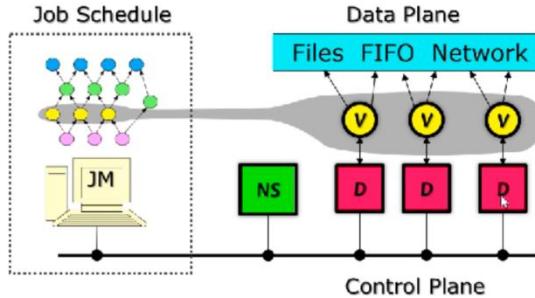
```
cat /var/log/nginx/access.log | ①
  awk '{print $7}' | ②
  sort | ③
  uniq -c | ④
  sort -r -n | ⑤
  head -n ⑥
```



上例展现了 Dryad 是怎样样得到最受欢迎的网页的。

The runtime of Dryad

Vertices (V) run arbitrary app code, exchange data through TCP pipes etc., and communicate with JM to report status



79

JobManager 控制了底下的节点，而程序员就去写这张图，以及对应的 input 和 output 是什么、数据来自哪、数据要给谁。程序员制定完这些以后，后面 job manager 就把节点分配到可用的机器上，然后通过各自方式做传输。

Scheduling in JM

General scheduling rules

- Vertex can run anywhere once all its inputs are ready
 - Prefer executing a vertex near its inputs (**locality**)

Fault tolerance

- Vertex **fails** → run it again
- Vertex's inputs are gone → run upstream vertices again (**recursively**)
- Vertex is **slow** → run another copy elsewhere and use output from whichever finishes **first**

What if the vertex execution is **non-idempotent**?

Advantages of Dryad

Big jobs more efficient with Dryad

- MapReduce: big job runs multiple ($>=1$) stages
 - Reducers of each stage with regard to distributed storage (GFS)
 - Output of reduce → 2 network copies + 3 disk writes
- Dryad: each job is represented with a DAG
 - Intermediate vertices write to local file

Dryad provides more **flexible** operations

2021/10/12

Map-reduce 可以让程序员的效率提升十倍，我们需要知道哪些是可以提取出来的，哪些是可以放进去的。有了 map-reduce，用户只需要实现 map 和 reduce 两个函数，但它也不是一个万能的，当它要去算 PageRank 的时候，实际上可能就会很慢。在做 PageRank 的时候，map-reduce 批处理的缺点就暴露的很明显，它一定要算完一个 iteration 算完，才能开始下一个 iteration。

所以谷歌最后还是发现要根据应用的不同去开发。目前迎来的硬件大爆发的时代，很多 AI 公司主打的就是 ai 加速芯片。CPU 大量成本是在预测哪些东西和增加 cache 面积，对于 AI 来说，更希望有计算矩阵单元，如果能砍掉 cache，就可以让同样面积的处理器性能达到几百倍。

Big learning example

Social Arithmetic

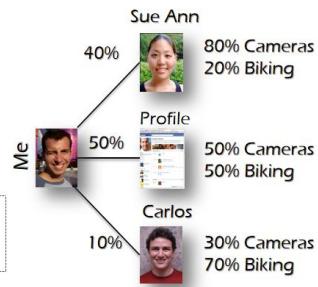
50% What I list on my profile
 40% Sue Ann Likes
 + 10% Carlos Likes
 I Like: 60% Cameras, 40% Biking

Recurrence Algorithm

$$Likes[i] = \sum_{j \in Friends[i]} W_{ij} \times Likes[j]$$

iterate until convergence

Label Propagation



Parallelism: Compute all $Likes[i]$ in parallel

举个例子来说，我们要预测一个用户到底是喜欢什么物品，如果我们能通过好友信息来推送广告，那么广告的点击率就会高很多。

我们根据这三个权重（自己、朋友 A、朋友 B），虽然对于自己的喜好来说 Camera 和 Biking 是相同的，但是加入了好友的权重以后就不一样了。这个算法存在一个递归的过程，很适合用并发的方式去做的。

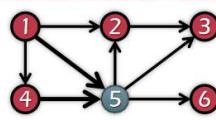
Big learning example

PageRank

$$R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j]$$

α is the random reset probability
 $L[j]$ is the number of links on page j

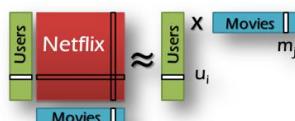
<http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf>



Alternating Least Squares

$$u_i = \arg \min_w \sum_{j \in N[i]} (r_{ij} - m_j \cdot w)^2$$

$$m_j = \arg \min_w \sum_{i \in N[j]} (r_{ij} - u_i \cdot w)^2$$

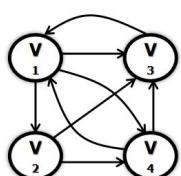


<http://dl.acm.org/citation.cfm?id=1424269>

指向自己的 rank 加在一起乘以 $1 - \alpha$ 。从大数据的角度来看，大量存在这种和图相关的算法。

Example: PageRank

$$PR_{k+1}(u) = \sum_{v \in B_u} \frac{PR_k(v)}{|F_v|}$$



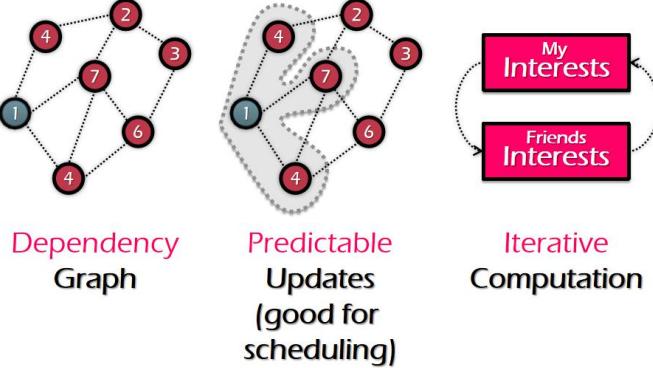
Iterative Batch Processing

Convergence

	K=0	K=1	K=2	K=3	K=4	K=5	K=6
PR(V ₁)	0.25	0.37	0.43	0.35	0.39	0.39	0.38
PR(V ₂)	0.25	0.08	0.12	0.14	0.11	0.13	0.13
PR(V ₃)	0.25	0.33	0.27	0.29	0.29	0.28	0.28
PR(V ₄)	0.25	0.20	0.16	0.20	0.19	0.19	0.19

一开始的时候，每个节点都分到 0.25，然后不断去迭代。

Graph Parallel Algorithm

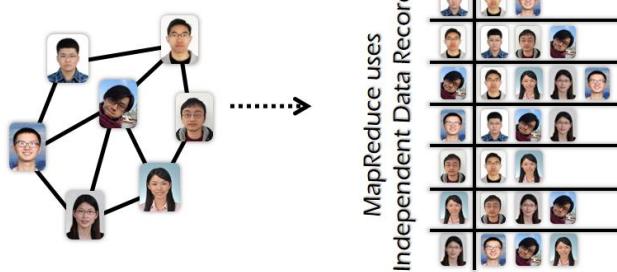


可以组成图的，节点之间是存在依赖的，图是可预测的（可以通过相邻节点来预测，这对于调度就比较友好）；第三就是在这些 PageRank 之间存在迭代这种关系，那么我们怎么样才能实现 graph 级别的计算呢？我们首先来使用上节课学到的 map-reduce，

Dependencies are difficult to express

Difficult to express **dependent** data in **MapReduce**

- Issue#1: Substantial data transformations
- Issue#2: User managed graph structure
- Issue#3: Costly data replication



如果我们要把这张图用 map-reduce 来说的话，因为 map-reduce 抽象层次太多了，每个人的信息至少要存两份，如果一个人出现在多个里面的就要存多份。

使用 MapReduce 来计算 PageRank

PageRank on MapReduce

Multiple MapReduce iterations

Each PageRank iteration:

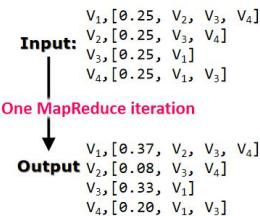
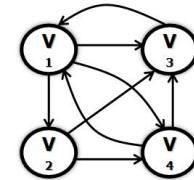
→ **Input:**

1. (**id₁**, [PR_t(1), out₁₁, out₁₂, ...]),
2. (**id₂**, [PR_t(2), out₂₁, out₂₂, ...]),
- ...

→ **Output:**

1. (**id₁**, [PR_{t+1}(1), out₁₁, out₁₂, ...]),
2. (**id₂**, [PR_{t+1}(2), out₂₁, out₂₂, ...]),

Iterate until **convergence**



当前节点的 PageRank，以及对外权重要出到哪个节点。Out 用来记录网络的拓扑结构，如果网络的拓扑结构没有变化的话，那么 out 都是不用变的。右下角，这是我们希望看到的结果。

那么我们怎么来做呢？在实现的时候，我们先要实现 map：

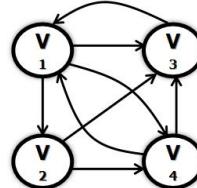
Map

→ **Input:**

- (V₁, [0.25, V₂, V₃, V₄]);
- (V₂, [0.25, V₃, V₄]); (V₃, [0.25, V₁]);
- (V₄, [0.25, V₁, V₃])

→ **Output:**

- (V₂, 0.25/3), (V₃, 0.25/3), (V₄, 0.25/3),
..., (V₁, 0.25/2), (V₃, 0.25/2);
- (V₁, [V₂, V₃, V₄]), (V₂, [V₃, V₄]), (V₃, [V₁]), ...



Map 输出的第一个 item，表示把 0.25/3 分别送给 V₂, V₃, V₄，每个节点都会生成这样一个 item。最后一个 item 是表示了图的拓扑结构。

一旦有了这个之后，我们就可以去做 shuffle 了，shuffle 毫无疑问就是拿 v₁ 作为 index 把所有 v₁ 打头的，全部排在一起。

接下来，我们 reduce 就是，对于每个节点来说，把后面的数字加起来，和后面不是数字的，做一个整合。

Shuffle

→ **Output:**

- (V₁, 0.25/2), (V₁, 0.25/1), (V₁, [V₂, V₃, V₄])); ...;
- (V₄, 0.25/3), (V₄, 0.25/2), (V₄, [V₁, V₃])

Reduce

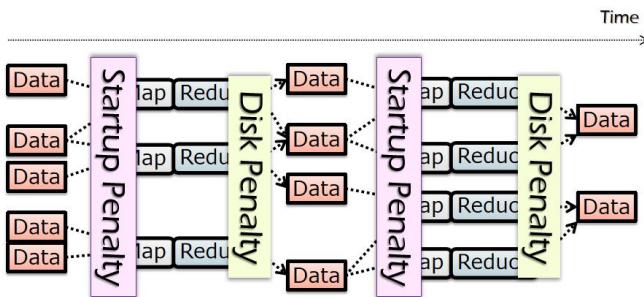
→ **Output:**

- (V₁, [0.37, V₂, V₃, V₄])); (V₂, [0.08, V₃, V₄])); (V₃, [0.33, V₁])); ...

Recall: Iterative computing is difficult for MapReduce

MapReduce runtime is not optimized for **iteration**

- Persistent I/O (e.g., GFS)



需要花很多时间的 **startup** 启动，还要做很多 GFS 的写，还有每一轮之间信息的交互，所以用 **map-reduce** 来解决 PageRank 问题，**overhead** 实在是太高了。当时还有一种方法就是 **pthread**，它是一个线程库。

Threads, Locks and Messages

Graduate Student

ML experts repeatedly solve **the same** parallel design challenges

- Implement and debug complex parallel systems
- Tune for a specific parallel platform

6 months later, a conference paper contains:

“We implemented ____ in parallel.”

However, the resulting **code** will be difficult to **Maintain** and **Extend**

couple learning model with parallel implementation

实现的并行代码可能换了机器就不能跑了，这就是把具体的算法和并发的实现耦合的太紧了。我们还是希望把容错和并发抽取出来，最终我们得到一个面向图的编程框架，需要满足：程序员只需要关注应用的逻辑，不需要关注底层到底是几台机器。

Pregel 框架

BSP 和 Pregel，BSP 是编程的一种抽象，而 Pregel 是具体的框架。

Pregel (Google, 2010)

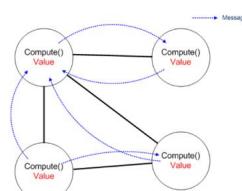
Inspired by Valiant's **Bulk Synchronous Parallel** (BSP) model

Communication through pure **message-passing**

- (usually sent along all the outgoing edges from each vertex)

Vertex-centric programming

- “Think like a vertex”

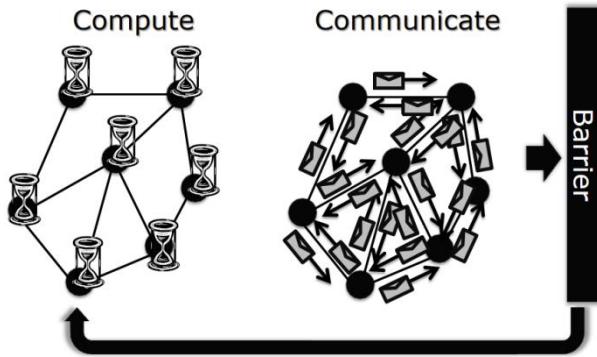


谷歌提出了用大量便宜的机器去搭集群，取得了很多成果。以点为核心的编程框架。之前我们提到的是假如自己是一个上帝，面前有很多台机器，怎么去交互。而用点的角度去编

程，程序员在思考的时候会更加的简单。

我们刚才说的是用 map-reduce 去实现 PageRank 算法，PageRank 是需要完成多次迭代之后才会收敛。

第一个是算，一声令下大家开始算，第二个是传，第三个是等。



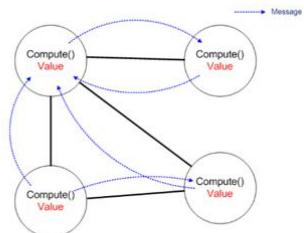
在算完之前不能进入下一次传。这个看起来很简单，但里面有很多潜在的选择，比如是单个节点算完了就可以传，还是大家都算完了才能开始传。

Vertex-centric programming

In Vertex-centric programming, each vertex will:

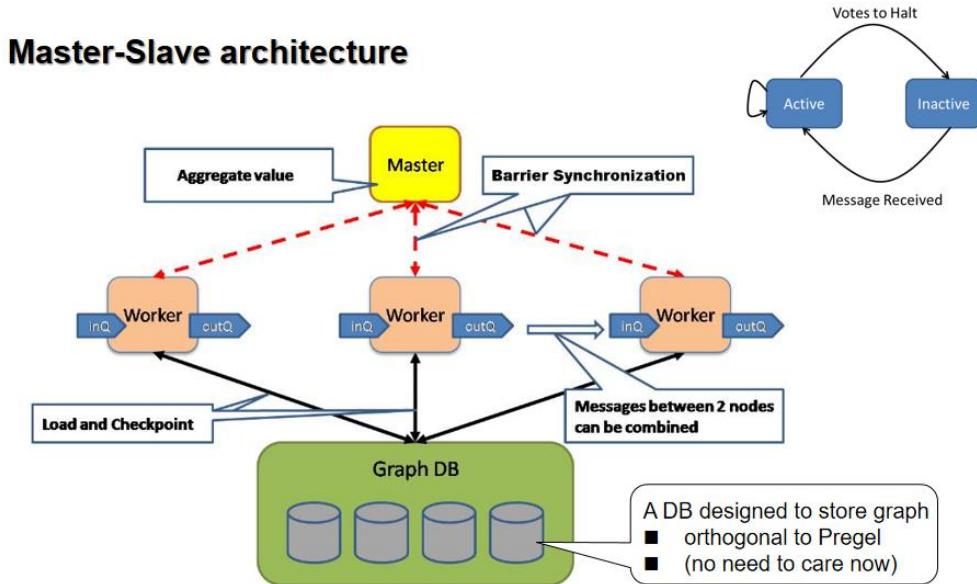
1. **Receives messages** sent in the previous **superstep** (aka, iteration)
2. Executes the same **user-defined function**
3. Modifies its value
4. If active, sends messages to other vertices
(received in the **next** superstep)
5. Votes to halt if it has no further work to do → inactive

Terminate when all vertices are **inactive** and **no messages** in transmit



第一步就去收消息，然后对本地的 PageRank 做更新，然后改完自己的以后，就去把消息发给别人，最后一件事情就是 vote to halt。因为我们发现新的 value 和旧的 value 没什么区别，已经差不多收敛了的情况下，我们不一定要真的做到值不变了，我们更新的值小于某个阈值了，然后开始投票，大部分节点都收敛了的情况下，这个运算就结束了。

System Architecture of Pregel



发消息的过程和收集哪些节点 inactive 的过程，都是 master 来做的。然后底下有一层存储，这是对 graph 的存储但是这个不重要。

接下来我们要看的是它的实现，是因为这种 message-passing 的流程很典型。

Pregel 计算 PageRank

```
Superstep 0: PR value of each vertex 1/NumVertices()

Class PageRankVertex {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep () >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
        }
        *MutableValue() = 0.15/NumVertices() + 0.85*sum;
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
}
```

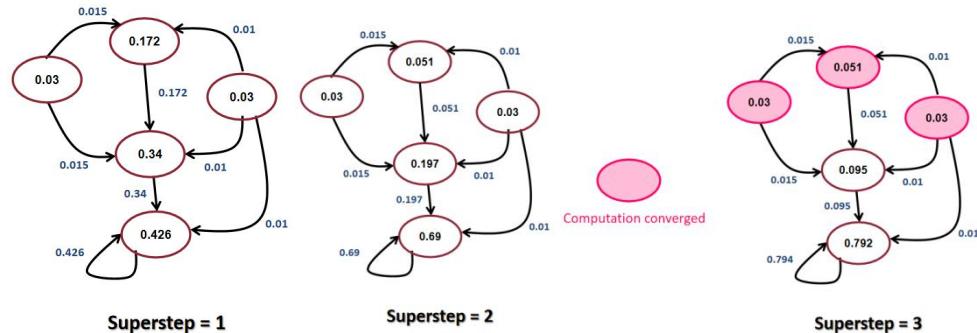
AC

Set to 30 by experience
GetValue: get value set by MutableValue()

参数就是别人给它发的一对 message，是谁给它发的这件事情是不知道的，因为对应每个节点来说，它是不知道整个图的结构的，它只需要知道 message 即可。Message 里所有的 value 全部加起来，然后算一个 sum 对应的函数，然后设置到 value 里去，然后调用 GetValue 的时候就得到了存的值，然后在没有收敛的情况下或者迭代次数小于 30 的情况下，可以知道有多少条边。然后直接调用 SendMessageToAllNeighbours 即可，底层记录了这个 graph 长什么样子，对于单个节点上的程序员来说，不需要 care 底层这张图长什么样子。如果程序

员去写了这个代码，我们作为系统程序员该怎么去做呢？

对于 Pregel 来说，它需要考虑 message 怎么传，对每个节点要暴露出什么对应的 api。这个思路和只需要让程序员写 map 函数和 reduce 函数是类似的。



Pregel 的容错机制

Failure detection: master pings each **worker** periodically via heartbeat

- Similar in MapReduce

Failure handling

- **Checkpointing:** at the beginning of a superstep, checkpoint the graph data to a persistent storage (e.g., GFS)
- After failure, can recover from checkpoint & re-compute
 - For either worker or master
- The frequencies of checkpointing is determined by the mean time to failure
 - Can measure MTTF in advance

首先，master 会得到 worker 的心跳，如果 crash 了的情况，会尝试 recover worker，为了保证不要完全重新再来，它会定期地写一次 checkpoint，写到持久化存储中，这样我们恢复的时候就可以从 checkpoint 中恢复，接下来的问题就是多久做一次 checkpoint 呢？在工程中，我们使用 MTTF (mean time failure) 参数，算完均值之后，我们就知道差不多有可能要出错了，那就可以做 checkpoint。

Pregel 就是算、传、等。程序员看到的只有收发 message，使用了 graphDB 来做存储。它是 vertex-centered programming，在 map-reduce 中一轮轮都要写到磁盘，而 pregel 不会写磁盘，中间算出来的值都直接变成 message 发出去了。

Problem#1: programming model can be more friendly

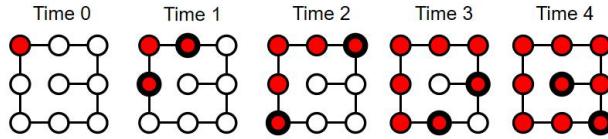
Message API is irreverent to the graph

```
Class PageRankVertex {  
public:  
    virtual void Compute(MessageIterator* msgs) {  
        if (superstep () >= 1) {  
            double sum = 0;  
            for (; !msgs->Done(); msgs->Next())  
                sum += msgs->Value();  
            *MutableValue() = 0.15/NumVertices() + 0.85*sum;  
        }  
        if (superstep() < 30) {  
            const int64 n = GetOutEdgeIterator().size();  
            SendMessageToAllNeighbors(GetValue() / n);  
        } else {  
            VoteToHalt();  
        }  
    }  
}
```

对于程序员来说，这里比较简单，但是序列化和反序列化的时候就比较复杂，我们希望能像单机一样来写这个程序，不用管 message。

Problem#2: the issue of BSP

Example Algorithm: If Red neighbor then turn Red



Bulk Synchronous Parallel Computation :

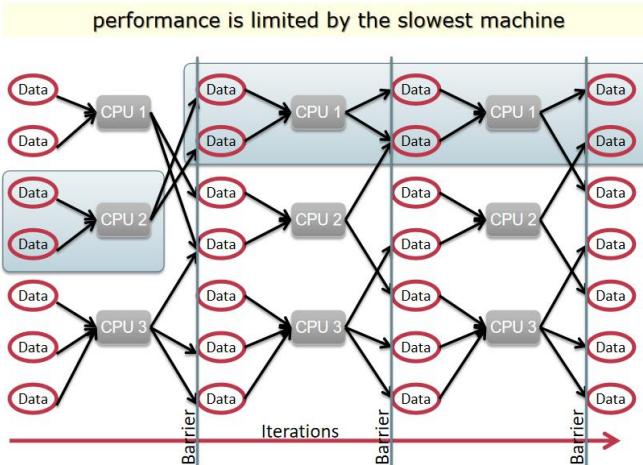
- Evaluate condition on all vertices for every phase
- 4 Phases each with 9 computations → 36 Computations

Solution: asynchronous Computation:

- Evaluate condition only when neighbor changes
- 4 Phases each with 2 computations → 8 Computations

在传染算法中，对 BSP 来说，有 4 个迭代，每个迭代都是算、传、停，但是每次都只有 2 个需要改，其实也只需要 8 次，但一共我们计算了 $4 \times 9 = 36$ 次。性能差了 4.5 倍。

Problem#2: the issue of BSP



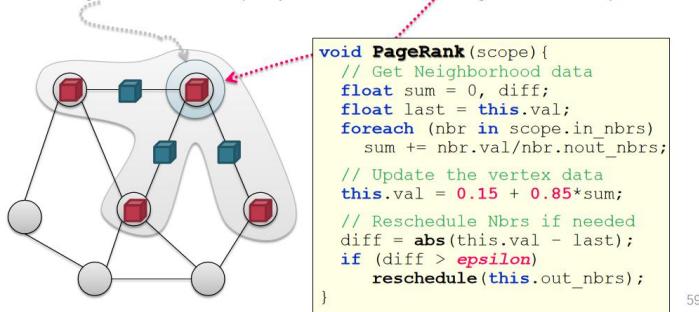
慢的机器会拖累快的那些人，因为同步不需要等待所有事情做完，接下来我们来看怎么

去解决这些问题。

Programming model: shared memory

Shared memory

- a user defined program which when applied to a **vertex** transforms the data in the **scope** of the vertex (scope is the set of all neighbor vertices)



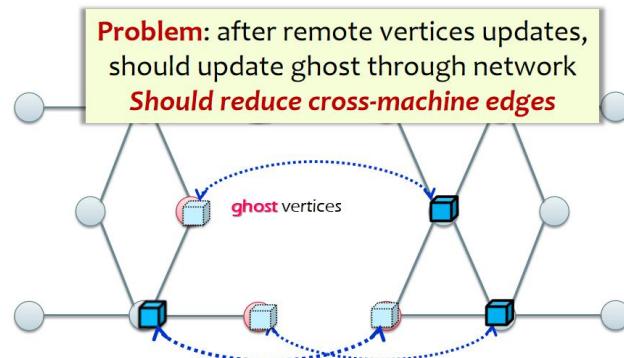
59

用户只需要定义一个 update function，每一个 vertex 会有一些邻居，跟这个 vertex 相关的，也就是那些邻居，我们就把画圈的 vertex 的周围的几个成为一个 scope。所以此时 PageRank 中传进来的不再是 message，而是 neighbour，然后遍历这个 scope 去获得 neighbour 的值。如果大于阈值，那我们就继续让我们的 neighbour 继续算，也就是 reschedule。这种方式更像是从 neighbour 的 scope 中把数据主动地拉过来。

Problem: what if the neighbor is at a remote machine?

Distributed Graph

Ghost vertices maintain adjacency structure and replicate remote data



Ghost vertices 就是镜像，当拆的时候会 copy 一份。对于 ghost 节点都会有一个主节点对应。有了 ghost 节点，对于这个图之后，对于每个非 ghost 节点，neighbour 都是全的，这样就可以在本地计算而不需要走网络了。

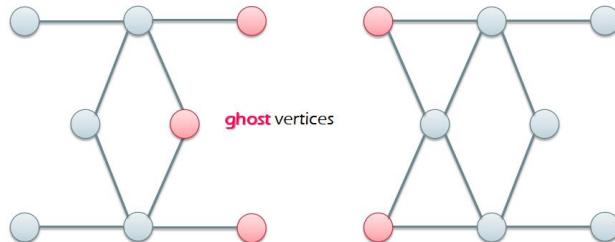
Problem: after remote vertices updates,
should update ghost through network
Should reduce cross-machine edges

在切的时候，有很多方法，这是一个 NP-Hard 问题。

Distributed Graph

Cut efficiently using HPC Graph partitioning tools (e.g., ParMetis, ...)

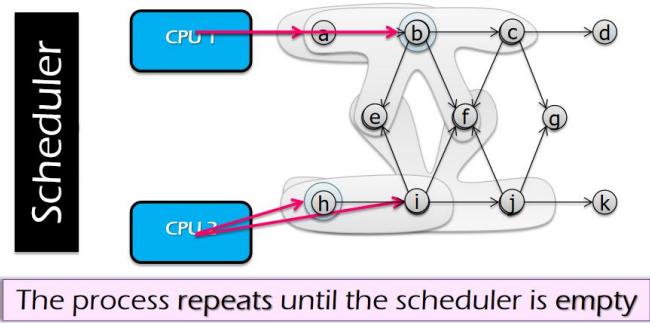
- Optimal partition is NP-hard
- Existing algorithms (e.g., ParMetis) provides good heuristics



加速 processing，我们使用的是 `async`，也就是没有严格的算、传、等。而是异步的，所以需要一个 `scheduler` 去做整体的协调。也就是最后一行 `reschedule`。

Asynchronous Scheduler

The scheduler determines the **order** that vertices are updated

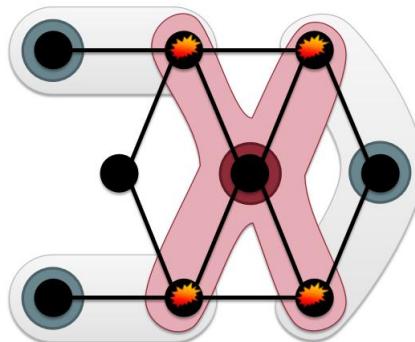


所以 `rescheduler` 就是谁要更新就去更新它，一定是上家要求你更新了才会去更新。但是在更新的过程中，并不存在一个时刻 t 可以让我们去把它刻画出来，异步意味着有可能导致正确性的问题。

Problem of `async`: race and race-free execution

Race: what happen if two UPDATE concurrently execute?

How much can computation overlap?

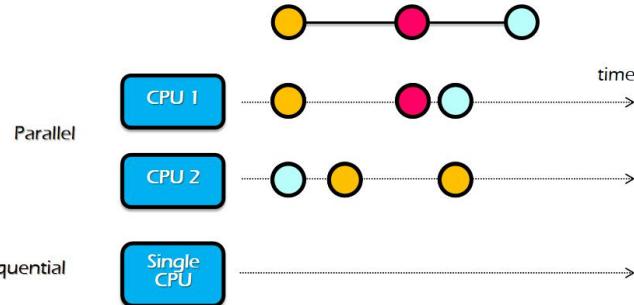


外围的三个更新是可以同步做的，但是更新中间节点的时候，因为 `scope` 有重叠，会导致前一次的丢失。这在同步的 `BSP` 中是没有的问题，我们为了性能丧失了 `BSP` 中没有 `overlap`

不会把对方覆盖掉。

Sequential Consistency

For each parallel execution, there exists a sequential execution of update functions which produces the same result.



这是一个非常非常复杂的问题，之后会有专题。我们提出了 **sequential consistency**，大家遇到冲突的时候就开始排队。正确性的定义：虽然是多核执行的，等价成单核之上的排队操作。怎么实现呢？其实也很简单，就是给节点的 scope 加锁。

Consistency through R/W Locks

Multicore Setting: R/W Locks (Pthread)

- Full consistency
- Users do not need to consider lock (done by the framework)



拿锁的顺序一定要按照节点的顺序拿。

2021/10/14

通用的系统效率不高，我们需要避免一件事情太通用的情况。所以在 map-reduce 的基础上，很有应用的拓扑结构通常是以图的形式存在，包括电影点赞、好友关系、网页链接，我们需要基于图的边和点做计算，这就提出了关于 Graph 的应用。我们提出了 Pregel 和 GraphLAB，在这些系统中都用到了存储设备，需要把数据存下来，有 file、data base、key-value store，我们提到 GFS 中是三备份。

Key-Value Store

但是这些里最简单的、用的最多的都是 key-value store。它的接口非常简单：

GET(K,V) SCAN(K,V)

UPDATE(K,V) INSERT(K,V) DELETE(K,V)

Scan 可以找到 n 个 key 的值，我们看到 key-value 的接口和前面提到的是更加简单的。如果我们把 key 对应到一个文件名，把 value 对应到文件的话，我们完全可以使用文件系统来实现 key-value store，如果有这个在，对 key 的搜索就可以变成文件系统中的 lookup，读取就是 open+read，insert 就是 create+write，update 就是 open+write。这是一个很简单的实现，为什么我们不真的用文件系统实现 key-value store 呢？

第一个问题，get 时用到了 open+read+close，但是我们很可能只需要读一个很小的字符串，我们需要调用三次系统调用，key-value store 的特点就是不会特别长，比如说淘宝页面下的访问量，只是一个数字。一个文件最小需要 inode+4K，这就造成了浪费。同样，对于读来说，我们可能只需要读几个 byte，但是为了读一个文件，我们要从目录开始读 inode-table 再去读 block。

一个直观的想法是：既然一个文件存一个 key-value pair 很浪费，我们是不是能用少量的文件来保存大量的 key-value pair 呢？底层复用了之前的文件系统。

如果存储的方式是 K,V,K,V,K,V。假如我们要更新这个 kv pair，如果 V' 比原先的 V 长怎么办？难道后面的 KV 都要往后移动吗？

在内存中，我们如果想要存储一个键值对，使用 map、hash 等结构的时候，我们不需要考虑这个问题。因为 map 结构里面都是指针，只需要改指向的地址就行了。

Kv 文件的这个问题，最简单的解决办法就是，我们把 update 变成 insert，每次都往后 append。换句话说，原先是 K1,V1,K1,V1,K1,V1，在 update 后就直接往后 append。K1,V1,K2,V2,K3,V3,K2,V2'。Delete 的话其实就是 append 一个 deleted，但是因为字符都可以作为 value 被存储，我们可以使用转义字符的方法：

假如我们使用 8 个 1 作为 deleted。

那么我们可以设置规则，读取到 7 个 1 后自动删除一个后面的 0。

所以真的要表示“11111111”的时候，实际上就是存储 111111101，因为 7 个 1 后的 1 个 0 在转义的时候被删除了。

这个是对于磁盘友好的方法，因为磁盘是顺着转的，我们在后面 append 就会好一点。换句话说，磁盘顺序写的性能远大于随机写。

Log-structured File

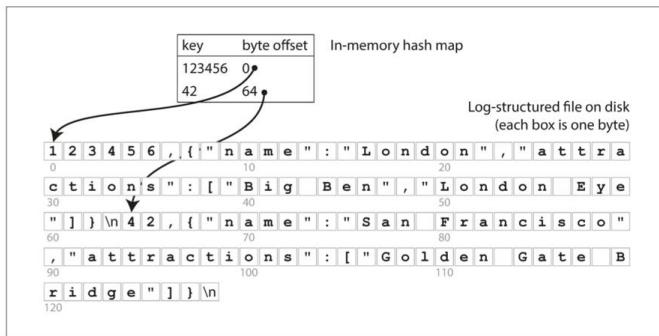
这个叫做是 log-structured file，好处是新的值和旧的值都在，甚至可以做恢复，在发生 failure 的时候可以做恢复。

接下来我们来看，刚才我们说更新是没问题的，但是 get 怎么办？我们需要从后往前读磁盘，但是如果一个 key 一开始更新了，后面再也没更新过，这就会导致要全部扫描一遍文件，性能很差。要解决这个问题，我们就需要加一个索引。这个数据结构会对整个性能造成很大影响，我们可以用之前提到的 hashtable。

In-memory Hash Index

The simplest possible indexing strategy

- Keep an in-memory hash map that map key -> byte offset in the log file



Byte offset 就是在磁盘上的 offset，整个下面的就是一个文件。这个 offset 就可以类比成内存中的指针，也不用考虑变长变短的问题，在内存里如果我们记录了这个 hash table，在 get 的时候，我们只需要先在内存中的 hash table 找到 offset，然后再去硬盘上找就行了。对于磁盘来说，只需要读一次对应 offset 的位置。

Q: 如果我们用这些结构，突然发生了断电，会不会会发生严重的问题？

A: 不会，因为内存里的 hash table 虽然丢了，但是我们可以重新通过从尾到头扫描一遍来重构这个内存中的表。所以它读写很快，容错也很强。

但是它有一个很大的缺点，就是 index 必须在 RAM 中塞得下，当我们在使用这种结构的时候，它比较适用于 workload 包含了大量的更新，但是比较少的插入的时候就比较适合。

做任何的 update、delete 操作都会让文件变长，我们是不是可以把之前的无效的 key-value 删掉呢？这就是 compaction 操作，但是 compaction 完，上面的 hash table 的同步又是一个问题。

可以做到如果让我们去设计的时候，是让整个系统停下来，还是让部分操作可以做。有一个问题就是，如果文件特别大，当我们这个文件变成一个 G 的时候，我们做压缩的时候，需要花十几秒时间，这十几秒都不可以去写。怎么解决这个问题，我们可以把它分成多个文件，变成一个个 segment，每个文件的大小是固定的，当我们做 compaction 的时候，能够以每个文件的粒度去做 compact，还可以把多个 segment 去 compact 到一个 segment，这样我们就可以把阻塞的时间粒度变细。

Recall: 如何处理 log-structured file 中的 delete 操作呢？

A: 在 delete 的时候添加一个 NULL 条目，在 compaction 的时候进行垃圾回收操作。

下面一个问题，我们想扫描一遍 Kitty0000 到 Kitty9999 之间的 key 以及对应的 value 是什么。到目前为止我们的系统不能很好地支持呢？因为内存中的 hash 不排序，所以我们不能用很好的方法去找到它。在实际中，range query 的应用非常常见。

我们先来讲 RAM 大小，index 在内存中放不下该怎么办？把 index 放到磁盘是比较慢的，在选择 hash 算法的时候的选择也很有讲究。如果冲突了，我们就可以使用 list 连在相同 key 的后面，这个的缺点就是会比较慢。我们还可以使用布谷鸟哈希算法，具体见高级数据结构笔记。布谷鸟哈希：在踢掉冲突的情况的时候，容易导致硬盘反复的随机写，这个比较慢；其次就是 rehashing 是个很慢的过程。

上节课，我们为什么要讲布谷鸟哈希，当我们遇到内存放不下 hash map 的时候，我们就要选择更适合存在硬盘上的哈希，为什么布谷鸟哈希更适合存在磁盘上呢？因为它的读的操作最多就是 2 次，把磁盘的访问开销降到了最低，但是它的插入操作因为涉及到了踢，也

会比较复杂，我们也可以限制踢的次数为两次。但是对磁盘上的数据做 `rehashing` 的时候，是非常非常慢的。

下一个问题是，我们使用布谷哈希作为 `hash` 策略的话，它更适合哪种场景呢？

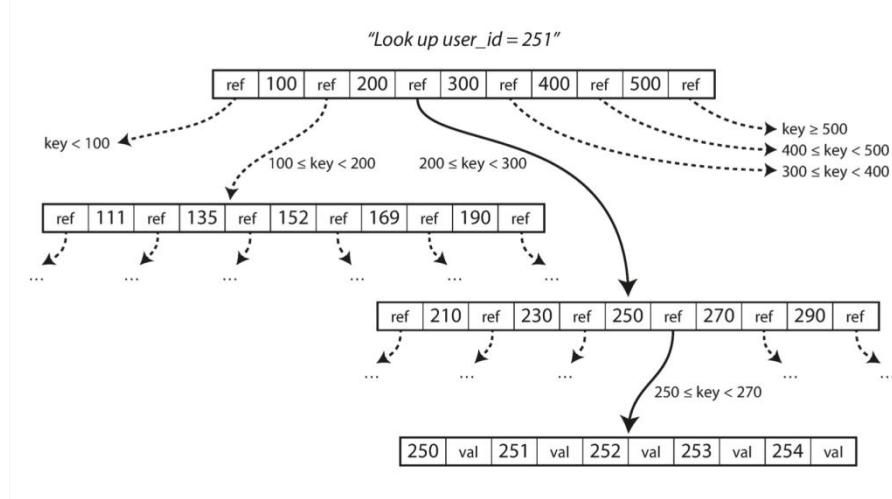
Not just choosing a data structure

Trade-offs of selecting indexes

- **Read** performance vs. **update** performance
 - An index needs to be updated upon inserts/deletions
- Whether index **is friendly to** the storage?
 - E.g., random vs. sequential
- Others (e.g., storage overhead)

但是它也不支持性能好的范围查询。

接下来我们就来到了 B 树：



读操作往往被 `cache` 挡掉了，所以在 `storage` 这个 `level` 读比写大很多是不成立的。所以 B 树的 `index` 也是不太适合，那么我们尽可能减少它的随机写操作呢？减少写操作的方法，就是把它 `append` 到最后，但是 `append` 就没有排序，这就不能做 `range query`。我们现在想能不能对磁盘上的 `segment` 做一个排序，这就是我们的 `sstable`。

SSTable

`SSTable` 是 `LSM tree` 的很重要的一个方法，对于每个 `segment` 都变成一个 `sorted string table`，如果太大了我们就创建新的文件。我们通过 `compact` 的方式来压缩旧数据。和 `log based` 的区别就是它在 `segment` 中是 `sorted` 的，但是因为排过序，它必然不是顺序写，而是在中间写，它的 `merge` 操作相对于 `log-based` 会快很多，因为是归并排序。查找的时候也会快很多，因为它排过序。这时候我们可以放一个 `sparse index`，一个 `sparse index table` 对应一个 `sstable`，在内存中维护了指向每个 `entry` 的指针。对应硬盘中的 `4T`，`sparse index table` 内存中的大小为 `128M`，这个大小肯定够了。我们排序的目的是做 `range query`，这样我们现在内存中找，然后再去磁盘上找。

随机访问怎么解决，这就需要 `memtable`，它是一个内存中排过序的内存结果。`Sstable` 太多了就做 `compact`，`memtable` 太多了写到 `sstable` 中。

有了 LSM tree 后，把 sstable 组成层级结构，这对于磁盘是非常合适的。

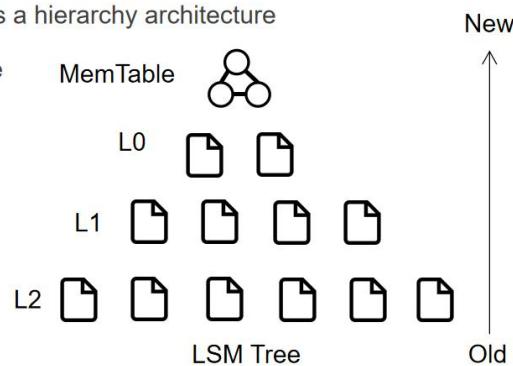
LSM Tree

The SSTable & operations are essentially LSM Tree

- Log-structured merge tree

LSM Tree organizes SSTables as a hierarchy architecture

- Each layer has maximum size
- Except L0, all files in layers are sorted



当发生 crash 的时候，会发生什么情况呢？Sparse index 可以重建，但 memtable 怎么办？

如果 memtable 丢了，那就真的丢了，为了不让 memtable 丢失，有一个方法就是为 memtable 单独放一个 log，它纯粹是 sequential write，没有排序，因为它是用来做恢复的。Range query 就是对每一层都做一个二分查找，把每一层查到的结果合在一块，这也不是一个非常优化的实现，比起之前提到的 B+Tree 还是差一点，但是比 Hash 的结果好一点。

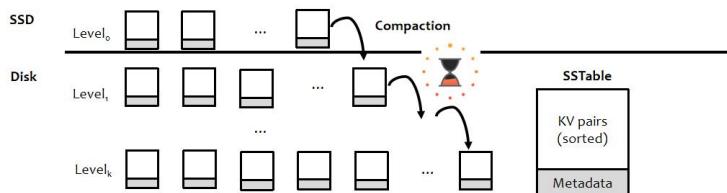
LSM tree 对于内存放不下的数据是很有用的，和 B tree 的比起来就是顺序写，缺点就是要额外的 compaction，range query 的效率比 B tree 低。

内存想写的时候，第一层还在做 compaction，这时候内存就停在那里了。那该怎么办呢？我们需要把 compaction 的过程变快。我们可以把前面这几层存到 SSD 里面，SSD 的速度还是比磁盘快一点。这里我们考虑的是用少量的 SSD 取得最大的优化的效果。

In principle, hard to prevent

Can only alleviate

- E.g., speed up compaction & merge process with **advanced hardware**
- **Observation:** SSD is much faster than disk on storage
 - Using it to store up-layer SSTables



这样就提升了第 0 层 compaction 的速度，降低了 memory 的压力。

第二个是 non-existent 的 key 是非常麻烦的事情，如果存在一个 key，那么在上层就找到了；如果不存在，需要找到最下层，所以我们需要增加 bloom filter。

对于 分布式的情况，我们可以使用 RPC 把 get 和 put 变成远程方法调用，这就变成了一个分布式的 key-value store，我们该怎么去分配这个数据呢？我们怎么去 shard 呢？这个

操作是要考虑到 scalability 的，如果把 hot 的数据放到一台机器上，那么 load balance 就做的不好。那么我们怎么找到数据呢？现在我们的 sparse index 在不同的机器上，是否要排序呢？如果不排序，我们该怎么做 range query 呢？最后我们会提到一致性哈希算法。

2021/10/19

Crash Consistency

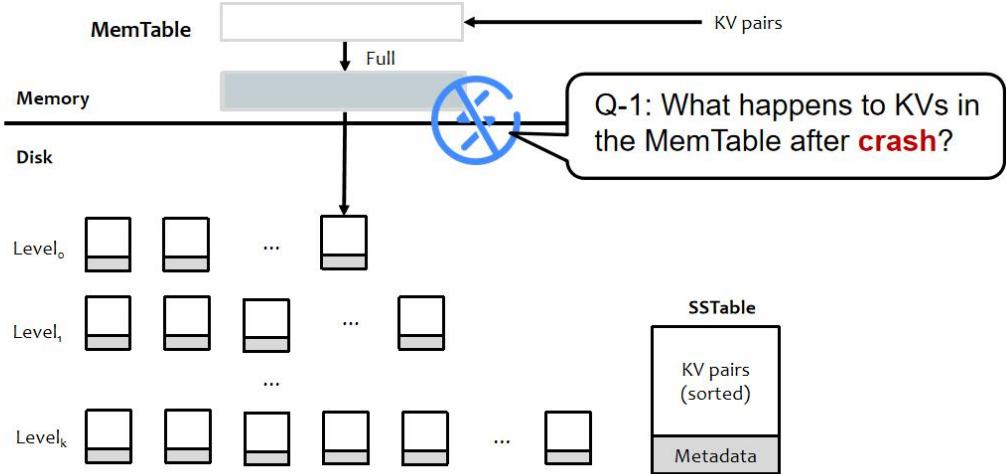
今天开始讲容错这块，我们关注 crash consistency 的问题。我们为什么要关注这个问题？我们讲了很多分布式系统中的组件、我们现在有 file server 存储 key-value store、data 等。用户只需要通过 key 就可以访问到 value，在最早的文件系统中，可以使用 key-value store 来实现文件系统，也就是把 block-number 变为 key，然后 file 就是对应的文件。到底谁更底层呢？实际上没有需要比较。可以认为是在 block 之上的 key-value 层，也可以认为是 block-number 和 4K 的 key-value pair。在分布式系统中，提供 kv 的方法可以有很多种，基于分布式的 kv store 提供一层文件系统，我们是不是可以构建出分布式的文件系统呢？

那么我们就要讲 abstraction 和 virtualization 的问题，abstraction 就是提供新的 API，而 virtualization 就是提供旧的 API。open/read/write 是 OS 提供的抽象，这就是 abstraction 的能量。再往上的話，大家到更上层的应用，比如我们的 database、前端、object，这些抽象都有自身应用的范围。Virtualization 是非常底层的技术，不创造新接口只复用旧接口，这就是为了兼容性、对接上层程序等，我们希望有一种对程序和人都比较熟悉的接口，那就是已经存在的接口。虚拟机对应磁盘中的一个镜像文件，可能是十几个 G。这个文件对虚拟机来说就是一个磁盘，对于虚拟机来说，我们需要把一个文件变成一个磁盘。磁盘的接口就是给你一个 block number，返回给你一个 4K 的 block。所以我们居然在一个文件的抽象之上构建了磁盘的接口，而文件本身是构建在磁盘之上的，这有点递归的感觉。因为虚拟机认为自己在访问磁盘，所以我们需要提供磁盘的接口；换而言之，如果有一个程序想要文件的接口，那么我们就要提供给这个应用文件的接口；如果一个分布式应用想要文件的接口，而我们底层是 kv store，那么我们也要提供一层文件的接口，这就是 virtualization，复用已有的接口。

所以在计算机中，所有的问题都可以加一层抽象来解决。

Kv store 建立在文件系统之上，不能每一个 key-value 用文件来保存，因为对文件读写的时候会导致大量的 Open 操作。一个非常直观的方法就是一次 Open，那么我们需要把文件整合到一个文件中。但是这就会遇到一个问题，当我们想要更新中间的 kv 变长、变短，那就会很麻烦，要把所有的东西往后挪动。我们是不是能利用磁盘顺序写非常快的特点（事实上今天我们有了 SSD、不依赖顺序写的速度等，假设快被打破了，那么对应的设计也要做变化），不断 append 往后加。我们需要支持 range-query，最后我们采取的是妥协的方案，如果我们在整个磁盘的维度一直写一个文件，我们可以切成一个 fixed size (SSTable)。我们可以再把随机写的操作抽象成 memtable 在内存中处理。L1 之后，因为访问的次数不是那么频繁，我们可以用一次比较“浪费时间”的操作，在做 merge 的时候顺便排序。

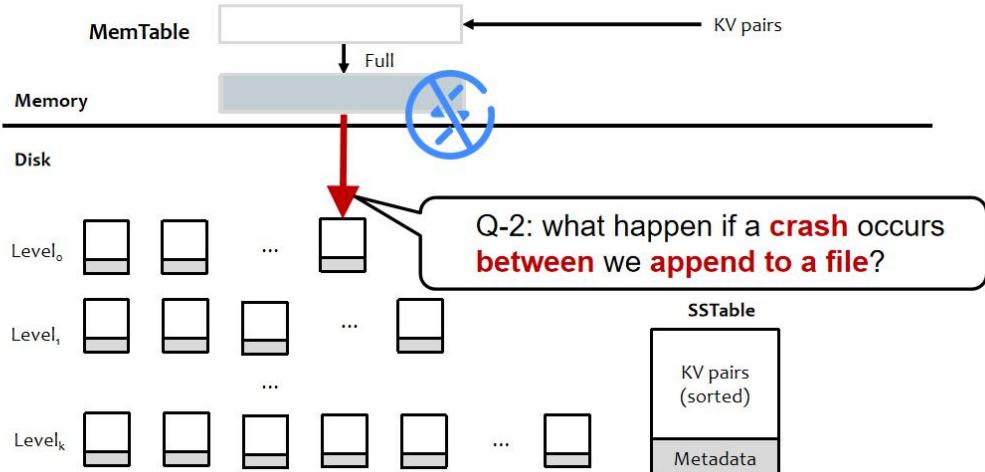
Example: LSM updates



12

问题 1: memtable 中的数据是易丢失的, flush 到一半断电了怎么办?

Example: LSM updates



问题 2: 数据在写到磁盘上的过程中丢了怎么办?

回顾一下: **append** 到磁盘上一个文件的时候, 我们需要更新 **inode** 区域、在 **data** 的 **bitmap** 中更新新使用的 **block**、数据 **block**, 也就是三次磁盘写操作。如果这三个写操作出错的情况, 就是 8 种可能。6 种是会有问题的。**Data block** 应该最先写, **data bit map**, 最后写 **inode** 中的 **block array**。如果 **inode** 先写, 数据对外就可见了。

我们期望是写的时候发生了 **crash**, 得到的系统是一直的, 也就是文件系统的不变量是保留的。

Our expectation: consistency

After rebooting & running **recovery** code

1. FS internal **invariants maintained**

- E.g., no block is both in free list and in a file

2. All but last few operations **preserved** on disk

- E.g., data I wrote yesterday are preserved
- User might have to check last few operations

3. No **order anomalies**

- \$ echo 99 > result ; echo done > status

Approaches: Recovering FS Metadata from Crash

1. Synchronous meta-data update + fsck

- During check, synchronize metadata, such as file size

2. Soft update (FreeBSD fs modified on FFS)

- not covered in this course

3. Logging (ext 3/4), and many others

- Before doing actual meta-data update, log the event
- After crash, recover from log

总体上保证文件系统 crash 后正确就是三种方法：

1. 同步元数据更新+fscheck。当我们做元数据操作的时候，就直接写穿同步到磁盘（保证了顺序）
2. Soft update，这需要依赖的文件系统的语义太多了，讲起来非常复杂。FreeBSD 在相当长的时间中被认为更先进。
3. Logging，这是今天最常用的方法。

同步元数据更新+fscheck

我们先来看第一种方法，很简单就是每当我们做元数据相关操作的时候就写穿。我们假设已经这么做了，重启之后做一次 fs check。

What does fsck do?

1. Check superblock

- E.g., making sure the file system size is greater than the number of blocks allocated
- If error, use an alternate copy of the superblock

2. Check free blocks

- Scans the inodes, indirect blocks, double indirect blocks, etc.
- Uses this knowledge to produce a correct version of the allocation bitmaps
- Same for the inode bitmap

Fscheck 保证 number of blocks 比 allocated block 大。磁盘中的 number of block 可能会不断减少（磁盘坏块），superblock 在磁盘中会有多个备份，如果不满足这个条件，我们就使用备份。

第二件事情就是 check free blocks，scan 的时候就可以知道被正在使用的 inode 指过去的 block。如果不一样，我们认为 inode 是正确的，free block bitmap 还是很容易出错的。换句话说，我们不 care 它的对错，我们只关注磁盘内部的一致性。（all or nothing）如果根目录遍历一遍所有 inode，如果有些 inode 对应的 bitmap 是 free 的，也需要更新 inode 的 bitmap。

3. Check inode states

- Check type: regular file, dir, symbolic link, etc.
- Clear suspect inodes and clear the inode bitmap

4. Check inode links

- Check link count by scanning the entire fs tree
- If count mismatches, fix the inode
- If inode is allocated but no dir contains it, lost+found

5. Check duplicates

- Two inodes refer to the same block
- If one inode is obviously bad, clear it; otherwise, copy the block and give each a copy

25

两个文件指向了同一个 inode，inode 的 counter 是 1 的情况，我们就把 counter 修改为 2；如果从根目录遍历完找不到 inode，但是 inode 的 counter 是 1，就放到 lost&found（失物招领目录下）

6. Check bad blocks

- E.g., point to some out-of-range address
- What should fsck do? Just remove the pointer

7. Check directories

- The only file that fsck know more semantic
- Making sure that ".." and "." are the first entries
- Ensure no dir is linked more than once
- No same filename in one dir

因为 fscheck 只能说出什么是错的，没有办法告诉你什么是对的，所以我们只能移除越界的指针，如果移除了 array 中间的一块，如果把后面往前挪，可能整个文件就废掉了。但是文件系统不 care 这件事情。

Fscheck 不一定能 100% 恢复上层的问题，而且整个过程非常非常慢，比如查目录这一步，我们需要把 inode 中的 counter 对应的 link 全部找到，它分布在磁盘中的任意地方，这就等于要把文件全部扫描一遍。

Fscheck 的条件，不是通过 reboot 重启的、每重启 20 次，所以 fscheck 性能很差，第二就是如果 order 是错的情况，fscheck 也不能做很好的判断。

当我们去 mark 一个 inode 已经被用了然后创建一个文件夹条目的时候，如果顺序反过来的时候，fscheck 的时候都会产生一些问题。

Ordinary perf. of sync meta-data update?

Creating a file and writing a few bytes

- Takes **8 disk writes**, probably **80 ms**
- (alloc, init inode, write dirent, alloc data block, add to inode, write data, set length in inode, ...)

Can create only about a dozen small files per second!

- Think about `un-tar` or `rm *`

一秒钟大概可以创建十个文件，而我们压缩包中有十万个文件，解压一下需要 3 个小时。
解决这件事情毫无疑问我们就想到了内存，利用内存的随机写特点和磁盘的顺序写特点。

How to get better performance?

Reality:

- **RAM** is cheap
- Disk **sequential throughput** is high, e.g., 200 MB/sec or more

Use a big write-back cache?

- No sync meta-data update operations
- Only modify in-memory disk cache (no disk write)
- `creat()`, `unlink()`, `write()`, etc., return almost immediately
 - If cache is full, write LRU dirty block
 - Write all dirty blocks every 30 seconds, to limit loss if crash

This is how Linux EXT2 file system worked



我们把大量的数据放到内存 cache 里，每隔 30 秒/内存每满一次，flush 到磁盘。

What can go wrong with write-back cache?

`unlink()` followed by `create()` an existing file `x` with some content, all safely on disk

One user runs `unlink(x)`

- 1. delete x's directory entry **
- 2. put blocks in free bitmap
- 3. mark x's inode free; another user then runs `create(y)`
- 4. allocate a free inode
- 5. initialize the inode to be in-use and zero-length
- 6. create y's directory entry **
- again, all writes initially just to disk buffer cache
- suppose only ** writes forced to disk, then crash

What is the **problem**? Can `fsck` detect and fix this?

我们先删除再建立一个文件的情况下，现在假设断电了，6 个磁盘操作都要写到磁盘，发生断电的时候，我们并不知道谁写进去了。假设 1 和 6 写到了磁盘而其他操作没有写到磁盘里，最后的效果就是 `x` 的名字改成了 `y`，`fscheck` 是检查不出来的，但是毫无疑问这个错误还是比较严重的。

Journaling/Logging

Journaling overview

Write-ahead Logs

- A concept from database

Record before update

1. Record changes in journal
2. Commit journal
3. Update

Crash before commit:

- No data is changed
- Discard journal



Crash after commit:

- Journal is complete
- Redo changes in journal

那就需要我们的 **journaling/logging**，日志文件系统，它最先是在 ext3 中加入，原先是在数据库中的发展而来的。我们要把所有操作写在 log 里，commit 才去更新，如果在红线前发生了断电，那么因为还没有 commit，会全部 discard 掉。如果 commit 了断电了，可以通过 log 中的操作去做 redo。

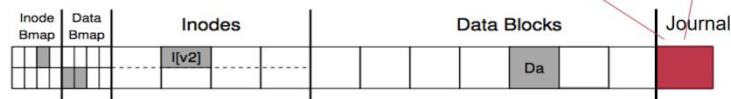
Append a File via Journaling

Inside of I:

- owner : yubin
- permissions : read-only
- size : 1 => **2**
- pointer : 4 => **5**
- pointer : null
- pointer : null

I:

size: 1=>2
2nd pointer: null => 5
Db:
Old-data... => Appended-data...
COMMITTED



我们就可以在整个磁盘的最后加一块 journal，把我们要做的事情全部记在里面，一旦写完 COMMITTED 整个日志有效，如果没写那就整个日志无效。（all or nothing）断电以后有没有可能已经做过了，该做的都做了应该把日志删掉。再做一遍会有问题吗？因为日志中记录的操作是幂等的，所以再做一遍也无所谓。

所以流程就是先写日志，再 commit，再做对应操作。

第一个问题：所有的东西都要写两次，第一次写 journal、第二次写到对应的位置。我们怎么样才能取得一个比较好的设计，即保证 crash consistency，又不写两遍。我们可以只去写 metadata。231G 的 metadata 可能只有几 M。也就是对 metadata 写两边，而大的数据只需要写一遍。所以我们能不能只对 metadata 做 journal 保护。

EXT4

ext4 有 3 种模式：

1. Write-back mode: 只写 metadata 到 journal 中，数据还是直接写到 disk 上。
2. Ordered 模式: order 和元数据中有顺序，先写数据再写元数据。先写数据，crash

之后是无所谓的。

3. full journaling, 也叫 datemode。所有数据写两遍，都进 journal。

1. journal

data=journal 模式提供了完全的数据块和元数据快的日志，所有的数据都会被先写入到日志里，然后再写入磁盘（掉电非易失存储介质）上。在文件系统崩溃的时候，日志就可以进行重放，把数据和元数据带回到一个一致性的状态，journal 模式性能是三种模式中最低的，因为所有的数据都需要日志来记录。这个模式不支持 allocation 以及 O_DIRECT。

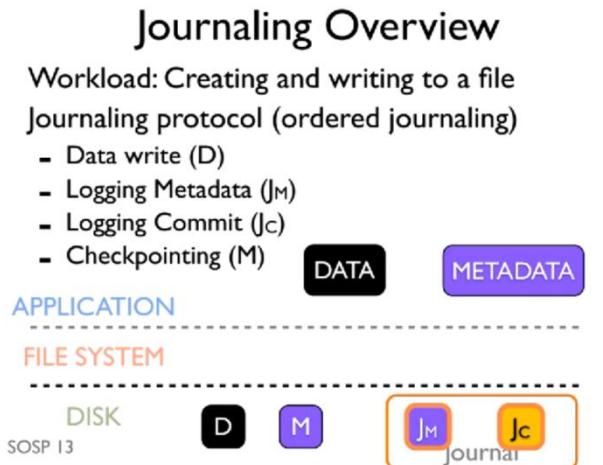
2. ordered (*)

在 data=ordered 模式下，ext4 文件系统只提供元数据的日志，但它逻辑上将与数据更改相关的元数据信息与数据块分组到一个称为事务的单元中。当需要把元数据写入到磁盘上的时候，与元数据关联的数据块会首先写入。也就是数据先落盘，再做元数据的日志。一般情况下，这种模式的性能会略逊色于 writeback 但是比 journal 模式要快的多。

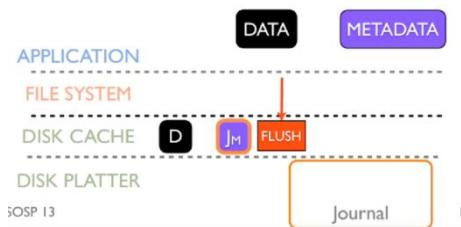
3. writeback

在 data=writeback 模式下，当元数据提交到日志后，data 可以直接被提交到磁盘。即会做元数据日志，数据不做日志，并且不保证数据比元数据先落盘。writeback 是 ext4 提供的性能最好的模式。

我们期望的是第二种 Ordered 模式，也是 ext4 默认的方式。现在我们有一个 DATA 和 METADATA，磁盘中有一块 journal 的区域。我们先把数据写到 file system，然后 meta data 写到磁盘上的 journal (Jm)，然后我们写一个 Jc (journal commit)。写完 commit 之后，我们再把 metadata 写到 disk 对应的地方。

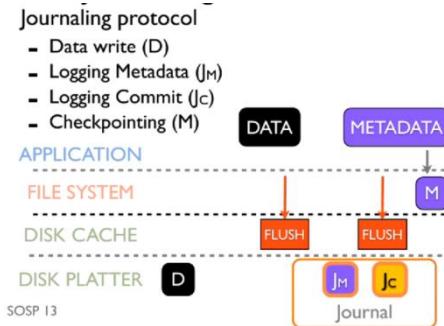


我们要保证 order 这件事情，怎么保证？注意磁盘中是有 cache 的，磁盘中有一个 buffer，我们写了 a 和 b 不一定是按照顺序写到磁盘上的，可能先写 b 再写 a。我们需要 a->flush->b。我们再回到前面这个例子中，我们要加几个 flush 操作？



我们先把数据和 Journal metadata 写到 disk cache 上，并且 flush 到磁盘上，这两个的顺序是无所谓的，因为我们的 journal 还没有 commit。后面是写 Journal commit 和真正要落到磁盘上的 metadata，这个要先写 Jc 到 data cache 中然后 flush，因为如果这两个也无所谓顺

序的话，可能导致磁盘中的 metadata 已经到位了，而 journal 中还没有 commit。这样崩溃以后 recover 的时候，没有 commit 本应该是 nothing 的情况，但是变成了 all。OS 误以为没有发生任何问题不需要恢复。



所以第二个 flush 是不能删掉的。

我们有没有把第一个 flush 优化掉？第一个 flush 是为了让 commit 在 data 和 JM 的后面，OS 有什么办法让这三个数据一起落到磁盘上的时候，重启恢复以后知道这个 journal 是 commit 的或者写了一半没有 commit 的。我们要把 data 和 JM 统一算一个哈希放到 Jc 里面，这样我们在 recover 的时候，比较一下 Jc 里的哈希以及读到的 D 和 JM 的哈希，如果一样那么就是 all，如果不匹配那么就应该是 nothing。

2021/10/21

AUTOMIC 是一个很重要的属性，因为人潜意识会认为代码是瞬间完成的，这一点本身从两方面来看

1. 从系统的角度来看，把一个操作变成原子的（`a++`），这是系统对上的抽象。在系统里，是要从内存读到寄存器里，寄存器`++`，然后写回内存里。

常见的 automic 是和 crash 相关，运行时刻发生了 crash，导致中间结果暴露给了上层。

有三种方法：

1. Syn meta-date update + fsch
2. Soft update
3. Logging

所以第一种方法慢，我们需要记住每秒钟最多创建 12 个文件，怎么解决这个问题呢？我们必须在磁盘和磁盘的上一层加一个 cache。但是一旦加了缓存之后，缓存内部的顺序没有办法保证，有可能出现磁盘依旧是 consistent，但和我们上层的操作是不符合的，因为在文件系统这个级别，fscheck 只能保证元数据是一致的，但不能保证其对错。如果上层操作使得元数据一致但是不正确，那么 fscheck 是无能为力的。

对于 log，有一种方法是 full journaling，也可以只保护 metadata，把 metadata 写到 journal 里去，顺序是先写 data 和 journal of metadata，flush 一下，然后再写 commit，这样我们就可以保证 data 和 journal of metadata 已经就位了，前提是 data 最好不要覆盖原有的数据，如果覆盖了，那么哪怕没有 commit，也回不去原来的状态。Append 一个文件的时候，data 是不会被覆盖掉的，memory map（mmap）是最容易覆盖的情况，也就是把文件映射到内存中，对内存的所有操作都会同步到文件中。

那么我们上节课末尾讲了一个小优化，为了去掉第一个 flush，也就是把 data 和 journal of metadata 算一个 checksum，写到 journal of commit，这样我们就可以判断这三者是否一直，

如果一致了，那么说明 commit 是 ok 的。

All-or-nothing Atomicity

接下来，我们就要来讲通过前面的这些方法，crash consistency 只对文件系统才有作用。我们把这个问题泛化一些，是否能够提供一种 all-or-nothing 的 atomicity。什么操作会变成多个原子操作呢？OS 里有一种 rename，实际上就是通过 unlink 和 link 实现的。Eg. mv /a /b，如果/b 存在，要先删掉/b，对/b 先做 unlink，然后再对/a 做 link。

这个操作可能被 crash、interrupt、并发打断。

Interrupt 是经常被提到的点，因为它可能任何时候发生，并且导致控制流的改变，它正好可能插入到 rename 的中间，导致看到 rename 的中间状态。

另外一种改变控制流的方法，就是 signal handler。应用程序是不知道 signal handler 的存在的，应用程序做到一半的时候被触发了 signal handler，也可能看到中间状态。

另外一种是并发导致的，中断可以认为是自己和自己并发。

Example: Renaming - 1

Normal 1 UNLINK(to_name) 2 LINK(from_name, to_name) 3 UNLINK(from_name)	Under failure 1 UNLINK(to_name) 2 LINK(from_name, to_name) 3 UNLINK(from_name)
---	--

Text edit usually save editing file in a **temp** file
– Edit in .a.txt.swp, then rename .a.txt.swp to a.txt

What if the computer **fails between 1 & 2**?
– The file **to_name** will be lost, which will surprise the user
– Need **atomic** action

Example: Renaming - 2

Normal 1 LINK(from_name, to_name) 2 UNLINK(from_name)	Under failure 1 LINK(from_name, to_name) 2 UNLINK(from_name)
--	---

Weaker specification without atomic actions
– 1. Changes the inode number in for **to_name** to the inode number of **from_name**
– 2. Removes the directory entry for **from_name**

If fails between 1 & 2
– What will happen after failure? The file **from_name** still exists!

在任何一个状态发生 failure 都会有问题。

EG:IBM System 370

TRANSMIT 指令，第一个参数指向一个 string，第二个参数指向一个 table，第三个参数指向一个 counter。把这个输入以后，会把第一个 string 中的时候对应 table 做替换，counter 为 0 的时候，任务就完成了。

情况 1：如果在其中 crash 了，string 的前一半翻译完成了、后一半翻译没完成。如果重新运行一遍、前面已经翻译的可能会被翻译成别的情况。（crash 问题）

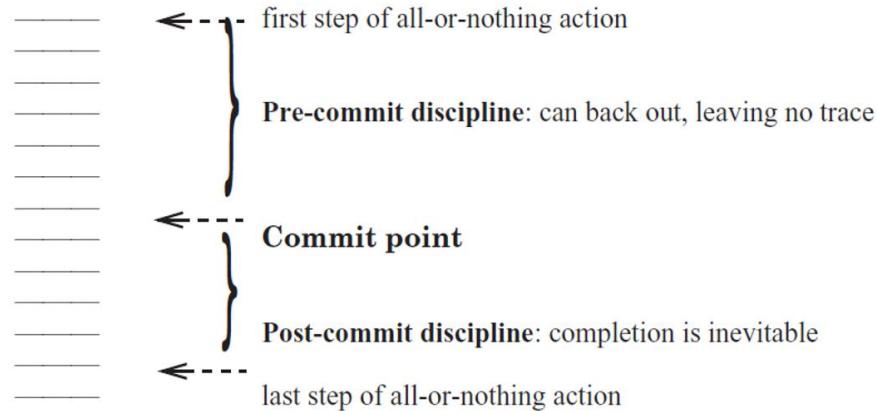
情况 2：如果 string 包含了两个 page，第二个 page 被 swap 出去了，读到第二个 page 的时候就会发生 pagefault。（exception flow control 的问题）

方法 1：指令运行的时候，把中断都关掉；对于 pagefault 的情况，它先空跑一遍，如果都在内存里就继续做事情；否则就处理 page fault，然后再跑。但是在空跑之后也有可能被 swap 走。

核心问题就是这个操作太复杂了，硬件不能做太复杂的事情。最后就变成硬件的操作越来越简单，后面就有了 risc 指令集。

在这个过程中，我们说 journaling 不能完全解决这个问题。Journaling 只能保证 consistency，而不能保证文件系统的语义。如果我们把 rename 整个过程记录到 journal 中，那么是可以解决这个问题的。

Commit Point



在 commit point 之前发生了 crash，就是 nothing；在之后发生了 crash，就是可以恢复。这整个操作序列就是 transaction。

Case study: bank transfer

1. Journaling only ensures the **consistency**, but not atomicity
 - Our previous example of `rename`
2. Even all file system operations (e.g., `rename`) are atomic is still insufficient
 - User can group different file operations in an arbitrary way
 - Write multiple records with file system API

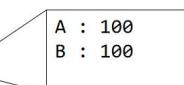


Case study: bank transfer

```
transfer(bank, a, b, amt):
    bank[a] = bank[a] - amt
    bank[b] = bank[b] + amt
```

Invariant:
 $\text{bank}(a) + \text{bank}(b) = 200$

- Suppose the bank is stored in a single KV
 - a and b are stored in a **single file**
 - And their offsets are known



我们希望保证 a 和 b 的总和没有变，只是完成了转账。我们假定 A 和 B 的余额以 kv store 的形式存在。

```
transfer(bank, a, b, amt):
    bank_a, bank_b = read_accounts(bankfile) ← sum=200
    write(bankfile, a, bank_a - amt) ← sum=150
    write(bankfile, b, bank_b + amt) ← Crash!
```

What is the sum after recovery?

发生 crash 的时候，我们发现中间 50 元就被丢掉了。核心在于好几次 write 操作需要变

成一次 write。一个很直观的方案就是我们加一个 shadow copy。

```
Implementation
```

```
transfer(bank, a, b, amt):
    bank_a, bank_b = read_accounts(bankfile)
    write(#bankfile, a, bank_a - amt)
    write(#bankfile, b, bank_b + amt)
    rename(#bankfile, bankfile)
```

这里有一个假设，我们把 transfer 的操作聚焦到了 rename 的原子性上的。Rename 分成如下操作：

我们假设 bank 的 inode 为 12, #bank 的 inode 为 13. 我们把 bank 的 inode 指向 13，再增加 inode13 的 ref_counter，再减少 inode12 的 ref counter，再把目录中的#bank 的 entry 删掉，然后再修改 inode13 的 ref_count=1.

如果第一步 crash, bank 和#bank 两个人指着 13 但是 ref_counter = 1, 而 12 的 ref_counter = 1 但是没有人指着。文件系统只能修改 ref_counter 到一致的值，但是没办法删除原先的 bank 文件。

有一个方法就是把所有的操作放到 journal 里面，我们的 journal 可以保证这一点。

我们接下来来想，我们在做 crash 的时候，我们在写文件（写磁盘）的时候，还是需要一个 all-or-nothing 的操作。写一个磁盘的 sector，我们能不能保证这件事情是原子的？假设磁盘上一个 section 是 4k，对于应用程序来说，如果写 4k 这件事情不是原子的，那么就少了很重要的特性。所以磁盘通常可以保证写 4k 的 sector 的原子的。磁盘会保存足够的能量来写 4k 这个操作，也就是有一个专用的小电容。

如果没有这个小电容该怎么办呢？

data state:	1	2	3	4	5	6	7
sector S1	old	bad	new	new	new	new	new
sector S2	old	old	old	bad	new	new	new
sector S3	old	old	old	old	old	bad	new

比如我们想写一个 4k 数据，我们一次性写 3 个 4k 的 sector，记为 S1,S2,S3。当我们开始写第一个的时候，S1 就变成了 bad，因为还没写完。最终从 old, old, old, 写到了 new, new, new。但是没有电容，断电了就停下来了。Commit point 在哪里呢？是在 4 的前面还是后面呢？如果出现了 new bad old, S1 != S2 && S2 != S3 && S1 != S3, 我们需要恢复成 new new new, 恢复的办法就是把 S1 copy 到 S2 和 S3，这样就变成 new, new, new 了。所以 commit point 是 3 到 4 转变的一瞬间，当我们中间的 4k 被写了若干个 byte 的时候，我们就可以识别出是不一样的。这个方法非常浪费磁盘，因为一次 4k 写操作要写 3 倍的量。

我们今天讲的是当我们的 bank 要完成一个转账操作的时候，a 把钱给 b 这个操作，是放在一个 kv store 中的，它是建立在一个文件系统之上的。会做两次 write 操作。两次 write 操作中间发生了断电就有可能不一致，我们使用 shadow copy 的方法把问题划归为了一个操作叫做 rename。但是 rename 又有若干次写操作（对于文件系统来说，至少有 4 次）。我们怎么把这 4 次写操作变成原子的呢？

我们可以通过 log 的方式，这个方法的 commit point 在 COMMIT 这个操作。COMMIT 是一次磁盘写，但是 commit 本身不是一个 bit 那么简单，它记录的是哪一个操作 commit 了。硬件提供了写 4K 是原子的，我们保证了”COMMIT:3”写入是原子的。

我们怎么保证硬件提供的 4k write 是原子的呢？

1. 硬件提供电容。
2. 磁盘上写 3 次。

我们回到 shadow copy，这就是我们前面说的最终落到 sector write 是原子的。核心的规则是，永远不要覆盖一个没有 shadow copy 的文件（孤本）。Shadow copy 方法支撑了 bank transfer，但是它的支撑点有很大的局限性，比如我们有多个文件怎么办？如果有多个 shadow copy，又没有 commit point 了。

方法 1：把多个 shadow copy 放到一个 shadow dir 下，其实可以。但是不能处理两个机器、两个文件系统的情况。所以 shadow copy 太依赖文件系统自身的局限性了。

如果我们要改很小的大小，我们需要把整个文件 copy 一份。所以它只能用来做单机、单磁盘、单文件系统的 all-or-nothing。

我们可以使用 log。我们能不能提出一种通用的 log 方式，使其对于文件系统或者其他的操作，都是可以实现的。

讲 log 的时候，经常会有 redo/ undo 组合。我们以 sysR 作为例子。

Introducing Transaction Terminology

<pre>transfer(bank, a, b, amt): begin bank[a] = bank[a] - amt bank[b] = bank[b] + amt if bank[a] < 0: abort //Not enough funds else: commit</pre>		<pre>transfer(bank, a, b, amt): begin write (a, read(a) - amt) write (b, read(b) - amt) if read (a) < 0: abort //Not enough funds else: commit</pre>
--	--	---

我们回到 bank transfer 这个例子。A- amount; b+ amount。Begin 标志着原子操作的开始，COMMIT 标志着原子操作的 commit point，abort 发生了就是 nothing。

Consider the Bank Account Example

Two accounts: A and B

- Accounts start out empty
- Run these all-or-nothing actions

Problem

- After crash, **A=3000**, but T3 never committed

Solution

- Undo to A's previous committed value
- T1 and T2: "All"
- T3: "Nothing"

```
begin // T1
A = 3000
B = 2000
"commit // A=3000; B=2000

begin // T2
A = A - 1000
B = B + 1000
"commit // A=2000; B=3000

begin // T3
A = A + 1000 // A=3000
--CRASH--
```

在 transaction 中间如果发生了 crash，就要 undo 掉这个操作。如果在 T3 运行的时候发生了 crash，那么 T3 就是 nothing，而 T1 和 T2 COMMIT 了，就是 all。

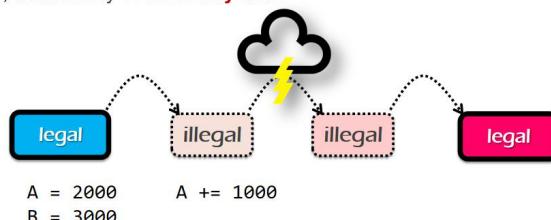
我们更加 general 一点，有：

More general

Files states are transferred based on application, **internal states may be illegal**

- E.g., bank transfer
- Goal: transfer between legal state to legal states

However, **Crash** may occur at **any time**



接下来就是解决方法，我们维护一个 log 文件，每个日志文件有若干个 action，每个 action 包含了旧状态、要做什么和新状态，我们发生 crash 的时候，比如新状态已经做了但是 commit，那么我们就 undo；反之，如果我们已经 commit 了，但是旧状态还没有改成新状态，那么我们就把旧状态改为新状态。

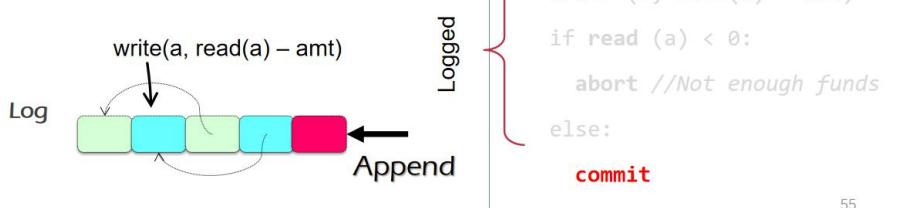
DO-UNDO-REDO Protocol

Each operations between begin ... commit is recorded in a log

- Append-only reduce random access

The begin ... commit is called

- **Transaction**

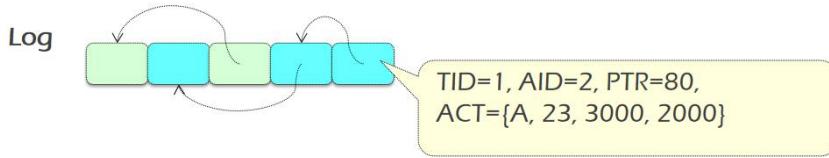


不同的 transaction 的 log 可能交织在一起，因为 transaction 是没有顺序要求的。所以 transaction 的 log 块会通过指针连在一起。

DO-UNDO-REDO Protocol: log content

Each log record consists of

1. Transaction ID
2. Action ID
3. Pointer to previous record in this transaction
4. Action (file name, offset, old & new value)
5. ...



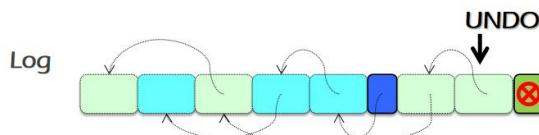
写 log 是在真正要改磁盘数据前做的事情，要保证这个顺序就可以用 flush。我们在遍历 log 的时候，是从后往前遍历的，把所有没有 commit 的，标记为 abort。对于标记为 abort 的日志，我们就找到它的 action undo 掉；如果 commit 了，那么我们就去 redo 掉。

Recovery rules How to recovery from crash?

Read the log and recover states according to its content

Rules:

1. Travel from end to start
2. Mark all transaction's log record w/o CMT log and append ABORT log
3. UNDO ABORT logs from end to start



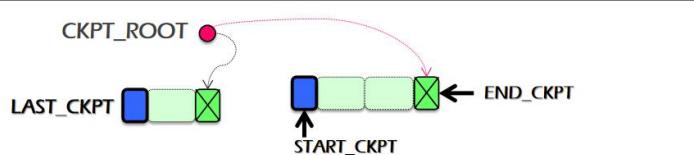
62

这是文件系统 log 的泛化。讲完之后就有一个问题，我们什么时候把日志删掉呢？防止日志大小不断增长？日志什么时候能够删掉这件事，没那么简单。我们把日志里的数据写回到文件中该写的地方，这就是 checkpoint。

How to checkpoint?

How to checkpoint? actions

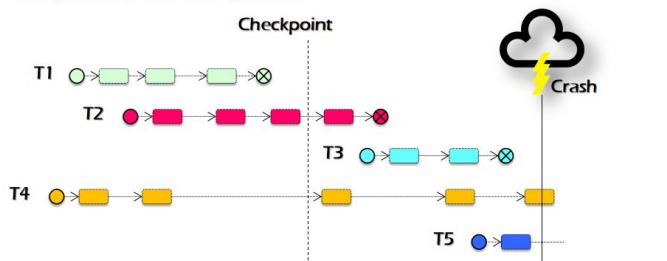
1. Wait till no transactions are in progress
2. Write a CKPT record to log
 - Contains a list of all transaction in process and pointers to their most recent log records
3. Save all files
4. Atomically save checkpoint by updating checkpoint root to new CKPT record



67

等到没有 action 在做的时候，我们写一个 CKPT record 到 log，告诉整个系统要开始做 checkpoint 了，在这个过程中把前面所有的操作保存到文件里去。Checkpoint 本身包含了前面所有日志中的 transaction，我们把这些 transaction 全部写到文件里去。然后我们写一个 END_CKPT。假设现在有这么多个 transaction:

Recovery with checkpoint

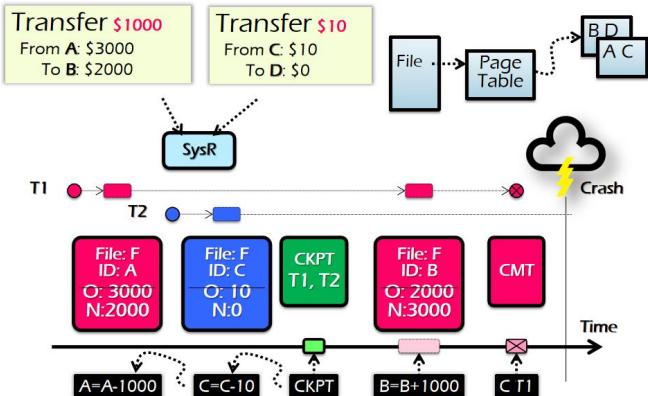


1. Read latest checkpoint
→ T2, T4 are ongoing transactions
2. Read log → T2, T3 are winners and T4 is a loser
3. Read log to UNDO loser (T4)
4. Read log to REDO winners (T2 and T3)

假如如上图所示，有一个 checkpoint，我们在 recover 的时候，我们知道做 checkpoint 就代表前面这些都已经写到磁盘上了。我们发现 T2 和 T4 在做 checkpoint 的时候还是 ongoing 的，这时候我们应该读 T2 和 T3，在 crash 前就 commit 了，这两个标记为 winner，而 T4 没有 commit，它就是 loser。所以 T2 和 T3 都是要 redo 的，而 T4 作为 loser 就是要 undo 的。T1 所有影响的改变，都已经在 checkpoint 前做完了，所以我们不用 care。

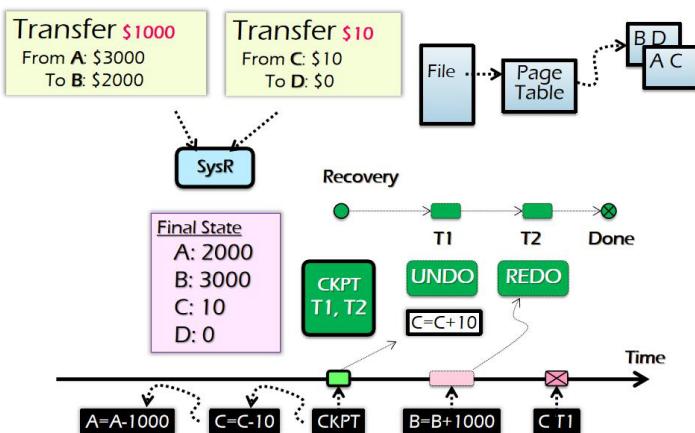
而对 T5，我们恢复到 checkpoint 的时候，已经 nothing 了，都不算 loser 集合里。所以 checkpoint 的核心就是当我们做完 checkpoint 的时候，其之前 commit 的 transaction 的 log 都可以删除。

Example: logging



做 checkpoint 的时候，代表前面的真的写到磁盘上去了。

Example: recovery



在 recover 的时候，我们知道在 checkpoint 的时候 T1 和 T2 还在做，T1 已经在 crash 前 commit 了，所以我们需要从 checkpoint redo T1，而 T2 是没有 commit 的 loser，所以需要 UNDO T2。

为什么我们需要 redo 和 undo，因为我们有 checkpoint 来删除已经完成的 log。但是我们 checkpoint 又不能把系统暂停。所以我们必须让 transaction 和 checkpoint 同时做，这就带来了 redo 和 undo 的问题。我们怎么实现只有 redo 的 log？

REDO-only logging

Logging rules

- Append **REDO** log record **before flushing state modification**
- Uncommitted transactions can not **flush state (w/o UNDO)**
 - E.g., achieved via **caching** the state modification to the memory

Recovery rules

- **ABORT** all log records w/o **CMT** log
- **REDO** all committed transactions

Can we have **UNDO-only** logging?

Checkpoint 的时候，日志一定要旧，这样我们才能做 redo。我们就要保证如果没有 commit，我们就不能把状态写到磁盘中。也就是没有 commit 的时候，checkpoint 就不做操作。我们

就可以实现一个非常简单的 recover 机制：只有 redo。

我们能实现一个只有 undo 的 logging 吗？说明磁盘上哪怕没有 commit 都是最新的，每次把所有状态都 flush，甚至连 checkpoint 都不需要等，在这种情况下一旦发生 crash 操作的时候，它就要去做 undo 操作。

UNDO-only Logging

Logging rules

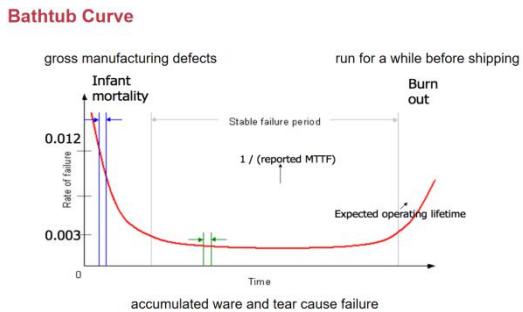
- Append UNDO log record **before** flushing state modification
- State modification must be flushed before transaction committed
 - w/o REDO

但是意义不是很大。

要实现原子性，我们可以使用 shadow copy 方法，还可以使用 undo-redo logging。大家要记住的一点事情，容错和 all-or-nothing、原子性有很紧密的关系。

我们以前提过 MTTF (mean time to failure)，MTTR (mean time to repair)，MTBF (mean time between failure)

$$MTBF = MTTF + MTTR$$



One way to measure "MTBF"

- E.g. 3.5-inch disk's MTTF is 700,000 hours (80 years!)
- Guess: ran 7,000 disks for 1,000 hours and 10 failed
- If the failure process were **memoryless**, then OK
- "expected operational lifetime" is only 5 years

出厂设置运行的时候没挂，后面挂的概率就很低，时间长了以后挂的概率会上升。这段平稳时期，就是他这样测出来的平均无故障时间。

当我们用到平均无故障时间的时候，通常是测的最低的故障率的期间。当我们用一个 RAID 的时候，它相当于用多块磁盘组成阵列，这样任何一块磁盘坏掉，可以用其他几块和校验值恢复出那块磁盘。在 70 万小时之内，有一块坏掉，我们替换掉以后又可以以 70 万小时的最佳状态换掉。我们只要考虑在替换的一个小时之内，另一块故障的概率是多少，这样不断地换，我们同时能每个磁盘保证在故障率最低的期间。但是这样校验的磁盘会写的很多，我们就可以把校验写平均一下。RAID5 是最常见的容错的技术。

2021/10/26

上节课，我们讲了 All-or-nothing Atomicity，也就是当一个系统在做一个很大的操作的时候，我们认为它是一步完成的，但是实际上它在计算机系统里面分成了若干步。那么当发生一些 crash 的时候，导致系统在执行到中间的时候崩溃了，或者发生了一个 interrupt 导致中间信息被暴露出来了。对于前者，我们要保证的是 all-or-nothing，而对于后者，我们要保证的是 before-or-after，所以我们要用 transaction 本质其以原子的方式去运行。

提到 all-or-nothing，我们就不得不提到 commit point，既然破坏的原因是因为完成一系

列操作分成若干步，那么势必就会有一个 **commit point**，在其之前 **crash**，回滚到 **nothing**，在其之后 **crash**，状态就会 **redo** 到 **all**。为了实现 **all-or-nothing**，我们提出了两个方法，第一个是 **shadow copy**，也就是我们把所有操作写到一个文件，通过 **rename** 这个文件来作为 **commit point**，也就是在 **rename** 之前所有人看到的是旧数据而 **rename** 之后所有人看到的是新数据。我们只需要把 **rename** 这个操作本身变成 **all-or-nothing**，从而用它去支持更普遍的操作的原子性。但是 **shadow copy** 有一系列的缺点，只能够对一个文件使用、不能跨文件系统使用，适用性比较低。所以我们提出新的 **logging**，我们把所有的操作以 **record** 的方式不断地 **append**，它既支持了 **all-or-nothing atomicity**，并且因为都是 **append** 操作，它变成的是顺序的写，对于磁盘这种介质来说非常友好。

有了 **logging** 之后，我们就有了 **redo-logging** 和 **undo-logging**。需要做 **redo** 和 **undo** 一定是因为数据不仅仅存在于 **log** 中，还存在 **Home** 上。因为如果数据只在 **log** 中，虽然我们查找某个特定数据只需要从后往前扫描就行，但是毫无疑问这样是比较慢的，所以我们需要把当前时刻所有数据的情况存在一个叫做 **Home** 的地方。对比文件系统，如果 **log** 是后面的 **block**，那么 **Home** 对应的就是 **inode-table**。

那么数据出现了两份就会导致 **log** 和 **Home** 的一致性问题。如果 **log** 上的数据更加新，**commit** 了以后，我们对 **Home** 就是 **redo** 去更新数据；如果 **log** 上的数据已经写到了 **Home** 中但是没有 **commit**，那么在 **crash recover** 的时候，这个 **transaction** 是一个 **loser**，我们需要在 **Home** 上 **undo** 这个 **transaction**（在 **log** 的这个 **transaction** 的 **record** 中，我们会同时维护新的值和旧的值，**undo** 的时候其实就是把旧的值写到 **Home** 中）。

All-or-nothing atomicity，这个 **logging** 技术可以用在 **cache**、**database**、**file system**、**message queue** 等，这种方式非常通用。

今天我们来看可见性和并发性的问题，如果我们不仅仅是在做一个转账的操作，我们同时还在做一个审计（**audit**）操作，也就是把所有钱加在一块。在没有人往银行中存钱的情况下，总金额应该是不变的。但是如果我们将转账和审计在两个 **CPU** 上同时运行，我们的审计可能会看到转账过程中一个加了钱，一个还没扣钱的情况下，总金额变多的情况。原因就是转账在现金情况下，要么在 **A** 的手里要么在 **B** 的手里，但是在计算机中是有中间状态的。

正因为这两个 **transaction** 被拆成了若干个子操作，同时运行的时候会交织在一起，以某种方式交织的时候就会出现这个问题。我们为什么要允许交织呢？因为我们要提高并发性，让 **transaction** 能够同时去做。但是这就会引发 **race condition**。

我们先来看这个经典例子：两个同时运行 **i++**，最终可能得到 **i=1** 的情况，究其根本是因为 **c** 语言的 **i++** 对应到汇编并不是一句指令。

Race 的定义：两个或两个以上的主体访问一个共享变量、其中至少有一个写操作。

所以管理数据是非常难的问题，**crash** 可能导致不一致和原子性被破坏、多台机器同时运行会导致 **race condition**，我们今天就来讨论这个问题。

我们是不是可以提出一种 **abstraction** 来简化数据的管理，**transaction** 就是数据管理的一种抽象。**Transaction** 提出了 **begin**、**commit**（结束）、**abort**（放弃）等几种源语。**Transaction** 不仅仅可以解决 **all-or-nothing atomicity**，也可以解决 **before-or-after atomicity**，所以 **transaction** 这个抽象是非常强大的。

Transaction 提供的四个属性是 **ACID**（**atomicity**, **consistency**, **isolation**, **durability**），我们把数据库中的 **ACID** 泛化到分布式系统。

第一个 **A**，提供 **log** 这个机制，就可以 **all-or-nothing atomicity**。

C 这一点，这里的 **C** 其实是用户态对于正确的定义，比如转账这个操作不会让求和操作不会有任何改变。这个定义是和业务相关的，并不是从系统角度来定义的。这就是说，如果

我们代码写错了，光靠 transaction 的底层机制是不能保证用户态的 consistency 的。所以这里的 consistency 和分布式系统中的一致性是不太一样的。

Isolation，它其实就是要满足两个同时进行的 transaction 执行起来要隔离的。对于并发的 transaction，执行起来的效果就是要么 A 在 B 前面，要么 B 在 A 前面。为了实现 isolation，有不同的级别，比如我们之前说的 before-or-after atomicity 是最容易理解的级别。

Durability，就是一个 transaction 一旦 commit 了，它的所有写操作都必须放在持久化存储中，不能因为各种各样的原因而丢失。

怎么保证 ACID

我们学过的东西里能保证哪些？

Atomicity：使用 shadow copy 或者 logging。

Consistency：程序写代码正确，是非系统方面的。

Isolation：这节课要讲的 before-or-after

Durability：也是 log 来保证，是 write-ahead-log，所有操作先写到日志上，commit 后再写到 Home 的位置。

Serialization

我们先来讲最简单的 isolation 要保证的特质，也就是 serializability。我们的目标是，我们会允许多个 transaction：T1、...、Tn。我们希望的效果是：看起来这些 transaction 的效果是和一个个串联起来跑的效果一样的。

T1	T2
begin	begin
read(x)	write(x, 20)
tmp = read(y)	write(y, 30)
write(y, tmp+10)	commit
commit	

假如我们现在有两个 transaction T1 和 T2，分别有一个 begin 和 commit。T1 是读 x 和 y 写 y，而 T2 是写 x 写 y。这两个 transaction 有很多种运行的方式。每一种运行的方式叫做一个 schedule，如果是两个核怎么排序呢？有没有可能真的就是某一步同时运行呢？如果我们测的精度足够高，那么我们一定可以看到顺序，但是这个顺序对我们来说没有意义，所以我们要从最后效果的方式来定义正确。

我们继续来看上面这个例子：

如果按照顺序 T1、T2 执行，我们可以得到结果为 x=20,y=30

如果按照顺序 T2、T1 执行，我们可以得到结果为 x=20,y=40

以下是两种 sequence：

T2: write(x, 20) T1: read(x) T2: write(y, 30) T1: tmp = read(y) T1: write(y, tmp+10)	T1: read(x) T2: write(x, 20) T1: tmp = read(y) T2: write(y, 30) T1: write(y, tmp+10)
At end: x=20, y=40	At end: x=20, y=10

所以根据线性序列化的定义，左边这个 sequence 是对的，而右边这个 sequence 因为没有一种顺序执行结果的答案与之匹配，就是错的。

为什么 serializability 是比较理想的呢？

它可以简化程序员的思考。

由于 serializability 可以保证结果是按序进行的。

T2: write(x, 20) T1: read(x) T2: write(y, 30) T1: tmp = read(y) T1: write(y, tmp+10)	T1: read(x) x=0 T2: write(x, 20) T2: write(y, 30) T1: tmp = read(y) y=30 T1: write(y, tmp+10)
At end: x=20, y=40	At end: x=20, y=40

In the second schedule, the results are correct. But T1 reads x=0 and y=30; those reads are not possible in a sequential schedule. Is that ok?

但是同样的结果，调度的 schedule 是可能完全不一样的，在这个过程中，我们发现有一个小的 trick，T1 在读 x 的时候读到的是 0，而 T1 在读 y 的时候读到的是 30，虽然这种情况最终答案是对的，但是 T1 能够读到 x=0, y=30 吗？如果 T1 在 T2 前，或者 T2 在 T1 前，那么 T1 一定也看不到 x=0, y=30.

上面这样也是可以允许的吗？我们需要引入 serializability 不同级别。

final-state serializability、**conflict serializability**、**view serializability**。之前我们说的就是 **final-state serializability**，也就是我们只关心最终的数据状态是否等价于某种线性 schedule 的情况。

Conflict serializability: 在 tx 的中间，我们只 care 存在冲突的情况。它只关心 transaction 中间有可能冲突的那些变量，它们之间是排序的。

而 **view serializability**，也就是我们要求事务之间读的时候也需要保证是和线性执行是读到的结果是一致的。

我们重点先关注 **conflict serializability**。

Conflict Serializability

Two operations conflict if:

1. they operate on the same data object, and
2. at least one of them is write, and
3. they belong to different transactions

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**

Conflict serializability 的定义是两个属于不同 transaction 的操作对相同的对象做操作并且至少有一个是写操作。这个定义和 race condition 很像，所以产生 race 的前提就是 conflict。

一个调度方式是 conflict serializable 的当且仅当冲突的 transaction 顺序是和顺序排列的

调度一致的。

我们下面来看一个例子：

T1	T2
begin	begin
read(x)	write(x, 20)
tmp = read(y)	write(y, 30)
write(y, tmp+10)	commit
commit	

依旧是这个例子，这个例子的 conflicts 如下：

T1.1	read(x)	and	T2.1	write(x, 20)	on x
T1.2	tmp = read(y)	and	T2.2	write(y, 30)	on y
T1.3	write(y, tmp+10)	and	T2.2	write(y, 30)	on y

Conflict Graph

如上例所示，conflicts 是成对出现的，所以我们可以画一个图叫做 conflict graph。节点是一个个 transaction，边是有向的。如果事务 T_i 和 T_j 之间存在一个 conflict，并且第一步是发生在 T_i 中，那么我们就把一条边从 T_i 指向 T_j 。

结论：一个 schedule 是 conflict serializable 当且仅当它的 conflict graph 是无环图。

Conflicts

T2.1	write(x, 20)	->	T1.1	read(x)
T2.2	write(y, 30)	->	T1.2	tmp = read(y)
T2.2	write(y, 30)	->	T1.3	write(y, tmp+10)

(T2 -> T1)

Conflicts

T1.1	read(x)	->	T2.1	write(x, 20)
T1.2	tmp = read(y)	->	T2.2	write(y, 30)
T1.3	write(y, tmp+10)	->	T2.2	write(y, 30)

(T1 -> T2)

```

T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)

```

~~T1: read(x)~~

~~T2: write(x, 20)~~

~~T2: write(y, 30)~~

~~T1: tmp = read(y)~~

~~T1: write(y, tmp+10)~~

T2.1 -> T1.1

T2.2 -> T1.2

T2.2 -> T1.3

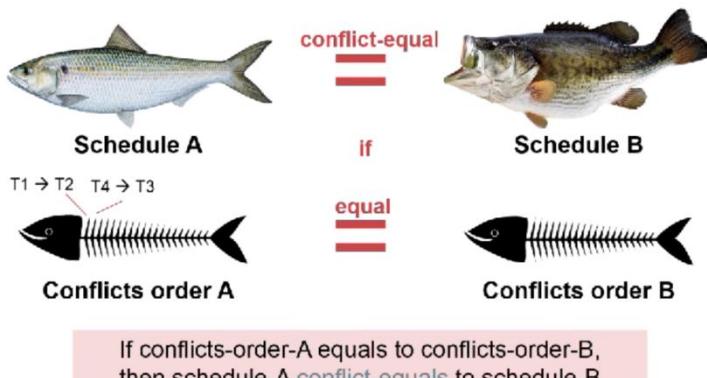
T1.1 -> T2.1

T2.2 -> T1.2

T2.2 -> T1.3

上图右侧的情况显然就是存在 conflict 的情况。调度有很多中，但是在 conflict serializability 的时候，我们只看冲突的那些。只要冲突的那几个，最后能够变成排好队的效果，我们就可以变成 conflict serializable。

Conflict Equivalence

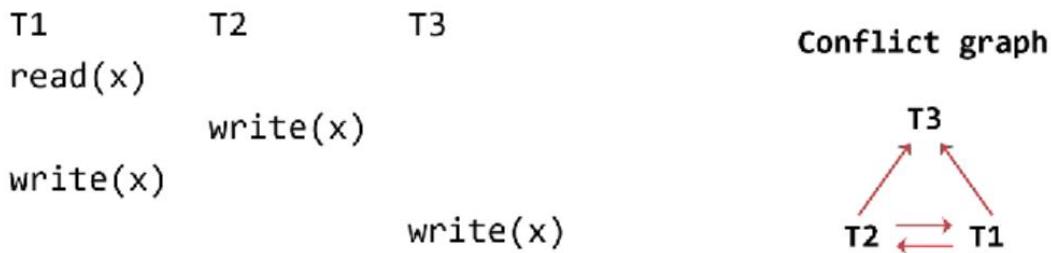


这样的好处就是我们只需要看存在 conflict 的情况，不需要考虑别的不共享的变量的情况。这种就叫 conflict equal。

View Serializability

允许 conflict graph 中有环的情况。

View Serializability



Cyclic -> Not conflict serializable

But compare it to running T1 then T2 then T3 (serially)

- Final-state is fine
- Intermediate reads are fine

Question: why shouldn't we allow this schedule?

它的 final state 的 ok 的，等价于 T1、T2、T3 这个序列往下执行。它不满足无环的条件，所以是不满足 conflict serializable 的。这种不允许调度是好还是不好呢？我们希望它可以执行，因为从结果和过程来看都是正确的。因为系统调度限制的越多，速度就会变慢，所以我们也不能加太多限制。在这个图里，我们是希望它是可以运行的。

所以我们需要 View serializability

Informal definition

- A schedule is view serializable if the final written state as well as intermediate reads are the same as in some serial schedule

Formally, for those interested

- Two schedules S and S' are **view equivalent** if:
 - If T_i in S reads an initial value for X , so does T_i in S'
 - If T_i in S reads the value written by T_j in S for some X , so does T_i in S'
 - If T_i in S does the final write to X , so does T_i in S'

A schedule is **view serializable** if it is **view equivalent** to some serial schedule

非正式定义：一个 schedule 满足 final return state 和线性是一样的，intermediate read 和某一种线性的 schedule 是一样的。

Question

Why conflict serializability when it seems **too strict**?

Why not focus on view serializability?

Final-state Serializability	View Serializability	Conflict Serializability
Care the final state only	Care the final state as well as intermediate read	Care the final state as well as all the data dependency

Final-state 是要求最少的，而 view 还要看中间状态，而 conflict 还需要 care data 之间的 dependency 关系。为什么我们的课依然要讲 conflict serializability 呢？是因为：

1. conflict serializability 从工程上更容易测试，view serializability 可能是 NP-hard 问题。
2. Conflict serializability 是很容易产生的，我们之后会讲到 two-phase lock，但是我们很难去产生一个 view serializability。

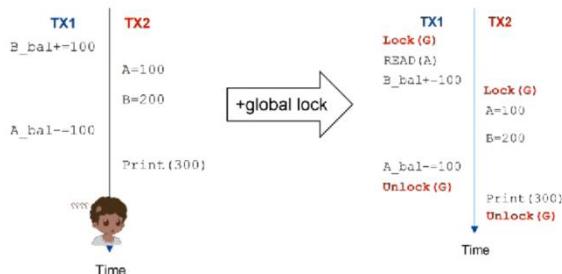
生成 conflict serializable 的调度

基本思路就是使用 lock。一个具体的例子就是 `pthread_mutex`。我们来看第一种：

global lock，也就是 tx 开始的时候 acquire lock，而 tx 结束的时候 release lock。这显然是对的。

Solution#1: Global lock

Can the previous non-serializable execution **happen**?



但是我们发明的 transaction 是为了并发，但是这样就效率很差。

Simple fine-grained lock

我们不是锁整个 transaction，而是锁一个数据。换句话说，改 xyz 和 abc 的两个事务是不需要互斥的。我们访问相同数据的时候才需要上锁。我们用完数据了以后就把锁放掉。

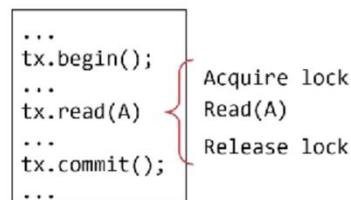
Solution#2: Simple fine-grained lock

Only acquire the data lock before accessing

Release the lock after accessing it

How to implement?

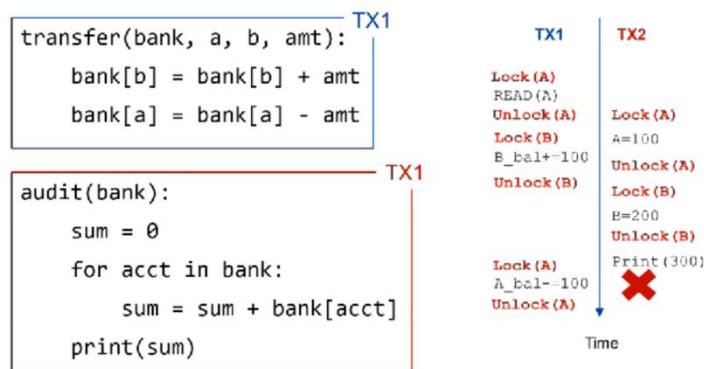
- Recall that TX must call TX's read/write interfaces to read/write data
- Intercept these call to add locks



大家觉得这样有什么问题吗？

Solution#2: Simple fine-grained lock

Can the previous non-serializable execution happen?



虽然写 b 会有保护，但是上例中我们会发现虽然 amount 加到了 B 上，但是统计的时候还是用 A 的原先的值，而不是转账后的 A 的值，A 和 B 之间没有保护，导致我们最后统计出来的总金额依旧是错的。因为这样的实现仅仅保护了单个数据的读写，而我们要保护两行的执行。于是我们就发现 simple fine-grained lock 在访问数据后放锁就会有问题，这个锁我们不应该放掉。所以我们提出 two-phase locking。

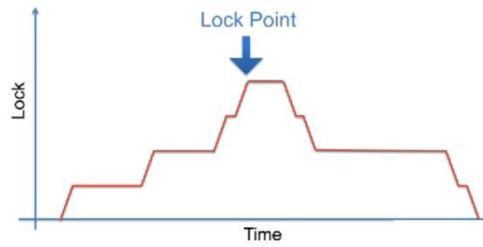
2PL(Two-phase Locking)

换句话说，我们选择在访问数据前去拿锁，然后我们到 transaction commit 的时候，我们再去放锁，这样就变成了我们在 transaction begin 的时候，我们每次读一个数据，我们就去拿它的锁，等到 transaction commit 的时候，我们一个个放掉 lock set 中的锁。

Solution#3: Two-phase locking

We can proof that two-phase locking can guarantee serializability

Why the name? Lock phase → lock point → unlock phase



第一个阶段就是拿锁，第二个阶段就是放锁。中间的 lock point 就是我们不再访问新的数据了，一旦我们开始放锁，我们就不能再拿锁。

为什么 two-phase lock 能产生 conflict serializable 的 schedule 呢？我们来看一个简单的证明：

我们假设 two-phase lock (2PL) 会产生一个带有环的 conflict graph。那么这个环的样子一定是 $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k \rightarrow T_1$ 。那么我们假设 T_i 和 T_{i+1} 之间的共享变量是 x_i 。这等价于：

```
T1 and T2 conflict on x1  
T2 and T3 conflict on x2  
...  
Tk and T1 conflict on x_k
```

```
T1 acquires x1.lock  
T2 acquires x1.lock and x2.lock  
T3 acquires x2.lock and x3.lock  
...  
Tk acquires x_{k-1}.lock and x_k.lock  
T1 acquires x_k.lock
```

既然 T_1 和 T_2 在 x_1 上 conflict，那么 T_1 要先访问 x_1 ，也就是 T_1 会想拿到 x_1 和 x_k 的锁。 T_2 就想拿 x_1 和 x_2 的 lock，……。

这意味着 T_1 要先 release x_1 的 lock，这样 T_2 才能获得 x_1 的 lock 继续执行，意味着 T_1 会在 release x_1 的 lock 后再尝试拿到 x_k 的 lock，但是这就破坏了 2PL 的假设：一旦我们开始放锁，我们就不再拿锁了。所以假设不成立，2PL 不会产生 circle。

优化：read-write lock

也就是我们发现读和写是不等价的，读的时候多人读都可以，但写的时候只有一个人写。所以我们要做到所有读者都能拿到读锁，而写锁只能被一个人获得。但是一旦有一个人在写，那么读者需要等在外面。而要等所有读者都走了，写锁才能进去。于是我们应用到 2PL 的时候，我们就变成了访问一个变量就要一把锁，等到 lock point 之后，我们基本上已经做完了，我们就先把读锁先放掉。这样 Lock-Time 图的面积就越少，性能就越高。

2PL 的主要问题

它并不能防止 dead lock，怎么解决？最简单，拿锁要按照顺序拿。如果拿了 b 的锁还想拿 a 的锁，那么先把 b 的锁释放再去拿 a 的锁。

2. 等到发现死锁了，再让一个人放锁。

这两个思路就是悲观（要设计一种机制让死锁不会发生）和乐观（不会死锁，发生了再去解决）的区别。

2PL 是否真的能满足 serializability 呢？

这个属于一个 tricky 的 case。

比如 tx1 是所有老师（原先有 4 个）涨 10% 的工资，而 tx2 是加入两个新老师。我们的执行的结果可能是给 5 个老师涨了工资，因为在锁的时候，我们只会去锁我们要去访问的变量，但是没有一个全局的区域锁。这两个操作它们之间并没有满足 conflict serializability。我们并没有能力保证 tx1 执行到一半的数据的范围发生了变化的情况。我们需要使用 predicate lock 或者 range lock 的情况。

2021/10/28

下单这个操作应该是在一个 transaction 中完成的，不能分开。Transaction 的本质是围绕数据的，本质上就是对数据操作造成了数据的变化，事务就是把多个数据状态的变化包在了事务中。

有了事务之后，我们就可以简化数据的管理，可以使用 transaction.begin 和 transaction.commit 把我们需要做的事务包起来。由于我们有 transaction.commit，所以事务的 commit point 非常明显。它俩提供了 ACID 的属性。

我们将 atomicity 显式地让程序员来定义。比如亚马逊提供了函数的接口，让程序员直接构建 serverless 的服务，这样亚马逊就可以提供 at most once。在这里也是一样的，告诉程序员写代码必须要写 commit，这就是抽象的力量。

Consistency：有两种，replica 和用户态应用的 consistency，它要做到的事情其实就是：

Review: Why serializability is ideal?

Assumption: programmers are pro at writing single-thread programs

- Specially, $\forall_{tx} C_i \rightarrow tx \rightarrow C_j$, in a single-thread context, tx can move data from a consistent state (C_i) to another consistent state (C_j)

Then, if transactions guarantee serializability, then the final state of concurrent execution is consistent

- i.e., the concurrent execution can reduce to $C_0 \rightarrow tx_0 \rightarrow tx_1 \rightarrow \dots \rightarrow C_n$. If C_0 is consistent, then C_n must be consistent

当前的状态的 **consistent**, 通过一个 **transaction**, 下一个状态也是 **consistent**, 那么我们就认为事务的 **consistent**。但是如果代码正确执行, 但是还是不 **consistent**, 那么我们就认为是代码的问题。

Review: Consistency

Transaction must change the data from a consistent state to another

- What is consistent is **defined by the applications**
- E.g., transfer should leave the $\text{sum}(\text{bank}[a] + \text{bank}[b])$ **unchanged**

Transaction alone is not sufficient for consistency

- E.g., If the programmer writes the incorrect program.
 - $\text{bank}[a] = \text{bank}[a] - \text{amt} * 2$

```
transfer(bank, a, b, amt):  
    bank[a] = bank[a] - amt  
    bank[b] = bank[b] + amt
```

如果代码写错了, 扣了两倍的钱, 那么我们事务 ACID 做的再好也没有, 因为这是代码的问题。当然也有一些工具去扫描代码去判断是否 **consistent**。

Isolation, 两个 **transaction** 要么你先要么我先, 这样可以避免 **race** 现象。

Durability: 存在持久化存储上。

通过 **log** 和 **recovery** 我们可以保证 A 和 D。C 是程序员和一些工具的事情。I 就是我们现在要考虑的事情, 对于 **isolation** 的定义要满足 **serializable**, **transaction** 执行的结果等于依次执行的值, 这是从结果出发的。

Serializability 为什么很理想呢? 因为写代码的时候有因果依赖, 这是程序员写单线程程序很擅长的思维, 而执行的时候可以变成多线程, 这就是提供 **Serializability** 的原因。

Review: Why serializability is ideal?

Assumption: programmers are pro at writing single-thread programs

- Specially, $\forall_{tx} C_i \rightarrow tx \rightarrow C_j$, in a single-thread context, tx can move data from a consistent state (C_i) to another consistent state (C_j)

Then, if transactions guarantee serializability, then the final state of concurrent execution is consistent

- i.e., the concurrent execution can reduce to $C_0 \rightarrow tx_0 \rightarrow tx_1 \rightarrow \dots \rightarrow C_n$. If C_0 is consistent, then C_n must be consistent

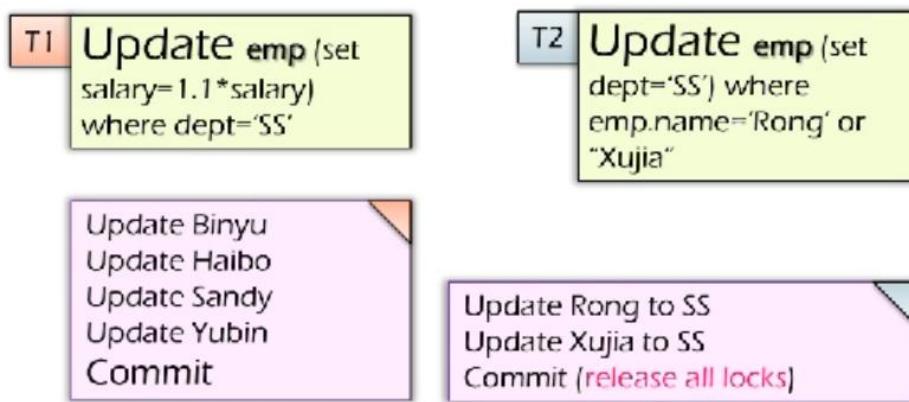
既然任取一个 transaction 都可以从 consistent 的状态到另一个 consistent 的状态，那么经过任何一个事务序列，只要 C_0 是 consistent 的，那么 C_n 也是 consistent 的。

我们定义了三种 Serializability,

1. final state
2. View
3. Conflict

它们之间的关系是 $finalstate \supset view \supset conflict$ ，我们最终讨论的是 conflict，我们可以 **通过 2PL 去实现 conflict**。2PL 比较容易实现，无非是加锁和放锁。它有一个 local point，一旦开始放锁就不能再拿锁。通过 2PL 实现 conflict serializable，这就和某种排队效果是一样的，那么我们就实现了 Serializability。

Phantom Problem



上节课提到了幻影问题，这是 2 阶段锁在现实中使用时遇到的实践的问题。比如 4 个人已经是软院的老师了，有两个人新加入软院，给老员工加工资，给新员工 set department。如果我们严格按照 QPL 来做的话，我们 touch 到的 data 是在整个这张数据库的表中，所有 item 的 department 先读一下，读完之后，我们要去做更新。然后就对四个人做写操作。假如我们现在有 100 个人，表有 name:salary:department。但是只有 4 个人的 dept 是 SS。

我们做第一个 transaction 的时候，如果严格按照 2PL，我们应该把所有的 department 这一块都得拿锁，改的时候，我们再去拿这 4 个 salary 的锁。所以我们应该拿了 104 把锁，在现实中这就太多了。很多人是在遍历 department 的时候，是不拿锁的，也就是最终只拿 4 把锁。但是只拿 4 把锁，会导致另一个事务在更新 department 的时候，会导致 T1 和 T2 不构成 before and after 关系。我们从上往下执行，结果第 6 个人是 ss。可能会导致我们希望给 4 个老员工加工资，又给了某个新人加了工资。现实中一般是使用 range lock，一把锁把整个列全部锁住或者就是直接 ignore 这个问题。

接下来我们反思，2-phase lock 是不是最完美的方法。它依然是有问题的，比如 dead lock。我们就需要排序，就是 abcd 有一个序列，我们的逻辑在执行的时候并不是按照这个序列的，按照这种情况访问的时候，我们该怎么样知道先拿哪些锁呢？我们怎么根据一个顺序去拿锁呢？

解决 dead lock 的方法

1. 就是 preorder 本身依赖 transaction，运行前就知道要拿几把锁。
2. 我们假设不会发生死锁，如果发生死锁了 abort 掉，但是这会要求我们维护锁的依赖关系才能判断是否出现了环，但是这个 cost 很小。
3. 使用启发式算法，使用超时时间来 abort 掉。但是有可能导致 live lock，比如事务的运行时间很长，每次都因为超时而被 abort。

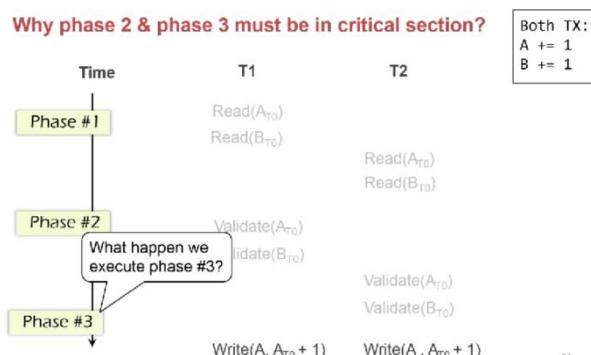
所以避免和检测 dead lock 都存在一些问题。我们能不能不要去拿锁？因为一旦有了锁，就会有死锁。不加锁怎么保证 before-and-after？

OCC（乐观并发控制）

我们可以在 commit 之前检查一次，如果不符合就 abort，这就是 OCC。

它的意思就是我们把整个 transaction 分成三个阶段

1. concurrent local processing，我们把所有需要读的数据读到 read set，把所有要写的数据放到 write set 中。Local processing 的意思是所有操作都是在本地执行的，没有加锁。当我们想要 commit，我们做一个 validation 操作，我们怎么知道 serialization 有没有保证呢？也就是 check read set 中的数据有没有被人修改，也就是它再读一遍要读的数据。
一旦确实读的一样，我们立刻用最快的速度把 buf 中写到该写的地方，也就是 commit result。
 - 如果读的不一样，那么就 abort
 - 第二次读完以后，读的数据再被修改了怎么办呢？我们必须保证第二步和第三步是一个 critical section（加锁变成一个原子操作）。



如果不加锁，那么还是有 race 问题，典型的 i++ 问题。

注意这里的加锁和 2-phase 的加锁是不一样的。因为 phase2 和 phase3 是很短的，phase2 就是再读入一遍 read set，phase3 就是把该写的写入。而整个 transaction logic 的时间，没有加锁，比起 2-phase lock 的加锁时间大大减少。我们可以支持更多的并发。

问题 1：怎么 lock？Read set 和 write set 哪个需要加锁？

对 T10

$$readset = \{A = 30, B = 20, C = 50, \dots\} \xrightarrow{T9} readset = \{A = 40, B = 30, C = 50, \dots\}$$

$$writeset = \{X = 30, Y = 40\}$$

A 一开始和验证的时候都是 30，相当于我们在 1 阶段和 2 阶段拿了一把锁。那么我们的 read set 是可以不加锁的。哪怕 validate 的时候拿到的数据没改，而之后被 T9 修改了，我们可以完全认为是 T10 发生在 T9 之前。所以 read 这件事情不用加锁，使得 OCC 这件事情的并发度更高。

如果已经被改了好几次，只是改的值相等，但是这也没问题，相当于别的操作做完了以后，我们再读了一次。只要它的值是一致的就可以。

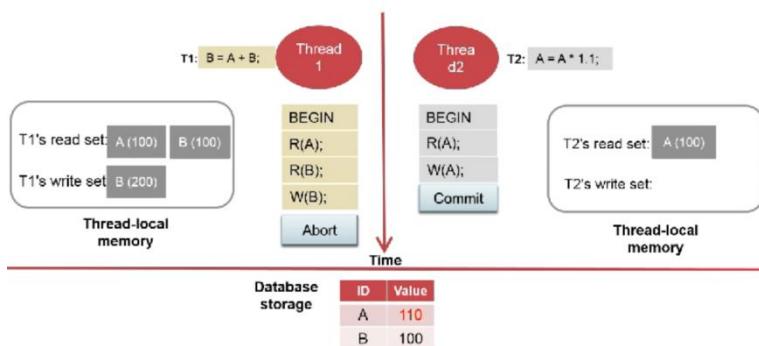
是否只需要在 commit 的时候，给 write set 加锁呢？事实上我们需要在第二步和第三步对 write set 加锁。

所以这里只需要给 write set 加锁就可以了。我们在实现 critical section 的时候，也可以使用 2 阶段锁。

问题 2：用 OCC 能不能完全避免 deadlock？

还是会有 deadlock，但是我们排序就可以了。在 2-phase lock 中，排序是很难的。但我们现在只有 write set，我们完全不用 care 2 阶段锁中存在的 if-else 导致顺序变化的问题。这就可以通过 transaction 底下的框架来帮我们做这件事情。使得我们 write set 的 deadlock 问题可以很方便的形式解决掉。

OCC 的例子：



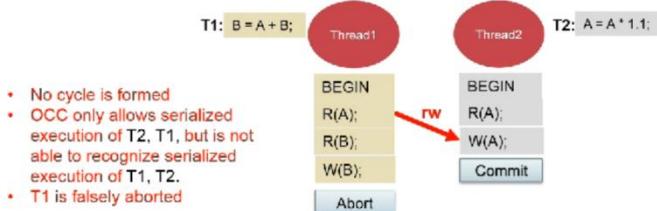
T2 先 commit 了，而 T1 发现 validate 时 A 和原先的数据不一样，就直接 abort。

但是这样执行的结果依赖于 T1 和 T2 执行的顺序，但是这个是应用的语义，而不是我们 transaction 的语义。我们只需要保证其顺序执行即可。

如果 B 不 abort，把 200 写进去了，那就相当于先做了 T1 再做了 T2。所以 OCC 也不是很完美的，会出现这种 false abort（不需要 abort，但是 abort 了）的情况。针对这个 case，有很多的研究工作，让它不要 abort。

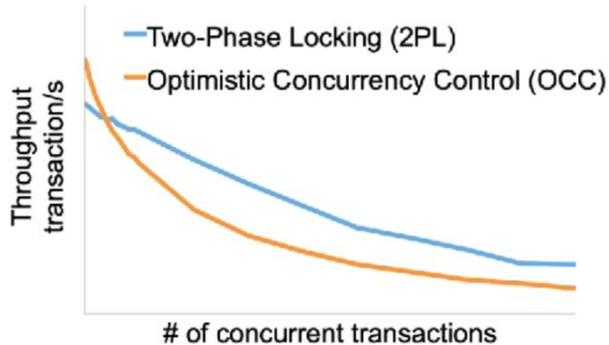
OCC's Problem: False Aborts

Some transactions aborted by OCC could have been allowed to commit without causing an unserializable schedule



OCC 的使用的条件是 low contention。否则就会导致更多的 abort。对于高竞争的事务来说，就没那么好。换句话说，如果我们的 read set 很大，它更容易有一个元素被修改，那么 abort 的概率越高。在 contention 很高的情况下，OCC 会持续地 abort。

Serializability is costly under contention



from: Rococo [OSDI'14]

随着 concurrent transaction 的增大，OCC 的性能就迅速变差了，因为大量时间都在 abort 中。2PL 最多就是等待，而 OCC 是要重做的，可能要重做 5~10 遍。可能做 OCC 和 2PL 的 hybrid。
*能不能缓存计算图的子图结果？

RTM (restricted transactional memory)

OCC 简单到硬件可以做 transaction memory。硬件能不能提供 transaction 来简化软件的开发？

Hardware Transactional Memory

A new CPU feature, inspired by the ACID properties of transactions

CPU guarantees the ACI properties of memory accesses

- i.e., no race conditions & no locks

Intel supports HTM named as RTM

- Restricted Transactional Memory
- In Haswell processor

Semantic of both all-or-nothing and before-or-after

- Not handles failure since it is memory



A little introduction to RTM

ISA support for transactional memory

Recall: Writing with TX is straightforward

- Use TX.begin to mark when a TX starts
- Use TX.commit to commit the TX

In RTM, the concept is similar (using new instructions)

- Use `xbegin` to mark an RTM execution start
- Use `xend` to mark an RTM end

我们需要用额外的硬件接口。硬件如果发现在 transaction 中有 load 指令，那硬件就帮你来维护 read set 和 write set。所以在英特尔的指令中引入了 `xbegin` 和 `xend`。中间所有的读写操作都会被记录下来，在 end 的时候 check read set 是否被修改过。它是怎么知道谁改了什么东西呢？

这就牵涉到 cash coherence。如果核 1 修改成了 $a=20$ ，所以 CPU 要广播给其他所有核，让它们一致从这个 CPU 中取最新的值，这就是目录协议，要维护一个很复杂的表。我们发现这件事情和 OCC 有点关系，当我们写了点东西的时候，我们可以通知别人。那么我们怎么处理多次修改成相同的值呢？硬件上是要一旦修改就要广播的。Abort 掉以后就可以 rollback 到这个指令。

Programming with RTM

If transaction starts successfully

- Do work protected by RTM, and then try to commit

Fallback routine to handle abort event

- If abort, CPU rollbacks to line `xbegin`, return an `abort code`, and goto fallback

Manually abort inside a transaction

```
if _xbegin() == _XBEGIN_STARTED:  
    if conditions:  
        _xabort()  
        critical code  
        _xend()  
    else  
        fallback routine
```

48

How to handle aborts? Can we use simple retry?

RTM provides new instruction set

- `xbegin()`, `xend()` & `xabort()`

Problem: RTM cannot guarantee success

```
Beginning:  
if _xbegin() == _XBEGIN_STARTED  
    /* do some critical work */  
    _xend()  
else  
    goto beginning  
  
Simply Retry
```

Can we use simple retry? No

Problem: RTM cannot guarantee success

- E.g., what if there is a page fault during critical work?

Must switch to a fallback path after some retries (e.g., using a counter)

- E.g., degrade to use lock in the fallback path (since it's not in a transaction)

```
if _xbegin() == _XBEGIN_STARTED  
    if Lock.held()  
        xabort()  
        /* do some critical work */  
        _xend()  
    else  
        Lock.acquire()  
  
Switch to Pessimistic Sync.
```

在现实中这样一直 rollback, 这就是 live lock。如果重试次数太多就退回到最传统的 lock。

那么 RTM 可以实现 ACI, 少了一个 D, 因为它是 memory 上的, 只能保证 ACI, 并且也不能保证一直成功。

Fun facts about the implementation of RTM

The benefits of RTM

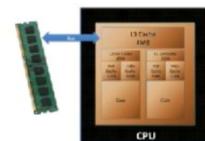
- Memory operations between `xbegin()` & `xend()` satisfy ACI properties

Drawbacks

- Cannot guarantee success

Intel implements RTM using OCC

- Use CPU cache to track the **read/write sets**
- How to detect conflicts? Cache coherence
- **Questions:** what are the other limits of using CPU cache to track read/write sets?



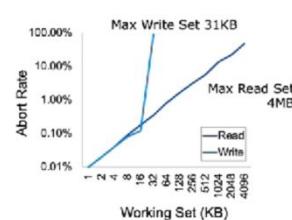
并且英特尔的硬件事务管理, 不能使得 read set 和 write set 无限大。如果 read set 在 cache 中放不下了, 那么就会 abort。

Another drawback of RTM: limited working set

Reason: CPU has limited cache size

How big is the RTM read/write sets?

- Depends on various factors
 - Hardware setup (e.g., CPU cache size)
 - Access pattern: read or write
 - L1 cache tracks all the writes
 - L2, or L3 to tracks all the reads
 - Why read set is bigger?

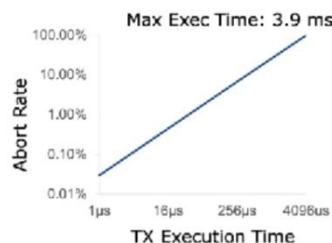


如果一个 transaction 要跑 100 毫秒, 那也是不行的。因为出现中断的情况下也是 100% abort。

How long can an RTM program execute?

Not very long

- The longer the execution, the higher the probability a TX aborts
- Affected by other facts, e.g., CPU interrupts



硬件实现 transaction memory 是大势所趋，因为它性能好、写起来简单。

下一个问题是：我们说了半天依然是使用 lock，只是压缩到了 validation 和 write 两个步骤。我们能不能实现 lock free 的 read 呢？

我们需要适当的放松一下 serializability 的要求，来增加并发。最常见的场景是，对大量的数据做分析（read-only analysis）、更新一小部分数据。

如果我们用 2PL 来做，

Performance

- Long-duration locks on all READ/WRITE
 - Scenarios:
 - run analytics on a companion data, while update a piece of data
- long running read-only transaction

Disadvantages of locking-based approach (2PL)

- May involve in deadlocks
- Transactions may hold a lock for a long time
 - Read-only transactions (e.g., *data mining*) can block update transactions

Disadvantages of OCC

- May involve in a livelock
- Transactions may retry for many times
 - Read-only transactions (e.g., *data mining*) is more likely to abort

Multi-version Concurrency Control

2PL 和 OCC 能够保证 transaction，但不是最完美的。需要 lock 的最根本的原因是因为数据只有一份，一旦改了被别人读到了就是最新的。所以我们需要锁起来不让别人读。既然根本原因是 one-copy，我们怎么解决这个问题呢？我们提出 multi-version concurrency control，也就是我们不只有一份数据，而是我们有多份数据的 copy。

如果我们数据库中每一个值都有多个备份，那么我们就可以不加锁。我们改的是新的，而我们读的是旧的。我们每次加新的 version。我们读的时候只需要读相同的 snapshot 就可以了。

Multi-version Concurrency Control

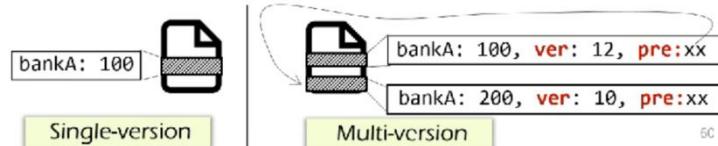
Each data item has **multiple versions**

- When accessing different versions of data, probably no conflict!

Key (high-level) idea

- Writes **don't overwrite** the original data
- Instead, writes **install new versions** of data
- Reads read from a "**snapshot**" of data

Benefits: no lock or validation during TX's execution



这就是我们说的 snapshot isolation

Snapshot Isolation

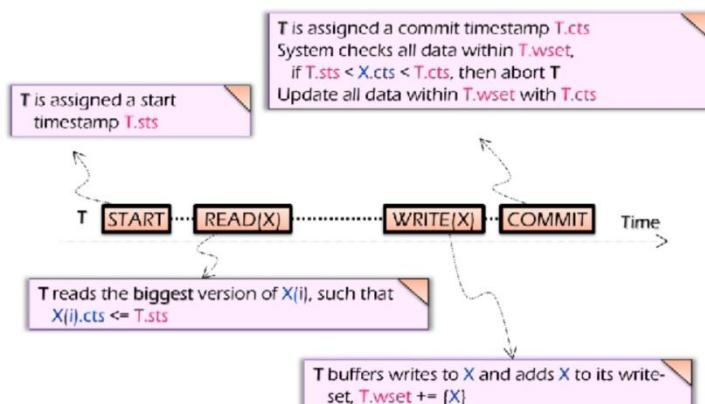
A popular multi-version concurrency control (MVCC) scheme

Transactions:

- **WRITES** a local **buffer** (similar to OCC)
- **READS** a "**snapshot**" of entire data image
- **COMMIT** only if no **write-write** conflict
 - Install new versions of data

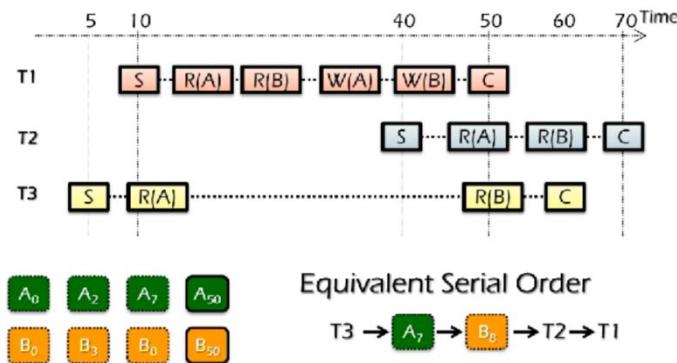
Commit 的时候，只需要没有写-写冲突，就可以把新的值放到新的地方。

Snapshot Isolation Implementation



Commit 的时候，检查如果在 write set 中如果 X 的 complete time stamp 在 start 和 complete 之间的，那么就要 abort 掉。X[i].cts 为什么会大于 T.sts 导致我们最终的 abort 呢？因为？这个例子可能有问题。

Snapshot Isolation Example

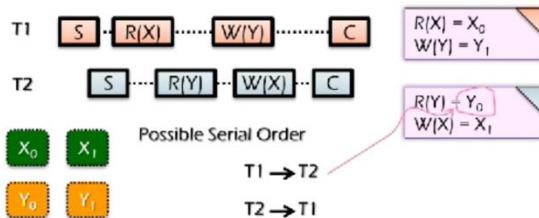


相当于 $T3 \rightarrow$ 写 $A7 \rightarrow$ 写 $B8 \rightarrow T2 \rightarrow T1$ 。

Question of Snapshot Isolation

Snapshot isolation (SI) differs from serializability due to one anomaly that is possible under SI but not under serializability.

Describe the anomaly and also give a concrete application for which the anomaly is undesirable.



但是 snapshot isolation 并不能保证序列化，一旦发生这个 case。在 T1 的时候读 X 写 Y，在 T2 的时候读 Y 写 X。这就会破坏 serial order。那出错了怎么办？让程序员选择这件事情即可。

2021/11/2

11/18 期中考

一个 transaction 就是数据管理的抽象。

Review: What is a transaction (TX)?

An abstraction to manage the data

Data is also an abstract concept, can be arbitrary computing data.

Concrete examples including:

- Key-value store entries
- File system metadata (e.g., directory, inode, etc.)
- Processor's metadata (e.g., child processors)

Look like similar program, with data managed by the TX system, and extra mark to denote the start/end of a TX

如果是文件系统的话，就可以用 metadata，如果是 kv-store 那么就有所不同。Failure 会

导致中间状态的暴露，如果允许暴露，整个系统会变的复杂无比。要从技术上没有人能把模块再分了，大家不用再关系操作里面在做什么，这就是 **enforced modularity**。对于 **transaction** 有基本语言，**begin** 和 **commit**，中间如果出现了失败，那么就 **abort**。有了事务就可以保证 **acid**，其中 **(all or nothing)atomicity** 和 **durability** 可以用 **write-ahead log** 来实现，**isolation** 就是 **before or after atomicity**，可以用锁来实现，强调的时候多个并发的时候的原子性。**Consistency** 就是事务在执行前后的状态是一致的，依赖于上层写代码的逻辑保证。

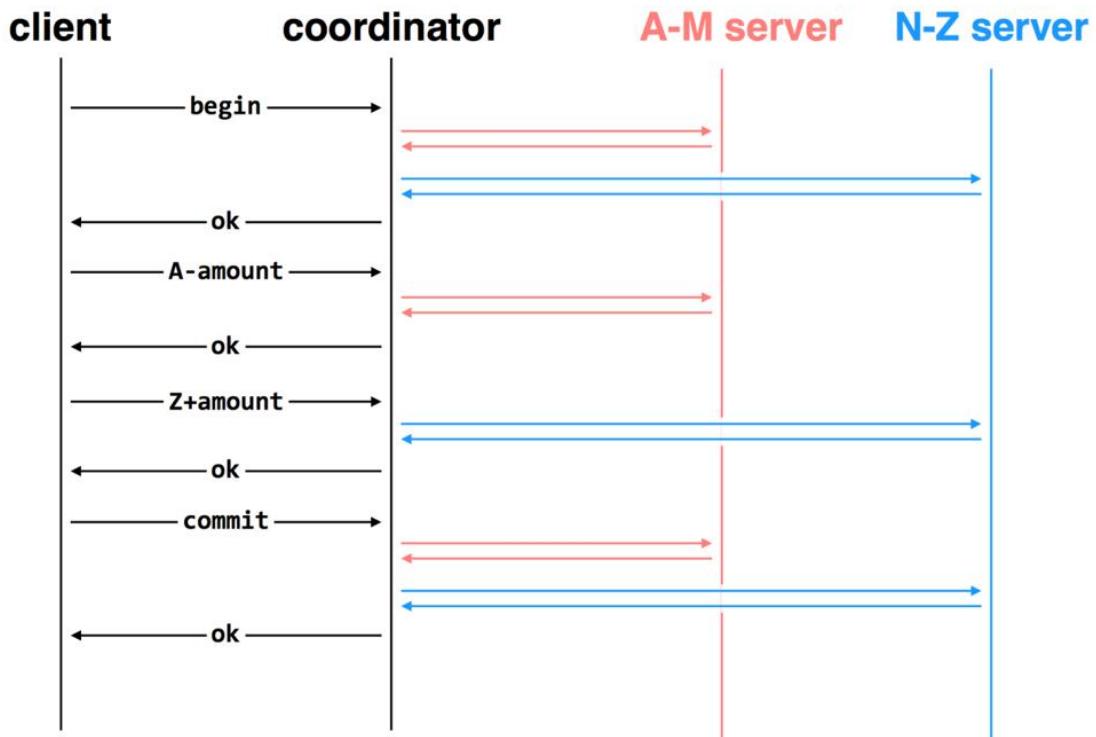
分布式的 Transaction

当数据分布在不同设备上的时候，我们怎么样保证语义呢？换句话说就是一个分布式的 **transaction**，在我们强调 **scalability** 中的多台机器的系统中是很重要的。也就是怎么用成千上万套便宜机器构建一个可靠的、性能、处理、存储能力不断上涨的分布式的系统。

我们先来看两台 **server** 的场景，两台 **server** 就可以用 **coordinator**，两台机器连到 **coordinator** 上，让它连接两台机器。如果两台 **server** 直接暴露给 **client**，那么会增加复杂性。

我们现在做一个简单的 **kv-store**，**key** 就是 **a-z**，我们把 **a-m** 放到 **server1**，**n-z** 放到 **server2**，我们假设 **coordinator** 和 **server** 自己有自己的 **log**，这用来保证 **consistency** 和 **atomicity**，**coordinator** 会把来自 **client** 的信息发给 **server**。

Client 开始做 **transaction**，发一个 **begin**，**coordinator** 会和 **server** 说要开始一个 **transaction**，然后告诉 **client** **ok**。**Coordinator** 就是一个传话的。



如果我们现在 **a** 想给 **z** 转账，这两台机器就要做一个分布式的转账，**coordinator** 需要同时和两个 **server** 说要开始一个 **transaction**。这是一个很简单的协议，无非就是让 **coordinator** 把来自 **client** 的请求分别发给 **server1** 和 **server2**

网络的引入导致了一系列问题，**message** 可能会丢掉。我们永远不知道一个 **message** 在半路上被丢了还是发送给对方以后 **ack** 被丢了。

我们可以使用 **RPC** 做这件事情，不断重发就可以得到 **at least once**，**RPC** 保证 **at most**

once 有几种方法？

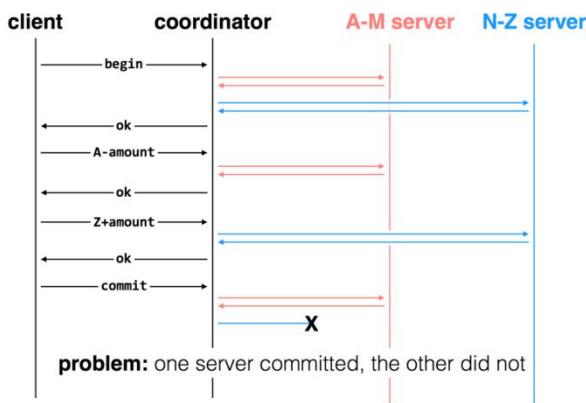
1. 幂等。
2. 记录下来。

在整个过程中，a-M 在任意节点 crash、coordinator crash、N-Z server crash、包在某个地方丢了 etc。我们需要想一个办法来满足在任意一个地方出现意外我们都能 handle 掉。

The Main Problem

Multiple servers can experience different events
– One commits while the other crashes, or
– One commits while the other aborts, etc.

我们先来看一个问题，最后在 commit 的时候，A-M commit 了，N-Z 此时故障了。



那么这个结果理论上分析应该是 nothing，但是对于 A-M 已经 commit 了怎么办呢？我们不能让 coordinator 告诉 A-M Server 要 commit。这就说明我们传统的 commit 操作已经不够了。所以我们提出 two-phase commit。

2PC(Two-phase Commit)

Phase-1: preparation / voting

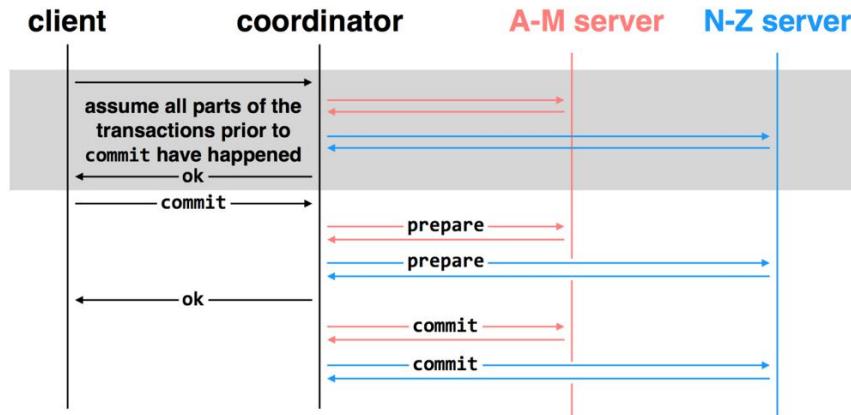
- Lower-layer transactions either abort or *tentatively committed*
- Higher-layer transaction evaluate lower situation

Phase-2: commitment

- If top-layer, then COMMIT or ABORT
- If nested itself, then become tentatively committed

两阶段 commit，当 coordinator 让 server commit 的时候，

- 第一阶段让 lower layer 事务要么 abort 要么临时地 commit（不是真的 commit，是暂时的 commit）。higher-layer 事务（coordinator）把底下的 server 情况收集起来。
- 第二阶段就是顶层要么 commit，要么 abort；如果是嵌套的过程让子节点全部变成 tentatively commit 了以后，最后顶层节点才可以 commit，然后一路通知下去。



two-phase commit: nodes agree that they are ready to commit before committing

此时如果最后的 commit 挂了，分为

- 网断了，coordinator 的 commit 请求没法出去，那么再发一次。
 - N-Z Server 挂了重启以后，coordinator 还是会重新发这个 commit 请求，并且 N-Z server 发现 log 中有一个 prepare 的还没有 commit，它也可以主动问一下 coordinator。
- Commit point: (在 point 前 failure, 就是 nothing; 在 point 后 failure, 就可以完全恢复)
- 在这个过程中，我们整个 transaction 的 commit point 在哪里呢？对于 coordinator 其 log 是在 ok 之前写，所以 commit point 是在 coordinator 发给 client ok 前写 log 的时候。一旦这个 commit log 被写到硬盘上了，整个 transaction 就 commit 了。

Multiple-site Atomicity

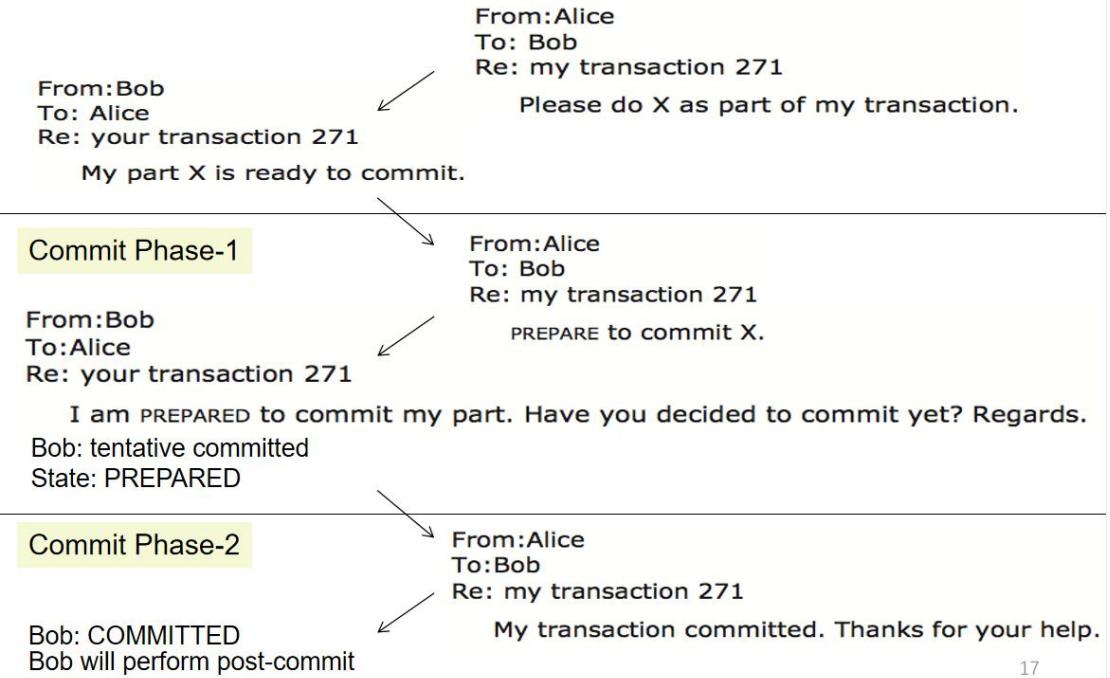
Worker: Bob, Charles, Dawn

- Does three transactions: X, Y, Z

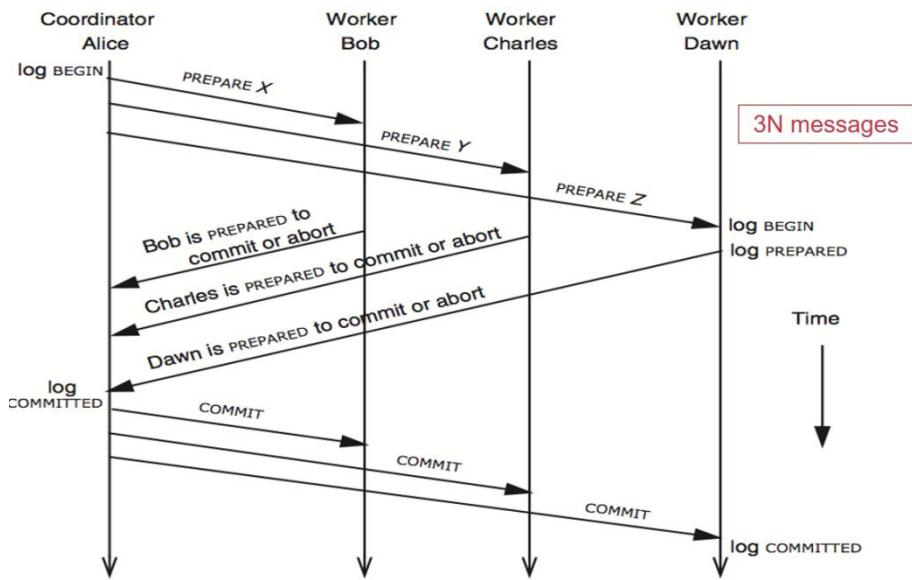
Coordinator: Alice

- Create a higher-layer transaction
- Send three messages to the three workers

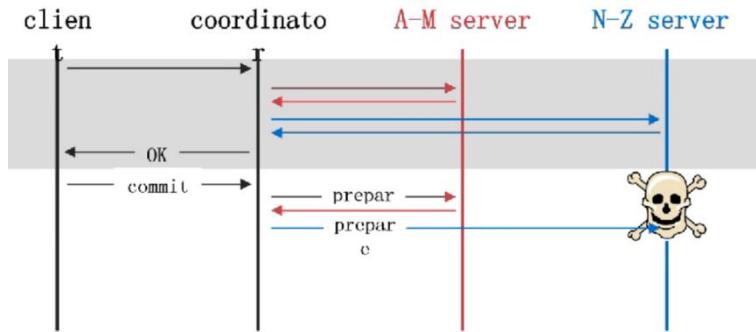
Challenge: un-reliable communication



Alice 让 bob prepare。 BOB prepare 完告诉 Alice prepared，然后 Alice 让 Bob commit。



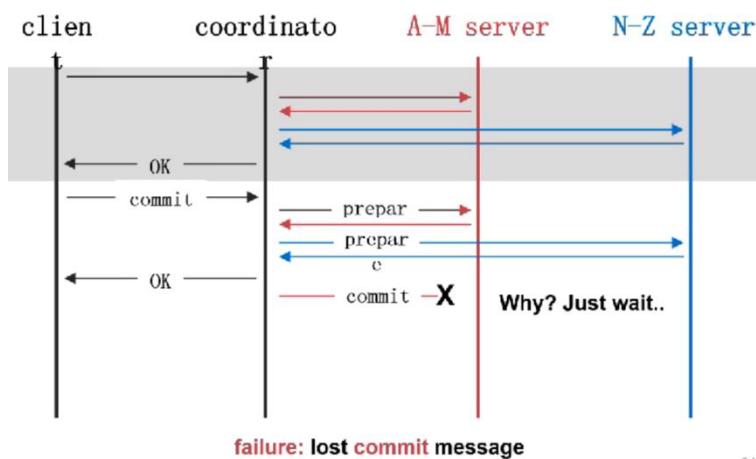
所以 coordinator 要收集 worker 信息，如果 worker 没有收到消息，它就会不断重试发送消息。



failure: worker failure during prepare

coordinator can safely abort transaction, will send explicit abort messages to live workers

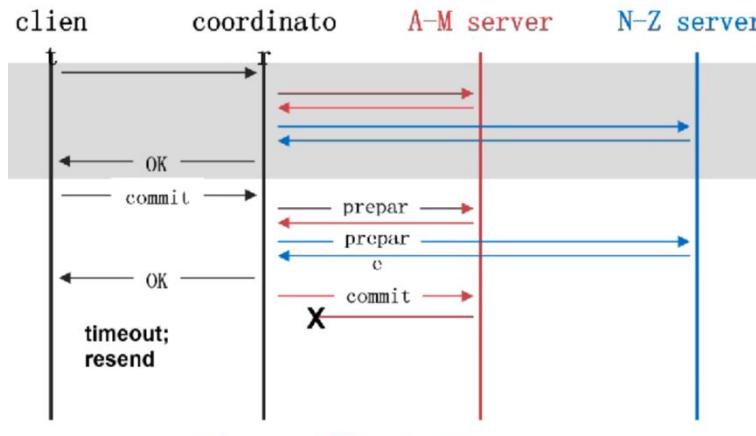
我们发现 N-Z Server 心跳包没了，那么 coordinator 就 abort。在 commit point 前发生了 crash，所以我们没有必要等待 N-Z Server 重启，直接 abort 即可。



failure: lost commit message

24

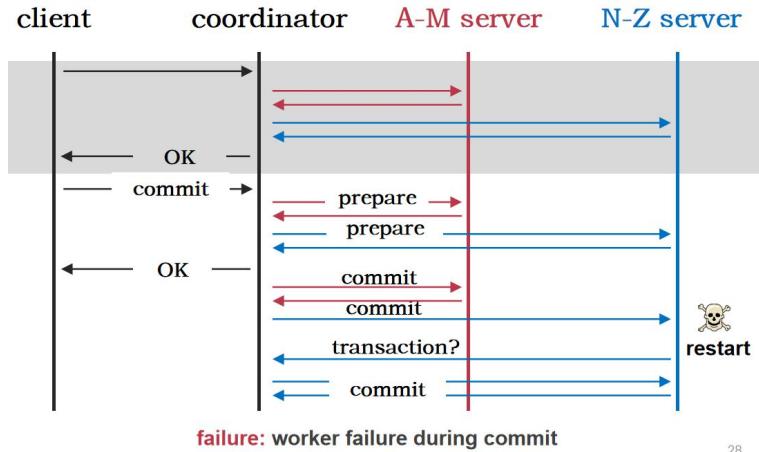
发完 ok 以后发现 commit 的 message 丢了，要么让 coordinator 主动重发，要么就是 server 主动问一下 coordinator 为什么没有消息了（因为已经 prepared 了，所以 Server 是知道自己在中间状态的）。



failure: lost ACK of commit message

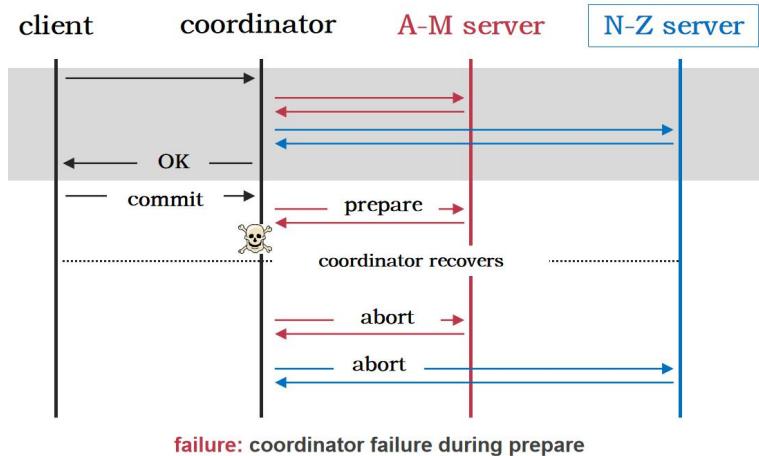
25

如果 coordinator 没有收到 A-M server 的 ACK，就 coordinator 再问一次。

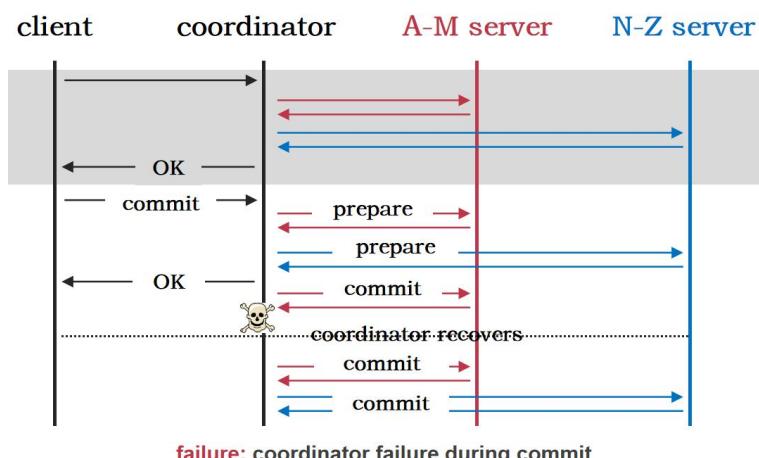


28

此时我们已经在 commit point 后了，不能够 abort，worker 在 prepare 的时候就需要把 prepare 的 log 写到硬盘上，重启以后发现 transaction 处在 prepare 状态，worker 因为不知道自己该怎么办就可以问一下 coordinator 刚才那个 transaction 到底是 commit 还是 abort。



重启完发现 commit 的事务并没有收到任何的 prepared，所以 coordinator 重启以后就直接发送 abort 给 server 即可。



重启以后再发一遍 commit 到两个 server 即可。

有了 2-phase commit，就是多个 site 不能形成一个 commit point，但是在一台机器

coordinator 上是可以通过 log 提供 commit point 的，但是在前面这个 case 中，A-M server 一旦挂了，所有数据都没了，仅仅这样实现的话，availability 是不高的。

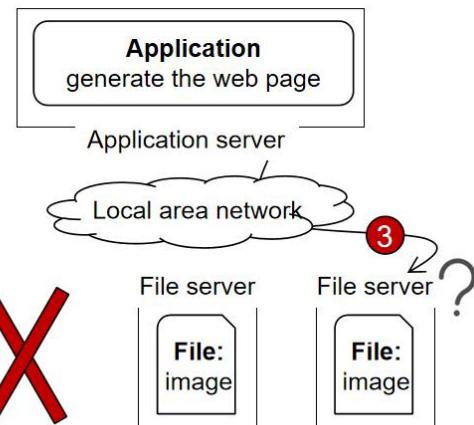
2-phase commit 和 2-phase lock 一点关系都没有。前者实现 all-or-nothing，而后者实现 before-or-after。现在我们的 availability 是很差的，一旦某台机器发生了 failure，那么这个服务就不可用了。解决这个问题也很简单，我们有两个 A-M Server 和 N-Z Server，冗余是容错的唯一方法。于是我们就来到了 replication，一旦讲到 replication 就势必讲到 consistency，这里的 consistency 是系统级别的 consistency，而在 transaction 中 acid 的 consistency 是语义级别的。

使用 replica 主要是为了容错和提升性能（不让一台机器成为瓶颈，尤其对于读操作）。通过 replica 可以得到更低的 latency，因为 cache 可以认为是上下层的 replication，还有一种是横向的 replication，比如 server A 和 server B 是相同的，用户可以选择连 A 和 B。

Review of the challenges of replication

Suppose a file (image) is replicated on three servers

1. Application puts file A to one server
2. The server crashed
3. What happens when read A from a live server?



我们再来看一下 replication 的 challenge，假如一个文件保存在服务器上，有三个 file system 的 replica，比如文件 1 通过网络放到一个 server 上，然后这个 server crash 了，那么通过其他 server 获取文件 1 就得不到了。在这种情况下，这种简单协议是不 work 的，我们怎么保证 replica 的文件服务器是一致的呢？OCC 就说 abort 的概率很低，而 2PL 是没有 abort 的，可以保证是排好队的。

*复习一下，悲观的死锁解决方法：排序；乐观地解决死锁的方法：放锁打破环。Transaction 做 isolation 的时候也有悲观和乐观，为了保证 before-or-after 这个属性，悲观的保证顺序执行的方法是我们认为不会 conflict，乐观就是有可能发生 conflict，那时候就 abort。

于是就有两种思路：

1. 乐观备份。
2. 悲观备份。

我们先看乐观的情况。

确定文件的新旧的前置问题：同步时间

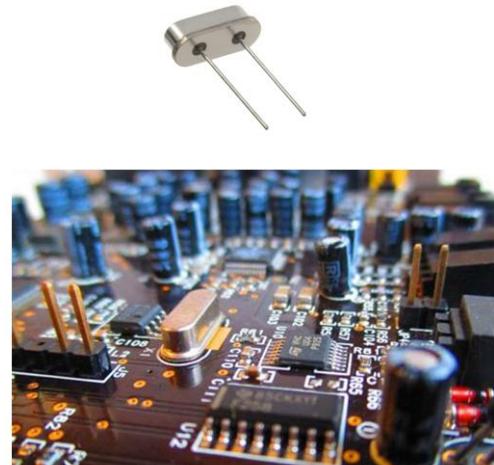
我们在生活中有很多设备，比如我们做了一个 PPT，这就涉及到在台式机和笔记本上怎么确定文件的新旧，一个最简单的方法就是通过文件的时间戳。时间在分布式系统中被用到的是非常多的，比如 DNS cache 的有效期是 24 小时，百度如果要换 ip 的时候，需要保证原先 ip 可以工作 24 小时。还有一种是做计量，计算一个操作花了多久。日历时间：当前的时间是多少，给事件打上时间戳做排序。

既然时间很重要，那么怎么样测量时间呢？

Time Measuring

Measuring time intervals

- Computer has a reasonably-fixed-frequency oscillator
 - E.g., quartz crystal
- Represent time interval as a count of oscillator's cycles
 - time period = count / frequency
 - e.g., with a 1MHz oscillator, 1000 cycles means 1msec



石英可以非常稳定地产生固定频率的晶振，比如 1MHz 的振荡器产生 1000 个 cycle 的时间就是 1 毫秒。所以我们在计算机中就可以把时间记录成从 1970.1.1 到现在这个时刻经过了多少秒。当我们把计算机关掉的时候，这个时间并不会消失。在我们主板上有一个小电池支撑这个时间不断往前走。这个时间并不是非常的准，晶振不是人类想的那么完美，振动的次数是存在误差和偏移的。

NTP 对外提供对时服务。

Clock Synchronizing

Need to take into account network latency

- Simple estimate: RTT/2
- When does this fail to work well?
 - Asymmetric routes, with different latency in each direction
 - Queuing delay, unlikely to be symmetric even for symmetric routes
 - Busy server might take a long time to process client's request
- Can use repeated queries to average out (or estimate variance) for second two

过去的时间和回来的时间可能是不一样的，这样我们计算的时间就不准了。

Estimating Network Latency

```
sync(server):  
    t_begin = local_time  
    tsrv = getTime(server)  
    t_end = local_time  
    delay = (t_end-t_begin) / 2  
    offset = (t_end-delay) - tsrv  
    local_time = local_time - offset
```

操作系统内部维护的就是这么一段代码，但是是有问题的。如果调时间这么调会有什么后果？比如我们时间慢了十秒钟。`t_end` 可能比 `t_begin` 小，`delay` 可能变成负数或者巨大的数字。

Clock Synchronizing

What if a computer's clock is too fast

- E.g., 5 seconds ahead
- Naive plan: reset it to the correct time
 - Can break time intervals being measured (e.g., negative interval)
 - Can break ordering (e.g., older files were created in the future)
- "make" is particularly prone to these errors

Principle: time never goes backwards

- Idea: temporarily slow down or speed up the clock
- Typically cannot adjust oscillator (fixed hardware)
- Adjust oscillator frequency estimate, so counter advances faster / slower

应用程序假设时间是不会往回走的。

Slew Time

```
sync(server):  
    t_begin = local_time  
    tsrv = getTime(server)  
    t_end = local_time  
    delay = (t_end-t_begin) / 2  
    offset = (t_end-delay) - tsrv  
    freq = base + ε * sign(offset)  
    sleep(freq * abs(offset) / ε)  
    freq = base } temporarily speed  
                                up / slow down  
                                local clock  
  
    timer_intr():      # on every oscillator tick  
        local_time = local_time + 1/freq
```

49

我们在更新时钟的情况下，我们每次更新的时候少加一点，这样我们的时钟就慢下来了。我们不能一次性地去调整，需要渐渐地去调整。

有了这个时间戳以后，我们就可以拿到一个比较准的时间。

Reconciliation 就是解决冲突和中断。文件产生冲突的核心原因就是版本不一样，最简单的方法就是文件 inode 中的 `m_time`。

File Reconciliation with Timestamps

Key Problem

- Determine which machine has the newer version of file

Straw-man

- Use the file with the highest *mtime* timestamp
- Works when only one side updates the file per reconciliation

如果两台机器开始时间是一样的，如果对同一个文件在两台机器修改，单纯看 *m_time* 是不够的，我们不能用 *m_time* 来覆盖另一台电脑上的文件的修改。我们的目标是写的数据不能丢，解决这个问题的方法就是 *vector_timestamp*，每个文件中维护了一个时间戳向量，记录了在每台机器上的时间戳。只有在时间向量可以比较的时候，才去做比较。在一个节点上的时间戳永远只和自己比较。

Vector Timestamps

Idea: vector timestamps

- Store a vector of timestamps from each machine
- Entry in vector keeps track of the last *mtime*
- V1 is newer than V2 if all of V1's timestamps are \geq V2's
- V1 is older than V2 if all of V1's timestamps are \leq V2's
- Otherwise, V1 and V2 were modified concurrently, so conflict
- If two vectors are concurrent, one computer modified file without seeing the latest version from another computer
- If vectors are ordered, everything is OK as before

比如：

$$A_1 = \{13:20, 14:20\}$$
 这就是可比较的情况， A_2 显然比 A_1 更加新。
 $A_2 = \{14:40, 14:20\}$

$$A_1 = \{15:20, 14:20\}$$
 这就是不可比较的情况，系统不能帮我们处理这种情况。
 $A_2 = \{14:40, 14:20\}$

这就是乐观带来的问题。

悲观的情况怎么处理？我们不希望产生任何 *conflict*。我们最后的目标想到达 *single-copy-consistency*。

1. 加锁，同时只有一个人可以改，但是性能会受到影响

一个 *client* 给向两个 *server* 发请求，一旦一个挂了，那么就发给另一个。但是因为网络关系可能导致 *network partition*。

处理 network partition: Quorum

Handling Network Partitions

Issue: Clients may disagree about what servers are up

- Hard to solve with 2 servers, but possible with 3 servers

Idea: require a **majority** servers to perform operation

- In case of 3 servers, 2 form a majority
- If client can contact 2 servers, it can perform operation (otherwise, wait)
- Thus, can handle any 1 server failure

我们现在有三个服务器，写的时候我们同时写三份，一旦两个服务器回复写完了，那么我们就认为写完了。在读的时候，如果从正好挂了的服务器读，那么就有问题了，所以也要发 3 个请求，收到两个数据是相同的，那么就认为读到了。

在这种情况下，我们可以保证读到的一定是最近写进去的数据，这可以容忍一个 server 发送 **failure** 的情况。这就是 **Quorum**，要求读进去的数量和写进去的数量必须大于 **replica** 数量 N ($2+2>3$)。这样保证我们读到的 **replica** 一定有新的。

Quorum

Define separate read & write quorums: Q_r & Q_w

- $Q_r + Q_w > N_{\text{replicas}}$ (Why?)
 - Confirm a write after writing to at least Q_w of replicas
 - Read at least Q_r agree on the data or witness value

Example

- In favor of reading: $N_{\text{replicas}} = 5$, $Q_w = 4$, $Q_r = 2$
- In favor of updating: $N_{\text{replicas}} = 5$, $Q_w = 2$, $Q_r = 4$
- Enhance availability by $Q_w = N_{\text{replicas}}$ & $Q_r = 1$

写 4 读 2 显然是对读友好，我们可以很快地读。写的时候只允许 1 个 crash，读的时候我们允许 3 个 crash。

注意 $Q_w = 2$ 并不是说只写两个，而是 5 个都要写，但是有 2 个返回写完就继续往下走了。写两个的话，**latency** 比写 5 个收到 4 个的 **latency** 要低。

那么如果我们通过这种方法，也可以优化 **availability**，也就是每次写都收到响应，这就要求写的操作相对比较少。

Quorum 的思路是非常简单的，但是写都要写 5，读的时候就可以读 2 个之类的。

2021/11/4

上节课讲了多台机器上要做 **transaction** 要去怎么做，**commit point** 在什么地方，上节课讲的是 **2-phase commit**，有一个 **coordinator** 问大家是不是可以 **commit**，如果可以 **commit**，

它收集到所有的 commit ready 消息之后，它就发 commit 发送给大家，然后让大家真的 commit。所以此时 coordinator 写了 commit 的日志就是一个 commit point。我们发现有一个问题，如果对于 A-M,N-Z 有任何一个 server 出了问题，我们不能 commit，会返回 abort。我们并不能保证其 availability，所以我们想了一个办法就 replication，它可以提高可用性和 throughput、latency 也是，有了 replica 以后可以在交大放一台 b 站服务器，这就会一点不卡。

现在我们的基站很强大，5G 基站大量的算力空着，可以让基站把自己的算力租给 B 站，那么我们无论到哪，我们最近的基站就可以去要我们想看的视频，这就是随身定制的 replica。这就是边缘云计算。

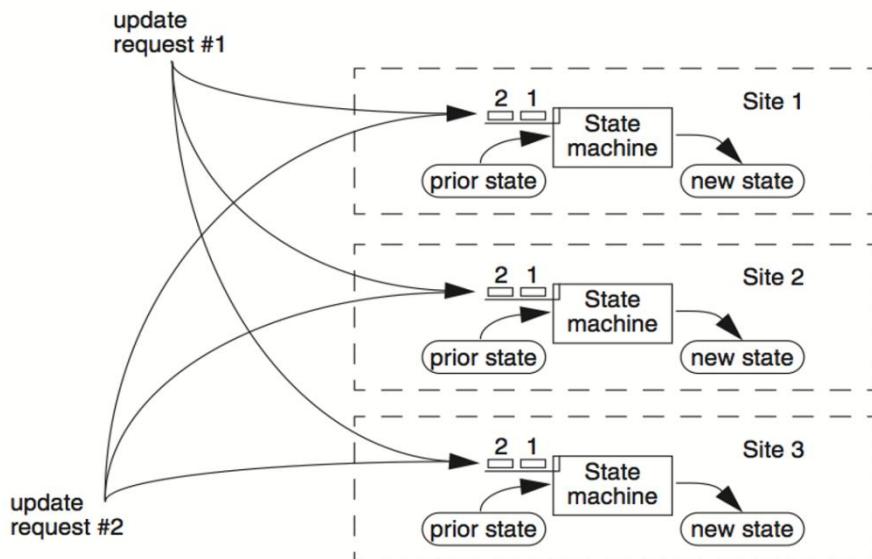
有了 replication 之后，转账以后一台服务器看不见怎么办？这就有两个大的思路，optimistic 和 pessimistic。

乐观的情况：允许出现 out-of-sync 的情况，让人和机器去判定，简单的方法就是时间戳，但是带来的问题就是怎么去维护机器间的时间戳，那么我们可以用一个 vector 去维护不同机器间的时间戳，如果存在偏序关系，那么我们可以直接覆盖；如果不存在偏序关系的情况，就产生了 conflict，需要我们手动 merge 的情况。

悲观的情况：我们不允许出现任何 conflict 的情况，single-copy（对外产生的效果和一个 copy 是一样的。）我们想了一个办法叫做 Quorum，每次写的时候都等到有若干个请求返回才成功。我们读写都是 n 次，但是我们等到对应数量的请求返回即认为 ACK 了。比如我们写了三台，其他两台没有回复，我们也继续往下走。我们只需要保证 $Q_r + Q_w > \text{replica}$ 即可，如果一个服务器告诉我们值是 2，两个服务器告诉我们值是 3，怎么办？比如原来的值是 3，我们要写 2 进去。读 3 个的情况我们读到了“2,3,3”，我们不会认为是 3。这和投票没有关系，必须要三个相同的情况下，我们才能认为读到了正确的数据。因为机器会 recovery，所以 3 最终是会变成 2 的，所以我们的 client 需要去读它，直到读到的数据符合条件。

Quorum 只能保证 consistency，不能保证 atomicity。在这个 case 中，它用到了一个分布式系统中的黄金原则“majority have the choose”。当 replica 不一致的时候，就要了今天的 RSM。

RSM（Replicated State Machine）：如果机器开始是一样的，输入、输入的顺序都是一样的，操作都是 deterministic 的，那么最终结果是一样的。



我们通过网络发过去请求，我们认为输入是可以一样的。但是顺序很难保证。

Deterministic 呢？相同的代码如果里面有锁、CPU 多核调度等都会影响 deterministic。所以确保 the same order 是根植于分布式的问题。

既然它问题的根源在分布式，我们可以改为一台机器 coordinator-worker 的形式，我们想保证 input 的 order，我们不需要让一台机器的 coordinator 去执行这个 input，它只被用来给 input 打上 order 标记。

RSM: Replicated State Machines

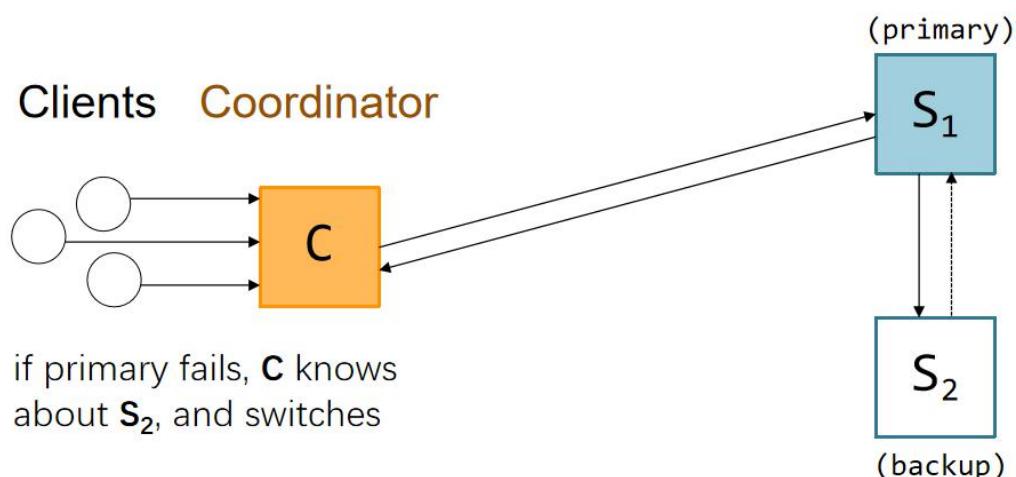
RSMs provide single-copy consistency

- Operations complete as if there is a single copy of the data
- Though internally there are replicas

RSMs use a primary-backup mechanism for replication

- Using **view server** to ensure only one replica acts as the primary
- It can also recruit new backups after servers fail

Primary/Backup Model



attempt: coordinators communicate with primary servers, who communicate with backup servers

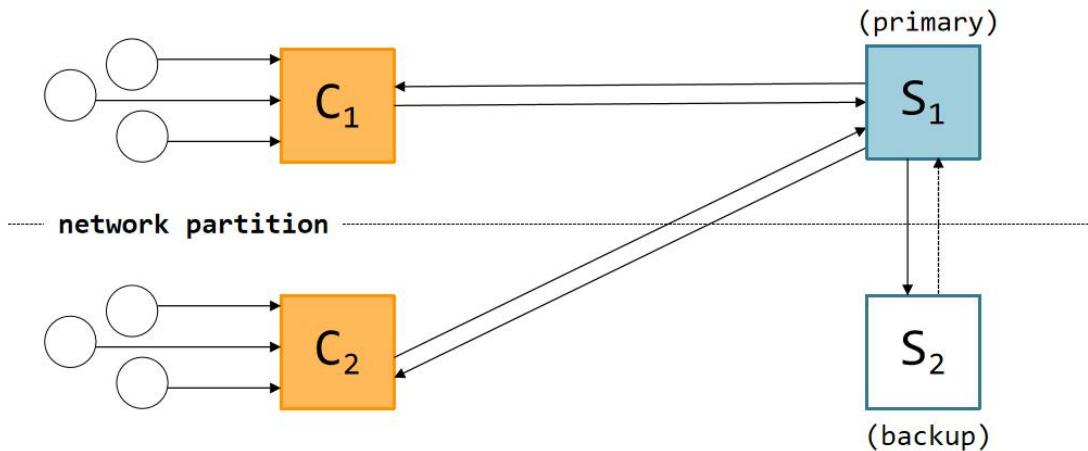
很多 client 会往 coordinator 发消息，coordinator 负责把消息发给 server，由于我们要维护一个 single copy，coordinator 只和 primary server 打交道，primary server 做完操作以后把状态同步给 backup-server。

在这种情况下，我们要考虑的是：

1. primary server crash 了，coordinator 就会把 **S₂** 认为是 primary server，而对于 client 来说，什么都不用改，不用管和谁在沟通。

Coordinator 不只有一个，否则它就会变成瓶颈，所以在实际中会有多个 coordinator，由于 primary server 只有一个，coordinator 都会连到这个 primary。

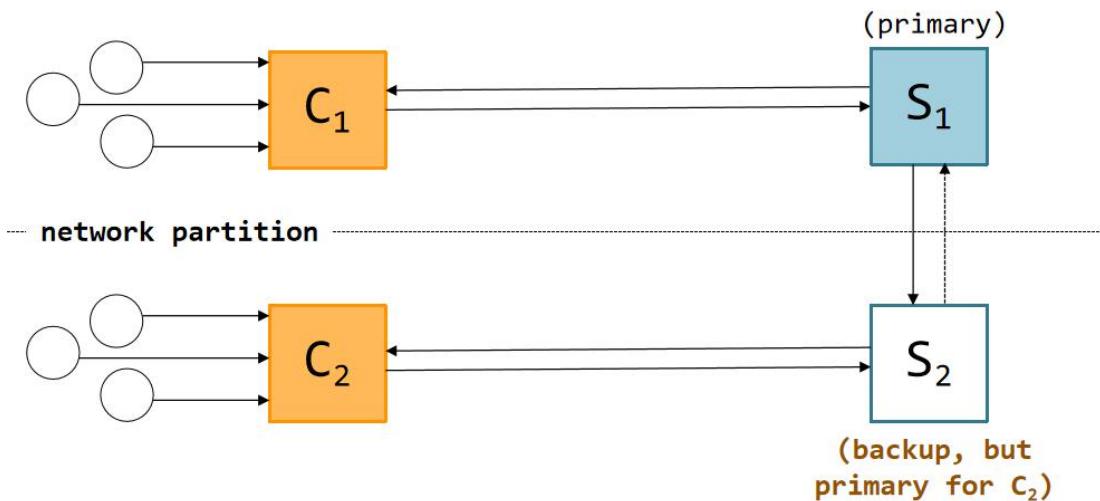
Multiple Coordinators + the Network = Problems



attempt: coordinators communicate with primary servers, who communicate with backup servers

如果出现了一个 network partition, 也就是对于 coordinator2 来说, 它认为 primary server S_1 挂了, coordinator2 要立即把 S_2 变成 primary server。

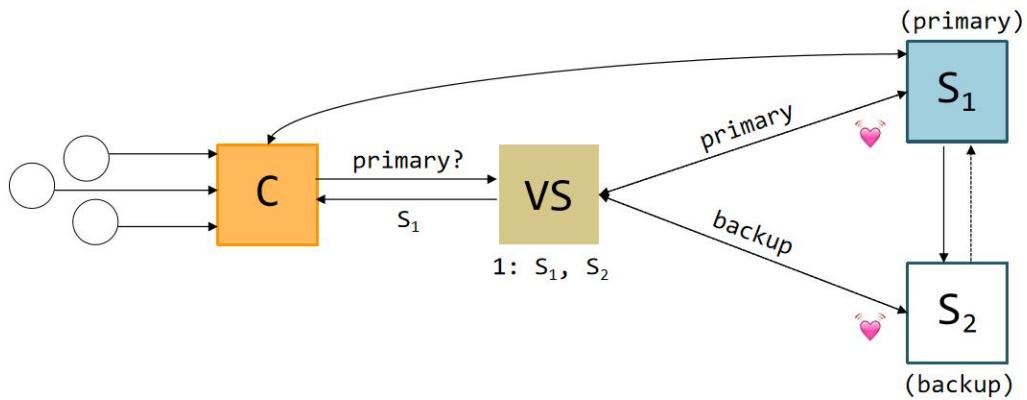
Multiple Coordinators + the Network = Problems



C_1 and C_2 are using different primaries;
 S_1 and S_2 are no longer **consistent**

此时 S_1 和 S_2 之间一定数据不一致了, S_1 和 S_2 信息不一样并且不能覆盖。要解决这个, 我们就要加一个 view server, 它记录谁是 primary, 谁是 backup。View server 会通知服务, 你是 backup 还是 primary。View server 只有一个, 来避免不一致的情况。

View Server



Use a **view server**, which determines which replica is the primary

Coordinator 开始问 view server 现在谁是 primary，回答是 S₁，那么 coordinator 就直接和 S₁打交道。S₁ 和 S₂ 定期地向 View server 发送心跳。这个 view server 只有一台机器，也会面临各种各样的问题。

1. S₁ 挂了，view server 立即发现 S₁ 的心跳没了。那么 view server 任命 S₂ 变成 primary。然后 coordinator 间的时候，返回 S₂，这是 view server 的正常容错。在 S₂ 只要自己是 primary 之前，它会 reject 来自 client 和 coordinator 的任何请求，为了防止两个 primary 同时出现的情况。

2. Network partition failure, S₁ 其实没挂，但是心跳没有到 view server，所以 view server 误以为 primary server S₁ 挂了。在这种情况下，它就只能认为 S₁ 挂了，它会把 S₂ 设为 primary。它设置 primary S₂ 的消息，还在 fly 的时候。Coordinator 给 S₁ 发送消息，它还会继续往下执行，还会和 S₂ 同步，并且 S₂ 同步成功。在还在 fly 的时候，coordinator 知道 primary S₂ 了，还是因为任命状还没到 S₂ 会 reject 掉 coordinator 的请求。任命状到 S₂ 以后，此时 S₁ 和 S₂ 都是 primary。虽然此时 coordinator 最新的请求都会发到 S₂，如果 coordinator 没收到新的 S₂，还是给 S₁ 发了就会导致不一致。

所以我们现在加一条新的规则，需要等到其他 backup server 发送 ACK 给 primary server 才继续执行操作。S₂ 因为当前没有 backup，因为 S₁ 还不知道自己不是 primary，所以 S₂ 在这个短暂停时间内是没有 backup 的，这种情况也是允许的。所以此时 S₂ 作为 primary server，它知道当前自己没有小弟（backup-server），所以 S₂ 不需要收到别人的 ACK 就可以继续执行别的操作。

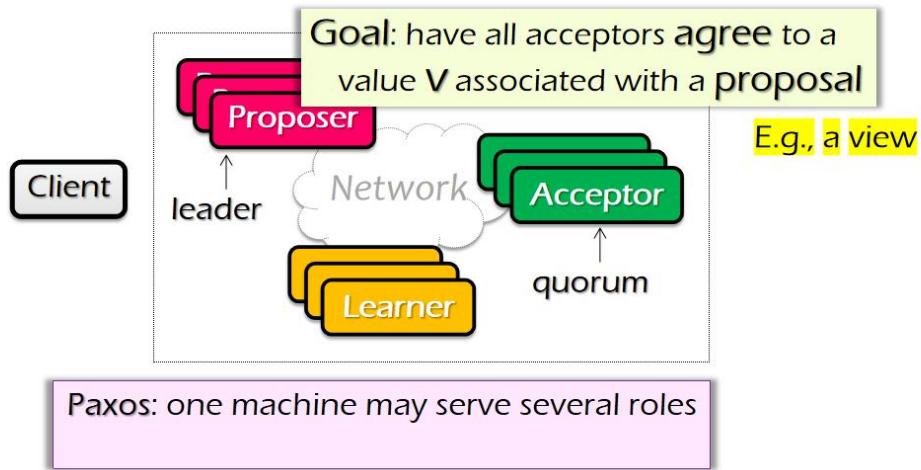
这里面最大的薄弱的点就是 VS，于是我们就需要多台 VS，但是我们为了解决数据的一致性，我们才希望 view server 是一台，现在多台又不能保证一致性了，这时候我们就引入了 paxos。

Viewserver 大家已经用过了， etcd

PAXOS

Paxos' properties: correct + fault-tolerance

- *No guaranteed termination (i.e., lack of availability guarantee)*

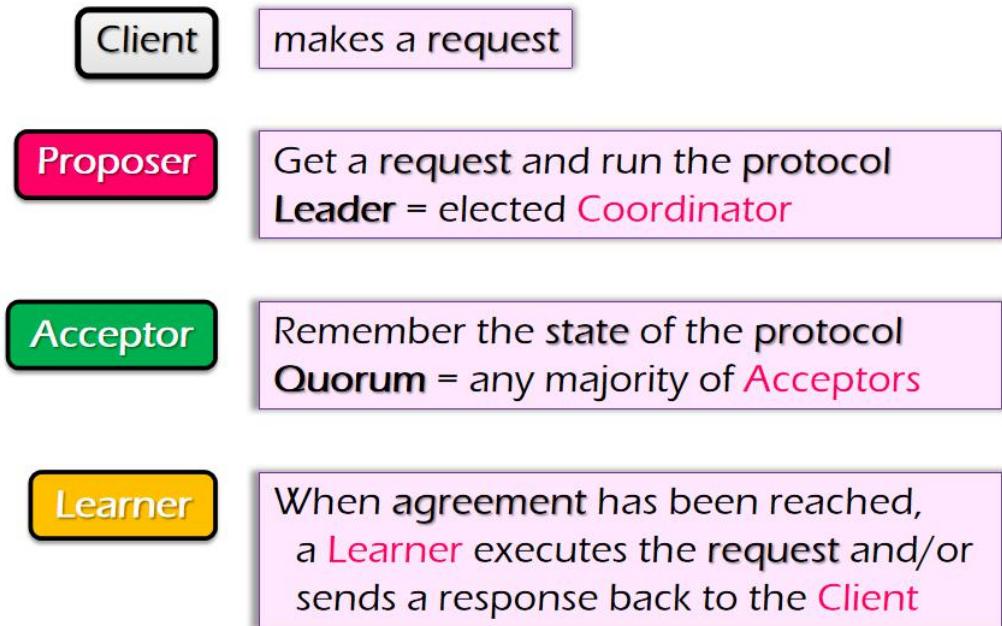


Paxos 要保证是数据是对的，所有参与的 viewserver 要讨论出一个一致的结果。第二个目标就是 fault-tolerance，这个系统中不依赖某个节点挂了与否都可以正常运行。

PAXOS 不能保证 termination，也就是它可能不能讨论出最终的结果，如果讨论不出结果，那就再讨论一次，但是在实际实践中，还是可以保证 termination 的。

分成若干个角色，有 Proposer: 提议把 S2 变成 primary，acceptor 说同意，而 learner 就是学习最新的节点。一台机器在不同 round 里面可能扮演不同的角色。Paxos 最难的是不同轮可能嵌套在一起，因为每个人通过网络连接，所以它直接导致了不同轮可能嵌套在一起。这是 Paxos 很多人吐槽太难懂了的情况。

目标: 所有的 acceptor 同意一个 proposal 的 value 是 v(proposal: 2:S2)，比如 S2 是 primary server。



我们直接看这个例子，整个过程首先有一个 proposer，它想成为一个 leader。Leader 会 proposal 一个 value v 发给所有人，收集所有人是不是 agree，如果 major 的 acceptor agree 以后就宣布 result。

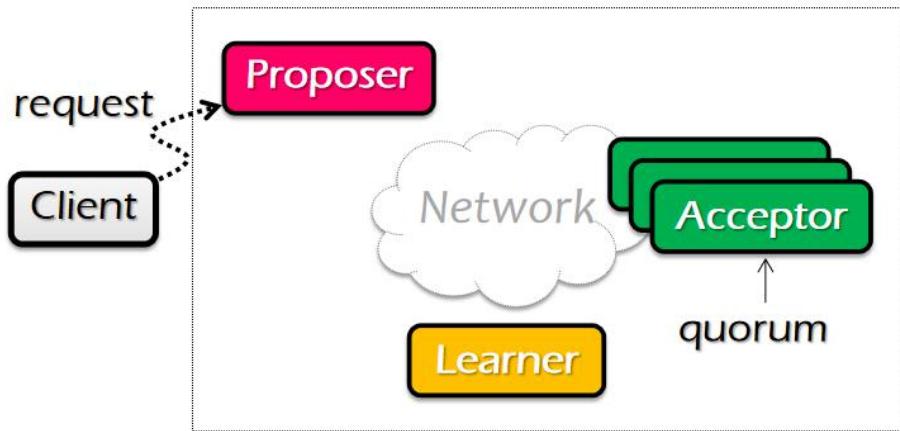
2-phase-commit 要收集所有人的 commit 才 commit，而这里 leader 只需要收集到 majority 的 acceptor 的 agree，就可以 announce。如果不是 majority，五个机器说等于 v1 和五个机器说等于 v2，那就不好解决。

如果有多余一个人希望变成 leader 怎么办呢？我们需要让所有人同意你变成 leader，这本身就是我们需要解决的问题。所以最终一个屋子里乱糟糟的，很多地方都在 propose 出新的 proposal。并且数据包也会丢，leader 也 crash 了怎么办呢？如果一个 leader 收集到 majority，刚想和大家说自己是 leader，但是自己挂了怎么办？

为什么叫 PAXOS，这是一个希腊的小岛，上面住了很多人选 leader，有一些人出来 proposal，大家就骑着小车送消息。它是有 round 的，轮和轮之间是异步的，第二轮不需要等第一轮结束才开始，因为大家不知道第一轮什么时候算结束，每一轮又分成若干个异步的阶段。

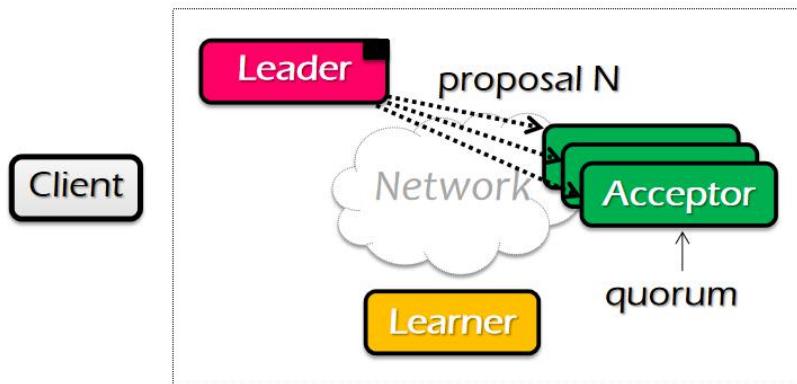
Paxos in Action: Phase 0

Client sends a request to a proposer



Leader creates a proposal **N** and send to quorum

N is greater than **any** previous
proposal number seen by this proposer



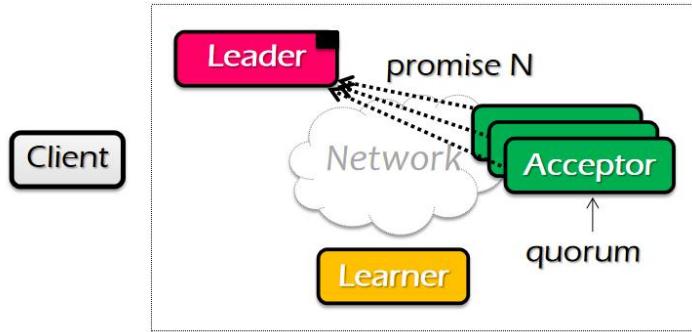
Client 让 proposer 开始, 然后有一个 proposer 变成了 leader, 它需要说“我就是 leader”, 不需要等别人同意, 他可以创建一个 proposal 叫做 N, N 的特点就是比它见过的所有 proposal 都要大。然后就把 proposal N 发给其他人。

Paxos in Action: Phase 1b (Prepare)

Acceptor: if proposal ID > any previous proposal

1. reply with the highest past proposal number and value
2. promise to ignore all IDs < N

else ignore (proposal is rejected)



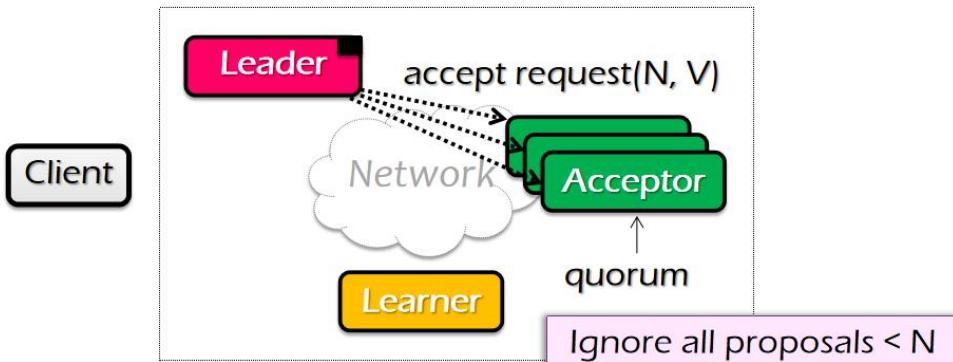
对于 acceptor，如果收到的 proposal 大于它见过的所有 proposal，那么它就把之前见过的最大的 proposal 的记录发给 Leader。比如它现在收到的是 5 号，而之前收到的是 4 号+value 的，它就会把 4 号+value 发给 leader，它会承诺未来忽略掉接收到的所有小于 N 的这些 proposal。

所以 acceptor 就做这个事情。

Paxos in Action: Phase 2a (Accept)

Leader: if receive enough promise

1. set a value V to the proposal N can be decided value or new value
2. send accept request to quorum with the chosen value V



如果 leader 收到了足够多 (majority) 的 promise，它就会选择一个 value V 发给所有的 promise 给它的 acceptor 的节点。在这里需要注意的是，如果收到的所有 promise 里面是没有 value 的，那就自己挑一个新的 v。如果带了 value，那么它就会直接选这个 value。

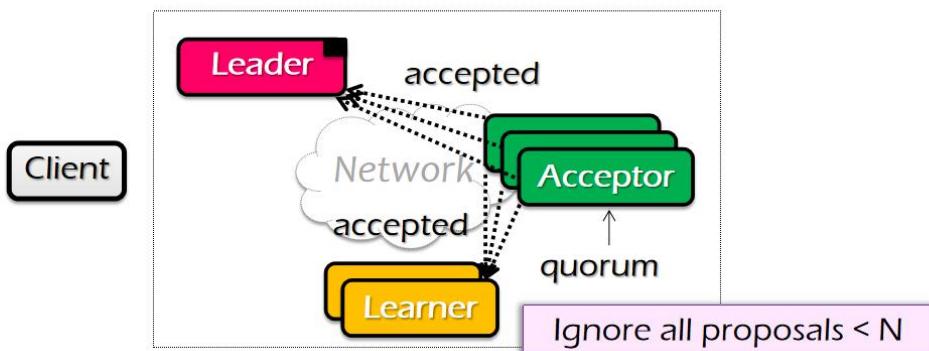
Paxos in Action: Phase 2b (Accept)

Acceptor: if the promise still holds

1. register the value V

2. send accepted message to Proposer/Learners

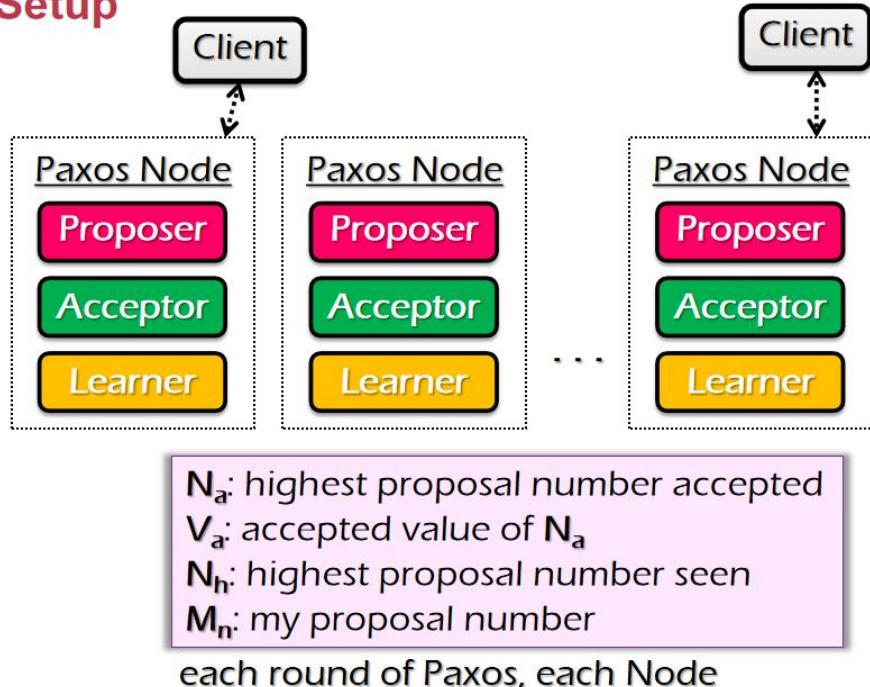
else ignore the message



Acceptor 收到这个新的消息之后,如果 promise 依然是,那么它就记录最新的 N 和 value,记下来以后,就给 leader 发一个 accept。记下来一个这个 leader 就定下来了,这个 learner 就可以回复给 client 我们定下来的 view server 是谁。

在整个 paxos 的节点中,每个节点都可以同时充当多个身份,需要维护一些数据。

Paxos Setup



N_a 和 V_a 就是接收到的最大的 n 和 v 是多少,还有见过的最大的 n 是多少,还有就是自己的 proposal number 是多少。

Paxos Pseudo-code

N_a : highest proposal number accepted
 V_a : accepted value of N_a
 N_h : highest proposal number seen
 M_n : my proposal number

A node decides to be Leader

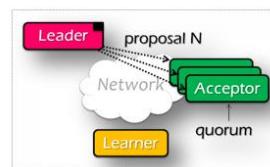
Leader chooses $M_n > N_h$

Leader sends $\langle \text{proposal}, M_n \rangle$ to all nodes

Acceptor receives $\langle \text{proposal}, N \rangle$

```
if  $N < N_h$ 
    reply <promise-reject>
else
     $N_h = N$ 
    reply <promise-ok,  $N_a, V_a$ >
```

Ignore all proposals $< N_h$



它选择一个比见过的所有 proposal 都大的 proposal number 发给其他所有节点。对于 acceptor 来说，如果比见过的所有 proposal number 都大，那么就返回 $\langle N_a, V_a \rangle$

Paxos Pseudo-code

N_a : highest proposal number accepted
V_a : accepted value of N_a
N_h : highest proposal number seen
M_n : my proposal number

If Leader gets promise-ok from a majority

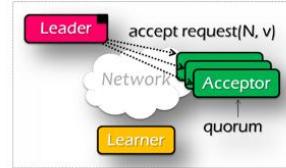
```
if  $V \neq \text{null}$ ,  $V =$  the value of the highest  $N_a$  received
if  $V = \text{null}$ , then Leader can pick any  $V$ 
send  $\langle \text{accept}, M_n, V \rangle$  to all nodes
```

If Leader fails to get majority promise-ok

delay and restart Paxos

Upon receiving $\langle \text{accept}, N, V \rangle$

```
if  $N < N_h$ 
    reply  $\langle \text{accept-reject} \rangle$ 
else
     $N_a = N$ ;  $V_a = V$ ;  $N_h = N$ ;
    reply  $\langle \text{accept-ok} \rangle$ 
```



If Leader gets accept-ok from a majority

send $\langle \text{decide}, V_a \rangle$ to all nodes

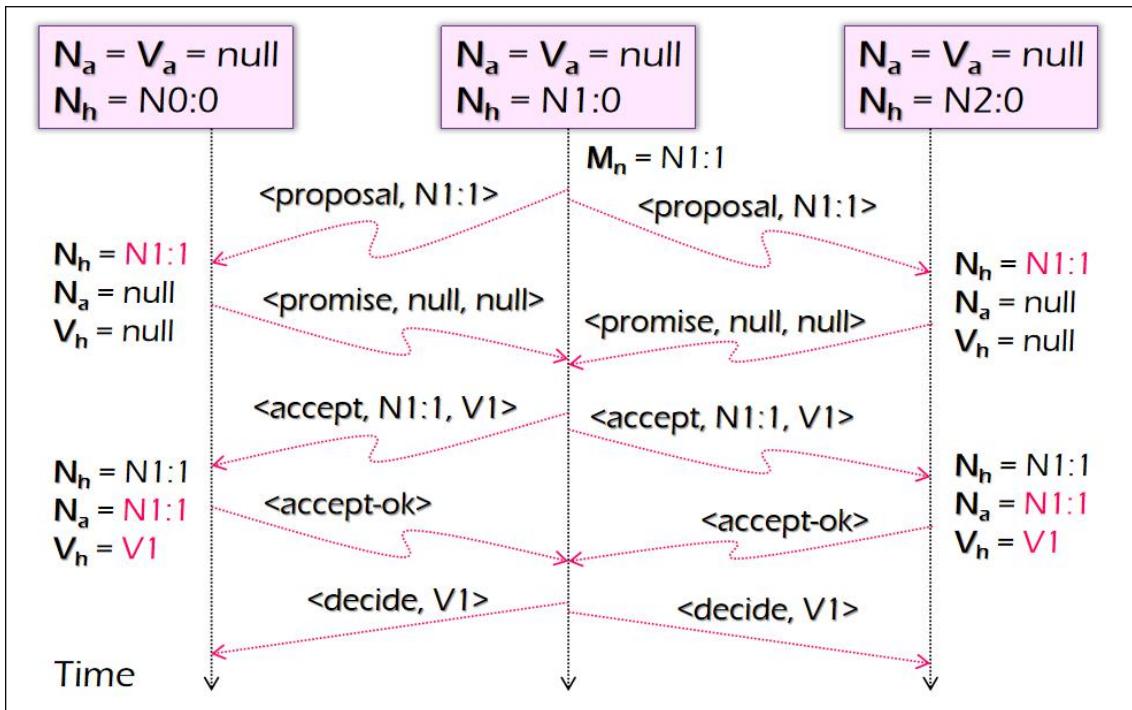
If Leader fails to get majority accept-ok

delay and restart Paxos

如果 V 不等于 null，那么就设置为最大的 N_a 对应的 V_a 。然后发给所有 acceptor。对于 acceptor，如果有大的，那么就拒绝，否则就直接设置成 leader 发过来的值。

如果 leader 收不到 majority，那么就要等一轮期待下轮能够收到。

一次没有出现任何情况的 paxos



我们为什么要多个 acceptor 呢？如果只有一个，acceptor 挂了就会影响整体，为什么不接受第一个 proposal，因为多个 leader 同时在，有没有可能两个 leader 同时见到 majority 呢？

A 想要成为 proposer，它发起 proposal N=1，同时发给 A、B、C，发给 B 的消息丢了，那么 A 就收到 2 个 promise，然后把<accept, 1, foo>发给了两个人，然后 A 挂了，B 又活了，然后 B 挑了一个 proposal N=2，此时因为 A 挂了，B 只能收到 B 和 C 的 promise。于是 B 就会继续发送<accept, 2, foo>，而 B 不能自说自话发送<accept, 2, bar>。

这样设计的好处就是如果 B 再挂了出现一个 D，那么后面的 leader 不管你是谁，都得到的是 foo。这样 foo 是在第一个人发 foo 到 C 的时候就决定了。

如果现在有 7 个节点，A 收到了 51% 的节点的 promise，但是在发 accept 的时候只发到了 C 就挂了，另外的节点都没有拿到这个 foo。另外五个人可是 majority，然后 C 也挂了。此时就没有人知道 foo 了，于是另外五个人可以决定 value=bar。我们现在的问题是，什么时刻 foo 成为了最后结果？如果有 7 个节点，C 收到 foo 是不够的，而只有当 majority 收到 <accept, 1, foo>的时候，才是真正的一个 commit point，后面的任何人的 proposal 里，收到的 promise 里的 majority 里一定有一个 foo。换句话说，上一轮 accept 的 majority 和下一轮 promise 的 majority 一定有一个交集 foo。

五个人里面大家又 proposal 一个 bar，收到了 4 个 promise。有人 proposal bar，刚刚有一个人收到了 bar，此时 A 和 C 又活了。此时 C 记录了同意过 foo，B 同意过 Bar，此时 E propose一个新的 N。此时 E 收集到了一些 null, foo, bar，他就要选最大的见过的，也就是 bar。此时这个值并没有确定下来，需要把<accept, E, bar>发给所有人，然后等 majority 收到才真正是 commit point。所以 paxos 里面 commit point 是所有人决定的，commit point 是 majority 收到了 accept 并且记录到 log 中。所以是没有人知道 commit point 在哪里的。这就是 paxos 和其他的不一样的地方，正因为没有人知道，所以它并不依赖于某一个节点。

有了 paxos，我们就可以保证一旦返回给了一个 client 一个值，那么一定是 majority 同意的情况，这就可以实现之前的 RSM 的情况。

Paxos for RSM

Fault-tolerant RSM requires consistent replica view

- View: <primary, backups> (e.g., <node1, node2>)

All active nodes must agree on the sequence of view changes

- <vid-1, primary, backups>, ...

Use Paxos to agree on the <primary, backups>

- Each Paxos instance agree to a single view
(e.g., <2, Node1, Node2>)

Paxos itself can also be used to implement RSM

- E.g., agree on the state
- Usually inefficient: e.g., two RTTs to agree on a value

在实践中 Paxos 实现比较困难，后来就有人提出了 raft 算法。

Raft Decomposition

1. Leader election

- Select one server to act as leader
- Detect crashes, choose new leader

2. Log replication(normal operation)

- (Only) Leader accepts commands from clients, appends to its log
- Leader replicates its log to other servers (overwrites inconsistencies)

3. Safety

- Keep logs consistent
- Only servers with up-to-date logs can become leader

2021/11/9

Paxos 算法回顾

Paxos 算法是用来解决新问题的一种方法，它是一个通用的协议，但是比较繁琐、很难理解，复杂性较高。简单回顾一下 paxos，一个 paxos 集群是一个小岛，我们有一个 client 想知道这个小岛上选举出来了什么，也就是 accept 了什么东西。client 通常来说是去问 learner 的。

课后大家问的问题：大部分时间都是 **proposer** 和 **acceptor** 之间在做交互，为什么又有个 **learner** 呢？

learner 其实就是来回应 **client** 的人，**client** 可以随便问一个 **learner** 当前的 **proposer** 的 n 为多少。**learner** 就问一圈，问到大多数的人回复的都是同一个答案，那么就返回给 **client**。所以 **learner** 本身是比较简单的，但是放到这里平添了复杂性，这也就是为什么在 **raft** 协议中没有 **learner** 的理由。在 **Raft** 中，只有 **proposer** 和 **acceptor**。

然后 **Paxos** 算法的核心是 **proposer** 和 **acceptor** 之间的交互，有很多 **round**，并且每个人可以扮演多个角色，所以每个 **proposer** 做的第一件事请就是把 **propose** 的东西发给自己，自己给自己回复 **accept**。目标就是同意一个值 v ，这个 v 第一次 **promise** 的时候是没有的，一直要 **proposer** 收到了半数 **promise** 之后，才会真的 **propose** 一个新的值。并且就算它 **propose** 了一个新的值 v ，也不代表这个值最后真的会被接收，因为：

1. 这个 **proposer** 可能还没发出 v 就 **crash** 了，
2. 发的人还没有到达半数就 **crash** 了。

结论：如果我们存在一个时刻，一个 **proposer** 已经把 **proposal** v 发给了超过半数的人后，超过半数的人返回了 **accept**，那么这个时刻 v 就真的确立下来了。

解释：哪怕现在这个 **proposer** 挂掉，又来一个新的人 **propose**，一定至少有一个人会告诉他曾经同一个某个值，因为第二个人问一圈 **promise** 的时候，如果要超过半数，那么势必第二个人所问的 **promise** 的集合和第一个人的 **accepted** 集合相交不为空。那么这个交集上的至少一个人就会把第一个人 **propose** 的 v 告诉给第二个人，这样第二个人就知道了曾经第一个 **propose** 了 v 并且拿到了半数的 **accept**，于是它就选择 v 作为自己的 **proposal**，而不是新出来一个值。就算第二个人也 **crash** 了，第三个人相同地，也会得知这个 v 。所以这个 v 被确认的时刻就是超过半数的人向第一个人发送 **accept** 的时刻，但是这个时刻本身是不可被观察的，因为我们不是上帝视角，我们不能同时观察所有节点，我们只能在一个节点的角度观察自己发生了什么事情。

Q: When is the value V chosen?

A: 虽然正确的情况下应该是大多数 **acceptor** 接收的那个时刻，但是因为那个时刻是不可知的，我们推迟到 **leader** 收到大部分的人回复 **accept** 的时刻，这是虽然不太准确但是可被观察的时刻。

- 当 **acceptor** **crash** 的时候，他就要记住见过的最大的 N_h ，当重启之后，它就要去判断新的 **proposer** 是否大于这个 N_h 。
- 如果一个 **acceptor** **accept** 了以后 **crash**，他就必须记住见过最大的并且 **accept** 的 N_a, V_a 。
- 如果一个 **leader** **crash** 了，那么有新的 **leader** 会出来 **propose** 一个新的 v 。

Raft 协议

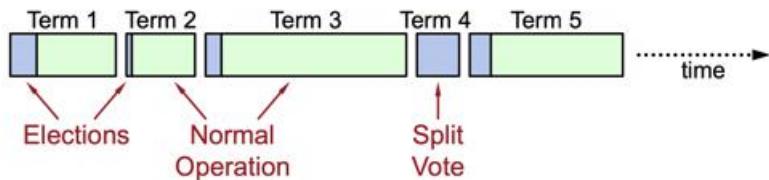
Raft 协议其实和 **Paxos** 很接近，但是简化了一些步骤，首先是删掉了 **learner** 这个 **role**，那么 **client** 去问谁呢？就直接去问那个 **leader**。第二个简化是在 **Paxos** 有一个 **proposer**，带

有 N 和 V，会比较迷惑，Raft 中，大家只需要同意一个 log，我们把所有东西记录在 log 里，log 里新加了一项，最终同意 leader 的，不同意非 leader 的即可。

Raft 说的是所有节点最终要实现的是 RSM，replicated state machine，也就是有一个 input 过来之后，我们要给三个机器，order 是一样的，那么 output 都是一样的。我们怎么保证 input 和 order 是一致的呢？我们写在 log 里就行了，谁先写谁在前面，大家同意这个 log，自然就同意了 input 和 order，所以把问题变成了同意 log 一致性的问题。因为同一时间 leader 只有一个并且只有 leader 能写 log，这就避免了多个人写 log 导致不一致的问题。

那么 Raft 协议到底怎么做呢？

1. leader，如果 crash 了，就选一个新的。Leader 写 log 并且发给其他 server，它能保证所有人看到的 log 是一样的，只有拥有最新 log 的 server 才能变成 server。比如 leader 在复制最新 log 给别人的时候 crash 了，导致一部分拥有最新的 log，而另一部分拥有相对旧一点的 log。这个协议可以保证只有拥有最新的 log 的那部分人才可以竞选下一轮的 leader。要保证 safety，我们选举出来一个拥有最新 log 的 leader，大家都听 leader 的话就行了，所以最难的是那个选举。



首先就是大家选一个 term，其中蓝色部分就是选举的过程，而绿色部分就是执行的过程。如果发生的一次 crash，就重新选举。一轮轮选举+执行，如果有一次没选出来，那么就再选一次。选 leader 的过程就和 Paxos 协议中的 propose、promise 比较像了。

数据库

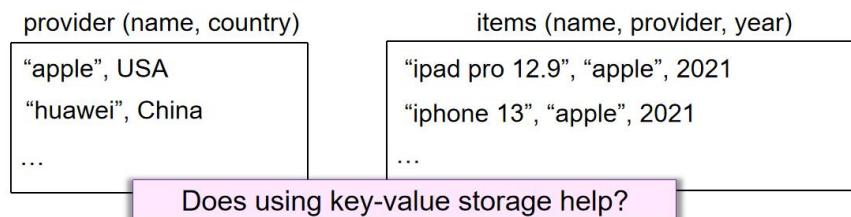
我们想知道数据库和 KV Store 到底有什么区别和特点。但是数据库是有它的局限性的，一个比较大的就是 scalability，acid 的束缚使得数据库曾经在扩展过程中遇到很多瓶颈，但是现在已经做了很多改进了。这是因为 SQL 具有一系列的特点，虽然安全性和稳定性不错，但是扩展性上不去。所以出现了 NOSQL。最早就是觉得 SQL 太复杂了，就用简化的 KV Store 和 document 来存储。

大家写过 web application，基本上都有一个 database，其实早期的数据库，最早用在银行里记录转账、利息。航空公司记录谁买了票、谁退了票。到今天的话，电子商务主要用到了数据库，一些重要的信息，比如商品价格就存在了数据库中。我们已经学过 kv-store 和 filesystem 之后，我们为什么还需要一个 database，核心在于我们希望对真实世界的数据 modeling，当我们要支持一个简单的购物网站的时候，我们应该如何去存储这些 item 和 provider 的信息呢？同样，我们还需要存储 item 和 provider 之间的一些信息，比如某个设备是由哪个供货商提供的。

Naïve database: using flat file

Store our database as **comma-separated value (CSV)**

- Use a separate file per entity
- How to find the item named “ipad pro”?
 - The application must **parse the files** each time they want to read/update records



一种最简单的方式就是使用文件，比如 csv 文件，中间使用 **comma** 分割的一些 **value**，我们就可以用一个单独的文件（相当于数据库中的表）。对于这样的一个文件来说，我们怎么去找到“ipad pro”这样的一个 item 呢？

首先，我们要 **parse** 这个文件，然后把对应的 **record** 读出来并去更新。这和我们学过的 **kv-store** 很像，我们完全可以把 csv 文件变成 **key** 和 **value**，要找的时候直接根据 **key** 找就行了，我们学过的 **kvstore** 在这种场景下是比较简单的。

那么如果我们想找所有发布在 2021 年的设备呢？那么此时 **kvstore** 的 **get** 和 **put** 就不够了，我们需要遍历整张表才能找到，开销很大，并且用户需要知道存储的数据结构长什么样子。

除了查找，在插入的时候，我们也要考虑数据对不对。

More questions of using flat file: data integrity

How do we ensure that the providers in items are valid?

- E.g., Are Apple Inc. and apple Inc. the same provider?

What if somebody set the album year with an invalid string?

- E.g., 2049

What if there are **multiple providers** for an item?

- Apple X Intel

What happens if we delete a provider?

- All its related items should also be deleted

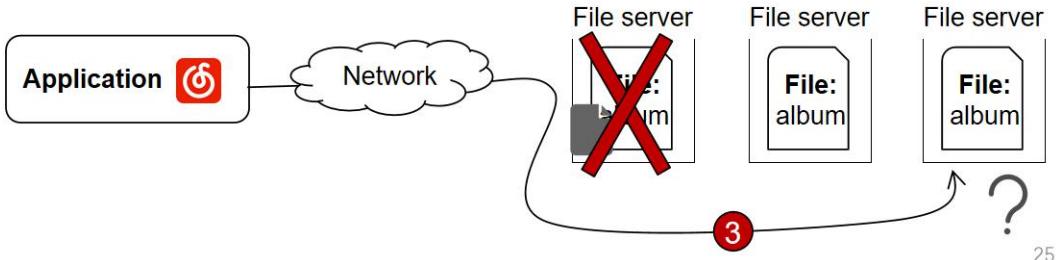
Does using key-value storage help?

单纯使用 **kv-store** 是很难检查上述的错误的和解决上面的问题的。所以 **kv-store** 是不够的。并且，当做成一个大的数据中心之后，我们势必要考虑如果机器 **crash** 了怎么办。并且为了高的可用性（**availability**），我们需要对数据做 **replica**，那么又涉及到了数据的一致性（**consistency**）的问题。

回顾：一致性问题

– **Consistency problem**, one replica gets updated while others are not

1. Update the albums on the first machine
2. The first machine crashes
3. What if we query the third machine?



25

像这样，文件写在了第一台机器上之后，第一台机器 crash 了，这种问题怎么解决？

1. **quorum** 机制，写的时候写两个，读的时候也读两个，这样我们就可以防御一台机器 crash 的情况。
2. 写文件的时候，先写两个备份，再返回写完的状态。每次返回给 **client** 的时候，就一定是三个备份上都有了这个文件。

所以一旦讲到 **database** 之后，**ACID** 就呼之欲出了。像 **transaction** 这种抽象依然使用，所以我们可以使用之前我们学过的 **transaction**、**fault-tolerance** 方法去实现。这就是早期的数据库的做法，早期的数据库可以认为就是满足 **acid**、支持 **transaction** 的 **key-value store**。但是大家觉得只想做数据查询操作，但是不得不考虑数据在磁盘中是怎么存放的，这导致逻辑层和物理层紧耦合在一起，我们希望解耦，让它不要和底层紧密地耦合在一起。一旦我们实现了解耦，就可以让逻辑层的人负责逻辑层、让物理层的人负责物理层，有了分工，效率就能提高，生产力就可以提高。

这就是 **DBMS** 出现的初衷

Database management system (DBMS)

A DBMS is software that allows applications to **store & analyze** information in a database

A general-purpose DBMS is designed to allow the **definition, creation, querying, update, and administration** of databases

Application developer does not need to consider the above questions

- The database helps resolving these issues
- Developers only need to write the program using **the DBMS data model & query language**

它使得开发者不需要考虑它底层是怎么解决这个问题、怎么管理数据的。本来应用程序要考虑所有数据相关的东西，随着解耦，数据库就诞生了。在 **DBMS** 中，最重要的就是两个部分：

1. **data model**
2. **query language**

我们需要一种语言精确地描述程序员所需要的数据、所需要对数据的操作是怎么样的。

Data Model

Data model: collection of concepts for describing the data in a database

Schema: a description of a particular collection of data, using a given data model

Example data model:

- Relational
- Key-value
- Graph
- Document
- ...

More expressiveness

Though some models are more nature in specific workloads

- E.g., the *graph* data model is more natural for graph workloads aka. **Polyglot**

Key-value store 也是一种 data model, 但是它的表达能力是非常有限的。关系型模型可以作为主体存在, 但是其他模型也会在特定任务上表现更好。

对于所有数据来说的话, 它是一个 **unordered** 的一个集合, 包含了各个 **attribute** 之间的关系, 每一行是 **attribute** 的值的集合, 当我们要表示 n 元关系的时候, 就是一个 n 列的 table。

Use tables and rows to represent relation

Concepts:

- **Relation(table):** an unordered set that contain the relationship of attributes that represent entities
- **Tuple(row):** a set of attribute values in the relation

Tuple

- Values are (normally) atomic/scalar
- Values belong to some **domain** (e.g., int)
- A special value **NULL** for every domain

Item (name, provider, year)

Name	Provider	Year
ipad pro	Apple	2021
iphone	Apple	2021

每个 value 可以指定一个范围。没有值就可以认为是 null。

如果我们现在插入了包含错误时间的产品会怎么样呢?

(ipad pro, Apple, 2049)

- 如果使用 key-value store 的话, 我们就要用如下的代码:

```
inserted_value = ...; // something from user
if inserted_value.year > 2021:
    report an error
kv_item.insert(key, inserted_value);
```

对于一个上层希望不写代码的人来说的话, 这就太复杂了, 如果我们使用 SQL 的话, 我们就可以使用如下代码:

```

CREATE TABLE Item
(
    Provider string NOT NULL ,
    Year int NOT NULL,
    ...,
    CONSTRAINT Year_Ck CHECK
        (Year BETWEEN 1940 AND 2021)
)

```

That is the **declaration**

如果我们要建立数据之间的联系，比如一个 item 属于某个 provider，我们需要先检查是否是一个合法的 provider。

```

inserted_value = ...; // something from user
if kv_provider.get(inserted_value) == NULL:
    report an error
kv_item.insert(key, inserted_value);

```

在 SQL 中，我们就可以直接通过外键绑定的方式，指向 provider name，当我们插入一个错误的 provider name 的时候，SQL 就会报错 “a foreign key fails”

Name	Provider	Year	Name	Country
ipad pro	Apple	2021	Apple	USA
iphone	Apple	2021	Huawei	China

```

CREATE TABLE Item
(
    Provider string NOT NULL ,...,
    FOREIGN KEY (Provider) REFERENCES Provider(Name)
)

```

外键可以解决手动写代码检查有没有 provider 的情况

如果我们要删除一个 provider 怎么办？就是把一个公司的所有物品删掉。在 key-value store 中实现这个功能，就是做一个遍历，把对应的 key 删掉

```

for k,v in kv_item:
    if v.provider == "Apple":
        kv_item.delete(k)

```

SQL 只需要加上这一行即可

```

CREATE TABLE Item
(
    Provider string NOT NULL ,...,
    FOREIGN KEY (Provider) REFERENCES Provider(Name)
    ON DELETE CASCADE
)

```

而在数据库中的 best practice 是标记为 “已删除”、“已离职”，因为数据太重要了，最好不要删掉。

kv-store 和 database 哪个更好取决于哪个更方便。

Key-value store 是相对简单的，SQL 能表达的数据是它的超集。我们在数据库中可以简单定义一个 primary key，那么整张表就变成了一个 key value store。

Query language

查询语言是比较重要的，比如可以用自然语言，但是自然语言的表达有歧义、不够精确。我们可以用专用的关系代数和集合论中的符号。有一套数学理论作为支撑。

我们先来看一个 query，如果用 key-value store 来做

▶ Example #1: Using SQL to query the data

Example #1: find the item released in 2021

- How to query the data with the key-value store model ?
- What about data query efficiency?
 - Solution: use an **index (e.g., hashtable)**

```
result = []
for k,v in kv_item:
    if v.year == 2021:
        result.append(v)
print(result)
```

Add an index offline

```
for k,v in kv_item:
    kv_item_year[v.year].append(k)

result = []
for k in kv_item_year[2021]:
    result.append(kv_item[k])
print(result)
```

Opt.

为了提高性能，最常见的方法就是加一个 **offline** 的 **index**。对于任何一年都有这个索引，当我们查询的时候就不需要遍历这个 **kv_item** 集合。**index** 建立的越多，就是空间换时间的过程。

如果我们用 SQL 做，只需要如下来写，也就是在整张 table 中运用 **Year = 2021** 这个谓词来做筛选，

Example #1: Using SQL to query the data

Abstraction: select

- Choose a subset of the tuples from a relation that satisfies a selection **predicate**
- Can combine multiple predicates using conjunctions / disjunctions

Syntax: $\sigma_{\text{predicates}}(R)$

- E.g., $\sigma_{\text{year}=2021}(\text{items})$

Example #1: find the item released in 2021

- How about adding an **index**?

```
select * from Item where Year=2021;
CREATE INDEX item_year_index ON
Item(Year);
```

```
mysql> select * from Item where Year=2021;
+-----+-----+-----+
| Name | Provider | Year |
+-----+-----+-----+
| ipad pro | apple | 2021 |
| mackbook 14 | apple | 2021 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

SQL 毫无疑问也是可以加 INDEX 的。

SQL contains three components

Data Manipulation Language (DML)

- E.g., query language based on relational algebra + aggregators **and more!**

```
select * from Item where Year=2021;
```

Data Definition Language (DDL)

- E.g., how to create a table

```
CREATE TABLE Item ( Provider string NOT NULL , Year  
int NOT NULL, ...)
```

Data Control Language (DCL)

- E.g., whether a user has access permissions on a specific table

This course does not intend to teach the languages of SQL.
But is why it is designed this way.

6

有了这三部分之后，我们就定义了数据的动态、静态和权限控制。

SQL 和 transaction 的概念

它们的相似点：两者都实现了 ACID

它们的不同点：没有 SQL 的话，开发者需要手写事务机制

```
...  
tx.begin();  
...  
tx.read();  
...  
tx.commit();  
...
```

Transaction w/o SQL

```
START TRANSACTION  
select * from ...;  
... // other SQL  
COMMIT;  
...
```

Transaction w/ SQL

关系型模型是理论角度去阐述，SQL 是关系型模型的实现，在 SQL 实现中 row 就是 tuple，table 就是 relation

Implementing relational model, actually is non-trivial

Relational model only defines how data is represented in concept

Relational algebra only defines what is the output of a query language

Example

- $\sigma_{b_id=102}(R \bowtie S)$ vs. $(R \bowtie (\sigma_{b_id=102}(S)))$

Will the results be the same? Which one is faster?

- The implementation decides **the query execution order**

- E.g., the role of a query planner (will be introduced in later lectures)

虽然结果是一样的，但是显然先 select 再 join 速度很快。所以 SQL 的写法存在很多优化空间。我们在实现 DBMS 的时候，就要考虑到顺序。

Table schema design of relational model

如果让我们来实现这么一个简历的页面，我们要怎么设计这个表呢？



Bill Gates
Greater Seattle Area | Philanthropy

Summary
Co-chair of the Bill & Melinda Gates Foundation.
Chairman, Microsoft Corporation. Voracious
reader. Avid traveler. Active blogger.

Experience
Co-chair • Bill & Melinda Gates Foundation
2000 – Present
Co-founder, Chairman • Microsoft
1975 – Present

Education
Harvard University
1973 – 1975
Lakeside School, Seattle

Contact Info
Blog: thegatesnotes.com
Twitter: @BillGates

我们可以用非常简单的方法记录。

ID	Name	Area	...
0	Bill Gates	greater seattle area	

...

但是住在什么地方这种事情，如果有很多人坐在同一个地方，最后这个位置改名了，那么需要全部都改。我们可以把 area 用像指针一样的外键来处理。这就是 many to one 问题，本质就是 replication 问题。

Many-to-one: A matter of duplication, and normalizing

Store the string instead of ID is actually a matter of **duplication**

- The same value has appeared in many entities
- E.g., the “greater seattle area” appeared in all the people who lived there

Drawbacks of duplication:

- Consistent style and spelling
- Ambiguity (e.g., several cities with the same name)
- Hard for updating. Suppose the city has changed its name

Normalizing: remove the duplicated entities in various entities

- Useful for many-to-one relationship
- How to normalize is a big topic (out of the scope of this lecture)

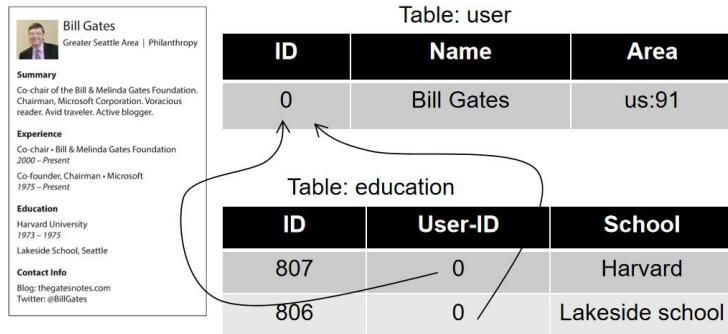
另外一个是 **one-to-many problem**，一个人可以有多个教育背景。我们该怎么设计这个表呢？

1. 把表设计的大一点，高中、大学、**master** 都作为一列，上过就填入对应信息。但是这种方式是把所有的可能都变成了 **column**，浪费空间。

2. 我们应该把教育单独地变成一个表。

Solution: using another table

- And constraint the relationship with foreign key!



Document, 比如 json 格式，解决了 **one-to-many** 问题，也就是 **jsonarray**。好处非常多，locality 非常好，所有信息都在一个文件中。Schema 也很灵活。但是我们要修改地名的话，**many-to-one** 还是有问题。

Summary: relational model vs. document model

Document model is a representative model in NoSQL databases

Benefits of document model

- Better locality
- Easy to represent one-to-many relation (but has duplication problem)
- Schema flexibility

Benefits of relational model

- Join supports
- Support one-to-many without duplication (e.g., with multiple tables)
- Better modeling many-to-one & many-to-many relationships

2021/11/11

今天讨论的是数据库的存储，为什么要用 **kv-store** 来实现 DBMS 呢？当我们把数据库变成分布式的时候，底层如果是 **kvstore** 是比较方便的，因为我们已经学过怎么把 **kv-store** 变成分布式了。

现在很多企业里，分布式数据库都是很重要的组件，比如阿里最近开源的 **oceanbase**。里面是大量的普通计算机，而不是计算机。这就是互联网公式的去 IOE 化(**IBM,ORACLE,EMC**)。就算是 **IOE**，也支撑不了双十一。

数据库包括 SQL，具有高可扩展性的 **TiDB**, **redis**, **OceanBase** 等。这些数据库学好了以后属于我们的核心竞争力。如果我们证明一下 **key-value store** 是没有 bug 的，这才是核心竞争力。**Amazon** 底层的 **S3** 组件就是证明过的，所以就很放心。

要保证 **transaction** 是 **consistency** 的，有 **2-phase-commit** 和 **3-phase-commit**（解决

coordinator)，我们今天有了 Paxos 和 Raft，就不需要原先考虑一些分布式的问题。

上节课提到，如果我们要设计一个 database，DBMS 本质上是对数据的管理，我们需要用一种准确的方式去定义数据是怎么样组织的，然后我们需要用一种非常简单的语言让程序员在 Query 的时候把自己的需求表达出来。这种语言不可以和底层实现紧密地绑定，必须是一种抽象的层次比较高的语言。所以分为了数据模型和 query language。

最终我们说 relational model 最终在 data model 中胜出，我们要注意 SQL 和关系型数据库并没有本质的关系。Data model 除了 relational 还有很多，比如 kvstore, document, graph 等。这些 data model 并没有好坏之分，只分为适用和不适用。对于 document 来说，一对多比较合适，我们还可以使用动态语言特性，哪怕一个字段在别的数据中都没有出现过，我们也可以往里加，但是如果要在关系型数据库中这样做，我们就得添加一列作为 attribute。

同样，对于一些社交网络、社交媒体，包括 PageRank 算法，用 graph model 就比较合适。但是关系型数据库的表达能力足够强，在没有特别要求的情况下，使用关系型数据库可以起到比较好的效果。

在关系型数据库中，每一行就是一个 tuple，每张表就是一个 relation。我们可以在定义的时候添加一些范围使得错误的值能在第一时间排除掉。

Query 包含了 DML,DDL,DCL。（访问、定义如何去做、控制权限）

讲到上节课的内容位置，我们发现 relational model 都是一个逻辑概念，但是如果它听起来很完美，但是放在硬盘上根本做不出来，也没有意义。

Big picture of how a SQL is physically processed

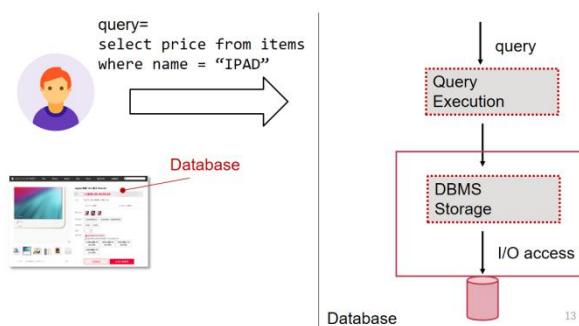


图 2 DMBS 的查询流程

数据最后肯定是存在磁盘上的。Storage 有哪些呢？CPU 寄存器、DRAM,SSD,HDD,网盘。这就有一个 trade-off，容量越大、速度越慢，我们还要考虑断电的问题，我们必须保证数据不能丢，数据库中对于数据 durability 是要求非常高的。我们要把数据放到非易失性的存储中，包括 non-volatile memory、SSD、HDD 等。接下来，如果我们知道数据最终在 SSD 或者硬盘上，这个 tuple 实际存下来的时候长什么样子呢？

我们要把 schema（一个 table 中有哪些 attribute，属于 table 的 metadata）和 tuple（一行行的 item）都保存到持久化的存储中。

我们在保存 DBMS 的数据的时候，需要考虑

- 上层是用什么语言来访问的
- 字符集用的是什么
- int 多大、其他类型多大

有没有让大家回忆起之前我们解决过的，RPC 从一台机器传到另一台机器的问题，两台机器之间对于数据的解释都是不一样的，在传参的时候，需要对数据做 marshall，我们就要充分考虑两边不兼容的问题，我们在存数据到磁盘的时候，也要充分考虑这些不兼容性，比如：little-endian 和 big-endian、大小等问题。

那么 Sun's RPC 中有一些定义，我们现在也有 JSON, Google protocol buffer，这样就定义

了刚才定义的一系列东西，这样我们就知道该怎么解析。

第二，tuple 该怎么表示呢？我们该怎么把 tuple 存到磁盘上呢？一个关系型数据库可以用很简单的方式来保存一个 tuple，我们需要在每个 tuple 上添加一个元数据。



我们第一步要确定 tuple 到底含几个 attribute，table 的 metadata 记录了表格有几项，但是具体到数据的 tuple 中可以有几项为空，所以我们要用一个 bit map 来记录下来。

那么表的 schema 怎么存呢？它在 database 内部也被认为是一张 table。

mysql> desc Item ->;						mysql> select * from Item;		
Field	Type	Null	Key	Default	Extra	Name	Provider	Year
Name	varchar(255)	NO		NULL		ipad pro	apple	2021
Provider	varchar(255)	NO	MUL	NULL		mackbook 14	apple	2021
Year	int	NO	MUL	NULL		mackbook air	apple	2020

左边这张表是在数据库内部的，通过这张表就把 schema 记录下来了。而 schema table 的 schema 都一样，每个数据库都只有一份。

有了 tuple 之后，是怎么保存到磁盘呢？

Recall: we should physically store the tuples on the disk

- However, programming on raw disk is non-trivial

What is the computer's view on disk hardware?

- Is programming this kind of abstraction simple?

Question: which systems we have learned in previous lectures can help?

- Log-structured key-value storage (lecture 09)
- File system (lecture 03-04)



首先程序员不能用 raw disk，我们是否有一种更好的抽象来满足对存储设备的操作呢？我们可以使用 log-structured kv-store 来保存数据。从磁盘角度来看，用文件系统或者 kv-store 是非常自然的事情。

我们先来看 kv-store, interface 如下

Review: key-value storage

Storage layout:

- Each data (**Value**) is opaque to the database
- Indexed by a key (**K**), which itself is also a data (e.g., int, string)
- Stored on disk (tolerate failure & support a large capacity)

Interface

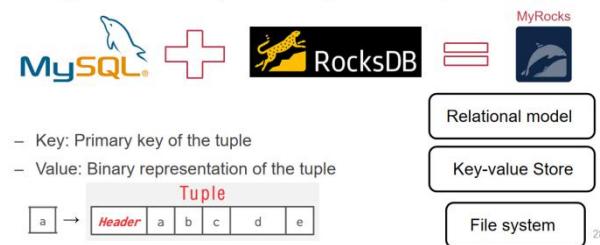
- Get(K) -> V, Scan(K,N)
- Update(K,V), Insert(K,V), Delete(K,V)
- The K and V can be arbitrary byte-sequence (e.g., a JSON file as a value)



Sol-1: Store tuples in log-structured key-value storage

Of course, we can build relational model on key-value stores!

- Example: MySQL can use MyRocks as its storage engine
 - MyRocks uses rockdb, a standard log-structured key-value storage



我们可以用 **a** 的值来找到这个 **tuple**。由于我们用的是 **kv-store**，当时用的是 **LSM-tree**，它尽可能去优化顺序写操作，这样对于磁盘来说可以充分利用顺序写性能高的特点。

LSM-tree 的缺点是：

- Write amplifications
- Space amplifications
- Read amplifications
- Poor range scan performance

所以我们又考虑到了 **B-tree**，它查找过程很快、**range query** 很快，对于大量读写的支持非常好，在 **DBMS** 中也支持 **B-tree**，但不是唯一的一种实现，**tuple** 可以完全没有 **index**。只保存 **B** 树的叶子结点的话，就需要全部扫描一遍磁盘。我们可以主动创建索引，这样 **INDEX** 就以中间节点的形式保存在磁盘上。

```
CREATE INDEX item_year_index ON Item(Year);
```

DBMS 中 **page** 就是 **B-tree** 中最小的单位。数据库默认为每一张表创建主键索引。我们到此为止已经知道了数据库中的每一行存在 **tuple** 里，而 **schema** 也存在特殊的表中。数据可以使用 **kv-store**、基于 **page** 的方法（**page** 在磁盘上组成了 **B-tree**）。

DBMS 中的 Page

接下来的问题就是，**database** 的 **page** 到底是什么东西。**Page** 就是一个 **fixed-size** 的数据

库，一个 page 可以包括多个 tuple 或者 metadata (B-tree 的中间节点)。它是最基本的存储数据的抽象，DBMS 有一层间接层，把 page 的 id 映射到物理的 id 上。一个磁盘有 2^{30} 个 block，数据库只看得到 page，所以需要把 page number 映射到磁盘。

给定一个 4T 的硬盘，该怎么样创建 database 呢？

方案一：创建一个 4T 的文件，在其中分块

方案二：它可以拆成 100 个 40G 的文件，对于每个文件，可以保存 100M 个 page。当 DBMS 要找第 340 万个 page 的时候，只需要去第三个文件中找就行了。

DBMS 可以比较方便地去查找数据。

上节课我们讲到了 tuple，每一行都有一个 header 的 bitmap，每个 schema 也作为 tuple 保存。

在硬件磁盘上，我们已经提到过 page 了，但是我们叫做磁盘上的 section (512B, 4K)；对于 OS，内存是 4K, 2M, 1G。对于数据库来说，page 就是 512B 到 16KB。到底 4K, 8K, 16K 怎么选择呢？



把 pagesize 变大的好处是索引 B-树会变矮、找的次数更少；坏处就是大的 page 没存满可能就浪费了空间(内部碎片)，另一个坏处是硬盘一次写保证的 all-or-nothing 是一个 sector (硬盘上的电容保证)。在这种情况下，如果我们的数据库不能利用硬件原子写的特点的话，我们就需要额外的机制来保证 atomicity。

How to manage pages on the disk?

Should provide methods to:

- Create a page
- Get a page
- Write a page
- Delete a page
- Iterate over all the pages
 - E.g., when scanning all tuples in a database

Shall the DBMS manage the disk like a disk driver?

问题在于，如果把 page 看出磁盘的一个个 block，DBMS 是不是要想磁盘驱动一样去操作磁盘呢？但是这会导致紧耦合。我们可以用文件系统，把 page 存放在文件中，这样我们就可以复用文件系统的 open/read/write 的接口。那么 page 和文件该怎么映射呢？每个文件放一个 page？元数据太多。

在数据库中 heap file 就是保存 pages 的文件，对于数据库来说，它们完全看不到磁盘，heap file 可以认为是磁盘的抽象。

Database heap file

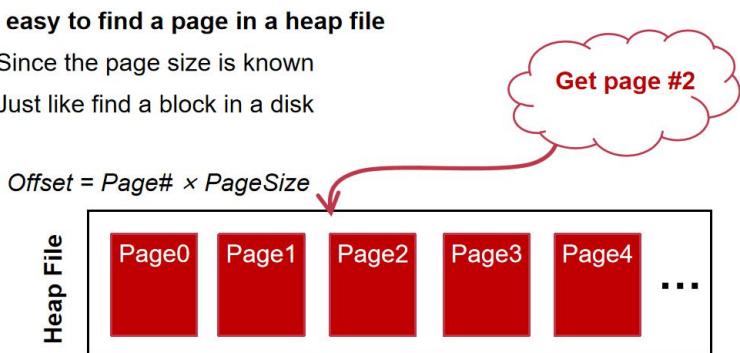
Heap file is an abstraction to manage the database pages

- An unordered collection of pages

Why don't we care about the order?

It is easy to find a page in a heap file

- Since the page size is known
- Just like find a block in a disk



当我们要去访问一个 page 的时候，我们就可以很容易地检索。这就是非常简单的 database heap file 的抽象。Database 尽可能减少了 OS 干的事情。

接下来，我们怎么找到哪些 page 是 free 呢？

1. linked list
2. Page directory

Linked list 在每个 page 的头放上 2 个指针，头尾相连变成链表。它会把第一个 page 作为 header，只维护两个指向 page 的指针，注意这里的指针是“大的 heap file 里的 offset”。注意这些不是内存中的概念。

第一个链表就是 free page list，第二个就是 data page list。

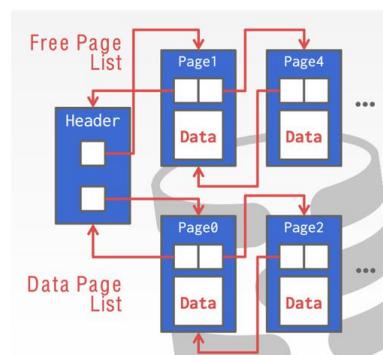
Solution-1: Linked list

Maintain a header page at the beginning of the file

- Stores two pointers
 - HEAD of the **free page list**
 - HEAD of the **data page list**
- Note: the data is not continuous
- A page can store data less than its size

Questions:

- When the lists are manipulated?
- Recall: heap file must provide:
Create, Delete, Iterate all



当 Create 和 Delete 的时候，会修改这个 list。注意每个 page 中的数据不能把 4k 撑满，因为还要留空间给头上的指针，所以数据是并没有占满 4K 的。

第二种方法是 page directory，也就是在最前方维护一个 bitmap。这和文件系统中的 block

bitmap 和 inode bitmap 是一样的。

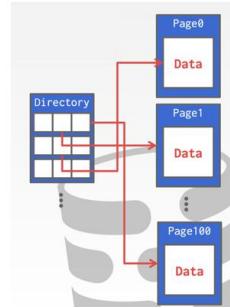
Solution-2: Page Directory

The DBMS maintains special pages

- Tracks the location of data pages in the database files
 - Like a bitmap
- Similar to the data bitmap in file systems (at the head of a disk)

The DBMS must make sure that the directory pages are in sync with the data pages

- To avoid crash consistency problem
- Similar to crash consistency in file systems
- E.g., by using logging



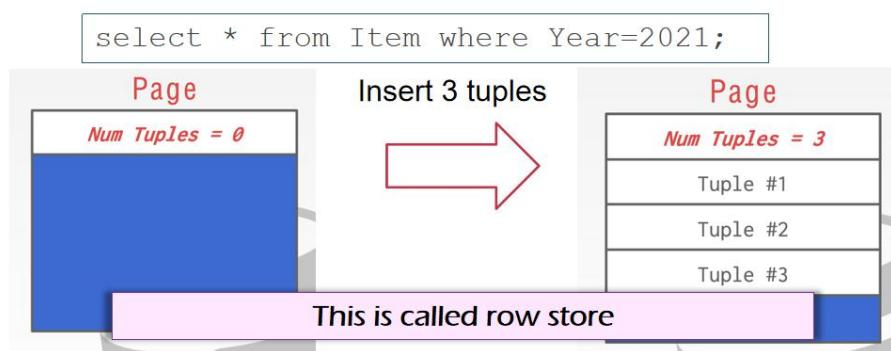
Tuple 是怎么保存在 Page 中的

如果是第一种方法，每个 page 头部还要两个指针。Data page 在存 tuple 的时候，还是放了一个元数据，也就是 number of tuples。

Tuple storage: Strawman Idea

Keep track of the number of tuples in a page and then just append a new tuple to the end

- And all attributes of a tuple is stored together (as a row)



- Benefits: easy to answer the query that acquire all fields

每一行都是一个存储，一行行存在 page 中。那么我们怎么删除一个 tuple，如果我们有变长的 attribute 怎么办？

因为 tuple 的顺序不是那么重要，删除的话就直接删除即可。但是可长可短的 tuple 该怎么办呢？如果留最长的，就会浪费空间。

我们会使用 slotted page。我们在 page 头部放下 slot array，记录的是每个 tuple 的头在哪。Slot 是从前往后找，而放 tuple 是从后往前写。

Tuple storage: slotted pages

The most common layout scheme is called **slotted pages**

The slot array maps "slots" to the tuples' starting position offsets

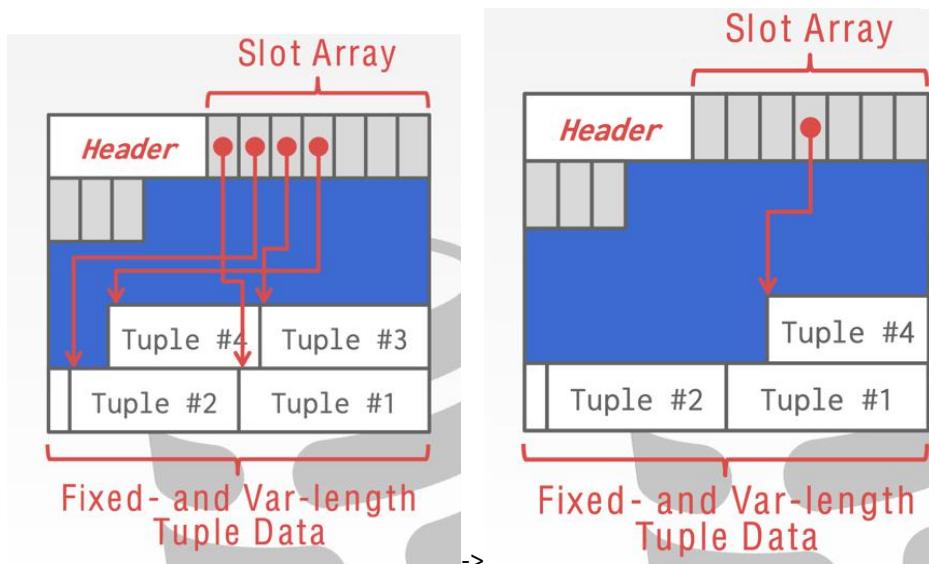
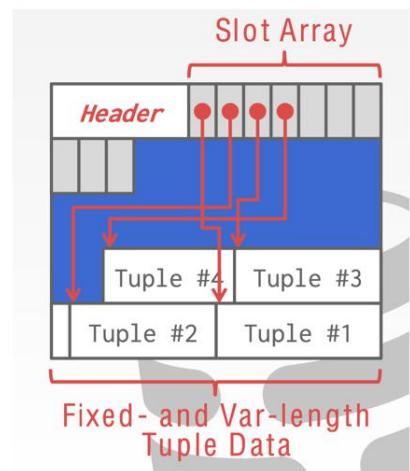
- Why not using an complete index at a page?
- A page is typically small so indexing is wasting

The slots are consecutive

- Easy to find all tuples in this page

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used



为什么要这样设置呢？这样元数据从前往后生长，而数据从后往前长，可以减少中间空间的浪费。当我们删除 Tuple#3 的时候，我们就要把 tuple #4 往后提，压缩多余的空间。**SLOT 为什么不用移动？？？**

元数据可以只记录每个 tuple 的长度。因为最终在修改的时候，都是把 4K 加载到内存中，该挪动的挪好、该写的写好再写回磁盘上。加载/写回的时间是更多的。哪怕修改了 tuple#1 要全部移动，我们也认为这是一个比较小的代价。

Q: 如果一个 tuple 比一个 page 还大怎么办？

A: 我们可以不允许这么大，也就是 MySQL 中不允许一行超过 63KB。对于那些非常大的数据，我们要用到额外的 blob (binary large object) store。它其实就是一个额外的文件。

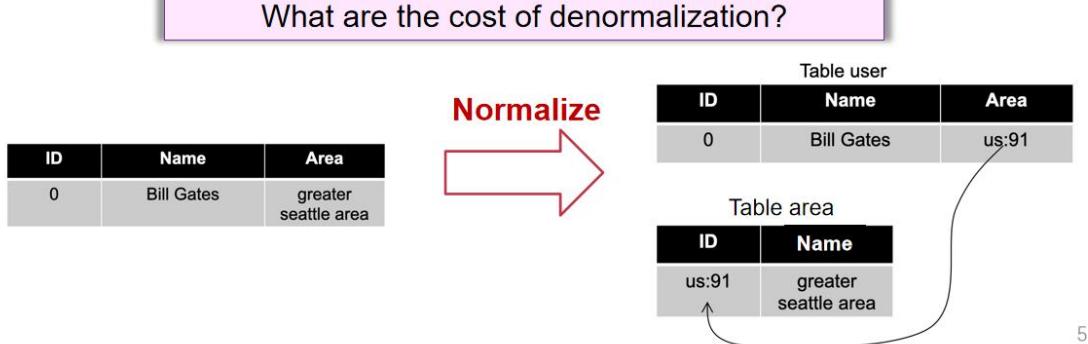
如果我们用到了 normalized

Tuple representation & denormalization

We can represent tuple in a denormalized way for better performance

Recall normalizing: remove the duplicated entities in various entities

- Useful for many-to-one relationship
- How to normalize is a big topic (out of the scope of this lecture)



当我们在做 join 操作的时候，locality 就非常差，因为这是不同 pages 中的不同 tuple。怎么解决这个问题呢？我们可以在保存的时候，就让他们保存在一块。

通过 pre-join 的技术，保存的时候是这样保存的：

Denormalized tuple data with tuple storage (pre-join)

Solution:

- Store related tuples in the same page

Pros

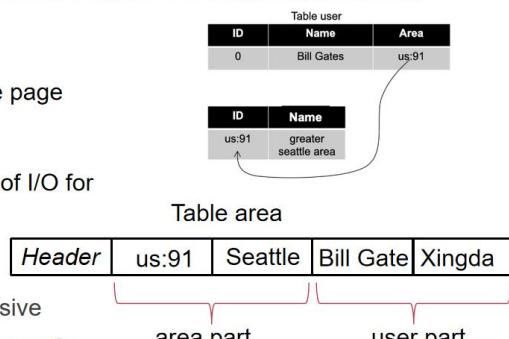
- Potentially reduces the amount of I/O for common workload patterns

Cons

- Can make updates more expensive
 - E.g., what if we insert another user?

Not a new idea

- IBM System R did this in the 1970s



多存这些东西的好处显而易见，当我们调用 join 的时候，就可以利用这个 locality 一下子把数据找出来，这是某一种优化。

所以在数据库领域有很多实现的方式，我们看到的和它内部保存的不一定是一样的。

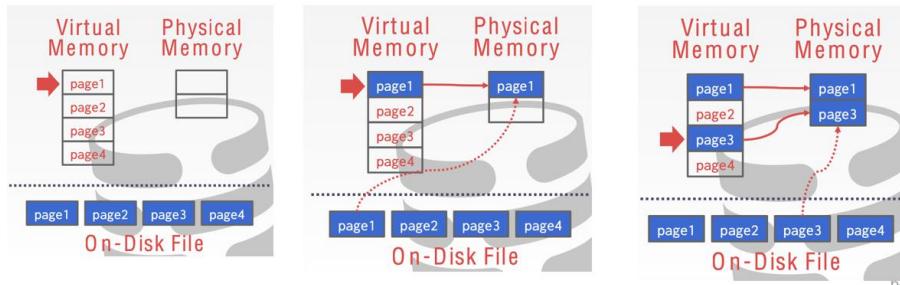
减少磁盘太慢带来的问题

磁盘太慢，就放内存中。怎么放内存呢？我们使用一个类似 cache 的机制？

Why not leverage the OS to manage the movement?

E.g., use `mmap()` to map the whole heap file

- The OS is responsible for moving the files' pages in and out of memory



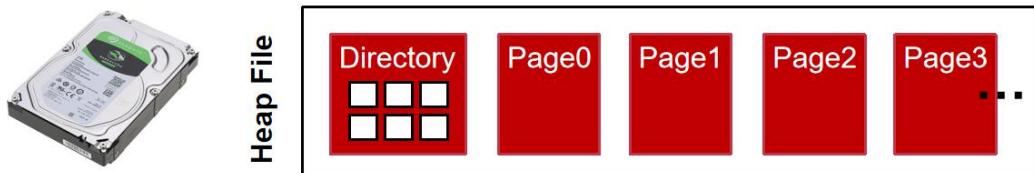
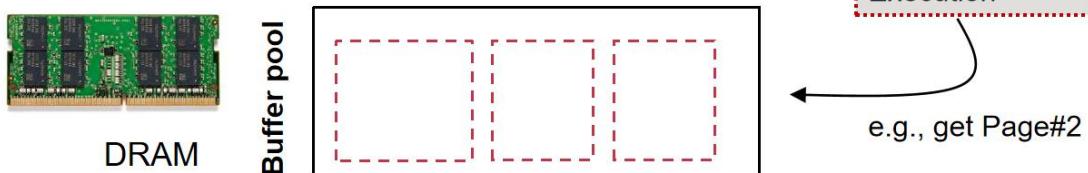
我们把一个很大的文件 `memory map` 到用户的虚拟地址空间。这就创建了 VMA，不过此时还没访问。当我们访问到某一块的时候，触发 `page fault`，就会进页表。

Virtual memory 和 page 建立起映射。等到我们访问 `page1` 的时候，`pagefault`，把这一块 `load` 到某一块 physical memory。通过 OS 的 `mmap` 机制，我们可以不用过分担心磁盘太慢这个问题。接下来，如果我们要访问 `page2`，物理内存不够了，就会产生 swap 问题，到底是踢掉 `page1` 还是 `page3`，OS 用的是 LRU 方法。如果 database 说，`page1` 是 index，而 `page3` 是 data，那么我们希望的是踢掉 `page3`。

对于 DBMS 来说，它希望它决定哪些页放进来哪些页踢出去，而不是 OS 来决定。OS 也提供了一些 `madvice`（告诉 OS，这几个页后面要读）、`mlock`（这些页不要 `page out`）`msync`。但是这只能建议 OS 这样去做。

Overview of buffer pool

Buffer pool to manage the cached copies of pages



如果要访问其他页了，怎么管理就是一个很有意思的问题

总结

今天主要讲的是 disk 的问题。DBMS 采用的是尽可能希望管理更多的细节，用一个大文件，内部每个 page 长什么样，DBMS 自己来管。对于每个 page 内部，再用 slotted 方式切成一小块来降低内部的碎片。我们可以通过 mmap 的方式把整个文件映射进去。

2021/11/16

期末考试可能考察的方向：给一个场景用学到的东西把场景解决掉。
今天我们讲的还是 database，

Heap file 就是给你一个 pagernumber，得到一个 4k，这个 heap file 被划分成 4k 的 page，头上还有一个 bitmap 记录哪些 page 是 free 的。在这层之上，就是提供一个 interface，给你一个 page number 返回给你一个 4k。在这层之下就是 file system。也就是，这本身就是一个文件，但这个文件可以非常大，大到几百个 G。有些数据库觉得一个文件太大了，就把一个 heap file 拆成若干个文件，但是对上提供的抽象依然是一个整体。

Q: 有了 page 之后，那么一个 page 里会存哪些数据？

A: 首先是 data，第二个是对 data 建立的 index。对于单个的 data page，我们从前往后存放一个 slot，从后向前存放 tuple。这相当于对 4k 内部继续做细粒度的拆分。

怎么解决访问数据慢的问题呢？我们就需要在内存和硬盘上来回地挪数据。它不是 OS 提供的抽象，而是数据库自己维护的一个 buffer。作为文件系统，很容易使用 mmap 来访问各种各样的 page，比如 index page 和 data page。

为什么不用 mmap 呢？Database 要自己实现一个 buffer pool。为了避免 OS 帮它做 cache，它使用了 O_DIRECT 的机制。当用户态读磁盘的时候，磁盘数据最终会被缓存到内存里，也就是 OS 在 kernel 中维护一个 page cache，读磁盘操作会先访问 page cache，如果没有 hit 再去读磁盘。而现在 database 希望不用 OS 的 page cache，自己维护一个 buffer pool，如果 hit 了就 hit，否则就读磁盘。所以我们在 open 整个文件的时候添加 O_DIRECT 选项来跳过 cache 缓存机制。

Buffer pool 依然是通过大数组的方式去管理。内存里的每一个数组元素叫做一个 frame。这个 frame 不是物理内存，访问它肯定是使用虚拟地址的。当一个 DBMS 要求访问一个 page 的时候，先把这个 page copy 到 frame 里，再访问 frame。Buffer manager 会选择一个 empty 的 frame，那么这么选择呢？无非还是两大类，

1. bit map
2. Linked list

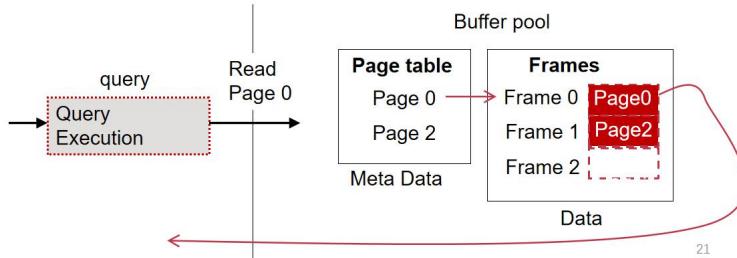
如果找到空闲的 frame，我们就使用它。我们怎么做 buffer pool 到 heap file 之间的映射呢？当我们应用程序访问 page2，我们先得看 buffer pool 中有没有 hit，我们去哪里找呢？这个在 ICS 中提过，cache 映射到 memory 的机制是全相连、组相连等。

在这里，我们使用的是一个 page table 映射。在这里的，比 OS 中的页表要简单得多，因为我们是软件实现这件事情。OS 中的 Page table 是硬件 MMU 去读，所以不能太复杂。但是在软件中，我们就可以使用一个 hash map 去实现。

Buffer pool meta data

Besides frames, buffer pool maintains several meta data

- i.e., the page table to record the mapping between frame <-> page



很明显, page table 是存在内存中的, crash 后重新建立一遍 page table 就可以了, 对于容错都不影响。在 heap file 这一层还有一个独立的 log 文件, 来增加容错处理的功能。给了我们一个 page number 的时候, 先查 page table, 如果 hit 了就返回 frame, 否则就读磁盘更新 page table 再返回。

注意 page table 和前面的 page directory 完全不一样。Page directory 记录的是哪些 page 是 free, 哪些不是, 这个是需要持久化在磁盘上的, 而 page table 不需要。

Page table 记录的就是 page 到 frame 的映射, 当我们写了一个 page 到 frame 之后, 我们应该把这个数据写回到磁盘, 但是如果我们每次写穿都会很慢。数据什么时候从 page table 中进磁盘, 我们需要记录哪些 frame 是 dirty 的, 等到我们空闲的时候, 一下子把这些 dirty frame 下回去。所以 frame 中我们有一个 dirty bit。

如果说, 我们的 frame 不够了怎么办? 我们就要做 evict 操作, 如果我们 evict 了一个 dirty page, 那么我们就要把这个 page 写到存储上去。

Q: 如果 buffer pool 和一个 query 同时访问一个 frame 怎么办? 也就是一个 cpu 在往里写数据, 另一个 cpu call 想 evict 掉, 怎么办?

A: 这就是 before-or-after 问题, 解决它只需要加锁就行了。我们可以加 latch (其实就是锁)。

Latch 核心就是说我们要保护内部的数据结构, 比如 database 的 metadata, 比如这个 pagetable 用户是看不见的, 用户只能看到存在数据库中的变量 a、b。至于 page table 和 B+树的 index, 是用户看不见的。所以从用户的角度来说, 这些数据就是逻辑数据。在实现的时候, 我们必须有 B+树、page table。所以保护这些数据结构就是用 latch, 保护用户的数据就是用 lock, 但是实际上都是锁。

Locks vs. Latches

Locks: (in two-phase locking)

- Protects the database's logical contents from other transactions
- Held for transaction duration

Latches

- Protects the critical sections of the DBMS's internal data structure
- Held for operation duration
- Decouple the internal data structure concurrency control from the transaction
 - E.g., Index & buffer pool

Though both are implemented as locks (e.g., pthread_mutex), they have different meanings in the DBMS

就算两者都是用 lock 来实现的, 在 DBMS 中也是不一样的。

Locks vs. Latches

For example, suppose two database transactions

- TX1: insert into Items {ID1, ... }
- TX2: insert into Items {ID2, ... }
- The ID is indexed by a B+Tree

Question:

- Does the two TX conflict?
- Will they wait for each TX's lock?
- What will happen if two TXs concurrently insert to the B+Tree?

从逻辑上来说，这两个 TX 不会 conflict，因为没有共享变量。

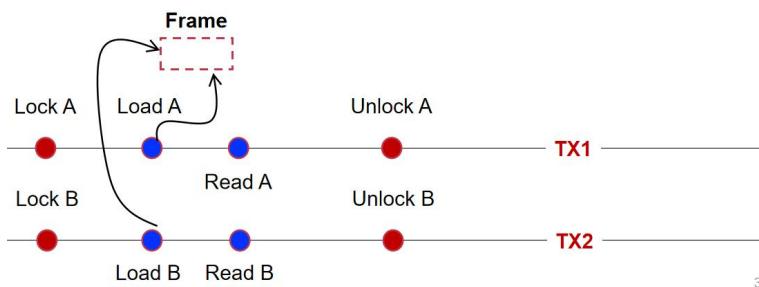
但从物理上来说，index 是共享的，page table 也是共享的。

如果我们使用 two-phase locking，对这两个数据一视同仁，那么我们应该这样做

Locks vs. Latches

To accelerate execution, the DBMS will first load A & B to the buffer pool

- Question: what if two TX concurrently write to the same frame?
 - The buffer pool frame would be corrupted (race condition)



34

如果它们用到的是同一个 frame，在 frame 这个地方产生了一个 conflict。那么我们不仅仅要 lock A，还需要拿到 page table 的锁，做完操作以后再去 release 两把锁。这样任何操作都有可能在内部数据结构上产生 conflict。当我们把 latch 和 lock 解耦开来的话，我们会发现如下：

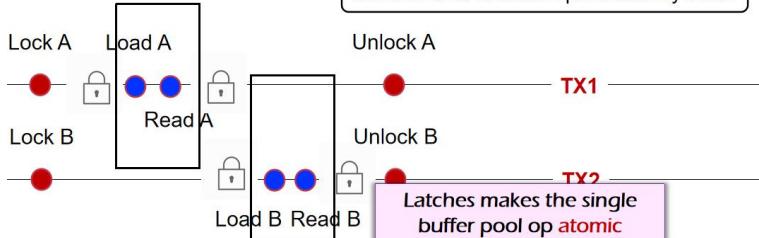
Locks vs. Latches

Latch

To accelerate execution, the DBMS will first load A & B to the buffer pool

- Question: what if two TX concurrently write to the same frame?
 - The buffer pool frame would be corrupted (race condition)
- Latch helps protecting the buffer pool w/o holding a lock on it during the whole TX lifetime

Internal data structure protected by latch



我们读完 A 以后，就放掉了锁。实际上是破坏了 two-phase-lock 的语义。

如果在这个 transaction 中还要读到 C，那么我们还要拿 page-table 的锁，再放掉。但是 two-phase lock 中说的是放完锁就不能在拿了。

对于 lock 来说，是满足 two-phase lock 的，只是对于 latch 说不满足。但是这并不会导

致问题，因为放掉 page table 的 latch 最最不幸的结果就是 A 的 frame 被 evict 了，这导致下次读 A 的时候从磁盘读。对于用户来说，并没有访问用户访问 A 和 C 是锁住的。

如果我们使用 two-phase lock 要求所有类型的锁都要先拿再放，这就会导致实现起来遇到很多问题。但是使用 latch-lock 的结构就可以解决这个问题。Latch 拿的时间是很短的，因为它无非就是保证读 A 的时候不要被人踢走。

对于 lock 来说，它保护的是用户看得到的数据，对于 latch 来说，它保护的是数据库看得到的数据，只要 operation 结束，我们就可以放锁，对于它来说，它根本看不见 transaction。

哪些 internal structure 需要保护呢？B+树的 index 和 page table。当我们新增数据的时候，b+树可能分裂，两个人同时操作 B+树就可能导致错误的结构，所以使用 latch 来保护。

latch 因为保护的都是易失变量，所以 latch 本身是不需要进 log 的。但是拿着 lock 的情况如果 crash 了，可能需要把 lock 进 log 从而更好地 recover。

Buffer Pool 已经存在了很长时间，为了进一步做优化，做数据库的人想出了很多的点，我们来看几个应用：

1. Multiple buffer pool

为什么要多个呢？不同的 page 可能有不同的 access pattern，一个 index page 可能更长地被访问，index page 和 data page 被访问的模式是不一样的，如果只有一个 buffer pool，互相的 pattern 可能就不能很好地被利用。如果我们分开，不同的 data pattern 就可以做不同的优化。并且单个 buffer pool 可能使得竞争更加剧烈，因为 buffer pool 的全局唯一的 latch 可能就成为一个大家都在竞争的资源。

我们可以每个 database 给一个 buffer pool，或者每个 page type（index、data）分配一个 buffer pool。

MySQL 中就会把 buffer pool 划分成若干个独立的 buffer pool，提高并发执行能力。

我们也可以对一个 page type 有多个 buffer pool 呢？们怎么知道我们要的 page 在哪个 buffer pool 中呢？

方法 1：哈希

```
Two classic partitioning algorithms
– Hash partitioning & range partition

Example: get_page(1)
– 1. id = Hash(1) %2
– 2. buffer_pool[id].get_page(1)
```

方法 2：range

```
Two classic partitioning algorithms
– Hash partitioning & range partition

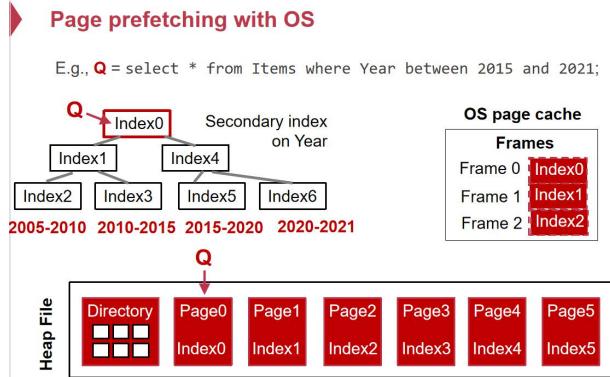
Example: get_page(1)
– 1. id = (1 <= 2)?0 : 1;
– 2. buffer_pool[id].get_page(123)
```

这两种划分方式的优缺点：通常来说，如果我们对数据什么都不知道，我们可以使用 hash，但是它会把原本一些顺序访问变得随机化。而 range 通常来说会对 locality 更好，但是这里的 range 是根据 page number 来的。但是相邻的 page 是否存着相同的 page 呢？不一定，这需要上层帮我们保证这件事情。

2. Prefetching

从 storage 来说，我们访问 page1 的时候，同时我们把 page2 和 page3 都加入到 frame 里去。对于这种简单场景来说，OS 其实做的非常好，比如使用 MMAP+MAP_POPULATE。但 database 不用 OS，因为 database 可以做得更好。

如果用 OS 做 page cache 的话，无非是读磁盘的时候读到什么 cache 什么。因为 OS 看到的是文件系统，但是 database 的 page 可能是 16K。OS 看到的太底层了，看不到上层的语义，所以不太智能。



当我们 index0 的时候，会自动加入 index0~index2，但是它下一个要访问的是 index4，这样就 miss 掉了。但是在访问 index4 的时候，会预取 index5 和 index6，就可以加速。核心就是 OS 不知道 B+树的底层结构是什么。

通常来说 DBMS 不会使用 OS 自带的 page cache，而是自己去管理 cache。

3. Scan sharing

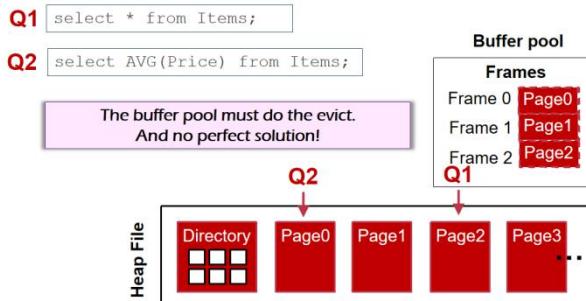
有两个 query:

The queries do not necessarily be the same, for example:

- Q1: select * from Items;
- Q2: select AVG(Price) from Items;

这两个都是要做 scan 操作。

Example of scan sharing



Q1 继续往下走的时候，要踢到一个，这会导致 Q2 一定会发生 miss，就变成了 Q2 要访问的所有东西在前一刻踢走了。怎么办呢？DBMS 会马上就知道 Q1 和 Q2 要访问的东西是一样的，它会尽可能让这两个 query 一起做，也就是 DBMS 会让 Q1 先停住，等到 Q2 到达以后再一起做。

Buffer Pool 的 Replacement Policy

一旦 buffer 满了之后，我们需要 replace。我们需要考虑准确性高、速度快、少一些元数据的 overhead。

LRU(Least Recently Used): 适用性广但不一定最适合。我们需要在 page table 中放一个 timestamp，说每个 entry 上一次被访问的时间是多少，我们要去排序。我们要记录 timestamp 本身就很难了，我们还需要让所有的 page 根据 timestamp 做排序，我们需要额外的数据结构。比如使用一个 queue 来记录访问顺序，队列的最末尾一定是最近访问的。

Overhead 太高带来的优化就是使用 clock 算法。每个 page 有 reference bit。时钟转的时候，遇到 1 就变成 0，遇到 0 就 evict 掉，它的效果近似与 LRU，实现起来也很简单。

但是 LRU/Clock 还是不够，它会遭遇到 sequence flooding 这个问题，也就是如下的 SQL 查询语句：

```
select * from Items;
```

它是顺序走一遍，但是不需要 locality，整个 buffer pool 做完这个操作之后可能就清空了，这时候我们可能需要 MRU (Most Recently Used，刚访问的就 evict 掉)。

如果我们在看电影的时候做一些需要 locality 的操作，那么看电影这件事情就会把 cache 的重要数据冲走。这时候我们应该把 cache 划分一下，把很少的一部分给电影，剩下的大部分给需要 locality 的操作。

方案 1：使用多个 buffer pool，对于 select * 中就分配一个很小的 buffer pool。

方案 2：使用别的策略，LRU (k，只在 k 个范围内做 LRU)；或者 MRU。

方案 3：不用 buffer pool 了。在 IBM informix 可以使用 light scan，只使用一块小内存滑动窗口。

通过 query execution 来优化 buffer replacement policy

假设我们要 insert into ITEMS values(id++, ..)

把所有商品从一家店 insert 到另一家店。

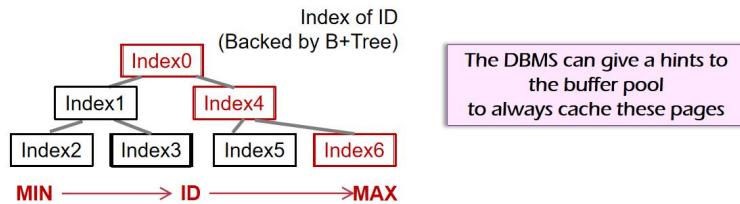
► Example: Insert into Items VALUES (id++, ...)

Suppose we add an attribute to our Items table

- ID as its primary key

What's the access pattern of this query if we execute it continuously?

Insert into Items VALUES (id++, ...)



因为是一直往下加的，我们可以让 DBMS 一直保存这个路径。

点搜索：select * from Items where ID = xxx

这样是不需要搜索整个磁盘的，对于这种情况 cache 的最合理的情况是存 index。因为我们不知道 xxx 的具体情况，底下的 index 理论上应该是随机分布的。但是越往下的节点被

访问的概率就越小，所以 cache 机制尽可能把上面这些放到 buffer 里去。这都是根据数据库被使用的时候，哪种 pattern 更多，对应的 cache 策略是不一样的。

I/O cost of evicting a frame

If a page is not dirty, then evicting is fast

- No disk I/O (write) involved

What if a page is dirty?

– Problem #1. Slow

- The DBMS must write back to disk to ensure that its changes are persisted

– Problem #2. Consistency

- The log of the modified page is not persisted

如果 page 没有被改过，那么就可以直接踢掉。如果 page 是 dirty 的，那么我们就要写磁盘。我们可以使用 background writing 的方式：在 DBMS 的后台等待系统空闲的时候，就会把 dirty frame 写到磁盘上的 page 中，然后清除 dirty bit。

Conclusion of DBMS managed buffer pool

The DBMS can almost always manage memory better than the OS

Can leverage the semantics about the query plan to make better decisions

- Evictions
- Allocations
- Pre-fetching

Yet, for simplicity, several systems still leverage the OS to manage its buffer

- E.g., the OS's pre-fetch & CLOCK works well under many scenarios

抽象会导致性能损失。一种合理的方式是 OS 提供一个申请磁盘的时候可以 propose 磁盘的外延还是内延，这样让 index 放到磁盘外延可以让访问速度更快。

2021/11/23

语句首先会进行一系列的处理，然后处理 DBMS 的 storage，最终通过 IO 访问到存储设备上，之前我们集中看的是系统是怎么存的以及上层的 cache，我们今天集中讲 SQL。

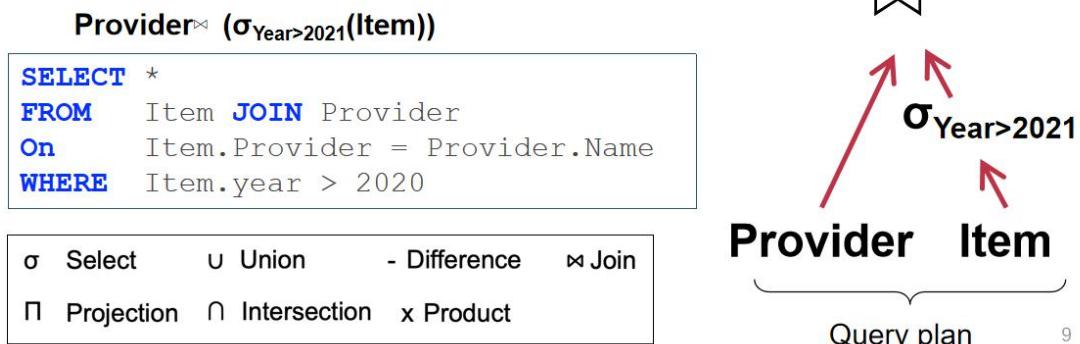
首先 Query Engine 会把 SQL 语言变成一个 tree (SQL plan)，它的每一个节点就是一个 operator，每一个 operator 做的额都是 relational algebra。数据是从叶子结点一路流动到根节点，就产生了 query 的结果。我们以之前卖东西为例，当我们要执行这样一个 query 的时候

Query plan

How to generate the query plan is out of the scope

The query engine will parse a SQL into a tree (called **query plan**)

- Each node is an **operator** in relational algebra
- Data flows from the leaves to the tree up towards to the root
- The output of the root node is the result of the query



最后我们有一个 `item.year > 2020`, 这是 item table 的 select 条件, 最终就会被解析为一个 tree。这个 tree 是要对 SQL 做解析的, 实际上这件事情不是很容易。但是它有一个好处就是 SQL 是一个标准, 一旦有一个人把解析器开源了, 大家都可以共享。需要注意的是并不是每个 database 都支持所有的 SQL 语句。

有了这个 tree 之后, 我们先不考虑这棵树是怎么生成的 (处理整个 string), 我们所关注的是它的 processing model 是什么, 也就是拿到这个 tree 之后 database 是怎么去处理的。Database 在处理 query plan 的时候有三种方法

1. **iterator model**: 让每个 node 都实现 `next` 这个方法。对于每一次调用的时候 operator 就会返回要么是一个 tuple, 如果没有 tuple 就返回 null。对于 operator 要实现一个循环不断调用下层节点的 `next`, 等到所有子节点都返回, 它就再向上返回。

Iterator model

Each operator of the query plan implements a **Next()** function

- Similar to the `iterator` in programming languages

Next()

- On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples
- The operator implements a loop that calls `Next()` on its children to retrieve their tuples and then process them
- Like the `iterator` in the programming language

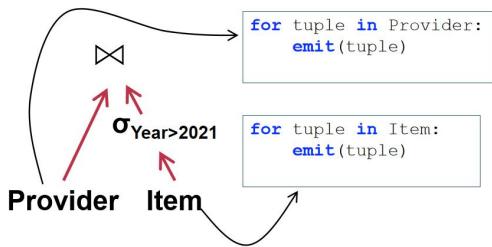
Also called **Volcano model**



我们举一个例子来看一下 iterator model 执行的过程。

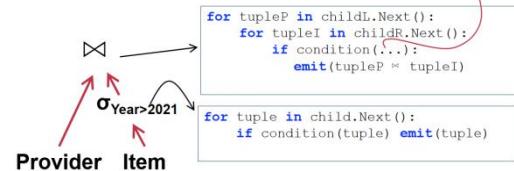
Iterator model

```
SELECT *
FROM Item JOIN Provider
ON Item.Provider = Provider.Name
WHERE Item.year > 2020
```



Iterator model

```
SELECT *
FROM Item JOIN Provider
ON Item.Provider = Provider.Name
WHERE Item.year > 2020
```

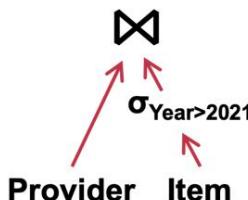


13

上方就是一个 plan，我们先看 item 这个 table，它的 next 就是对于 item 中的每一行，都生成一个 tuple 返回上去。这就是 table 最基本的功能：一行行返回。对于上面有一个条件，那就是当条件满足的时候，我们去 emit。对于 join 操作就是两层 for 循环嵌套分别对应两棵子树，如果笛卡尔积满足的条件的话，那么就把两个 tuple 连接起来并返回。

▶ Iterator model

```
SELECT *
FROM Item JOIN Provider
ON Item.Provider = Provider.Name
WHERE Item.year > 2020
```



tupleP.Name == tupleI.Provider

```
for tupleP in childL.Next():
    for tupleI in childR.Next():
        if condition(...):
            emit(tupleP < tupleI)
```

```
for tuple in child.R.Next():
    if condition(tuple) emit(tuple)
```

```
for tuple in Provider:
    emit(tuple)
```

```
for tuple in Item:
    emit(tuple)
```

所以整个 query plan 最后实现的话就是这么一张 table。在运行的时候，我们是从上往下去执行。我们先去调用 childL.next，返回了一个 tuple 到 tupleP 中，然后我们在调用 childR.Next 就会到条件中去访问，然后就会到 item 这个迭代器，返回一个满足条件的 tupleI。如果 tupleI 和 tupleP 满足条件，那么我们就返回了既包含 tupleP 和 tupleI 的一个 tuple。如果不满足的话，它就会继续调用下一个 next。

Iterator model 是比较简单直观的，基本上每一个 DBMS 中都存在这个。如果我们要做排序的话，我们需要等产生了所有 tuple 后才能返回（需要阻塞住等到所有 tuple 返回）

Iterator model 的缺点：next 的调用开销是很大的，开销为什么大呢？因为我们要保存 caller-save, callee-save, return address，在处理过程中我们要保存大量函数调用之间的开销。C++ 的函数调用实际上是一个 virtual function call。虚函数在调用之前需要查一下继承表，这会进一步恶化开销。

在传统的数据库里，到访问 tuple 的时候，是要访问磁盘的，一旦牵涉到访问磁盘，上层的 overhead 都可以忽略掉了。因为访问磁盘的时间是上层的 overhead 的一千倍以上，所以在传统的磁盘为存储介质的数据库中，这种方法是比较容易的方法。

但是现在内存越来越大，出现了一种 in-memory database，访问 tuple 的时间非常短，这样我们就不能忽略大量的函数调用的 overhead 了。

2. materialization model (物质化模型)

思路很简单，既然前面的问题是上调下每次都是一点点的 next，我们能不能一下子把全部的东西都返回？我们希望一下子输入所有的输入，一下子产生所有的输出。

好处就是函数调用的 overhead 大大降低了，但是 output 太大了怎么办？比如对一张一千万行的表做 select * 呢？所以对于用户来说，它可以写一个 limit 关键词。

Materialization Model

Each operator processes its input **all at once** and then emits its output **all at once**

- I.e., the operator "materializes" its output as a *single result*

What are the benefits?

- Less function call overheads

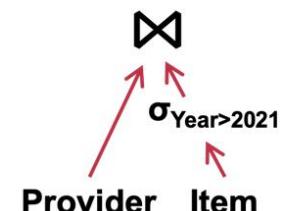
What happens if the output is too large?

- Should co-work with the DBMS *push down hints* (e.g., LIMIT) to avoid scanning too many tuples
 - LIMIT: set by users, which limits the number of tuple returned by a query

Limit 可以限制每次调用 tuple 返回的个数，每次调用限制只返回多少个，这样就可以很好地在内存中返回。

Materialization Model

```
out = []
for tupleP in childL.get_out():
    for tupleI in childR.get_out():
        if condition(tupleP, tupleI):
            out.add(tupleP < tupleI)
return out
```



```
out = []
for tuple in Provider:
    out.add(tuple)
return out
```

```
out = []
for tuple in Item:
    out.add(tuple)
return out
```

我们看到叶子结点是把整张表直接都返回了。最终函数调用可能就三四次结束了。所以 materialization 比较适合 OLTP (online transaction process) 在淘宝上买一个东西就是一个 transaction，基本上我们只需要访问很少的一些 tuple，比如货物-1，余额-1，已购物品中又多了一个东西。牵涉到的东西比较少，所以执行时间也比较短。但是对于 OLAP (online analysis process) 比如我们要生成今天最受欢迎的物品是什么，对于这种 analysis 要大量的扫描，使

用物质化就不太合适了。有没有办法折中一下呢？

3. Vectorized/batch model

Vectorized model

Somewhere between Iterator & Materialization model

Like Iterator model, each operator implements a `Next()` function

- On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples **a batch of tuples**

The operator's internal loop processes multiple tuples at a time

- The size of the batch can vary based on hardware or query properties

与 iterator model 类似，每次调用的就是 `next`，以前返回的是一个 tuple，现在返回的是一组 tuple，比如 10~20 个。上层拿到 tuple 处理完再调用 `next`。

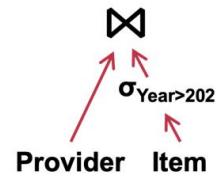
Vectorized model

```
out = []
for tupleP in childL.Next():
    for tupleI in Next():
        if condition(tupleP, tupleI):
            out.add(tupleP < tupleI)
        if len(out) > N: emit(out); out = []
```

```
out = []
for tuple in child.Next():
    if condition(tuple) out.add(tuple)
    if len(out) > N: emit(out); out = []
```

```
out = []
for tuple in Provider:
    out.add(tuple)
if len(out) > N:
    emit(out); out = []
```

```
out = []
for tuple in Item:
    out.add(tuple)
if len(out) > N:
    emit(out); out = []
```



29

这样底层就等于维护一个 N ，每次最多返回 N 个。这样函数调用次数比起 iterator model 来说降低了 N 倍。显然这样对于 OLAP 模式就比较合适了。

更有意思的一点是，对于 DBMS 来说可以有针对性地设置这个 N 。比如跑在英特尔的 CPU 上是 SIMD(single instruction multiple data)。这样一条指令就可以做 16 个加法，有效 IPC (instruction per cycle) 就是 16. 这样一次给 16 个，就可以很好地利用硬件的能力。所以 database 在选择 N 的时候就可以根据底层处理器不同的能力来设置 N 。之前的 LIMIT 虽然很像，但是我们希望用户是不知道底层应该设置多少的。

Operator 的优化

对于火山模型中的每一个节点，我们说都是一个实现关系代数的 operator。对于每个

operator 是怎么优化的呢？

当我们要 `select * from` 的时候，我们很可能需要去遍历整个表，但是这是一件很慢的操作，并且当我们还存在一个 `where item.year > 2020`，我们为什么还要遍历这个表呢？我们能不能利用这个 `where`。我们需要建立 `index` 来加速查找。Database 在内存中的结构是 `buffer-pool` 和 `index`；Database 在硬盘上的结构就是 `index + tuple + log`。所以我们的假设是访问 `index` 比访问 `tuple` 要快一些，因为 `tuple` 可能很长很长，但是我们的 `index` 仅仅是对一列做了一个 `index`，然后就是一个指向 `tuple` 的指针。

Q: `index` 的指针是什么？

A: `heap file` 的 `offset`。

既然访问 `index` 比访问 `tuple` 要快，我们希望减少整个 `tuple` 的时间。对于不同的操作比如说 `join`、`aggregate`、`select` 中的优化都是完全不同的。我们来看 `select` 是怎么做的。

Select 的优化

通常 `select` 有很多 `where` 的 predicate，我们假设现在是存在 `index` 的。DBMS 挑选一个 `index` 找到哪些是需要的。

Accelerate `select operator with index scan`

Index scan

- DBMS picks an index to find the tuples that the query needs
 - Instead of scanning the whole table to find the target tuples

```
SELECT *
FROM Item
WHERE Item.year > 2020
```



Accelerate `select operator with index scan`

Example: suppose there is an index on `Item.year` (named `Item.year_index`)

- Cursor: the current BTree node to traverse

```
SELECT *
FROM Item
WHERE Item.year > 2020
```

```
for tuple in child.Next():
    if condition(tuple)
        emit(tuple)

for tuple in Item:
    emit(tuple)
```

w/o index

```
cursor = Item.year_index.seek(2020)
for tuple in cursor.Next():
    if emit(Item.get(tuple.value))
```

w/ index

通常情况下，数据是以 b 树的结构来组织的。没有 `index` 的情况下，我们就需要调用 `iterator` 去遍历整张表。有 `index` 的时候，我们就可以直接找到 `year` 列对应的 `index` 找到 2020，然后返回后面的 `tuple` 即可。

Item (name, provider, year, price)			
Name	Provider	Year	Price
Ipad pro	Apple	2021	9999
Iphone	Apple	2021	6666

```
SELECT *
FROM Item
WHERE
    Item.year > 2020
    AND Item.price > 8000
```

接下来，如果我们的 predicate 更加复杂，我们在 year 后面再加了一个 price。我们就产生了一个问题，这两个哪个先做呢？

There are **99** items released after 2020, but only **2** of their price is larger than 8000

这种情况下，我们就应该先做 price 的索引，在做 year 的索引。所以不同的顺序会导致我们调用 next 的次数是不一样的。

Q: 在没有执行这个 query 之前，我们该怎么知道一个是 2 个，一个是 99 个呢？也就是怎么选择谁先做谁后做呢？

A: 使用 bitmap scan

How to reduce tuple fetched? Multi-index scan

If there are multiple indexes that the DBMS can use for a query

- Compute sets of **tuple addresses** using each matching index
- Combine these sets based on the query's predicates
 - (union vs. intersect)
- Retrieve the tuples and apply any remaining predicates

Postgres calls this **Bitmap Scan**

How to find the intersected data efficiently?

Naïve solution:

- Nested loop takes $O(N * M)$

Fast set intersection can be done with

- Bitmaps, hash tables, or Bloom filters

Example of bitmap:

- Suppose we have two sets {1,2,4} and {1,2,5}
- Then we can compute two bitmaps: **00010110** & **00100110**
- Then using an **and** operation will get the intersected results

我们先找一个 $year > 2020$ ，再找 $price > 8000$ ，把这两个变成 bit map，然后求一个交集，最后 **and** 一下即可。所以用 **bitmap** 方法并不是一个很聪明的方法。更容易的一个方法就是把搜索记录下来。

除了 **bitmap** 之外，我们还有一些别的方法，比如我们可以使用 **bitmap**。那么其他的一些 **operator** 就不管了。

于是我们发现到这里，我们就把 database 的 query storage 就讲完了，传统的 database 是提供 ACID 的。其中有很多是提供 transaction 来实现的。但是随着互联网的发展，使得传统的学院派的 SQL 不能适应互联网的高速发展，于是进化出来 noSQL 和 newSQL。

我们继续我们的课，上节课讲完了 SQL 的 DBMS 的实现，

OldSQL = Relational Model + SQL + ACID

The architectural or historical baggage of SQL

Abstraction

- Relational + Transaction

Semantic

- ACID

Architecture

- Mostly focus on Single-node & On-disk

除了关系型模型，还有 document, kvstore。关系型模型又包含了 relational model。Query 有包苦熬了存储的模式和 query 的模式。传统的 SQL 就等于 model+SQL language (存储访问配置) +ACID。

早期 SQL

有了这个之后确实挺好用的。但是互联网的发展对于 scalability 和 availability 的要求也迅速上升，互联网具有很强的 pattern，比如 read 很多而 write 很少。

The trend of Web and Big Data

Requirements

- Scalability: Multiple concurrent users
- Availability: Minimal User-down time
- Read-mostly
- Many data operations do not need transaction
- Large dataset

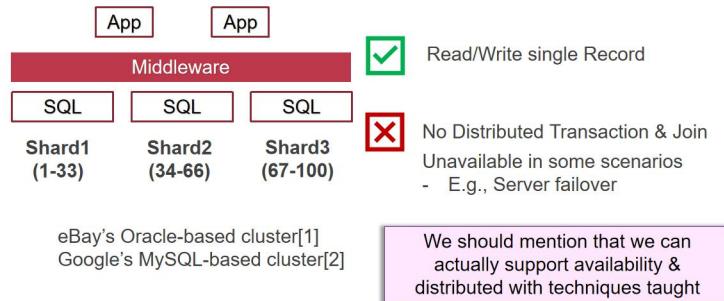
Note: some requirements are compatible to the SQL (and that's why there is NewSQL)



当需求不再被技术满足的时候，就会出现波动和机会。

传统的 SQL 基本上只支持 vertical scale (小型机变成大型机)。水平扩展是互联网公司更喜欢的。86 年提出 share nothing 就可以得到 scalability，我们可以做 shard

▶ Scale Horizontally with Middleware (in old time)



[1] <https://tuftsdev.github.io/WebEngineering/spring2017/notes/ebay2006.pdf>
[2] <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/38125.pdf> 53

在上个世纪很流行的就是 **middle ware**，把切成片的 **shard server** 合成一起。有了这个 **middle-ware** 之后，对于单个记录的读写没问题，但是没有 **distributed transaction**。如果机器 A 要转一笔钱到机器 B，没有 ACID 的保证，因为 **middle-ware** 早期没有 ACID 的功能，现在开发出了很多提供 ACID 的 **middle-ware**。一旦有一台 **server** 挂了，我们要提供 **availability** 是没有的。

由于不能提供分布式 **transaction**，所以是不能实现分布式转账（对 **transaction** 很高的操作）的。对于 **join** 也不支持，对于 **select *** 也不支持，这对于早期的 **middle-ware** 要求太高了。

NOSQL

于是就有人提出的 **NOSQL**。到底什么是 **NOSQL**，它可以认为是互联网厂商对数据库的一种叛逆行动。

NoSQL - Build from scratch

How does NoSQL make trade off?

1. Specific (simplified) data model
2. Weaken Transaction
3. Async Replication



Reducing the complexity
e.g., distributed query plan, join support, etc.



Sacrifice consistency for scalability & availability

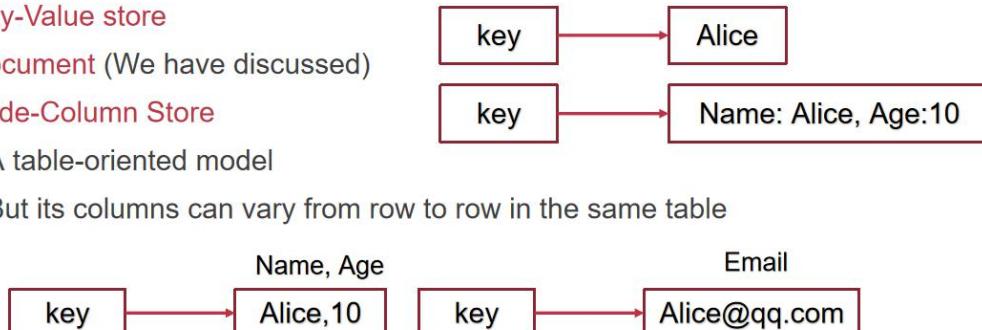
我们不用关系型，我们恢复成 Kvstore，我们把 **transaction** 的语义弱化，然后就是为了性能，我们可以降低一部分 **consistency**。

首先是简化的 **datamodel**，所有的 **record** 都可以通过 **Lookup** 找到。

1. Specific (Simplified from SQL) Data Model

All records can be found by simple lookup

1. Key-Value store
2. Document (We have discussed)
3. Wide-Column Store
 - A table-oriented model
 - But its columns can vary from row to row in the same table



Why?

- Easy to build from scratch, can focus on being more scalable!
- E.g., no need to implement the iterator model used in SQL

17

因为 kvstore 简单，不需要 SQL。

Weaken transaction

第二：transaction 变得弱化，对于 local transaction，是可以保证的，对于全局的 global transaction，就要求我们要在两个 shard 之间建立 transaction，但是一旦有了全局 transaction 就会很慢。我们能不能去掉呢？

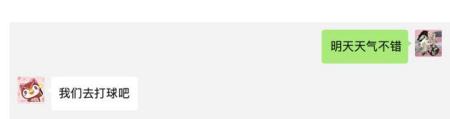
Drawback of weaken transaction

What may happen when weakening the transaction?

- Can have incorrect results (anomalies)
 - Messages reorder

In several workloads, can tolerate such incorrect results

- E.g., social network applications



WeChat on my Mac



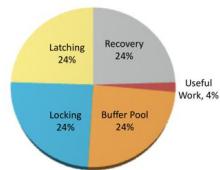
WeChat on my iPhone

这种事情用户不 care。所以 consistency 在这里是可以被牺牲的，用户可以忍受这种不正确的情况。就等于我们不用 transaction 了，acid 都没了。为什么互联网厂商敢这么做呢？Facebook 的 paper 统计过我们担心的情况在一百万次读中就一次。如果我们要避免这 100 万分之一，加上 transaction，那么就会导致真正的业务逻辑只占了 4% 的时间。

Why weakening the Transaction?

The Problem of OldSQL

- Huge software overheads introduced by ACID transactions



OldSQL spends **little time** for useful work

Figure from slides presented
by Michael Stonebraker[1]

[1] <https://www.usenix.org/legacy/events/lisa11/tech/slides/stonebraker.pdf>

记 log 又花了 24% 的时间，lock 和 latch 各占了 24%，维护 buffer pool 又花了 24%。

3. 异步复制 (asynchronous replication)

3. Asynchronous Replication

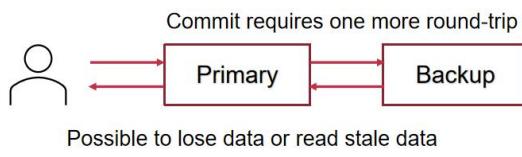
Replication is used for Availability

Synchronous

- Wait for backup's responses before return

Asynchronous

- Only waits for the primary



What are the trade-offs?



我们现在不等 backup 返回，直接返回给用户。这么搞会导致 primary 和 backup 之间的 inconsistency，一旦 primary 挂了，backup 提供的服务可能不是最新的或者是错的，等到 primary 恢复之后，最终状态才能同步到一起。如果 backup 错得很离谱，最终状态可能都同步不了。比如我们在 primary 挂了前花了一块钱，而在 primary 恢复前又花了一块钱。这就可能导致最终真的亏了钱。所以淘宝可以系统 consistency，而支付宝可以牺牲 availability。

NoSQL	Data Model	Replication	Transaction
BigTable (2005)	Wide-Column	Sync+Async	Local
Cassandra (2008)	Wide-Column	Async	Local
MongoDB (2009)	Document	Async	None
Redis (2009)	Key/value	Async	None
Dynamo (2012)	Key/value	Async	None

The developers' burden without Transaction

Application-level Transaction & Join

- Can cause error/performance issues

```
Tx Begin  
A = A + 1  
C = C + 1  
Tx End
```

With Transaction

```
Try {  
    Lock(A)  
    A = A + 1  
    Lock(C)  
    C = C + 1  
} catch {  
    reset(A)  
    reset(C)  
} finally {  
    Unlock(A)  
    Unlock(C)  
}
```

Without Transaction

于是 Google 内部就有很多抱怨，认为应该再次加上 transaction。于是 MongoDB 加上了 multi-document transaction。

NewSQL

因为 nosql 又支持了 transaction，所以命名为 newSQL。

NewSQL

A class of modern RDBMS, which provides:

- NoSQL's Scalability
- SQL's ACID Transaction & Relational Model

First proposed in 2011 [1]

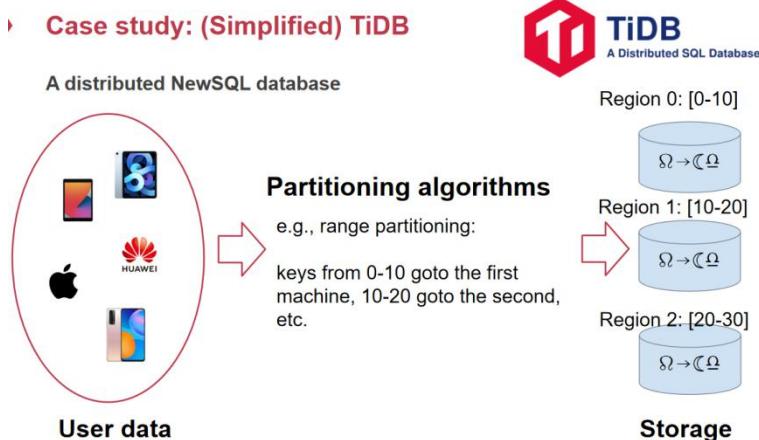
NewSQL 的定义：对外提供的是 SQL 语句；支持事务的 ACID；并发控制不能有锁，有锁太慢；对于单个节点的性能要更高；scale-out 和 share nothing，一定要支持多机的情况。

怎么去实现一个 newsql？

我们用到的方法都是我们学过的。

Case Study:TiDB

是中国主导的开源数据库项目，受到 Google 的 percolator 和 Spanner 的启发。看看怎么样既支持 SQL 又支持更好的 scalability。TiDB 有一个很好的特点就是搭积木，它复用了很多开源软件。



要做 scale, 数据库一定要 partition, 我们可以通过某种算法, 在最底层分成若干个 region。Kvstore 怎么做呢? 首先基于文件系统, 在上面放了一个 rocksDB, 它是基于一个 LSM-tree based kv-store。

2021/11/25

在学 RPC 的时候, 它和本地的 PC 相比的话, 可能就多了一个网络的问题。网络是一个很好的解释控制系统复杂性中的 layer 的方式。我们在第一节课的时候提过复杂系统之间因为组件之间的交互太多了, 很难去精细地控制。所以我们提出了 M.A.H.L 这四个方法 (modularity, abstraction, hierarchy, layering), 而网络就是在 layering 中很好的例子, 还有一个是 7 层的 inode, 每一层都担任不同的职能。

Layers in Network

- Application Layer
- End-to-end Layer
- Network Layer
- Link Layer

对于网络来说, 我们见得最多的是 Application Layer。在 ICS 中我们写 socket lab 的时候, 它是一个很高层次的, 我们不需要管底层的实现。

End-to-end Layer 强调的是从一个点怎么找到另一个点, 一个例子就是发快递, 只需要填自己的地址和对方的地址, 但是商品可能从仓库->中转站->集散点->到达。

最后一公里, 我们可以认为是 link layer, 最后一公里可能是骑着电动车的快递小哥, 也有可能是开一辆货车。所以 link layer 在网络里有很多方法, 比如 WiFi、有线以太网、有线电缆、光纤, 这负责点到点的传输, 也有可能是一段段传输在不同的介质中。

Network Layer 就是把 Link Layer 一段段连起来, 找到从一个点到另一个点的最短路径, 但是特殊的点就是找的时候那条路径可能还在, 但是在发包的时候可能就没了。所以 Network Layer 只负责找到一条路, 但是这条路通不通并不保证。

End-to-end Layer 如果发现发包的时候这条路是断的, 那么我们就让 Network Layer 再找一条路。

Application Layer 和 End-to-end Layer 有一些重叠的例子，它可以跳过 End-to-end Layer 直接调用 Network Layer.

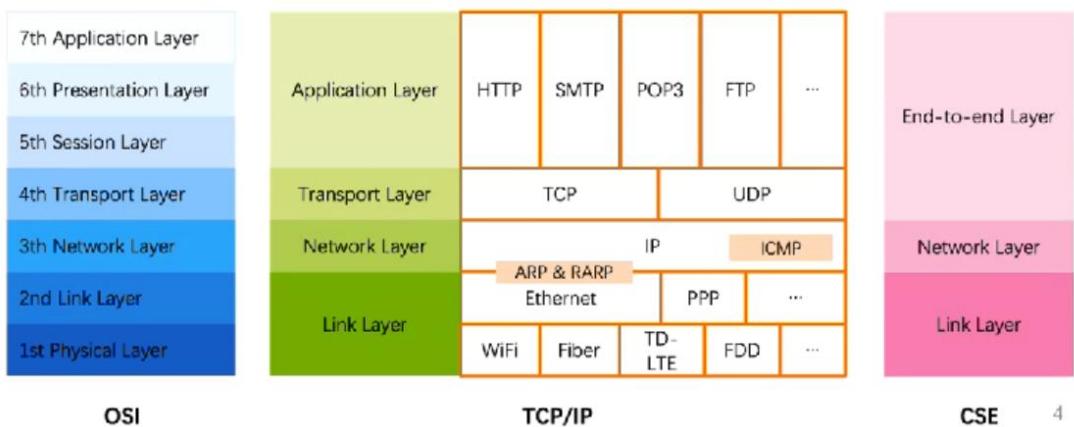
TCP/IP 协议

网络划分最有名的 TCP/IP 协议，它分为 4 个层次，由下至上为 Link Layer（网络接口层）、Network Layer（网络层）、Transport Layer（传输层）和 Application Layer（应用层）。

OSI, TCP/IP & Protocol Stack

OSI

- The open systems interconnection (OSI) model
- 7-layer architecture



所谓 transport layer 就是负责 TCP 和 UDP 等，而应用层负责的就是 HTTP 等。实际上最早的时候 OSI 提出了网络的 7 层模型。在我们课里，我们认为 application layer 和 transport layer 都是端到端之间的协议。

我们可以在 end-to-end layer 中进一步去实现嵌套的 end-to-end layer、network layer、link layer，即网络可以是无限嵌套的。

Q: 什么场景下会有这种嵌套的 end-to-end layer 的应用呢？

A: P2P Network。它就是建立在 internet 之上，但是有自己的路由协议、找节点的方法。比如：磁力链接等要去记录谁有数据，因为数据并不是放在中央的服务器上，可能分散在不同的用户机器上。

The Internet "Hour Glass" Protocols

More people, more useful

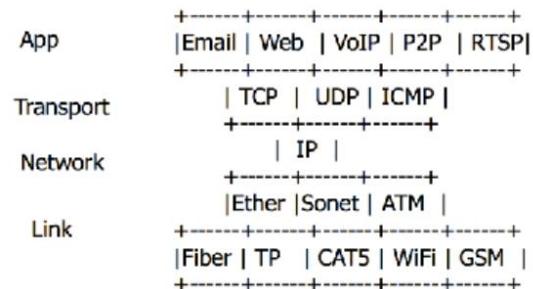
- Value to me = N
- Value to society is N²

Network, dumb vs. smart

- Standardize vs. flexibility

Network is a black box

- Simplify the system that uses it



"Everything over IP, and IP over everything"

ICMP 其实就是 ping 的协议，它是一个优先级非常高的协议。当我们 ping 一台主机的时候是每台机器上的 kernel 响应我们的，而不是我们运行的 application 响应。只要我们的笔记本上跑了个 Kernel，绝大部分 Kernel 都提供了 ping 的 API。所以在不知道的情况下就提供了一个 ping 的服务，这可能是一个很危险的事情，我们的电脑开了一个服务我们缺不知道，并且这个服务还会去响应别人。

我们发现，在底下有以太网，已经有一点点偏传输了，最下面还有 Fiber、光纤等。在这样一个 Protocol 下，它是一个典型的沙漏结构。

Q: 中间是一个很窄的点，为什么是一个沙漏型呢，也就是为什么中间只有一个 IP 呢？

A: 因为网络的价值取决于加入网络节点的数量。它期望的是越多的机器连在一起越好。

一旦有了 IP 之后，它就成为了一个占据了最大优势的协议，把所有的节点连接在了一起，这样我们上层就不用 care 下层到底是怎么连的。

相似的场景我们可以推广到 OS, OS 的下层是千奇百怪的硬件，上层是各种各样的软件，而 OS 就那么几个，成为了一个沙漏的中间部分。这样应用程序就不需要做很多兼容。所以沙漏模型是和生态紧密相关的，现在所说的卡脖子，其实就是卡中间这个瓶颈。比如哪天 IP 协议不让用了，就牵扯范围非常大。

当时在网络层还有一个争议就是 dumb 和 smart 的争议。当时有一些人认为，我们的网络是应该提供可靠的、点对点的可以保证质量（顺序、数据、时间有保证）的传输，这就是一个聪明的网络；另外一些人认为网络只应该发一个包，这个包可能顺序不对、发错了、很耗费时间甚至发错了。

最后选择的是一个非常不可靠的网络，可靠性只有 95%。所以，我们的网络是一个 best-effort (尽力而为) 的网络。它把这些责任全部推给了上层的协议。

Q: 为什么它把协议推给了上层，但是在现实中还是能 Work 呢？

A: 因为有些应用场景下 95% 也是可以接受的，并不是所有需求都是需要 100% 的可靠、数据完全正确。我们做不到 one-fit-for-all，必须做 trade-off。

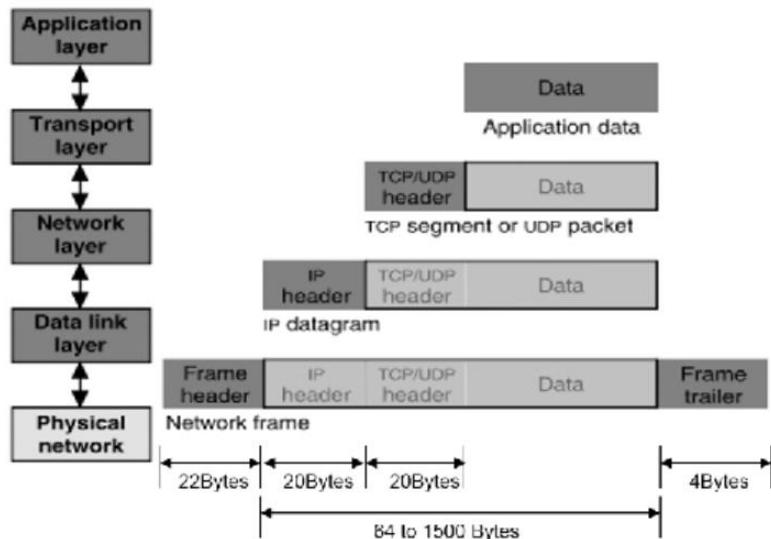
一个最基本的权衡就是传输的时延和传输的质量的权衡：快&质量差或者是慢&质量好。如果此时我们的网络是一个 smart 网络，此时我们是没有选择的。但是在实时通话、直播等场景下，为了时延，是可以牺牲掉一部分正确性的。

所以在 IP 这一层最终选择的是 Dumb 网络，有了这个以后，network 就可以作为一个黑盒来使用了，对于有安全性和稳定性的应用就发展出来了 TCP、对于别的一些应用场景发展

出了别的协议。

那么，这些协议的包长得是什么样呢？

Packet Encapsulation



发到 transaction layer，就会加上 TCP/UDP 的包头，到了 Network Layer 就会加上 IP 的包头，到了 Data Link Layer 就再加上了以太网的包头，这也是我们为什么使用 stack 来表示网络协议，因为它就像一个 stack 一样。

在 OS 实现的时候，data 这个字段有一个 socket buffer，也就是我们在创建 Application Data 的时候预留好这些字段，这样我们在往下传输的时候就不用反复拷贝这些数据了。

最后整个包是有大小限制的，64~1500Bytes。

Q: 为什么我们把数据切分成 64~1500 Byte 的小的包的形式去发送，而不是以流数据的形式一直发送呢？

A: 这也是一个 trade-off。我们包小了，利用率就低了，overhead 就变高了。

Q: 为什么包最小是 64 个 Byte？

A: 在第一节课讲过，我们要做冲突检测。

考虑如下极限的情况，主机发送的帧很小，而两台冲突主机相距很远。在主机 A 发送的帧传输到 B 的前一刻，B 开始发送帧。这样，当 A 的帧到达 B 时，B 检测到冲突，于是发送冲突信号。假如在 B 的冲突信号传输到 A 之前，A 的帧已经发送完毕，那么 A 将检测不到冲突而误认为已发送成功。由于信号传播是有时延的，因此检测冲突也需要一定的时间。这也是为什么必须有个最小帧长的限制。

按照标准，10Mbps 以太网采用中继器时，连接的最大长度是 2500 米，最多经过 4 个中继器，因此规定对 10Mbps 以太网一帧的最小发送时间为 51.2 微秒。这段时间所能传输的数据为 512 位，因此也称该时间为 512 位时。这个时间定义为以太网时隙，或冲突时槽。512 位=64 字节，这就是以太网帧最小 64 字节的原因。

512 位时是主机捕获信道的时间。如果某主机发送一个帧的 64 字节仍无冲突，以后也就不会再发生冲突了，称此主机捕获了信道。由于信道是所有主机共享的，为避免单一主机占用信道时间过长，规定了以太网帧的最大帧长为 1500。

参考：<https://zhidao.baidu.com/question/2139047735544496148.html>

如果我们不切包，整个一起发，那么一旦错了一点点就要整个文件重发，效率低。而如果我们切了太细了，那么导致包头的 overhead 又比较大。所以 64~1500 Byte 也是一个经验的数据。

Application Layer

Entities

- Client and server
- End-to-end connection

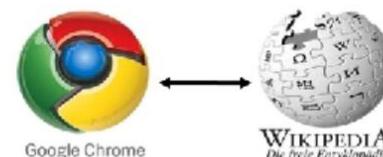
Name space: URL

Protocols

- HTTP, FTP, POP3, SMTP, etc.

What to care?

- Content of the data: video, text, ...



```
<html>
  <head>
    <title>Google</title>
    <script>window.google=....</script>
  </head>
  <body> ... </body>
</html>
```

对于 Application Layer 来说，我们可以自己定义一套 application 的 protocol。对于网页来说，它要考虑 data 的内容是什么。

Transport Layer

Transport Layer

Entities

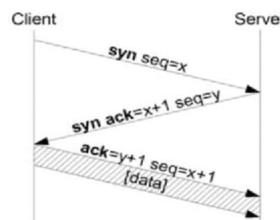
- Sender and receiver
- Proxy, firewall, etc.
- End-to-end connection

Name space: port number

Protocols: TCP, UDP, etc.

What to care?

- TCP: Retransmit packet if lost
- UDP: nothing



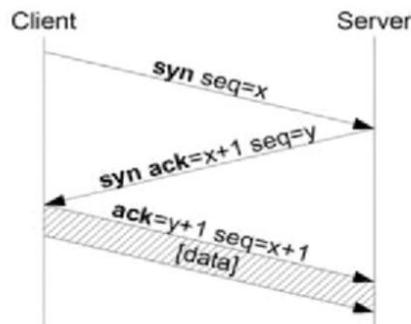
而 Transport layer 需要考虑去识别不同的 session，比如我们的服务器既服务了同学 A 和同学 B，我们在响应的时候怎么保证同学 A 和 B 收到了自己所对应的包。那么我们就要加一个 namespace，也就是 port。当我们连到一台服务器的时候，大家通过 port 和 application 连接，所以在一台笔记本上，我们就可以同时发起 65535 个请求。因为一台机器只有 1 个 IP，

我们就可以使用 port 来做扩展。

这一层还需要考虑的是我们的数据丢了怎么办、数据的顺序乱了怎么办。所以需要使用典型的三次握手。

TCP Segment Header Format							
Bit #	0	7	8	15	16	23	24
0	Source Port			Destination Port			
32	Sequence Number				Acknowledgment Number		
64	Data Offset		Res	Flags	Window Size		Urgent Pointer
96	Header and Data Checksum			Options			
128	Header and Data Checksum				Options		
160...	Options						

TCP 协议的包头会比较复杂，这里有一个 sequence number 和 ack number。



也就是传过去的时候，client 会给 server 一个 sequence，也就是告诉 server：“接下来会传输给你一个包，这个包的编号就是 x”，Server 收到以后会返回 $ack=x+1$ ，并且把自己的一个 y 作为 seq 返回给 client。下一次 client 发送的时候就继续发送 $ack=y+1,seq=x+1$ 和自己想发送的数据。

如果一共发送了 1000 个包，那么 seq 就会被加 1000。这样的好处就是，我们作为旁边一个人看到两个人在发消息，我们想冒充其中一个人来发消息，但是我们猜不到 sequence number 是多少，这就很难。所以 sequence number 起到了避免旁边人冒充的功能。

UDP Datagram Header Format							
Bit #	0	7	8	15	16	23	24
0	Source Port			Destination Port			
32	Length				Header and Data Checksum		

而对于 UDP 来说，头非常简单。发数据就不需要考虑 sequence number。所以可靠性并不高，明明发了 10 个包可能服务器收到了 12 个包。

Network Layer

Network Layer (the Internet Layer, IP Layer)

Network entities

- Gateway, bridge
- Router, etc.

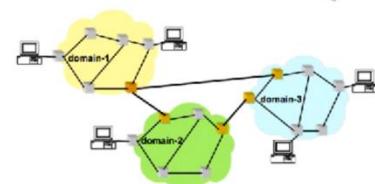


Name space

- IP address

Protocols

- IP, ICMP (ping)



What to care?

- Next hop decided by route table

到了 Network Layer 后，我们看到了就一台台这样的设备了。我们要考虑怎么去做路由，怎么去找段线。IP 这个 Layer 中，ICMP 可以认为是在 transport layer 和 network layer 的中间。

Network Layer 需要关注的是网关、网桥和路由器，整个 Internet 就是建立在很多个路由器互联的基础上的。那么我们该怎么知道谁在哪、我要发给谁、www.baidu.com 的 IP 在什么地方。我们就需要用到路由表，这些 IP 设备最核心的功能就是维护路由表。

0	4	8	16	19	31
Version	IHL	Type of Service	Total Length		
Identification		Flags	Fragment Offset		
Time To Live	Protocol		Header Checksum		
Source IP Address					
Destination IP Address					
Options		Padding			

Header

包核心的部分就是 source ip address 和 destination ip address。一个网卡上可以设置很多个虚拟 IP。所以 IP 这个概念就是一个逻辑上的概念，并不会直接地和某个设备绑定。

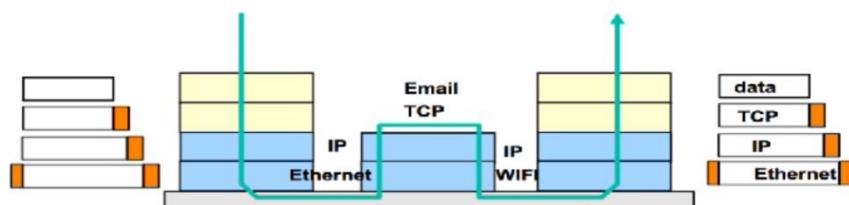
TCP/IP Architecture

Each layer adds/strips off its own header

Each layer may split up higher-level data

Each layer multiplexes multiple higher layers

Each layer is (mostly) transparent to higher layers



其实我们要把数据从 A 发到 B，就是先加一堆包头放到网络上，中间经过一些转发，然后到 B 了一个逐层解包。

Link Layer

目标就是从一个点到另一个点传输数据。它有几个目标。

1. 在物理上，它最底层的传数据，是真的传输数据的节点。
2. 一条线怎么被多个人复用。
3. 怎么把数据变成实实在在的编码的传输信号。
4. 找到传输过程中的错误来避免重传。
5. 对上提供 interface。

我们先从计算机内部传输数据的角度看起。假设我们现在要把数据从一个寄存器挪到另一个寄存器。比如我们在 CPU 里要把%eax 复制到%ebx，那么两个寄存器中一定有一些线连在一起。

我们把一个个 bit 放到线上，等到每根线上的信号稳定了，并且信号传递到目标寄存器了。那么目的寄存器就可以在下一个时钟的上升沿读到这个。那么它怎么知道什么时候要读数据呢，是要通过时钟来告诉它的。

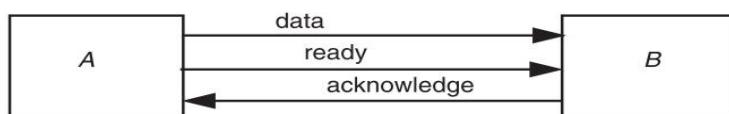
问题在于：如果没有一个时钟，我们该怎么样去发送这个数据。因为在两台计算机中存在一个共有的时钟是比较困难的事情。

我们可以换成这样一个问题，如果交大没有时钟，那么我们该怎么保证大家在一个教室中上课。我们只能一个个找。

数据并行传输(Physical Transmission without Shared Clock)

所以没有全局时钟的情况下，我们需要 ready 和 ack 的协议。我们先把数据（要传输的内容）放在这根线上，等我们把数据放好了，我们设置一根线叫做 ready（我们已经把数据放置好了）。B 一看到 ready 设置成了 1，那么就知道 data 就可以读了，如果 ready 没有设置成 1，那么我们的数据可能就处于 all-or-nothing 的中间状态。

这样 B 就可以把数据读走，然后 ACK 设为 1，那么 A 就知道可传下一个数据了，它先把 Ready 设置成 0，B 就知道要传下一个了，此时就把 ACK 设置为 0。然后继续传下一组数据。

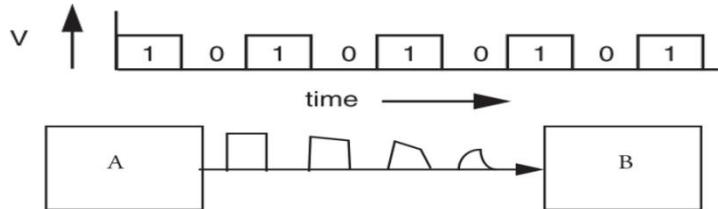


通过 ready 和 ack，a 和 b 之间就可以去做交互，这就比较适合在两台计算机去做。假设传输时间是 Δt ，那么我们每传一个 bit，就需要 $2\Delta t$ 时间，最快传输速率就是 $\frac{1}{2\Delta t}$ ，为了加快速度，我们可以把数据线加粗，比如从 1bit->32bit->64bit，这就是加粗总线，也就是 Parallel Transmission。

但是数据在并发传输的过程中，很难做到大家一样快，并且线和线之间会存在一些干扰导致传输速度上不去。

数据线性传输

这就发展出了线性的传输，用的最多的就是 USB。通用串行总线，真正传输的是其中的两根线。它就是一根线传信号，而不是 64 根线。



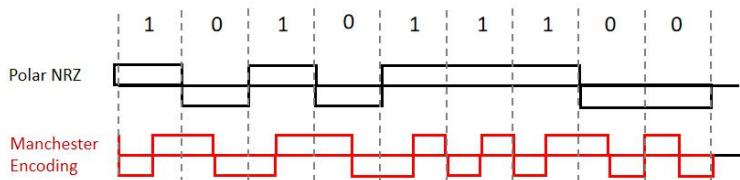
Q：一根线怎么做到传输准确度和效率都高呢？我们通过电压高低来表示 0 和 1，可能会随着距离衰减。当信号一旦发生衰减了之后，B 就不知道该怎么样解释这个信号，如果还有时钟信号，那么我们上升沿采集一下即可。我们现在只有一根线，只有 0 和 1 的变化采样率是什么让 B 很纠结。

Q：没有 Clock 的情况下如何解释信号（在信号会衰减的情况下）。

A：我们要提到一个神奇的组件：压控振荡器（voltage controller oscillator）。它的效果是当有 01 变化的时候，只要 01 变化是有节奏的，它可以还原出时钟信号。

Q：但是它有一个非常大的弱点，因为它之所以能恢复出时钟，是因为我们的信号在不断发生变化，但是我们传 100 个 1 或者 100 个 0，就无效了。

A：曼彻斯特编码。我们可以做一个编码，把 0 变成 01，把 1 变成 10，这样连续的数据也是会发生变化的，最多产生 0110 或者 1001。曼彻斯特编码等价于把时钟信号和数据二合一是，变化率可以体现在数据传输过程中，但是缺点是数据利用率只有一半了。



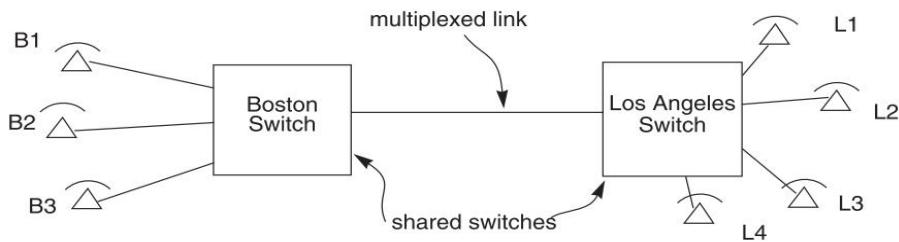
接下来，我们一根线怎么让多个节点用这根线去传数据。

这里就有两种基本模式，分为同步模式和异步模式。

一根线传数据的同步模式

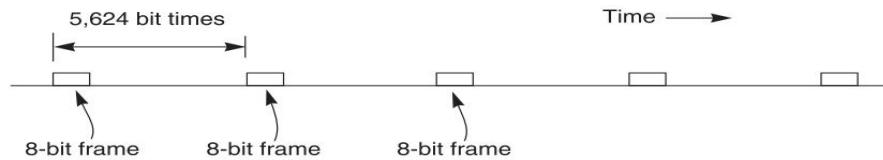
在同步模式中，有一个 connection 的概念，可以建立连接(setup)和撤掉连接(tear down)。数据是通过流的方式进行同步传输。

我们来看一个例子，老电影是转盘打电话，接线员在帮忙转接。这个过程就是建立连接的过程。接线员在插拔接头，这个连接是专属于你的。



在我们计算机中，可以认为存在两个 switch，当有人要打电话的时候就会产生一个虚拟的信道，也就是在这根线上给你预留一段时间。

比如线的传输能力是 45Mb/S，电话需要的是 64Kbps。我们怎么才能够建立连接呢？



我们把通话需要的 64K 在每秒内均匀划分成 8 个 bit 的 frame。那么每秒就是 8000 个 frame。每来一个人来我们在 45Mbps 中再分配 8000 个。我们一共可以支持 $45 * 1024 / 64 = 703$ 个人同时打电话。

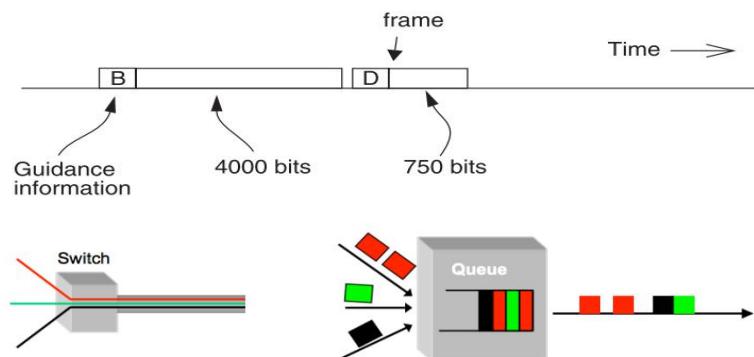
Q：分的时间片都是离散的，为什么电话听起来还是连续的呢？

A：因为播放的时候也是一点点放出来的。

当我们没说话的时候，我们也是占用着 8000 个的，这就是建立连接的过程。如果第 704 个人，会直接 reject 掉（请稍后重播）。这就是同步连接：不管我们什么时候说话，一定能把消息传递过去。

Data communication network (包交换网络)

说一句话打包变成一个 package 发送过去。这个过程没人预留 frame，而是发出去就占用，不发就不占用，好处就是资源利用率是很高的，不用就不会占据资源。缺点就是不能保证时延，可能被排在很后面，这在实时通话场景中是比较不好的，但是我们现在都在用这个。



一根线可以来自多个人发包，多个人来的时候就需要排队，有一个队列一个个发。

Q：既然我们要去做发包操作，我们怎么确定这个数据的头和尾呢？如果是同步方法，我们不需要费力去区分数据，但是现在我们怎么决定数据的头和尾呢？因为现在都是 01 怎么办呢？

A：我们可以设置一个 magic number，比如连续 7 个 1 就是代表着开始，数据中连续 6 个 1 我们就加一个 0（转义数据中的 magic number）。在读的时候，我们可以读到 6 个 1，发现

后面是一个 0 就删除掉这个 0，这样就不会出现数据中连续 7 个 1 的情况。

Q: 如果出错了怎么办？

A: 我们知道容错一定需要冗余，我们需要在最后加一个 checksum，如果 checksum 不对，要么包错了要么 checksum 错了。

如果错了我们现在只能丢掉包。我们能不能恢复呢？

海明距离

定义：从一个 number 变到另一个 number，需要反转多少个 bit。

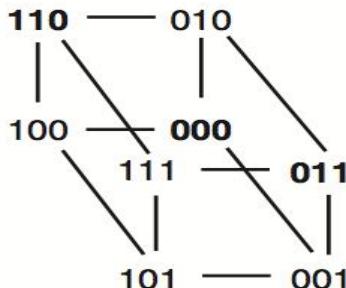
举个例子：100101 和 010111 的海明距离是 2。我们可以把这两个数字做一个异或，然后数一数有多少个 1。

2bit->3bit 做奇偶校验

我们就可以去做一个简单的校验，可以把 2 个 bit 编码成 3 个 bit 从而检测一个 1bit 的错误。

比如把 00->000, 11->110, 10->101, 01->011。

原先我们只有两位，00, 01, 10, 11 它们互相之间的海明距离只有 1。如果有 bit 位发生了翻转，我们是不知道的。



而我们现在变成了 3bit，我们就发现整个编码空间中 8 种方式中，只有 4 种是合法的，任何一个 bit 发生了翻转就会到海明距离为 1 的编码上去，落到的位置都是非法的编码。

Q: 那么既然我们可以检测发生的错误，我们能不能恢复发生的错误呢？

A: 既然我们有了海明距离，两个合法数字的之间海明距离为 2，我们把合法海明距离扩展到 $\begin{smallmatrix} 1 & 000 & 1 & 000 & 1 \\ \text{合法} & \text{合法} & \text{合法} \end{smallmatrix}$ ，这样合法编码的海明距离为 4。如果我们发生了一个 bit 的反转，到

了 $\begin{smallmatrix} 1 & x00 & 1 & 000 & 1 \\ \text{合法} & \text{合法} & \text{合法} \end{smallmatrix}$ ，那么我们就可以让 x 就近还原到最近的 1。如果发生了 2 个 3 个 bit 的翻转就不行了。

4bit->7bit 自动纠一位

Example-2: 4-bit -> 7-bit

4 bits \rightarrow 7 bits (56 using only extra 7)

- 3 extra bits to distinguish 8 cases
- e.g. 1101 \rightarrow 1010101

$$\begin{aligned}P_1 &= P_7 \oplus P_5 \oplus P_3 \\P_2 &= P_7 \oplus P_6 \oplus P_3 \\P_4 &= P_7 \oplus P_6 \oplus P_5\end{aligned}$$

Correct 1-bit errors

- 1010101 \rightarrow 1010001 : P1 & P4 not match
- 1010101 \rightarrow 1110101 : P2 not match

1	2	3	4	5	6	7
1	0	1	0	1	0	1
1	0	1	0	1	0	1
1	0	1	0	1	0	1

Not Match	Error
None	None
P1	P1
P2	P2
P4	P4
P1 & P2	P3
P1 & P4	P5
P2 & P4	P6
P1 & P2 & P4	P7

33

我们来看一个很有意思的编码方式，可以自动纠一位。它的思路其实就是增加合法和非法之间的海明距离。一个 4 位的数可以变成一个 7 位的数。 $1101 \rightarrow 1010101$.

对于 1010101 来说，如果发生了 1 个 bit 的翻转，得到了 1010001。我们会发现等式 P_1 和 P_4 是不成立的，因为这是第 5 位发生了翻转，所以包含第 5 位的等式都不成立了。我们就知道这一位有问题了。我们查表，P1 和 P4 不成立，说明第 5 位不成立。

通过这种编码，我们可以在这个三个等式成立或者不成立，也就是 8 种情况，如果都成立那 OK，如果 P1P2P4 都不成立，那么 P7 对应到 3 个等式都不成立的情况。当我们受到数据的时候，可以重新算一个 3 个等式，我们就可以倒推出哪一位出错的情况。

Q: 为什么不把红的 4 位放前面呢？

A: 看这个表 $3=1+2$, $7=1+2+4$ ，我们根据这个出错的位置就可以算出来第几位出错了。理论上把 3 位红的放后面，也是可以的，但是写代码麻烦。

如果错两位会怎么样？P1P2 都错了，但是我们认为是 P3 错了，这种情况对于海明编码来说是无能为力的，我们有两个选择，可以进一步扩大海明编码，比如 4bit->9bit 让它纠 2bit。

真的所有人都 care 纠错能力有多强吗？在 ip/tcp 这一层都有 checksum。所以我们会发现整个数据传输的容错能力最强的是用户态 check 哈希，底下都不用做，但是在网络的不同层次都有容错的方式。并不能保证 100% 的安全性，如果我们要更安全我们就在不同层再加。

2021/11/30

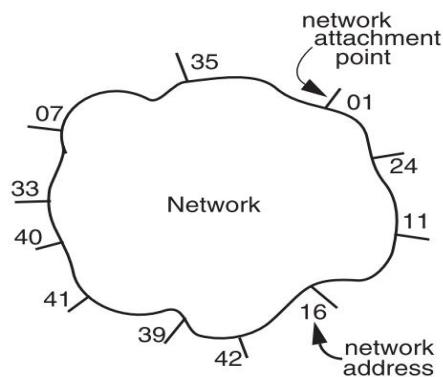
今天我们来讨论网络层，它是 CSE 分类中的第二层。上一节课我们介绍了 link layer，它和 tcp/IP 中的 link layer 是一样的，包含了 WiFi、光纤、手机基站。这些都是一个点到另一个点之间的连接，这都是直接相连的两个点。我只关心从一个点到另一个点怎么传输数据，

没有时钟我们就可以使用曼彻斯特编码来传输数据。

包括发生错误的时候，我们该怎么去纠错，我们可以通过海明编码把 4 位到 7 位，从而我们可以去纠一位错。所以合法的值的海明距离要足够大，如果海明距离为 1，我们就不知道是否出现了位的偏转。手机号打错一位就错了，但是身份证号码如果输错了一位，大概率是不能通过校验的。

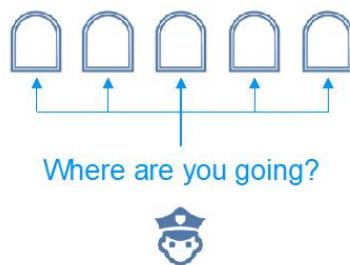
但是纠错能力再强，我们要区分出检验一位错误和纠一位错误的所需要的冗余是不同的。奇偶校验可以 **detect** 一位的错误，但是我们如果要纠错的话就要用到海明距离 $4\text{bit} \rightarrow 7\text{bit}$, **overhead** 一下就扩大了 75%。纠错的好处就是不用重传数据了。

当我们能在两个节点之间传输数据了之后，那么我们怎么用这一段段数据来拼成一个网络层呢？这就是我们常用的 IP 协议，它是 **best-effort network**，如果找不到就丢掉。如果我们在网络这一层来做 **absolute guarantee** 就很复杂，底层还是 **best-effort** 即可。

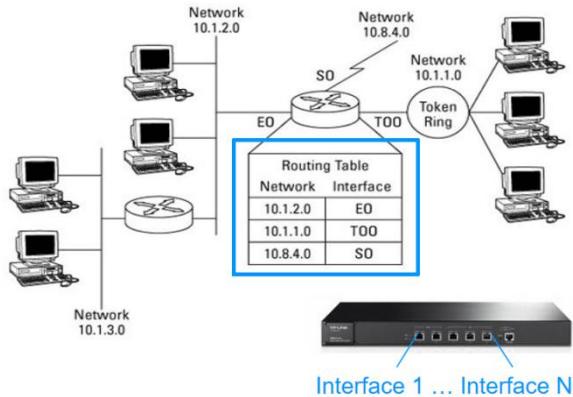


对于网络层来说，它对外有一些 **attachment point** (AP)，当我们接到网络的时候，我们要找到 AP，把设备通过 AP (WiFi、网口) 接入网络。AP 对应了一个数，就是一个网络地址，一旦知道了网络地址，数据就通过 AP 来发送传输。主要是发包接口和收包接口 (**network_handle**)。在网络中，我们不仅仅是一个终端节点，还有可能是中间节点去做一个 **forward**，这就是我们说的路由器，对于路由器来说，大量的包只是中转一下。

那么既然说到中间有一个中转的过程，**routing** 就是中转的过程。既然我们有一个目标和起始地址，网络层怎么找到这条路径呢？

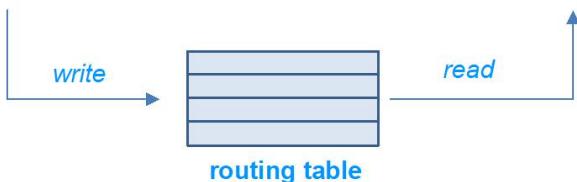


其实一个机器只要插了两个网卡（一个内网、一个外网）就可以做一个简单的路由器。路由器通常有很多个网口。它的本质就是插着很多网卡的电脑。高端路由器非常贵，因为在处理的时候对吞吐量要求非常高。



整个路由器是怎么工作的呢？其实非常简单。每一个 **router** 都有多个网口，它核心的数据结构就是一个 **routing table**，记录每一个口对应了哪一个网络。有人认为这上面都不需要跑 OS，只需要跑网卡驱动和数据结构即可。不过有了 OS 之后，我们可以做防火墙、流量管控等。

在这里我们就要讲控制面和数据面的分离，这是一种经典的分离方式。



在控制面上，它做的事情就是去做如何定义这个规则。也就是包从一个口进来之后，从哪个口出去。用到的就是一些路由的算法，它最后要产生这张表。比如我们有一个网络包要去 LA，这个路由表该怎么记录呢？它显然不能记住全世界每个国家的 **destination**。

对于数据面来说，它不 care 表是怎么生成的，它只负责把数据从一个网卡 **copy** 到另一个网卡，主要做的事情就是查表+转发，它和 **performance** 相关的一个需求。一个数据进来到出去可能就几千个 **cycle**。所以整个查表算法都需要在几千个 **cycle**。注意，访问主存大概是一两百个 **cycle**。所以对于一个包，我们只能访问 10 次内存，性能要求非常高。所以关心的就两件事情，**control plane** 实现更好的算法来提升容错性，**data plane** 就是不断压榨 CPU 和硬件的性能来降低转发的 **cycle** 数量。

Routing

我们先来讲 **global** 的 routing，整张 Internet 网是怎么组成的，我们是怎么找到去别的国家的服务器的网路上的通路的。

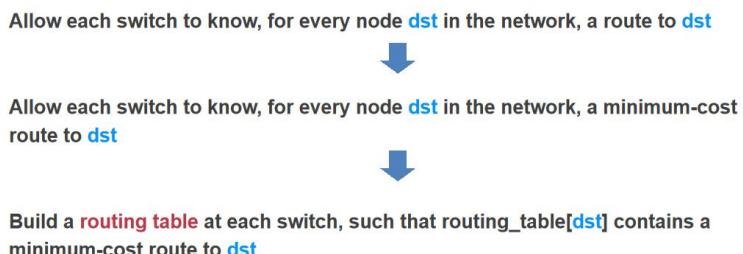


图 3 路由协议的目标

总体来说 routing 的协议就是让每个交换的节点知道该怎么走能到 destination，希望尽

可能找一条 cost 最小的路径。注意，这并不是距离最小的路径。

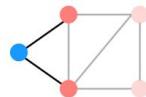
Q: 怎么访问 `github` 最快呢？

A: 最快的路径可能是我们先到阿里云，再通过专用通道到美国的阿里云去访问 `github`，这可能在距离上多一点，但是速度会更快。

所以这并不是完全依赖距离。然后，完成之后我们就要构建出 `routing table`。注意，每时每刻都有不同的服务器 `crash`、网络的拓扑结构在不断变化，所以路由表也要变化。

分布式路由的三个步骤

1. Nodes learn about their neighbors via the **HELLO protocol**



2. Nodes learn about other reachable nodes via **advertisements**



3. Nodes determine the **minimum-cost** routes (of the routes they know about)



首先，我们要知道 `neighbour` 是谁，当我们知道 `neighbour` 是谁之后，我们就可以把自己的 `neighbour` 告诉我们的 `neighbour`。这样就可以进一步告诉别人，这一步就是 `advertisement`。

第三步就是从广告中算出一个最短开销的路径。

这里我们讲两种常见的路由协议：

Protocol 1: **Link-state**

- A node's advertisements contain a list of its neighbors and its link costs to those nodes
- Nodes advertise to everyone their costs to their neighbors
 - via *flooding*
- Integrate using **Dijkstra's shortest path algorithm**

Protocol 2: **Distance-vector**

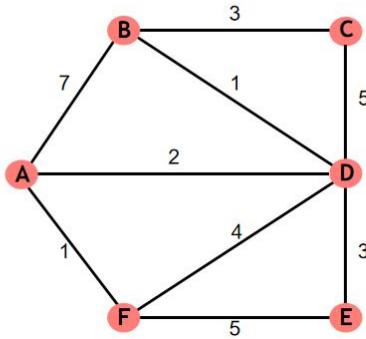
- Nodes advertise to neighbors with their cost to all known nodes
- Update routes via **Bellman-Ford** integration

Link-state 路由协议

我们以下面这个具体的例子为例。

这是一个上帝视角观察的网络。

link-state routing: disseminate full topology information
so that nodes can run a shortest-path algorithm



我们以 a 的视角到每一个 neighbour 的 link cost。

A's advertisement: [(B,7),(D,2),(F,1)]

A 就会把这个广告放到自己的名片里。

A 不断往外发这一条广告，B,D,F 作为邻居收到 A 的广告，然后进一步地 flooding（广播）给所有人，这样所有人都知道 A 认识 B,D,F。

当 C,E 收到了 A 的广告之后，也会 flooding 传回给 B,D,F，但是 B,D,F 看到自己已经帮 A 发过一次了，所以这轮就停了。

一旦发完之后，每个节点都知道另外几个节点是怎么连接的。接下来我们就用 dijkstra 算法找到最短路径即可。

dst	route	cost
B	A-B	7
C	?	∞
D	A-D	2
E	?	∞
F	A-F	1

Link-state Routing

Keep track of W, the set of nodes haven't processed yet

- Initially, W is all nodes in the network

Keep track of the current costs and routes to all of the nodes. Initially:

- $\text{routing_table}[\text{self}] = \text{Self}$; $\text{routing_table}[\text{anyone else}] = ?$
- $\text{cost_table}[\text{self}] = 0$; $\text{cost_table}[\text{anyone else}] = \text{infinity}$

While W is not empty:

- 1. $u =$ the node in W we have the minimum cost to so far
- 2. Remove u from W
- 3. For every neighbor v of u :
 $d = \text{cost_table}[u] + \text{cost}(u, v)$
if $d < \text{cost_table}[v]$
 $\text{cost_table}[v] = d$
 $\text{routing_table}[v] = \text{routing_table}[u]$

这个就是具体的 **dijkstra** 算法，我们简单回顾一下。W 集合：还没被处理的节点。我们从 W 中选一个 cost 最小的 u ，我们根据 u 的权重来更新 cost table 和 routing table。

Distance-Vector 路由协议

问题是不需要让全网的人都知道我们有什么 neighbour，这个的开销是非常大的。所以我们要引入 **distance-vector routing**。

Distance-vector Routing

In link-state, nodes calculate full shortest paths. But actually they only need the route (first-hop) to a destination

Advertisement format: Each node's advertisement is a list of all the nodes it knows about, and its current costs to those nodes

- Initially, this advertisement is just $[(\text{self}, 0)]$.

Nodes who receive an advertisement: Node X's advertisement will be received only by its neighbors

Integrate step: When node X receives an advertisement from its neighbor Y, this advertisement will be a list of $[(\text{dst}, \text{cost})]$ pairs. Each cost represents Y's cost to dst

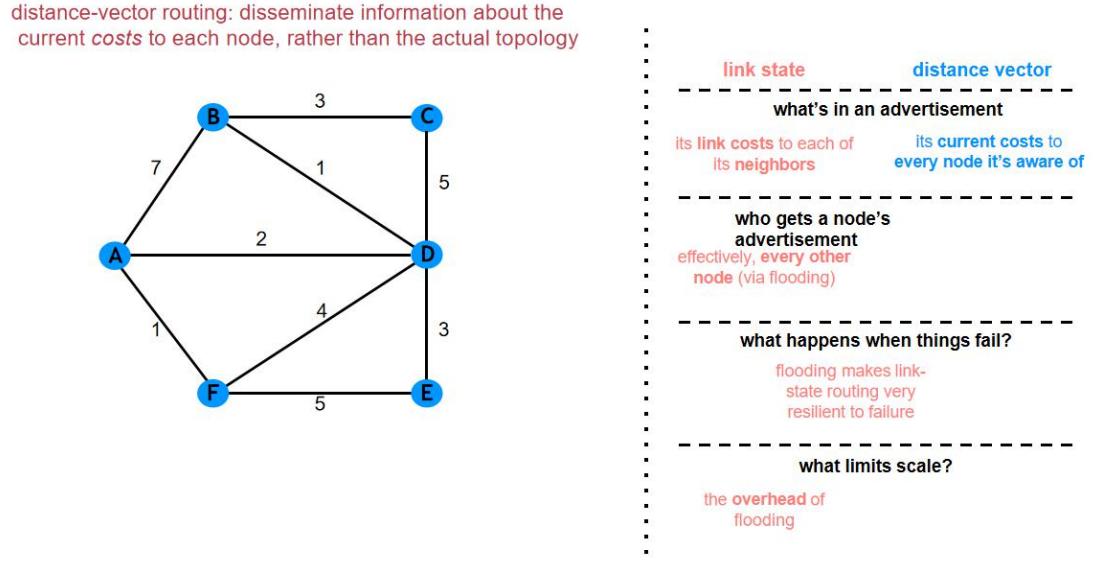
首先在 link state 协议里面，路由器实际上它们只需要 first-hop，因为我们路由表只记录从哪个口出去，我们不用考虑更多的事情。

所以在 distance-vector 协议中，它每个节点的 **advertisement** 是一个列表，也就是能够 reach 到的节点，而不仅仅是我们的 neighbour 是谁。一听它要放一个列表，我们就知道所以这个算法需要迭代，因为一轮做不完这件事情。

For each (dst, cost) in the advertisement, X needs to check for two things:

- If X is already using Y to get to dst, update the cost information (remember, costs can change!)
- If X is not already using Y to get to dst, see if Y could provide a better path; if so, update the routing and cost information

对于每个 destination cost, 需要检查两件事情。



首先 A 把自己知道的 distance vector 发给邻居节点。

A's first advertisement: $[(B, 7), (D, 2), (F, 1)]$

邻居 B,D,F 收到 A 的消息之后, 更新自己的 distance vector, 所以发送自己的 advertisement 给自己的 neighbour。

B's first adv: $[(A, 7), (C, 3), (D, 1)]$

D's first adv: $[(B, 1), (C, 5), (E, 3), (F, 4)]$

F's first adv: $[(A, 1), (D, 4), (E, 5)]$

所以对于 A 来说就收到的如上的 3 个消息, 然后把对应的数据更新到路由表中。

A's routing table

dst	route	cost
B	A-D	3
C	A-D	7
D	A-D	2
E	A-D	5
F	A-F	1

A's second adv:
 $[(B, 3), (C, 7), (D, 2), (E, 5), (F, 1)]$

A will learn about the correct min-cost path to C in the next round of advertisements; try that out for yourself!

所以 A 还要继续去发 advertisement 给邻居节点。注意现在这个表 C 的位置还不是最优的。

link state	distance vector
what's in an advertisement	
its link costs to each of its neighbors	its current costs to every node it's aware of
who gets a node's advertisement	
effectively, every other node (via flooding)	only its neighbors
what happens when things fail?	
flooding makes link-state routing very resilient to failure	failures can be complicated because of timing
what limits scale?	
the overhead of flooding	failure handling

注意这是一个去中心化的。每个人向周围发 advertisement 即可。一旦加入到社区，并不需要任何人同意，它只需要向周围人发 advertisement，就可以进其他人的路由表中。去中心化就会带来一些问题，比如我们要新加一个路由器，和所有人说到某些 destination 非常快，那么它就会成为一个中心节点，这就是 BTB 劫持。

INFINITY

A sends advertisements at t=0, 10, 20,..; B sends advertisements at t=5, 15, 25,..



A: Self, 0	A: B->A, 1
B: A->B, 1	B: Self, 0
C: A->B, 2	C: B->C, 1

t=9: B<->C fails

B 和 C 中间网络这条线断了，会发生什么问题？一旦在 t=9 的时候断了，B 就会发现连不通了，设置为到 C 的距离为 inf。等到 t=10 的时候，A 给 B 说认识 C。

INFINITY

A sends advertisements at t=0, 10, 20,...; B sends advertisements at t=5, 15, 25,..



A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 2 C: None, inf

t=9: B<->C fails

A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 2 C: B->A, 3 (2+1)

t=10: B receives the following advertisement from A:
[(A,0),(B,1),(C,2)]

A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 4 C: B->A, 3

t=15: A receives the following advertisement from B:
[(A,1),(B,0),(C,3)]

A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 4 C: B->A, 5

t=20: B receives the following advertisement from A:
[(A,0),(B,1),(C,4)]

continues until both costs to C are INFINITY

其实它们没有一个人可以到 C。最主要的原因是我们只传了认识谁，而不是说是因为谁认识到的 C。如果 B 知道 A 是因为 B 认识 C 的，这就可以避免这种事情。

这时候就要用到 split horizon。

Split Horizon

A sends advertisements at t=0, 10, 20,...; B sends advertisements at t=5, 15, 25,..



A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 2 C: None, inf

t=9: B<->C fails

A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: A->B, 2 C: None, inf

t=10: B receives the following advertisement from A:
[(A,0)]

A: Self, 0 A: B->A, 1
B: A->B, 1 B: Self, 0
C: None, inf C: None, inf

t=15: A receives the following advertisement from B:
[(B,0),(C,inf)]

split horizon takes care of this particular case

link-state 的好处：收敛很快；缺点就是 flooding，需要发 $2 * \text{node} * \text{connection}$ 数量的

advertisement。

Distance-vector 好处：只需要发 2^* connection 个 advertisement，缺点就是最长的路径和收敛时间存在相关性，还有 infinity problem。同样它也只适合来做小的网络。

所以怎么 scale routing 呢？

3 Ways to Scale

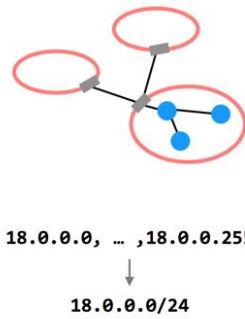
Path-vector Routing

- Advertisements include the path, to better detect routing loops



Hierarchy of Routing

- Route between **regions**, and then within a region



Topological Addressing

- Assign addresses in contiguous blocks to make advertisements smaller

1. **path-vector:** 我们把 path 也作为一个 vector，advertisement 中传的信息多一点，并不会花太多时间。也可以避免 infinity 问题。
2. **Hierarchy of routing:** 我们不能要求每台机器上都记录全世界的所有主机的路径，我们可以让到美国的路由只有一条路。
3. **Topological addressing:** 压缩连续地址，来做路由表的压缩。

核心就是路由表不能太大，否则就不能在 2000 cycle 做完查表操作。

Path Vector Exchange

Each participant maintains a path vector

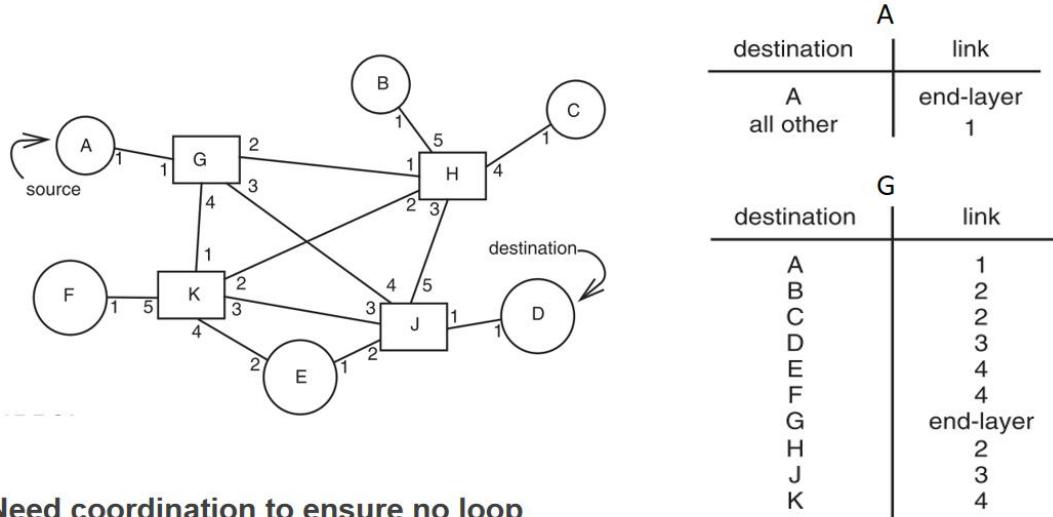
- A complete path to some destination
- E.g., zero-length path to itself
- Gradually learns about other paths
- Construct a new forwarding table from its new path vector

Algorithm

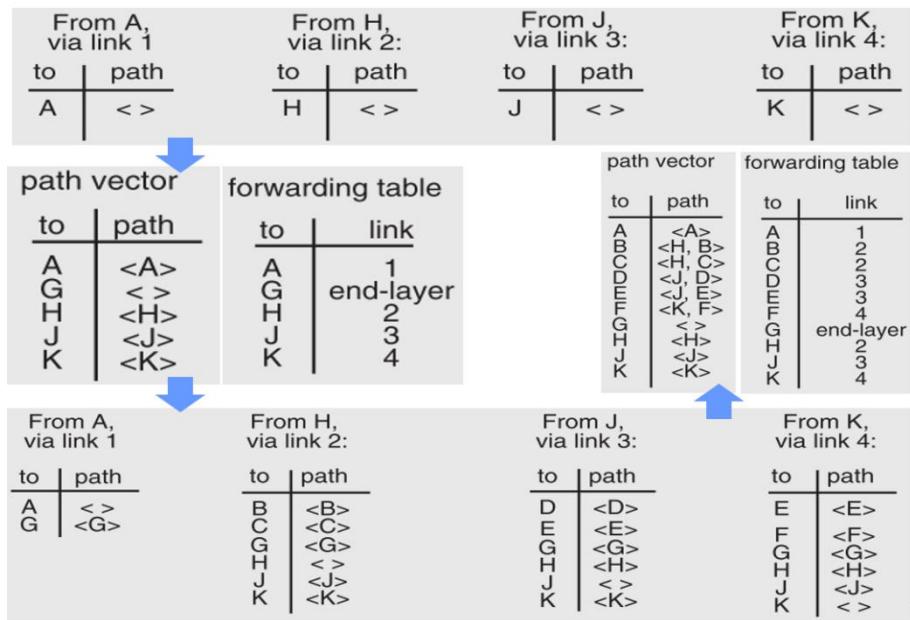
- Advertising
- Path selection

to	path
G	< >

Path Vector Exchange



我们以 G 为核心，它有 4 个网口，也就是 4 个邻居。



Path vector 整个 path 都是全的，不会出现 infinity 问题。有了 path vector 之后，scaling 是更好的，因为 path 信息更全，导致收敛速度降低。

Questions on Path Vector

How do we avoid permanent loops?

- When a node updates its paths, it never accepts a path that has itself

What happens when a node hears multiple paths to the same destination?

- It picks the better path

What happens if the graph changes?

- Algorithm deals well with new links
- To deal with links that go down, each router should discard any path that a neighbor stops advertising

Hierarchical Address Assignment & Routing

Two problems of the path vector implementation

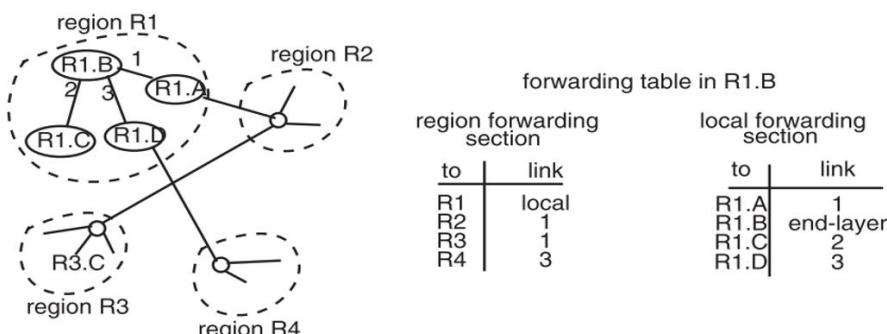
- Every attachment point must have a unique address
- The path vector grows in size with the number of attachment points

Hierarchy for better scalability

- Two parts of network address: region & station, e.g., "11,75"
- Regions correspond to the set of closely-connected entities
- E.g., region-11 has **only 1 entry** in other region routers' table

region 就是替代具体的机器，比如 region11, number 75，这样我们就先去找到 region 11，再去寻找其中的第 75 个机器。这样就大大降低了路由表的大小。

Hierarchical Address Assignment & Routing



Region is also known as **AS: Autonomous System**

所以对于 R1.B 来说，只需要记录两个 table，一个是 region 级别的，还有一个是内部的

region 内的路由表。

Hierarchical Address Assignment & Routing

Problems introduced by hierarchy: more complex

- Binding address with location
 - Has to change address after changing location
- Paths may no longer be the shortest possible
 - Algorithm has less detailed information

More about hierarchy

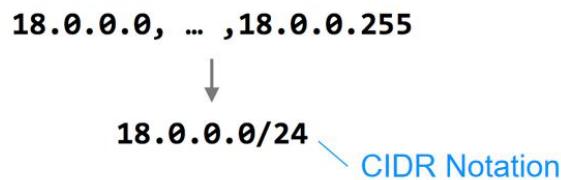
- Can extend to more levels
- Different places can have different levels

我们的 IP 地址为什么要因为我们的地理位置发生变化呢？从最最直观的角度来说，物理地址和 IP 地址应该是没有关系的，但是我们为了降低路由表的大小，我们才选择把 IP 和地址绑定在一起。

Topological Addressing

Further reduce the routing table

- Despite being between regions, BGP still routes to IP addresses (e.g., to 18.0.0.1, not to *region-3*)
- Addresses are given to regions in **contiguous blocks**, so that they can be specified succinctly via a particular notation ("CIDR" notation)
 - CIDR: Classless Inter Domain Routing
- Keeps advertisements small



98

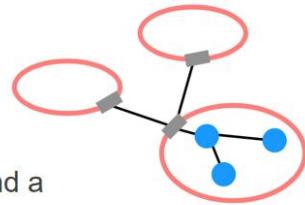
18.0.0.0/24 这个 24 代表 24 个 1，也就是子网掩码。如果我们要在路由表中表示 18.0.0.0 到 18.0.0.255。其实我们在表示 18.0.0.xxxx。所以我们的子网掩码就是 255.255.255.0。我们把地址和子网掩码做一个 and。这样我们就得到了 18.0.0.0/24（子网掩码开始有 24 个 1），注意子网掩码不会在中间出现 01 间隔的情况。

在我们的路由表中，就可以写成

网口	路由表规则
1	18.0.0.0/24
2	18.1.0.0/22
3(default)	0.0.0.0/0

我们拿到 ip 先和掩码做一个 **and**，如果结果和 18.0.0.0 匹配，那么就从 1 号网口发出去。所以如果我们实验室访问，那么就直接根据 1 和 2 网口，内部转发即可。如果是一个乱七八糟的 IP，那么我们就要通过 3 号的外网路由出去。这样就可以减少路由表的大小。

Routing Hierarchy



Across Region

- Use one routing protocol to route **across regions**, and a different protocol to route **within regions**
- Implies that there are devices on the edge of each region that can "translate" between or "speak" both protocols

BGP is the path-vector protocol used across regions

- Border Gateway Protocol

BGP (border gateway protocol) 是当前 Internet 用到的协议。这个本身是一个 path-vector protocol。它是一个很容易被攻击的协议。

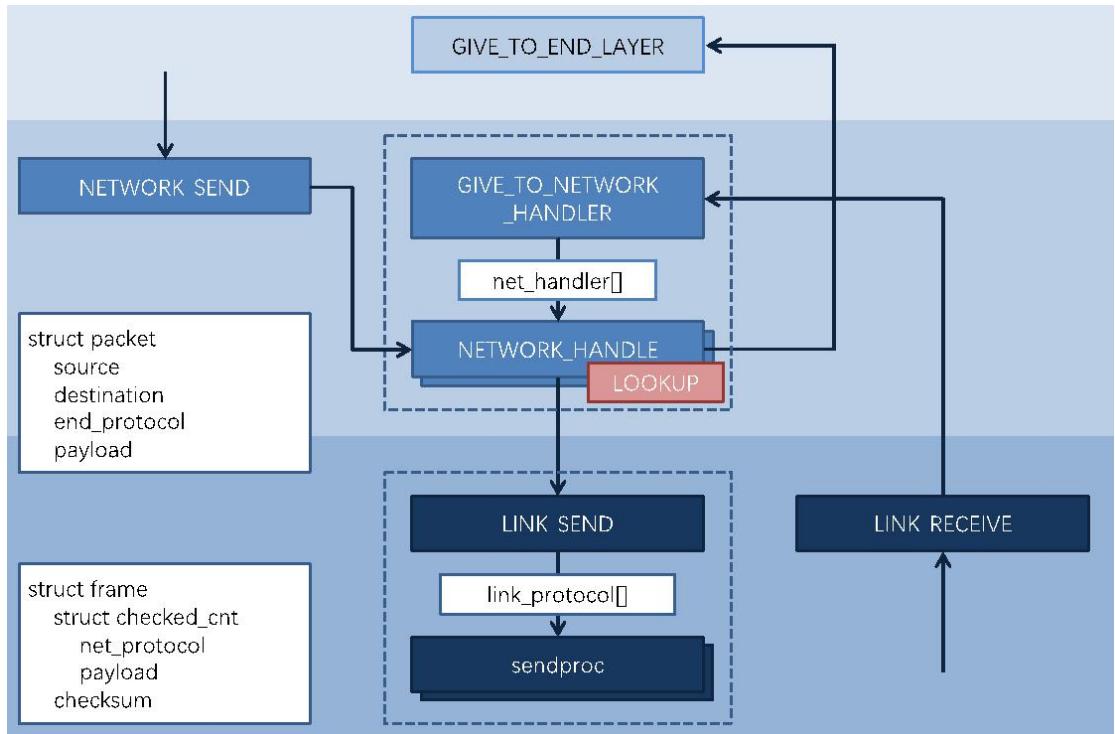
Data Plane

Network Layer Interface

```
structure packet
    bit_string source
    bit_string destination
    bit_string end_protocol
    bit_string payload
```

```
1 procedure NETWORK_SEND (segment_buffer, destination,
2                         net_protocol, end_protocol)
3     packet instance outgoing_packet
4     outgoing_packet.payload ← segment_buffer
5     outgoing_packet.end_protocol ← end_protocol
6     outgoing_packet.source ← MY_NETWORK_ADDRESS
7     outgoing_packet.destination ← destination
8     NETWORK_HANDLE (outgoing_packet, net_protocol)

9 procedure NETWORK_HANDLE (net_packet, net_protocol)
10    packet instance net_packet
11    if net_packet.destination != MY_NETWORK_ADDRESS then
12        next_hop ← LOOKUP (net_packet.destination, forwarding_table)
13        LINK_SEND (net_packet, next_hop, link_protocol, net_protocol)
14    else
15        GIVE_TO_END_LAYER (net_packet.payload,
16                            net_packet.end_protocol, net_packet.source)
```



Send 会调用 handle，是因为 handle 里有一个 lookup 路由表，如果发现 destination 不是自己，那么就 link-send 找到对应的 link_protocol 并且发出去。在网卡这个级别是没有办法发给自己的。Link receive 收到包以后，就交给 handle。

Forwarding an IP Packet

Lookup packet's destination in forwarding table

- If known, find the corresponding outgoing link
- If unknown, drop packet

Decrement TTL (Time To Live)

- Drop packet if TTL is zero

Update header checksum

Forward packet to outgoing port

Transmit packet onto link

`TTL = 64` 就是这个包最多被 64 个人转发，为什么要这样呢？防止不小心出现了 loop，这样包就一直占用着资源。还要更新 header checksum。因为我们 `TTL` 改了，`checksum` 也要改，这就是一个写操作。整个这个操作，Linux kernel 提供了一个很好的转发机制，但是它

不够快。

所以 intel 做了一个 DPDK 的一个库。

Data-plane Case Study: Intel's DPDK

DPDK: Data Plane Development

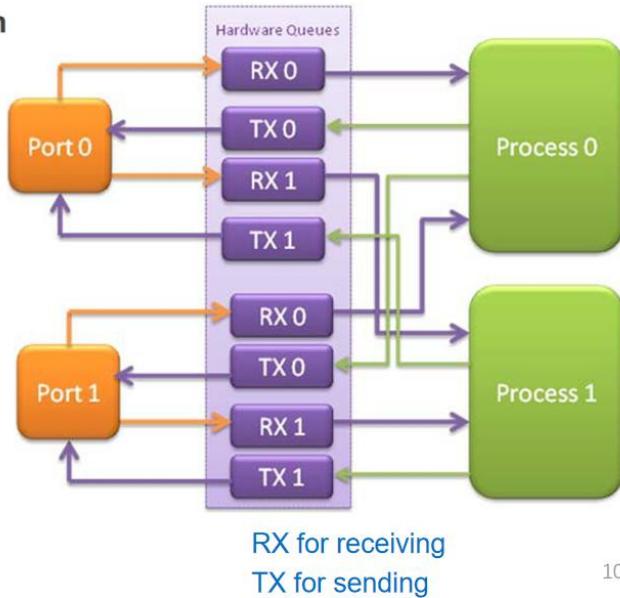
- Bypass kernel

Network card

- Has several ports
- A port has RX/TX

Processor

- Read packets from RX
 - Polling
- Find output port
- Write packets to TX



104

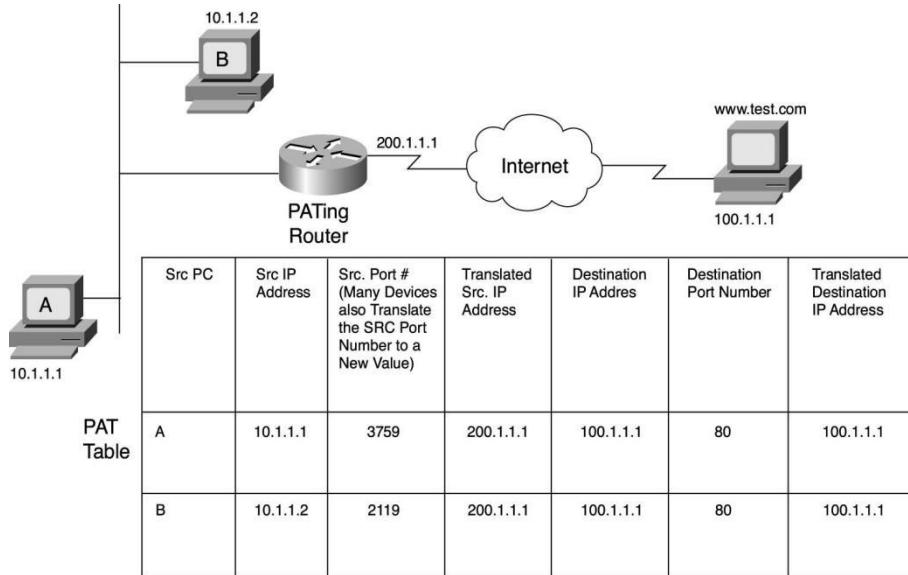
Intel 为什么要做这个事情呢？因为 intel 希望它家的网卡被越来越多人用。正好有人想用 intel 的 PC 当做路由器，但是这样麻烦就来了，一旦我们要更新路由算法，就要在 kernel 中改东西。

所以 intel 想做一个用户态的转发包的 library。Port0 和 port1 就是我们网卡的两个端口，对于这个端口，提供 receive 和 send 队列，有多少个 call 就有多少个 queue。轮询就可以看有没有包过来，一旦有包就立即处理。这样我们就可以忽略掉 kernel 这部分的作用。可以通过 mmap 把网卡写的内存映射到用户空间，这样用户态就可以直接去根据这块数据做转发。并且不让 kernel 调度，就可以降低 context switch 做的开销。所以有了这个之后，软件路由器一下子变得很可能。

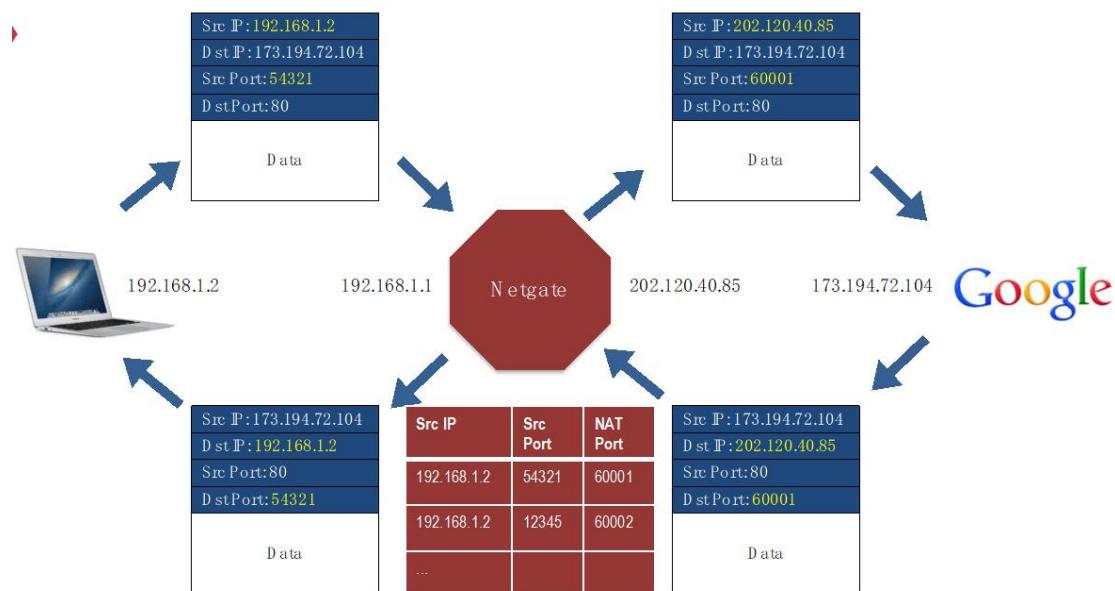
NAT

NAT (network address translation) 是因为电脑太多了，网络太少了，所以要分为内网和外网。内网开头是 10. 和 192.。

以 10 地址为了，出了内网，这个 10 地址是没有意义的，当我们要连到外面地址的时候，要把 ip 做一个转换。比如我们请求百度，百度的 destination 是什么呢？肯定是学校出口公网 ip，那么学校里那么多机器，我们就要根据端口号做一个转发。



需要根据端口再转换为 ip。



路由器有对内 ip 192.168.1.1 和对外 ip 202.120.40.85，负责对内对外的转换。负责 NAT port 和内网 IP:端口的映射。这里有好几个问题的。

1. NAT Port 的数量是有限的。
2. 这是典型的破坏了层级的设计。ip 地址不够了，我们用到了上层 end-to-end layer 的 port 的概念。我们用上层的概念解决了下层的问题，所以这并不是一个很漂亮的设计。而且必须要求上层必须是 port，如果我们用的不是 TCP/UDP 的 port 概念，那么 NAT 就不能用了。

2021/12/2

上节课我们主要讲的是网络层，解决的问题是去中心化的设计里，并没有一个全局的组织来统计哪里到哪里是通的，所以我们需要在局部构成一个网络体系，和其他连在一起构成一个整体。

在做 **advertisement** 的时候，并没有任何机制保证 **advertisement** 是真的，如果我们对外 **advertisement** 的信息是错误的，对方是没有判断机制的。

在形成路由的过程中，分为 **link-state** 和 **distance-vector** 两种方法。这两种模式，后者肯定是比前者更适合大规模的网络，但是依然不够。所以我们提出了 **path-vector**，我们要把整条路径都放在 **advertisement** 里，这样的好处就是不会出现 **loop** 的情况。但是有这个还是不够，因为量太大了，所以我们引入了 **hierarchy**，把一堆节点拼在一起变成一个 **region**。在互联网上，这叫做自治系统（AS），它对外和对内分别负责包的 **forward**，这样就可以大大降低路由表的数量，通过掩码就可以把 255 项变为一项，这意味着查表的速度可以更快。但是在一个非常高速的路由器里面，要做到这么快还是挺难的，因为每次查表我们只容许一两千个 **cycle**。

这时候我们就需要在 **Data Plane** 上提升查表的速度，如果我们能把表做一个很好的组织，使得最常用的项在 **L1 Cache** 中，这样我们查表的时候一个 **cycle** 就可以搞定，那么偶尔查 **L2, L3** 和内存，对整体的查表速度也没有太大影响。

上节课最后我们讲了 **NAT** 机制，这和我们在讲自治区域的时候也是类似的。比如学校网对外只有 3~5 个节点，那么所有访问外网的请求都会先发到 3~5 个边界节点。它解决的事情是网络 IP 不够了，这样我们就可以让内部变成 10.打头或者 192.168.打头。为了记录下对外的映射，我们要在路由器里记录下对外的端口，到时候发回数据的时候再映射回内网的 IP 和端口。

Src IP	Src Port	NAT Port
192.168.1.2	54321	60001
192.168.1.2	12345	60002
...		

如果我们有十几个家庭设备，每个设备都有多个进程发包，这个表就可能非常大。

我们如果想要配置家里的私有云，但是家里的 ip 是 192.168 的，怎么办呢？我们就要在路由器上做一个配置，比如把路由器的对外 ip 端口 60080 映射到内网笔记本的 80 端口。

以太网

接下来我们要谈一谈以太网，我们用的无论是有线的还是无线的，都是以太网，它本来设计的野心是很大的。它的全称是载波监听冲突检测的网络（CSMA/CD）。它的特点是 **listen before sending**，也就是先听，没有人发的时候我们再发。

以太网分为半双工（发收分开）、全双工（一条连接又能收又能发）

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

MAC 地址（48 位）和 IPV4 地址（32 位），如果以太网只负责从一个小的网络中的连接，为什么局域网要用 48 个 bit 作为 **destination** 呢？其实是因为这个协议最终没有赢过 IP 协议，所以它最终被降格成了 **link layer**。

它分为两种模式，

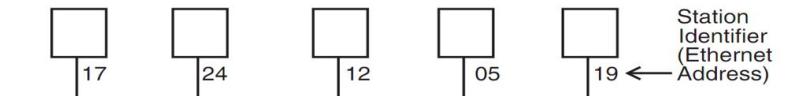
Hub 模式

它有一条线，上面有很多机器，每个机器发一条消息其他人都听得见。

Switch 模式

- Keeps a record of the MAC addresses of all the devices
- A 10/100Mbps switch will allocate a full 10/100Mbps to each of its ports

Broadcast Aspects of Ethernet



Broadcast network

- Every frame is delivered to every station
- (Compare with forwarding network)

ETHERNET_SEND

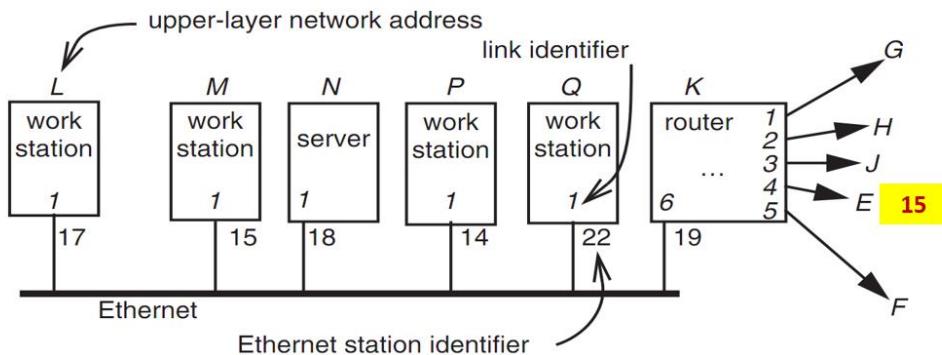
- Pass the call along to the link layer

ETHERNET_HANDLE

- Simple, can even be implemented in hardware

我们还可以设置为 broadcast 的包，这样它的 destination 就是全 f，这样整个 hub 上所有以太设备都会收到这个包。

Layer Mapping: Attach Ethernet to Forwarding Network



L sends a RPC to N by sending to station 18 of link 1

L sends a RPC to E by sending to K, E may have 15 as address, as well as M

问题来了，我们现在有多个路由器，K 是一个路由器，它是整个局域网的边界。在这样的一个配置里面，假设上层的网络地址 LMNPQ, 17 15 18 14 22 是以太的地址，当我们一个

包发过来的时候，真正要做索引机器的时候，我们其实是在靠以太地址。假如 L 要发一个 RPC 到 N，那么发包的过程是：设置包的 destination 是 N 传到 OS 以后，要加上一个包头变成一个以太包，加上 destination 是 18，这是一个 hub 模式，N 监听到 18 的包，然后就收上去了。

如果 L 要发一个 RPC 给 E，E 是另外一栋楼里的机器（在另一个以太局域网中的以太地址为 15）。虽然以太网的地址是 48bit，很大，但是没有人规定不同局域网间的以太网地址必须不一样。所以实际上很多网卡在卖的时候，以太网的地址是一样的。如果 L 还是根据刚才的办法把 E 的以太网的地址 15 填到里面去，那么 M 会收到这个包。所以发包的时候，L 其实应该把以太网的地址设置为 19，路由器 K 收到这个包以后，再根据路由表去查表，然后从 4 号口发出去，并且 K 发出去的时候需要再加上包头 15.

IP 到以太网地址（mac 地址）的这个表的映射是怎么来的呢？

ARP (Address Resolution Protocol)

NETWORK_SEND ("where is M?", 11, ARP, ENET, BROADCAST)

NETWORK_SEND ("M is at station 15", 18, ARP, ENET, BROADCAST)

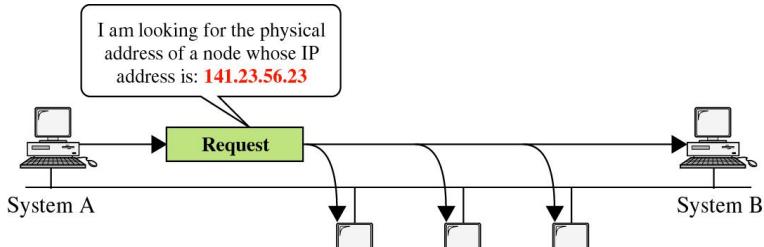
L asks E's Ethernet address, E does not hear the Ethernet broadcast, but the router at station 19 does, and it sends a suitable ARP response instead

Manage forwarding table as a cache

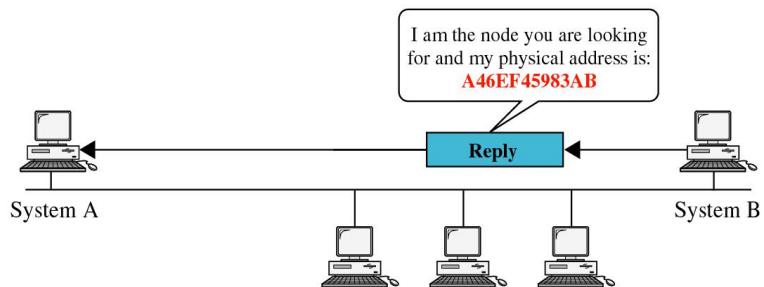
internet address	Ethernet/ station
M	enet/15

internet address	Ethernet/ station
M	enet/15
E	enet/19

ARP 协议用到了以太网上的广播协议，到一个新地方的时候，第一件事情就是向全网广播这个地址。也就是谁知道 192.168.3.3 的 mac 地址是多少，有人听了这个广播，就回复了一个 ARP 的请求。如果发的包是局域网外面的包怎么办？其他人可能也没有这个表项，所以没有人回复的时候，路由器就会回答它 19。所以 ARP 协议就长成这个样子，核心就是 IP 映射到 mac 地址。如果我们知道 mac 想找 IP 呢。我们可以用 RARP (reverse-ARP)



a. ARP request is broadcast



b. ARP reply is unicast

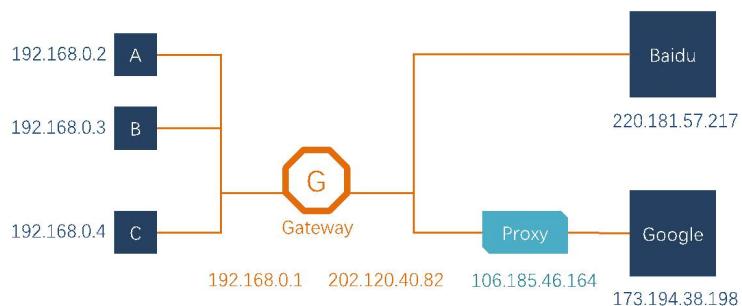
Network Topology

Take SJTU network for example

- Subnet: usually like 192.168.0.2 or 10.0.0.2
- Gateway: usually like 192.168.0.1
 - Get the global IP address: 202.120.40.82
 - A gateway usually has two (or more) IP address
- Proxy: get proxy's address
 - E.g., 106.185.46.164 (Japan)

我们想要访问一个日本的地址。

Network Topology



每次经过的时候，ip 就会发生一些变化。

首先应用程序说要访问百度，把百度的 ip 放到 target 里，扔给 OS，OS 就去查表，查到了路由器（gateway）的 mac 地址，而 source mac 就是这台电脑的 mac 地址。所以这个包就发给了路由器，路由器收到了以后，通过路由表查询。Source IP 改为路由器 1 的 source ip，source mac 改为路由器 1 的 source mac。

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

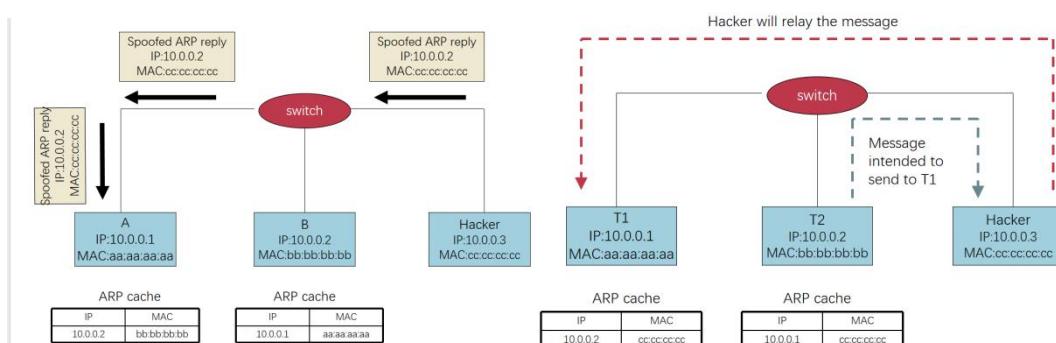
The router-1 (gateway): I get a packet with my MAC as target address. Is it my IP? No... So I'll just forward it to next hop, by changing the target MAC address to next hop's MAC address (NAT: change source IP and source port as well)

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

Router-2: I connect directly to Baidu, I'll just change the target MAC address to Baidu

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

Target MAC 和 source MAC 在网络传递的时候会一直发生变化。如果 IP 做底层，MAC 地址在网络上做索引也是完全可以的。



比如我们寝室三台电脑，A 和 B 都有一个 ARP 表。如果此时有一个 hacker，在 A 问 B 的 mac 地址是多少的时候，告诉它 B 的 mac 地址是 cc.cc.cc.cc，就会导致 A 的 ARP 表被篡改，这会导致当 A 要给 B 发包的时候，C 会收到这个包。这样就是中间人攻击了。

BBS 那时候用到的是 telnet，它是 22 号端口，注意 telnet 的所有网络流量是不加密的，这意味着我们在登录的时候输入账号和密码，中间人就可以截获。

因为这是协议的设计，所以没有一个很好的防御措施。

- 不要有一个动态的 ARP 表，可以交给用户自己去配置这个 ARP 表。
- ARP watch，专门监听是否存在异常行为，比如经常有一个非网关的地址在发送包。

上节课我们讲的是网络层如何和以太网做映射。从理论上来说，以太网是不需要地址的，但是由于我们要去复用以太网的 hub 连接。

End-to-end Layer

接下来我们就要讲更上层的 end-to-end layer。网络层到现在为止，依然没有一些绝对的保证，比如 delay。这不像我们之前说到的打电话的网络，我们把连接预留一小段。而且网络层还不能保证网络包的 order 是一致的。甚至连包能不能到都不能保证，也有可能包的错的，或者包发错了人。所以整个网络非常地不靠谱，这才是我们需要 end-to-end layer 存在的理由，因为没有一个单独的设计可以解决所有的问题。

UDP: 纯粹发包，不管丢包，和 IP 协议相比只是多了端口号，有了端口号一个 IP 就可以复用给不同的应用程序。

TCP: 保证数据是有 order 的、没有丢包、没有其他的包

RTP(real-time transport protocol): RTP 协议是基于 UDP 的，对于时延的要求更高一点，所以直播的实时流媒体会用到这个协议。

1. 确保 At-least-once 的分发

怎么确保 at least once，我们要设置一个 RTT 时间（发过去的时间+处理时间+返回时间 ack），超过 RTT 这个时间我们再重发这个包。

需要做如下事情：

1. 发包的时候放一个 nonce，不能发完就把包删了
2. 如果发现 timeout，就要 resend。
3. Receiver 要把 nonce 返回给 sender
4. 我们不能无限制地去 retry，我们最多重试 3~5 次，如果还是不行就往上报错。

At least once，要面对的就是两难问题。1. 数据没送到 2. 数据收到了但是没有收到 ACK。对于 at least once 来说，是没有一个 absolute 的 guarantee 的。我们只能保证 a 和 b 之间如果有两条通道是通的，我们最终可以保证收到这个请求。

下一个问题：如何确定超时，这个问题非常关键。

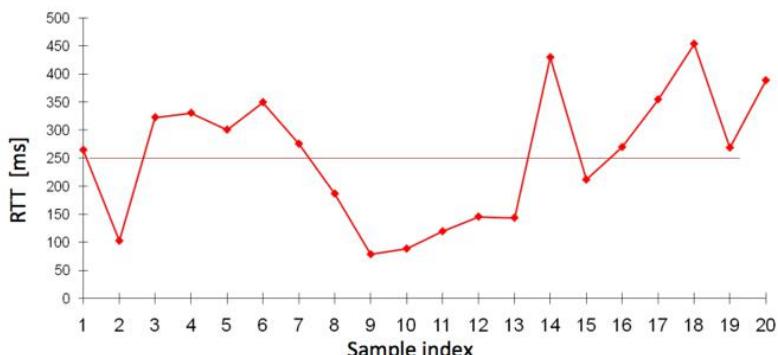
第一种方法是 **fixed timer**，设置超时时间 1s 等，这种太短就是无谓重试，太长就是浪费时间。并且 **fixed timer** 会导致 **collapse**，举个例子，当我们在收邮件的时候，大家都有邮件的客户端。邮件客户端有一个特点，它要设置比如 5 分钟收一次，收到一次以后改成 1 分钟收一次。对于这种邮箱来说，可能到固定时间以后整个局域网都在收邮件。比如云服务器厂商每到晚上两点，服务器就 crash 了，服务器的硬盘在狂转，因为在杀毒。

另一个例子：

网件路由器代码是用来同步时间的，写死了美国威斯康辛大学的 SNTP 的对时服务。如果有问题就会重试，每秒尝试一次。然后整个大学就一直在承受路由器的 ddos 攻击。所以 **fixed timer** 我们要非常小心。

那么既然 **fixed timer** 有问题，我们能不能设置一个 **adaptive timer** 呢？比如我们把超时时间设置为 RTT 的 150%，并且会指指数型的设置超时时间。比如 1s->2s->4s->8s。

RTT Could be Highly Variable



Example from a TCP connection over a wide-area wireless link
Mean RTT = 0.25 seconds; Std deviation = 0.11 seconds!

Can't set timeout to an RTT sample; need to consider variations

对于一些特殊情况，会导致系统在没有超时的情况下误判为超时。真实的 linux 里的代码长这样。

Calculating RTT and Timeout (in TCP)

Exponentially Weighted Moving Average

- Estimate both the average `rtt_avg` and the deviation `rtt_dev`
- Procedure `calc_rtt(rtt_sample)`
 - `rtt_avg = a*rtt_sample + (1-a)*rtt_avg; /* a = 1/8 */`
 - `dev = absolute(rtt_sample - rtt_avg);`
 - `rtt_dev = b*dev + (1-b)*rtt_dev; /* b = 1/4 */`
- Procedure `calc_timeout(rtt_avg, rtt_dev)`
 - `Timeout = rtt_avg + 4*rtt_dev`

1/8 和 1/4 都是 magic number。最终我们判断是否超时的时候，是用平均值加上 `dev*4`。所以其实这个值非常不靠谱。没有超时而误判为超时的情况是非常常见的。

NAK

NAK (Negative AcKnowledgment)

- Receiver sends a message that lists missing items
- Receiver can count arriving segments rather than timer
- Sender can have no timer (only once per stream)

我们有没有不依赖时间的方法呢？我们可以用 NAK 方法。它的逻辑是返回来的，我们发一个 3 号包，对方和你说 3 号包没收到重发一下（而不是告诉你哪些包收到了），这样我们就能 100% 确定对方确实没收到。这样我们就可以直接把那些丢掉的包发回去。在实际时间的时候，NAK 也要考虑很多问题。

2. 保证 At most once

在 at least once 中，`sender` 要记录某些包收到了还是没收到的，如果误判了超时，那就可能发了 `duplication`。所以 at most once 需要记录哪些包收到过，一旦我们记录了一个 `table`，它就可能变得非常大，导致空间和搜索时间都变长了。我们需要用到幂等和其他方式去解决。

有一种提供 at most once 的方法，就是我们的 `nonce` 是顺序的，我们只记录一个 `watermark`（水位）。比如说，一个 `client` 发过来 1 号、2 号、3 号 `nonce`，那么我们只记录现在已经收到 3 号 `nonce`，再给我们发一个 2 号，我们就会说已经收到过了。这意味着，如果我们有 10000 个 `client`，那么我们就有 10000 个水位要记录。

还有一个方法就是新的端口去接收新的请求，比如说原先使用了 80 端口，我们继续使用 81 端口、8008 端口。但是我们这些旧的端口也不能删掉。

为了能够解决这个问题，我们需要去接收犯错的可能。我们假设 `sender` 在 5 次 RTT 之后就不会再发送。我们就可以把 5 次 RTT 前的给删掉。

Accept the possibility of making a mistake

- E.g., if `sender` always gives up after five RTT (cannot ensure at-least-once), then `receiver` can safely discard `nonces` that are older than five RTT
- It is possible that a packet finally shows up after long delay (solution: wait long time)

Receiver crashes and restarts: lose the table

- One solution is to use a new port number each time the system restarts
- Another is to ignore all packets until the number of RTT has passed since restarting, if `sender` tries limit times

Anyway, duplicate suppression makes the system complex

对于 `receiver` 来说，记录包的编号在 5 次 RTT 之后就可以删除掉了。但是还有可能 `receiver` crash 重启完整个 `table` 就没有了。一种方法是重启以后使用一个新的 `port`，但是旧的 `client` 的 `request` 就失效了。另外一种就是忽略掉所有的包，直到超过 5 个 RTT 的情况。

3. 数据的 integrity

完整就是数据没有被改来改去。`Receiver` 收到的内容必须是和 `sender` 发的是一样的。一个基本做法就是 `checksum`。我们在之前说道，在 `link layer` 已经有 `checksum` 了，为什么我们在 `end-to-end layer` 还要做 `checksum`。因为 `link layer` 的海明编码一旦错了 2~3 位结果可能是错的。所以底层的不一定那么可靠，并且网卡检测完 `checksum` 复制到内存的时候，也有可能出错。所以 `end-to-end layer` 再加一层检测就可以增加安全性。又可能我们的包误发到别人那里去了，所以也不能依赖于底层的 `checksum`。

4. 分片和重组

4. Segments and Reassembly of Long Messages

Bridge the difference between message and MTU

- Message length: determined by application
- MTU: determined by network

Segment contains ID for where it fits

- E.g., "message 914, segment 3 of 7"
- Can be used for *at-least-once* and *at-most-once* delivery

Reassembly

- Out-of-order, mingled with other message's segments
- Allocating a buffer large enough to hold the message

比如我们传包的时候说，我正在传 7 个分片中的第 3 个。当收到了之后，顺序可能会混在一起。所以我们需要预留一个足够大的可以容纳整个 message 的 buffer。这样整个 message 就可以还原出来。一旦发生了 *out-of-order*，分为几种情况。

1. 不允许发生乱序的情况，收到 1,2 号包之后，丢弃收到的 4 号包，必须等 3 号包。这样浪费带宽。
2. 收到包之后，先放到 buffer 里，但是如果有一个 package 是坏的。这个 buffer 得一直维护住。这个对于 receiver 来说也是一个很大的内存压力。

所以最终我们 combine 一下两种方法，如果 buffer 满了但是还有包没有收到，那么我们只能丢掉 buffer 重传。

我们可以用一种为 common case 优化的方式，我们可以使用 NAK 来优化这种方式，一旦 NAK 导致了 duplicate，那么我们就把 NAK 转化为 ACK。

所以 *out-of-order* 的关键问题在于我们要用一个多大的 buffer 去对数据排序，如果我们 receiver 无原则的容忍 sender。Sender 可以利用这一点来消耗 receiver 的资源。

5.Assurance of Jitter Control (抖动控制)

5. Assurance of Jitter Control

Real-time

- When reliability is less important than timely delivery
- A few error in a movie may not be noticed
- **Jitter**: variability in delivery time

Strategy

- Basic: delay all arriving segments

Measure the distribution of delays in a chart showing delay time vs. frequency of that delay

Choose an acceptable frequency of delivery failure

Determine D_{long} that longer than 99% delay

Determine the shortest delay, D_{short}

Calculate number of segment buffer:

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

- $D_{headway}$ is average delay between arriving segments

Buffer 就是用来对抗 jitter 的。缓冲的时间怎么确定呢？我们可以用这个简单的公式来确定我们要缓冲多少个 segment。换句话说，我们最长的一次 delay 是 100ms，最短是 9ms，平均是 10ms，那么我们应该缓存 9.1 个包($(100-9)/10$)。我们收到一个包开始看视频，哪怕遇到了 100ms 也没关系，因为我们已经缓冲了 9 个包，可以抵御一次 D_{long} 。所以对抗 jitter 的方法就是缓存。但是游戏就没有办法用这种方法了。Jitter control 实际上是不同类型的包有不同的要求。

6. 确保权限控制和隐私

6. Assurance of Authenticity and Privacy

Internet is dangerous

- Hostile intercepts and maliciously modifies packets
- Violate a protocol with malicious intent

Key-based mathematical transformations to data

- Sign and verify: establish the authenticity of the source and integrity of contents
- Encrypt and decrypt: maintain privacy of contents

Consideration

- False sense of security, worse than no assurance

我们需要对网络加密，核心就是加密的公钥私钥体系。

Public Key VS. Private Key

- Public key: encrypt to identify *reader* (only me can read this)
- Private key: encrypt to identify *writer* (yes, it's me who wrote this)
- Poor performance, so just used to exchange symmetric key

Questions

- What is a certificate? Why using a CA (Certificate Authority)?
- How to exchange a symmetric key in HTTPS or SSH?
- What is the root of trust?
- Will be addressed later, in Security part

公钥和私钥其实是人为分开的，在数学上它们的地位是一样的，只是我们在使用的时候公开了一个而已。有一个缺点就是一旦我们使用了公钥私钥机制的时候，性能就会差很多。我们在密钥交换这件事情，我们用安全的公私钥方法，在大量数据传输的时候，我们再使用比较快的传输方式。

2021/12/7

从网络整个层次角度来看，底层专业知识更多，上层软件知识更多。上层的协议是用来解决各种各样的问题。学习协议的最好方法就是读手册看什么情况下会发送什么样的数据包。但是为什么要这么设计，背后有很多理论性的知识。

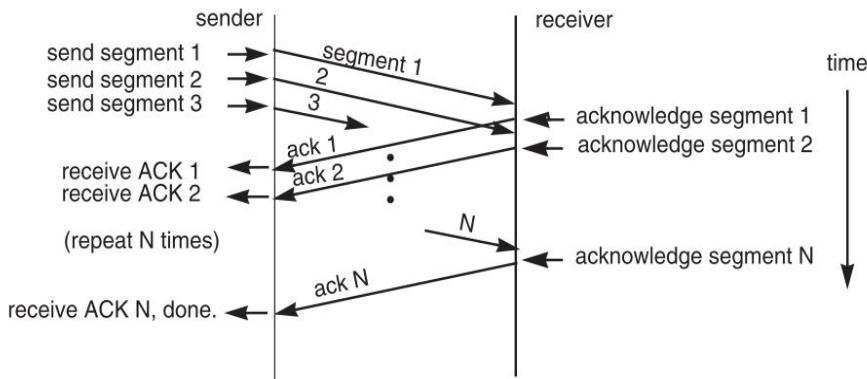
我们来看底层的 IP 层，包括底下的 link layer 不能解决 at most/least once、数据的完整性、数据的顺序、jitter control。

end-to-end performance 是希望发包的速度足够快，但是发包太快可能会拥塞整个网络。发包的速度不是决定网络瓶颈的点，而是网络本身带宽的限制，以及接收者能不能接收这个频率的发包速度。如果两台一模一样的机器旗鼓相当，那么是可以做到的，但是接收的机器可能同时收多人的包，这就很难坚持住了。我们有一个很多人发包的路由器，在收包的过程中，人越多，收包的压力就越大。前提是怎么样才能避免这种拥塞

1. lock step

发送者发一个 segment（包的碎片），然后 receiver 返回 ack，然后再发下一个。这可以很好地满足准确性，一旦收不到就重新发。但是这毫无疑问很慢，大量时间都在等待 ACK。所以这个方法很慢。

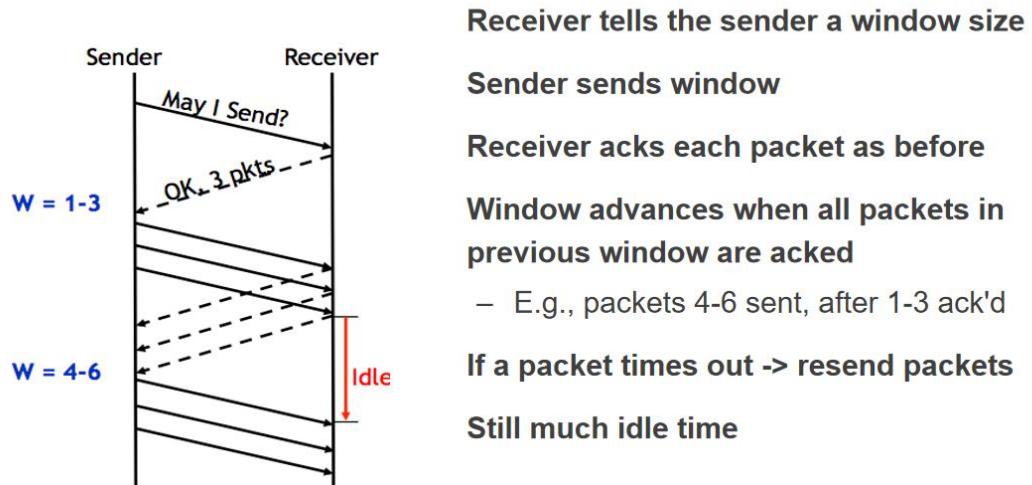
一种直观的想法是 pipeline，不管你收没收到，我先一直发。



Pipeline 打破了 sender 和 receiver 之间的关系，没有做等待，这就会导致太快，可能导致网络包和 ack 被丢弃。

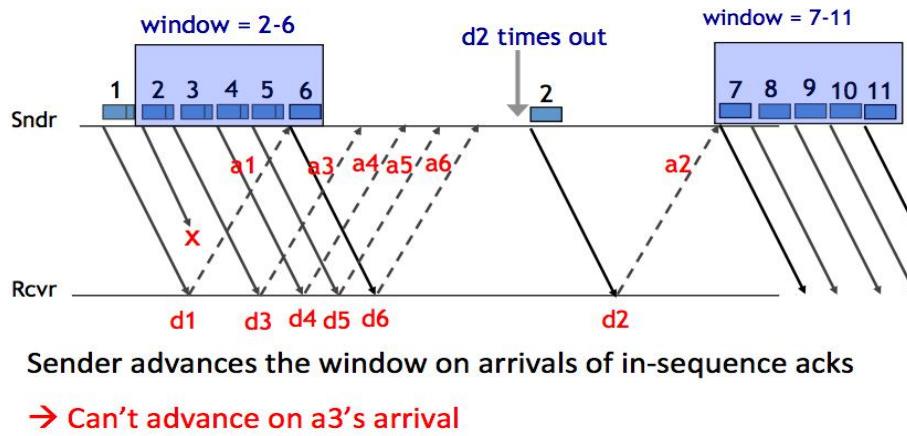
我们能不能让 sender 保持一个已经发送的 segment 的 list，如果 list 中全部 ack 了，那再往下发一个 list。这就是在 lock step 和 pipeline 之间的权衡。 n 怎么确定呢？比如我们要发 n 个包，实际上这个数字是由 receiver 的 buffer 来决定的。

Fixed Window



这个 n 个包的块就叫做 window, receiver 是在这个 ack 和下个 ack 之间差的时间还比较长。这个 idle 期间的 buffer 是没用的。我们就觉得这段时间可能是可以优化的。在这段时间之内, sender 有没有可能再发一些包。当 sender 收到第一个 ack 的时候, 是不是能发第 101 个包。

所以我们希望做到的是滑动窗口,有了滑动窗口之后就可以把 receiver 的 idle 时间减少,在这个过程中一旦出现了一个丢包,会出现如下的情况:



上图中的滑动窗口因为 2 丢包就堵塞在那里了。要等待 2 发包过来才能进去走。这里还是可以改进的,比如用我们上节课的 NAK 方法,告诉 sender 没收到再发一次,这就是一个可以优化的点。在实践的时候,TCP 就会发一个 duplicated ack,只发送 ack1,而不发送 ack3,因为就算发了滑动窗口也过不去。

我们这个包应该多大呢?

Sliding Window Size

window size \geq round-trip time \times bottleneck data rate

Sliding window with one segment in size

- Data rate is window size / RTT

Enlarge window size to bottleneck data rate

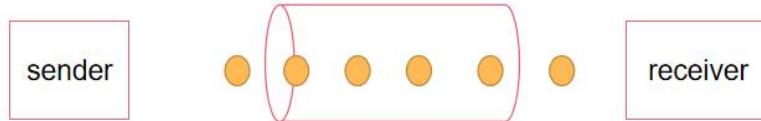
- Data rate is window size / RTT

Enlarge window size further

- Data rate is still bottleneck
- Larger window makes no sense

- Receive 500 Kbps
- Sender 1 MBps
- RTT 70ms
- A segment carries 0.5 KB
- Sliding window size = 35KB (70 packets)

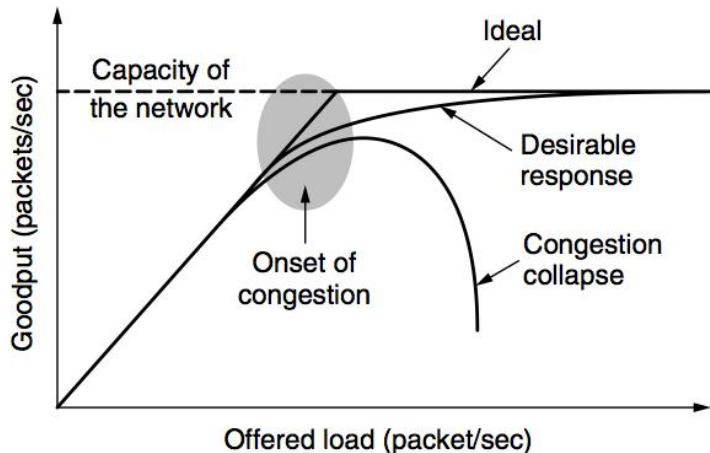
我们把 bottleneck data rate 可以看做一根管子的截面积,而 round-trip time 是管子的长度。而 window size 就是不等收到 ack 前还要继续发的数量,我们希望管子中都是我们扔出去的纸飞机。我们希望 receiver 收到一个包, sender 就发送一个包,并且此时管子中(网络中)充满了包。



那么我们在整个滑动窗口的过程中，`sender` 不知道 `bottleneck` 多少，但是它正好卡在 `bottleneck` 的角度去发送，一旦 `slide window` 都填满了，就不能再发送了，`receiver` 产生 `ack` 的速度也不会比网络发送的速度更快。但是我们依旧要估算这个 `RTT`，在 `Linux kernel` 中通常是通过 $3/4$ 加权来迭代计算 `RTT`。

前面我们说的是怎么让 `sender` 发包充满整个网络，接下来我们要控制 `congestion`，这是因为大量的包同时出现在网络上导致大量的包出现丢失。一旦网络上的包太多，导致排队的包还没有处理的时候就导致超时重发了。此时的排队就没有意义了。所以交换机单纯地变大，只要我们有超时机制，只会导致更多的排队超时，并不能解决问题。所以来 `tcp` 采取了一些比较极端的措施，让自己的发包率不要太高。`TCP` 一旦发生丢包，就立刻不传了，`sliding window` 就变成 `0`，等到网络正常之后才一点点去填满。

在真实世界上，网络层（快递公司）和 `end-to-end layer`（用户）都会做这些控制，而 `end-to-end layer` 最终做这些控制，因为发包快慢是这一层控制的。快递公司不能控制用户减少下单，只能体验到拥塞这件事情，所以网络层只能把这件事往上层做传递。



我们希望逼近网络包的极限，但是实际情况就是网络 `collapse` 了。对于 `congestion` 来说 $window\ size \leq \min(RTT \times bottleneck\ data\ rate, Receiver\ buffer)$

Load Shedding: Setting Window Size

For performance:

- window size \geq round-trip time \times bottleneck data rate

For congestion control:

- window size $\leq \min(\text{RTT} \times \text{bottleneck data rate}, \text{Receiver buffer})$
- Congestion window

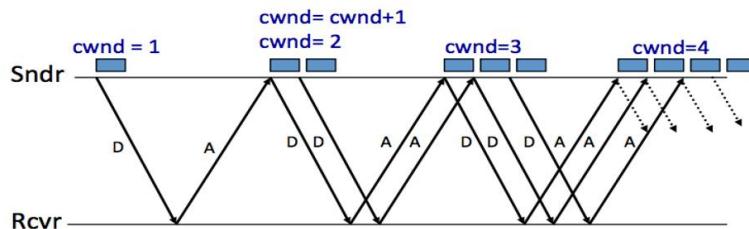
2 windows become 1

- to achieve best performance and avoid congestion

这两个可以合并成一个，也就是既达到网络的最大利用率也避免 congestion。Congestion window 应该是慢慢地变大的，如果没有发生 congestion，就继续上涨，如果发生了，就立即下降。

AIMD (线性增长、指数下降)

AIMD (Additive Increase, Multiplicative Decrease)



Every RTT:

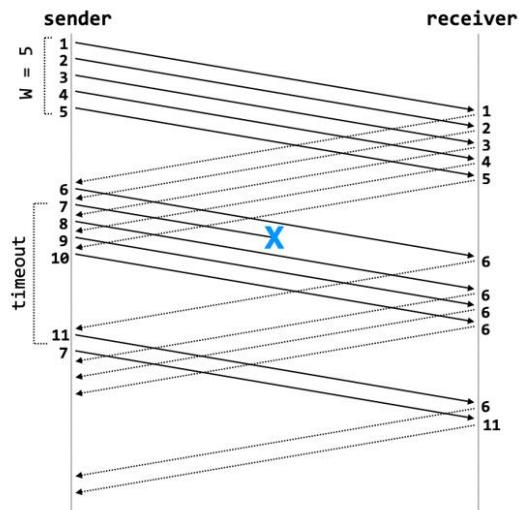
- No drop: $cwnd = cwnd + 1$
- A drop: $cwnd = cwnd / 2$

在 TCP 中用到了一个 AIMD 算法，这是一个比较保守的算法。但是 AIMD 有一个问题，就是开始的时候非常慢。我们可以在开始的时候用指数级的方式去增加，这就是 Slow Start 阶段（理论上应该叫 fast start）。还有一个 faster retransmit。

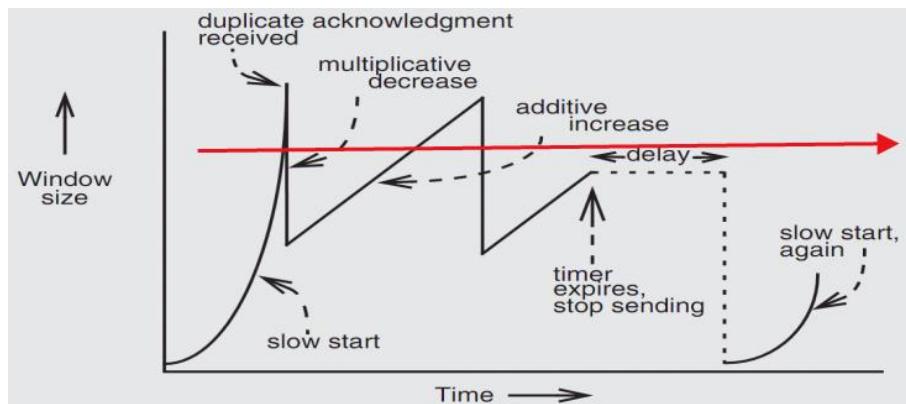
TCP Fast Retransmit / Fast Recovery

When a sender receives an ACK with sequence number X, and then three duplicates of that packet, it immediately retransmits packet X+1

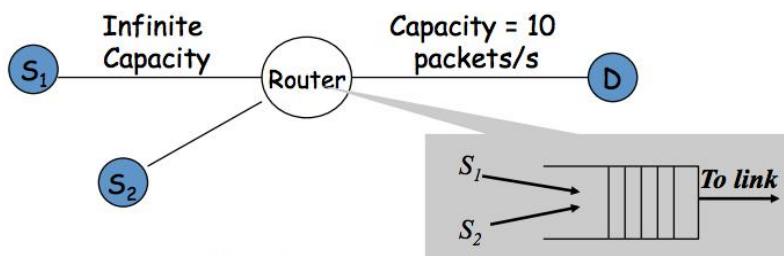
- Example: sender receives 5 6 6 6 6
- Infers that packet 6 is lost, immediately retransmits
- On fast-retransmit, window decrease is as before: $W = W/2$



第一次收到 6，是正常 ACK，然后收到 3 次重复的 6，就说明 7 号包丢了，一旦发生了丢包，那么立刻把整个 window 降到最低，并且重新开始 slow start。



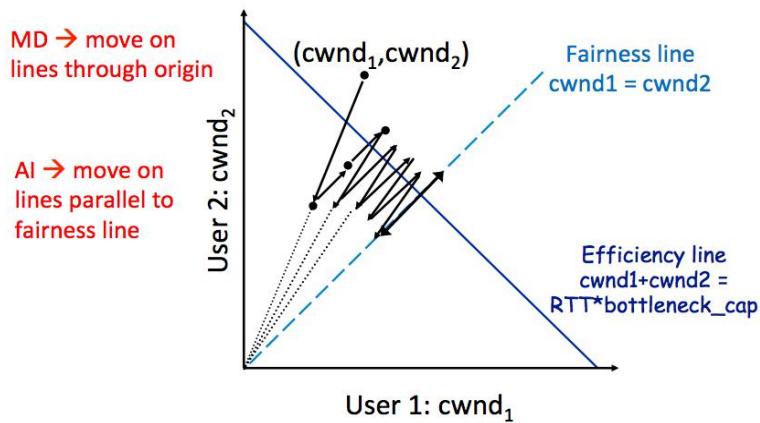
这个就包括了 slow start、丢包腰斩阶段，重新 slow start 阶段。很可能极限就在红线为止。但是红线下的锯齿状空白部分（没有被积分到的位置）都是被浪费掉的空间。这个可以很好地实现公平性。



Bottleneck may be shared

注意到 S1 和 S2 要竞争 10 个容量，最好的结果是 5+5. 我们来看怎么通过 TCP/IP 达到公平性。

Consider two users who have the same RTT



$y=x$ 这条虚线就是 user1 和 user2 的 window 大小相等的情况。如果超过了蓝线，说明加起来比 capacity 大，就会发生 congestion。腰斩就是两个人都腰斩，所以最终都会收敛到交点出，所以我们同时做到 efficiency 和 fairness。如果我们是线性下降，虽然对单个节点来说可以利用多余的带宽，但是多个节点时不能保证公平性。

Weakness of TCP

If routers have too much buffering, causes long delays

Packet loss is not always caused by congestion

- Consider wireless network: if losing packet, sender may send faster instead

TCP does not perform well in datacenters

- High bandwidth, low delay situations

TCP has a bias against long RTTs

- Throughput inversely proportionally to RTT
- Consider when sending packets really far away vs really close

Assumes cooperating sources, which is not always a good assumption

如果 buffer 很大，但是 data rate 很小，buffer 再大也没有。丢包也不一定是因为 congestion 导致的，假设我们手机连到了 WiFi 但是信号不是很好，此时应该是多发包，争取发十个包有一个包是对的，但是 TCP 会把你的 congestion window 降低到 1，误判为 congestion。并且 TCP 在数据中心（时延低、带宽高）做的不是很好，所以在数据中心有新的 DCTCP (data center TCP) 协议。它就针对数据中心的网络特点设计 TCP，并且 TCP 对 RTT 的要求有很大的依赖，因为 RTT 越长，throughput 越低，这实际上是不太准确的，因为 RTT 长了以后和 throughput 是没什么关系的。

Summary of Congestion Window

Reliability Using Sliding Window

- Tx Rate = W / RTT

Congestion Control

- $W = \min(\text{Receiver_buffer}, \text{cwnd})$
- Congestion window is adapted by the congestion control protocol to ensure efficiency and fairness
- TCP congestion control uses AIMD which provides fairness and efficiency in a distributed way

DNS 的设计

接下来，我们要来讲 DNS (domain name service)，它也是在 end-to-end layer 中的，在网络中可以把 IP 藏起来。当我们去访问一个域名 www.sjtu.edu.cn 的时候，最终可以解析为 IP(202.120.2.119)，这就是一个 look up 的过程。

IP Address as a Type of Name

An IP address itself is a type of name

- A structured name that is used to locate an object
- Use IP address to identify the server
 - Recall your labs in ICS on socket
- On Internet
 - The router will know where to send a packet with destination IP

Hostname has no such semantic

- A router does not know how to send a packet to "baidu.com"

在网络上其实已经够用了，但是我们希望有 **human-friendly** 的方式。

一个域名可以有多个 IP，这样用户端就可以挑一个距离近的。一个 IP 也可以对应多个域名，挂载不同的端口上即可。

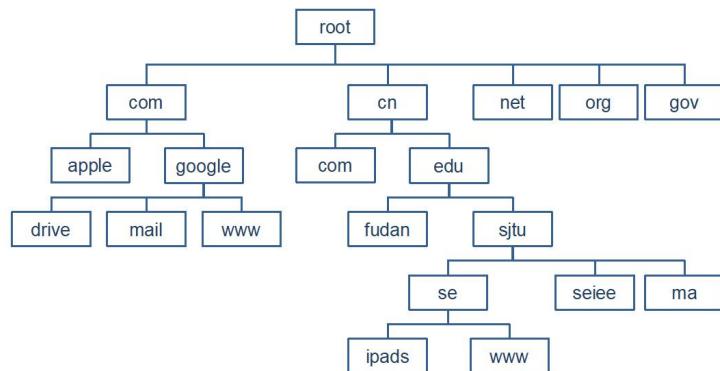
做 name change 的时候该怎么办呢？也是可以修改 name 和 ip 的映射关系，不需要修改 hostname，这样就把修改对最终用户屏蔽掉了。

DNS 查找域名和 IP 映射的算法

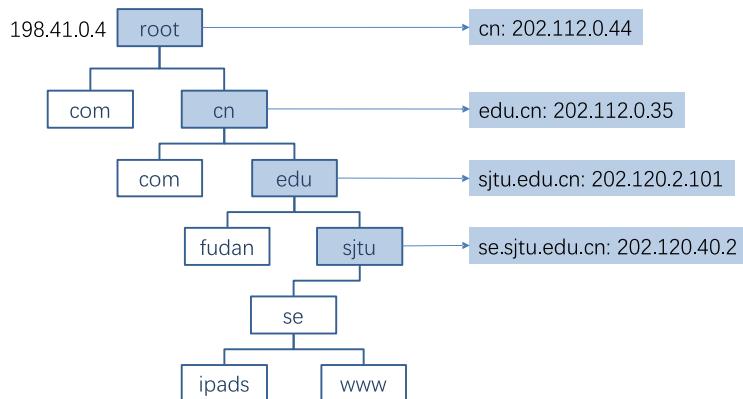
1. 每个人维护一个 hosts.txt，把域名和 IP 写下来，就和我们的目录一模一样。但是毫无疑问这个算法是不够 scale 的，文件会越来越大。所以在 1984 年，出现了 BIND(Berkeley Internet Name Domain)，这个系统今天还在用。

2. Binding 一定不能做成一个中心化的服务器，因为全世界这么多人在访问，但是我们依然要把数据存在很多个服务器上，只做提供 name service 的服务。我们怎么样才能组织这么多的域名服务器。我们首先分成一个个 zone，然后产生很多个 name server，最 root 的这些域名是由 ICANN 这么一个组织来维护。通过这种方式，做了一个分工的解耦。如果我们要注册一个 cse.sjtu.edu.cn 要层层审批到 ICANN，那就太麻烦了。去中心化就是放权到子 zone 中。

DNS Hierarchy (a partial view)



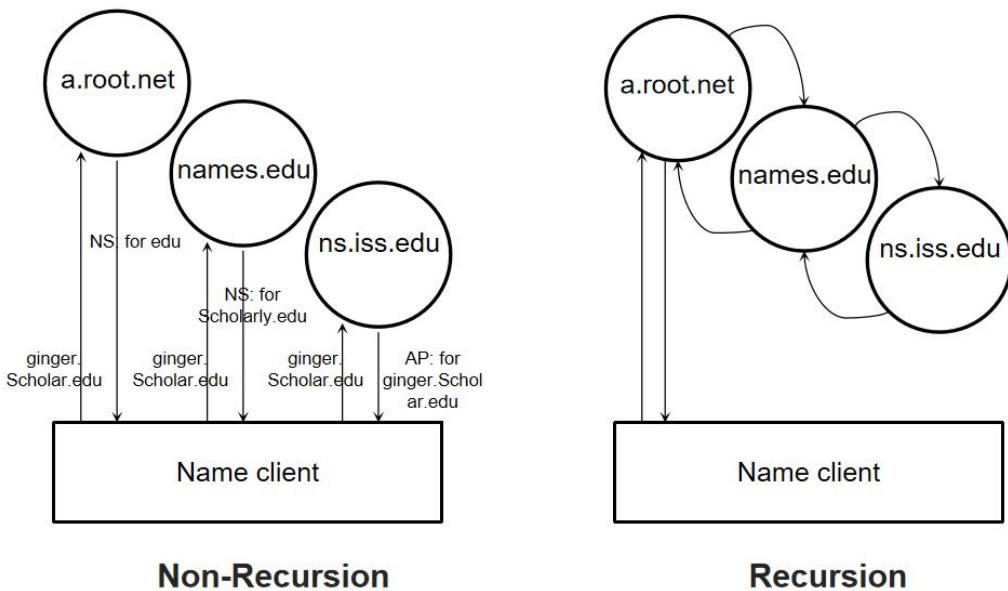
当我们去找 `ipads.se.sjtu.edu.cn`（省略了最后的一个点）的时候，我们先要去问根目录的服务器找`.cn`的服务器在哪，然后我们一层层往下找。



所以在整个这个过程中，`context` 很有意思，`name` 过程是一个 `global` 的过程，所以它是 `context-free` 的（不需要说是哪台机器上的哪个目录下的文件）

怎么做 `fault-tolerant` 呢？每一个 `zone` 都有多台服务器，一个挂了就问另一个。如果每一次都会问 `root` 的话，那 `root server` 会先 `crash` 掉。所以在现在的场景下，我们在浏览器里按下回车的时候，初始的 `DNS` 可以到任何一个 `name server`，比如先问交大的 `DNS`。如果我们不知道交大的 `DNS` 是多少，这是不可能的，因为在分配 `IP` 的时候，就会知道交大的 `DNS`。Google 给全世界提供 8.8.8.8，还有 114.114.114.114，这样我们可以加进 `/etc/hosts`。如果没有的话，那就是我们第一次连到无线网的时候，会拿到一个交大的 `DNS` 服务器 `IP`。直到没有人知道才会找 `root`。

DNS Request Process



Non-Recursion

Recursion

第二种的情况就是可以 cache 住，对于交大的 DNS 服务器来说比较合适。但是这种有一个问题，因为第一种是无状态的，第二种是有状态的，如果服务的人太多了，消耗的资源就会更多，所以在具有同样资源的情况下，recursion 的方法能够服务的人会比较少。

Three Enhancements on Look-up Algorithm

3. Caching

- DNS clients and name servers keep a cache of names
 - Your browser will not do two look-ups for one address
- Cache has expire time limit
 - Controlled by a time-to-live parameter in the response itself
 - E.g., SJTU sets the TTL of www.sjtu.edu.cn to 24h
- TTL (Time To Live)
 - Long TTL VS. short TTL
 - **Q: what are the tradeoffs?**

修改一次绑定关系后，24 小时后就会生效。

在 2008 的时候，全世界根目录的 name server 有 80 台，分散在有很多地方。交大的 DNS 也有 replica，至少有一台得在交大外网。外网如果没有 DNS，那么别人就不能解析交大内部的域名。在最早的时候，我们必须要通过 name discovery 来发现 DNS 服务器，这个今天基本上已经没有这个问题了。

我们要对 hostname 和 filename 做一个比较。Filename 映射到 inode number 就类似于 hostname 映射到 ip 地址，并且都是 hierarchy 的。所以我们发现 hostname 和具体的网站是割裂开来的。所以千万记住，name 不是文件的一部分，所以 hostname 和主机和网站都是

没有关系的。

在这里的绑定操作有点区别，对文件来说，一个文件名映射到多个 inode 是不存在的，但是对 DNS 来说是存在的。n 个文件名映射到 1 个 inode (hard link) 是存在的，对于 DNS 来说也是存在的。

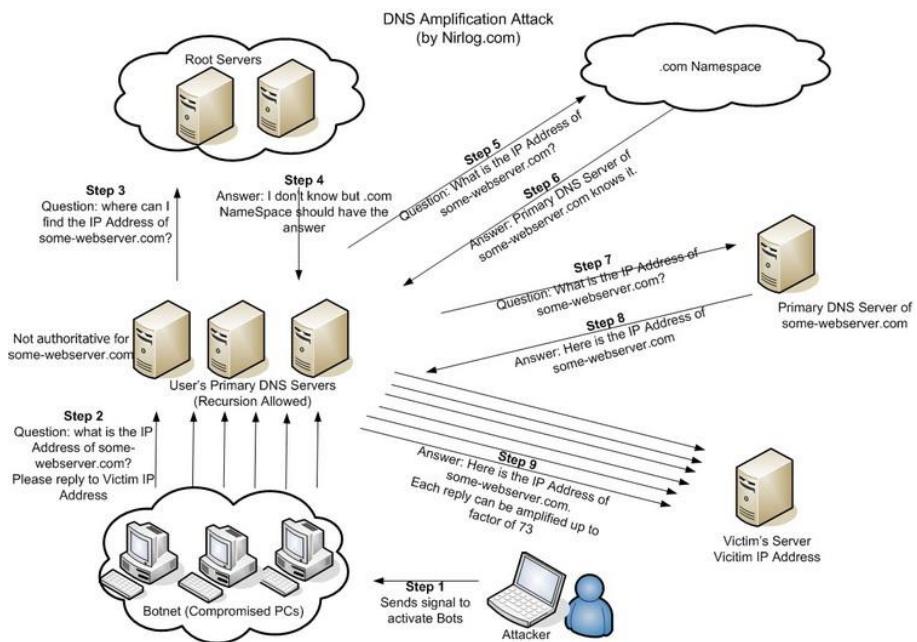
我们讲完 DNS 的设计之后，我们要来看一下在 DNS 设计的背后做对了哪些事情。DNS 利用 **hierarchy**，把责任分割成了 zone，这样做到了去中心化。Global name 很好地避免了对 context 的要求。并且 cache 可以减少 query 的量，还可以通过 delegation 来分配到多个。管理起来也很方便。

Fault-tolerant 也很重要，因为每个 DNS Server 有多个。

缺点就是管理的人到底是谁，政府还是 NGO，能不能给犯罪团伙提供服务，这都是有很多争论的。

第二个就是对于 root server 来说还是有很大的漏洞，我们讲数据库的时候，一旦 cache hit 就直接拿到了，但是如果我们要查找一个不存在的东西，一定会 cache miss。如果我们找一个不存在的域名，那么所有 DNS 服务器都不知道，就要问根服务器，这就可能成为一个 Dos 攻击。

并且在安全性方面，有如下的例子（DNS Amplification Attack）



Attack 在僵尸网络中发起 DNS 请求，在网络中会放大，然后把所有人的流量都导入到被攻击者。其实就是让 100w 台电脑认为 www.baidu.com 的 IP 是你的公网 IP，这就是 DNS 放大攻击，因为 DNS 并没有很好的认证机制。

DNS DoS (Denial of Service) attack

- BAOFENG.com & DNSPod
- 2009-5-18: DNSPod was attacked and banned
- 2009-5-19: The Internet in China was almost down
- Fixed timer: query for BAOFENG.com once per second!

Solutions

- /etc/hosts, dnsmasq, OpenDNS, etc.
- DNS shield to defend against DoS attack

攻击 DNSPod 之后，被攻击的游戏就开始卡了。主要是 DNSPod 还用来服务暴风影音，里面有一个 check update 功能。DNSPod 不能服务了就往上家发了。导致整个中国网络瘫痪，在安全性上，我们也可以通过 DNSSEC 得到来自正确服务器的反馈。

2021/12/9

如果交大的同学都想访问同一个网站，那么我们就可以在出口的地方做 cache，同样，我们也可以在 server 端主动设置推送。

我们打开淘宝的时候有各种各样的小图片，可以缓存在本地，如果我们通过 proxy 去访问这些网站的话，大家出口都从这个地方走，也可以做缓存。我们也可以通过 DNS resolver 的时候缓存 URL 到 IP 的映射，一旦我们访问过一个网址以后基本上就不会再更新这个映射了。

在之前上 NFS 的时候，client 也会有 cache，这时候就要考虑 cache 一致性的问题。

但是这些都是被动的方式，第一次的 cold miss 是不可避免的，那么我们在网络中怎么样把第一次的 cold miss 都避免掉呢？

这时候我们就提到了 CDN，CDN 是一种专门提供内容服务的厂商，独立于不同的网站，比如 B 站可以把流行视频分发到全国各地，而 CDN 厂商的广告就是在全球存在 xxxx 台服务器，可以同时存放内容来加速访问速度。



为什么 CDN 这么重要？因为互联网刚开始的时候并不是为了视频设计的，文本很快但是视频很慢。我们怎么让 client 知道 CDN Server 呢？

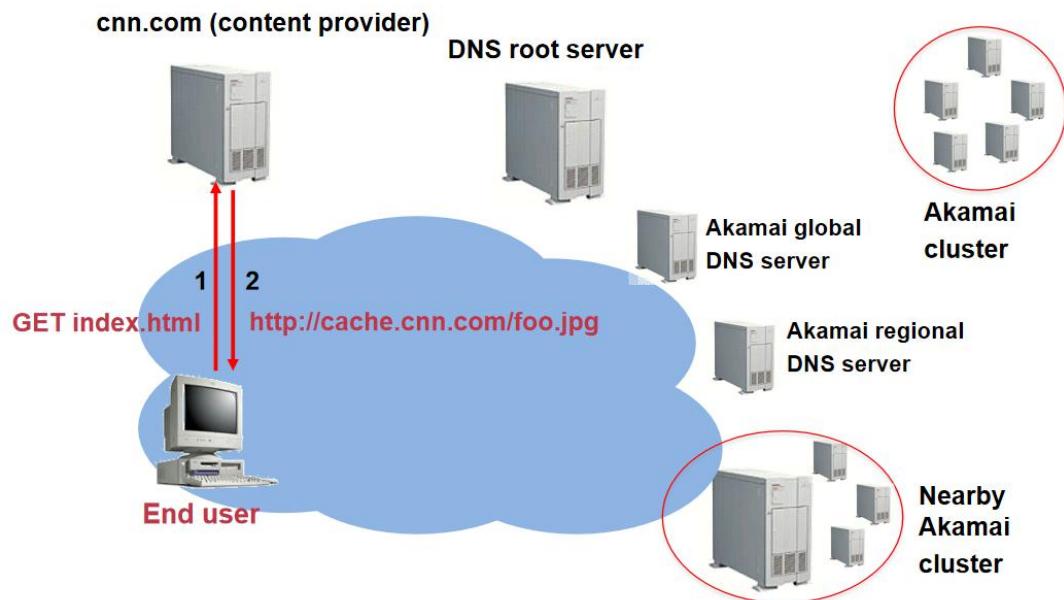
1. HTTP 重定向，B 站 Server 返回一个重定向的 CDN 服务器，B 站 Server 负责 redirect 到里请求 IP 最近的服务器。优点：1.控制比较细粒度，网页中的每个元素都可以重定向，比较方便。缺点：client 有 2 次 connection，server 为了告诉 client 要重定向要哪里，要做一些

运算。

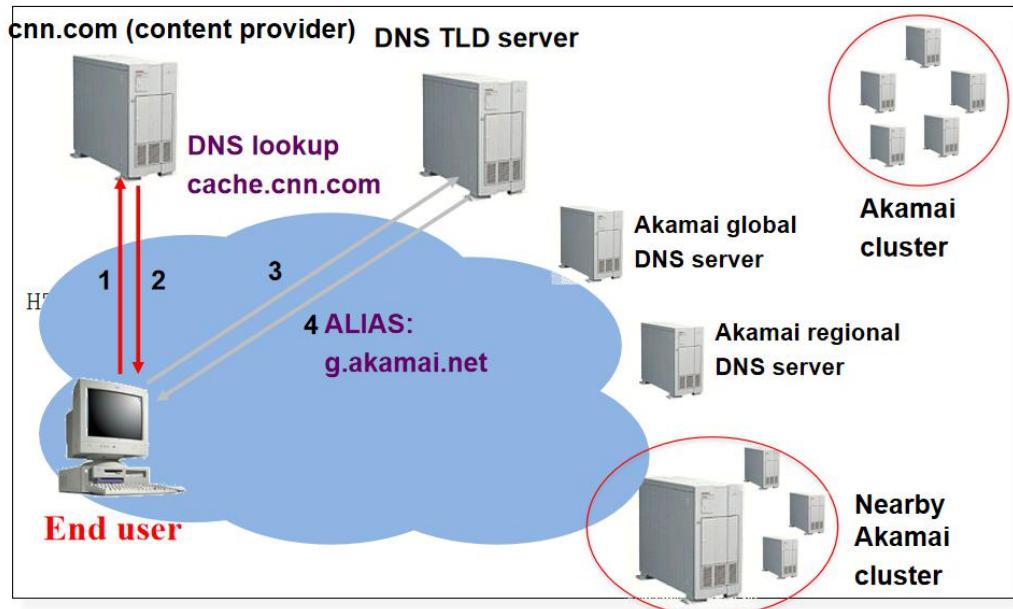
2. Naming, 基于 DNS 的 server selection。

这里有一个 local 的 client，它最终会通过 DNS 解析找到最近的 server。这样就可以把一部分工作放到 DNS Server 上，DNS 的 local 也有 cache，它可以有效避免 TCP 的 setup delay，并且可以 cache。缺点就是基于 local DNS Server 的 IP 地址，并且会有 hidden load effect。

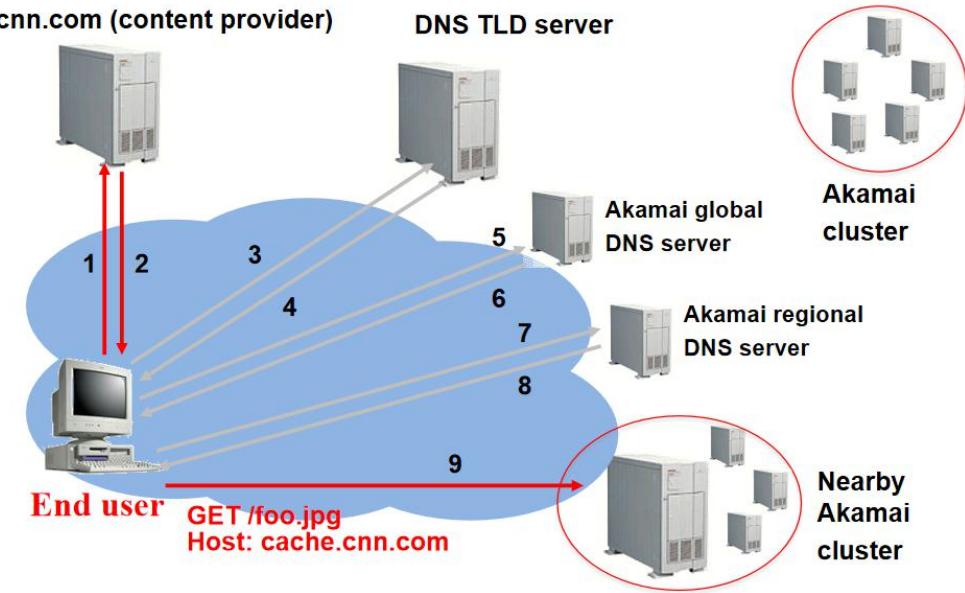
我们来看一个具体的 case：



这个是用户直接访问的例子。



做了一个地址到地址的映射，end-user 拿到 g.akamai.net 就会再去问这个地址对应的 DNS Server。



然后层层下问，就可以找到最近的 **cluster server**。当用户第二次访问的时候，DNS 已经 **cache** 住了，就不用一个个问下来了。

所以 CDN 对用户是透明的，如果没有域名映射到 IP 的机制的话，用户只能通过 IP 来获取资源。有了 DNS，我们可以直接在域名和域名之间做 alias。

有了 CDN 之后，淘宝和支付宝等就会大量购买 CDN 厂商的服务，进一步提高了全球范围内的用户体验，有了这个思路之后，CDN 很像一个全球分布的网络，严格意义上是一个多中心化的网络。但是它控制的还是单中心。对于多中心的网络就可以认为是中心化的控制面和去中心化的数据面。淘宝买 CDN 的服务的时候只需要和 Akamai 联系，而用户访问的时候就是去中心化的。所以 **to-B** 是中心化，而 **to-c** 是去中心化的。

一个想法，如果服务器允许自行加入会发生什么情况？如果每家每户都可以接入这个网络，那么就是 P2P 的 network。中心化的系统都会带来一些诟病，比如 **single point failure**。中央服务器挂了，全部都挂了。在没有中心化的网络之下，点到点怎么通信。

P2P 网络面临的挑战：1.整个 P2P 网络中到底有多少节点存了多少对象，2.怎么找到其他人呢？3.不同数据在节点之间怎么 **split** 呢？4.数据有没有可能放在某个节点，某个节点挂了就丢了呢？5.保证 **consistency**。6.保证 **security**。

BitTorrent

最早提出的就是 BitTorrent 协议。它在整个网络中有 3 种角色：**tracker**（记录哪些 **peer** 节点去服务哪些数据），**seeder**（做种，免费提供整个文件的节点，其他节点可能只有文件的 10%），**peer**（拥有一部分的文件，当它拥有 100% 的文件后，就成为了 **seeder**）。

使用了协作性的使用模型：用户使用一些简单的用户接口从其他人那里下载文件，下载的时候 BitTorrent 协议也会把文件发给其他人，并且在下载完成后，这个协议也会运行一小段时间。

首先有一个人的文件要分享给所有人，它就要发布一个 **.torrent** 文件到 WebServer 上，注意这个网站还是一个中心化的文件。我们现在讨论的是拿到 **.torrent** 的文件之后的操作。BitTorrent 说一旦我们拿到了这个 **.torrent** 文件，后面的过程都是去中心化的。我们要去连接

一个 tracker，通常是我们下载 BT 客户端的时候，它会写几个 tracker 在配置文件中。这一步也是中心化的。找到 tracker 之后，seeder 就会把 torrent url 给到 tracker，一旦我们下载的时候，我们就作为 peer 也为别人提供了服务。当我们去问 tracker 这个文件的时候，我们被 tracker 记录为了一个潜在的 peer 节点。

A torrent file

```
{
  'announce': 'http://bttracker.debian.org:6969/announce',
  'info':
  {
    'name': 'debian-503-amd64-CD-1.iso',
    'piece length': 262144,
    'length': 678301696,
    'pieces':
      '841ae846bc5b6d7bd6e9aa3dd9e551559c82abc1...d14f1631d
      776008f83772ee170c42411618190a4'
  }
}
```

把一个文件分成了很多个 pieces。客户端拿到 pieces 怎么下载呢？

1. 严格按照顺序下载
2. 有些 piece 比较稀有，稀有优先
3. 随机下载
4. 并发下载

BitTorrent 策略：第一个随机下，然后就是 **nearest first**，最后一个就会并发下载。

但是 BitTorrent 所有的记录都是有 Tracker 来完成的，但是 Tracker 还是一个中心化的节点，它要记录数据 ID 和数据所在的 peer 之间的对应关系如果有大量的 torrent，tracker 可能不够用。

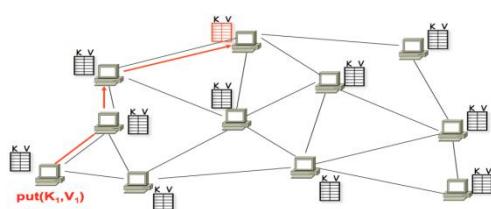
DHT (Distributed Hash Table)

所以有人提出了 DHT，一个文件分成 100 片，每一片都是一个哈希对应到一个 IP，也就是某一片数据谁有。之前的数据是保存在中心化的 tracker 上，而现在是保存在不同的分布式机器上。所以它就是一个去中心化的 key-value store，但是它很难保证 data alive，也就是存数据的机器可能关机睡觉去了。BT 协议就是能够在这个去中心化的情况下 work，所以我们也会有大量的数据的 replica。所以 BT 其实很难去做文件需要频繁更新的文件。

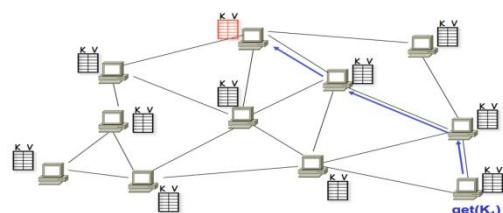
我们必然是会问一个节点数据在哪，如果不知道就再问下一个，这种方式其实和 IP 里的路由非常像。所以这种方式就是我们说的 network overlay。节点到节点之间的连接是已有的 IP 网络，但是它有一套自己的路由协议，其实就是拿着哈希找数据（类似于拿着 IP 找网络），所以我们要在现有的网络上再叠加一层路由协议。

最终目标是，网络如果很大的话，我们至少要做到 $\log(n)$ 的效率。

A DHT in Operation: put()



A DHT in Operation: get()



它可以通过一条路找到应该存放的机子上。Get 同理。所以核心是这个路由算法。

我们来看一个 Chord 协议，每次搜索保证了 $\log(n)$ 的节点，每个节点上的状态的大小也是和 $\log(n)$ 成正比，并且可以忍受大量的 failure。

Chord IDs

Key identifier = SHA-1(key)

- SHA-1 is a hash function

Node identifier = SHA-1(IP address)

Both are uniformly distributed

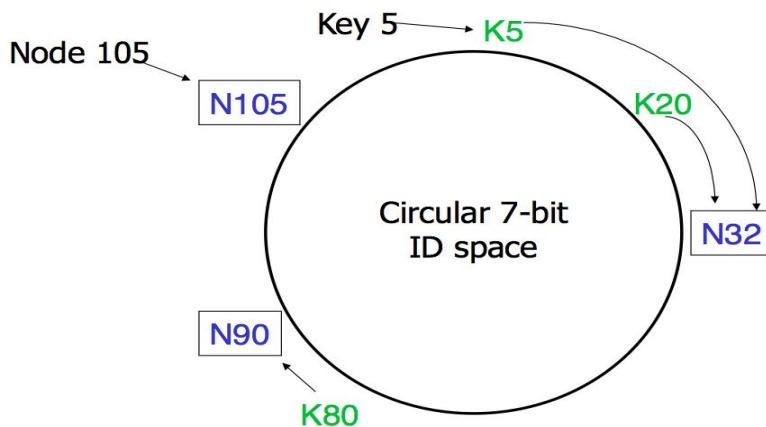
Both exist in the same ID space

How to map key IDs to node IDs? by mod?

对 IP 地址用 SHA-1 得到唯一 ID，这个就是在 P2P 里的网络地址，对于 key 来说的话同样也是做 SHA-1。最终这两个哈希值的域是一样的，所以它们的 ID Space 是一样的。

最理想的情况：如果我们要找一个数据 K，对数据 K 哈希一下得到 Hash(K)，我们想得到 IP 的哈希值 Hash(IP)，如果完全一致，找一下就行了。

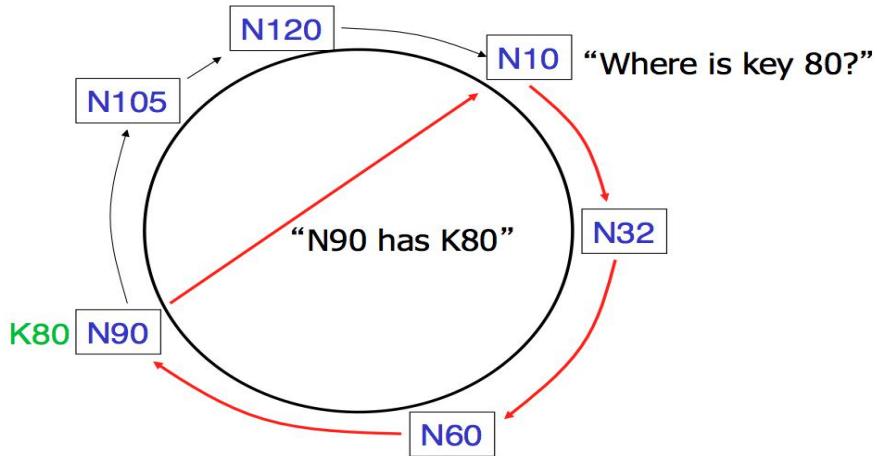
但是哈希空间很大，我们不能保证完全匹配。所以我们不能做点到点的匹配，所以我们只能保证点到范围的匹配。



A key is stored at its **successor**: node with next higher ID

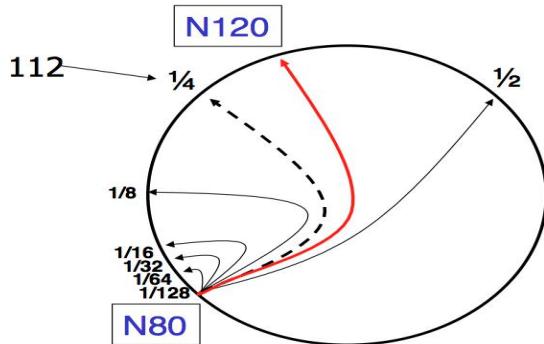
圈上的每一个点是一个哈希的结构，最上方是 0。我们全部哈希完后，顺时针找 key 的 hash 碰到的第一个 node hash，这样我们就可以通过哈希把所有的 peer 在 hash 空间上面分割成若干段，接收对应的 key。

一致性哈希的 $O(N)$ 算法



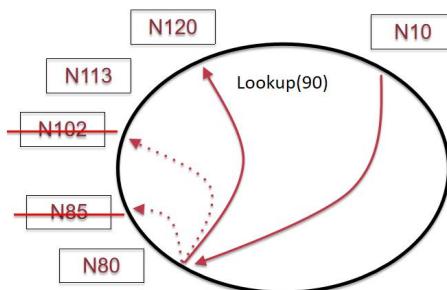
上图就是节点 10 希望找到 key 80 的情况。

一致性哈希的 $O(\log N)$ 算法



其实就是二分，我们每次找对家，这样就可以每次减半搜索空间。所以它每次只要额外的 $\log N$ 空间来维护对应的二分跳转点。每一个节点记录的表就是 finger table.

Failures might cause incorrect lookup

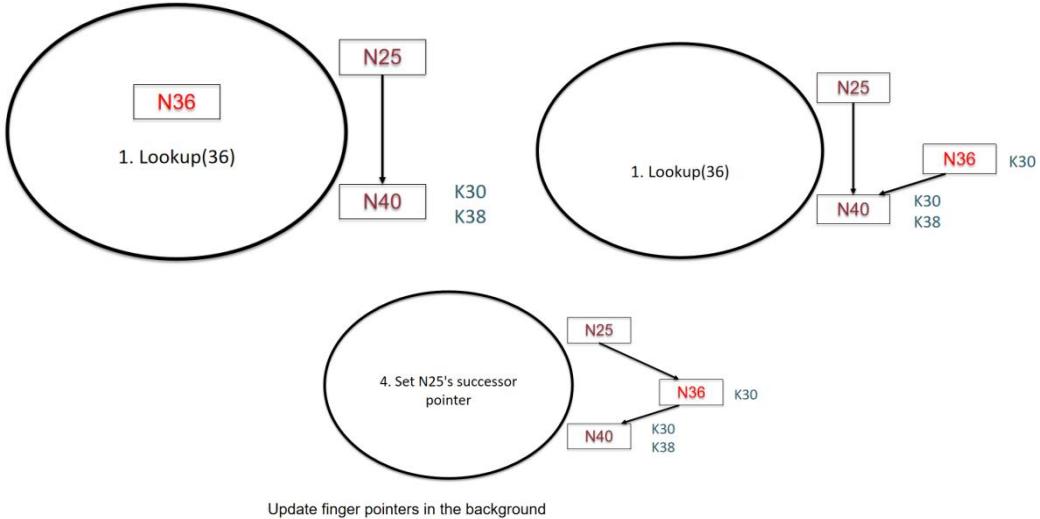


N80 doesn't know correct successor, so incorrect lookup

但是我们的节点可能是会 fail 的情况。比如我们要查 90，应该是 N102。所以我们还需

要维护一个 Successor List，也就是每个节点为了容错要去保存其后面的 r 个节点。如果 finger list 找不到，我们就用 successor list 去找，但是这样最坏情况退化到 $O(N)$ ，但是可以保证结果是对的。还有一个是 Join 的情况。

网络中新节点加入的情况怎么办？



此时 N36 要加入，通过 key 的 hash 查找机制，可以定位到 N40。然后 N40 一看对方是 36，它就会把一部分的 key (K30) 分给 36，最后一步就是把 N25 的 successor pointer 执行 N36。

用取模的方式完全可以实现分布式哈希表，但是考虑到 10 个 Node 加入 1 个新的 node 变成 11 个 Node 的情况，那么每个节点的数据都会发生变化。而在 consistency hash 中，只需要修改 N40 的数据。所以取模的计算方式不适合分布式 hash 的场景。

为了避免某些 Key 过于集中在一些 node 上，为了实现负载均衡，我们把 Node 变成 Virtual Node，也就是一台物理机可能同时包含 100 个虚拟节点。这样我们就可以把节点不再和物理机做一个绑定。

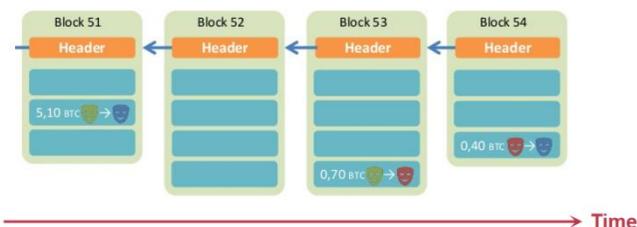
P2P 实现账本，就是 block chain。金融数据都是记录在银行/微信的服务器，一旦数据被篡改了，钱就变多/变少了。有没有可能我们把钱存在 P2P Network 中，全世界的机器都记录。

比特币假设全网 51% 的算力是 honest 的，为了保证数据不被篡改，它需要用 block chain 的方式。

Overview of BitCoin

BitCoin is a decentralized public-accessable ledger

- Using hashes as a pointers, which form a chain
- Tamper-proof (assuming 51% honest)



每个 **block** 有很多转账记录。这个转账记录可能被伪造，我们需要防止伪造。这个转账操作需要付钱的人的签名。把交易用私钥签名然后放到网上。我们把上一个 **hash** 值放到这个里面，**Block53** 的 **header** 就是整个 **block52** 的哈希值。随着时间增长，**block** 越来越多，形成了很长的一个 **Block**，这样没有人是可以篡改前面的 **log**。

Q: 如果 **block** 生成的速度太快了怎么办？

A: 会导致整个 **chain** 没办法在一台机器上存下来，但是作为个人不可能拿十台机器去存比特币的数据。最终有能力存比特币的又变回了中心化的数据中心。所以比特币严格限制每 10 分钟出一个块。存在一个随机数，必须使得算出来的哈希的前 100 位是 0。谁找到了这个随机数，谁就找到了一个合法的 **block**。这个协议就叫做 **Proof of work**，因为你找到了这个随机数，就代表是做了很多工作，所以这个就保证了 **fairness**。比特币的难度会随着全网算力的增长而增长，保证基本上 10 分钟出一个 **block**，这就保证了一秒钟最多做 7 笔交易。

现在新的区块链宣传 1 秒钟 10 万笔远超比特币，但是这事实上就等于会导致新节点加入到网络中的时候，下载 P2P 网络中日志的速度会慢于 P2P 网络中数据生成的速度，最终就会被全网的几个节点的垄断。

2021/12/14

上节课介绍了分布式的去中心化的系统比特币，它用到的核心数据结构的 **block chain**，每个 **block** 包含的是 **transaction**，也就是从 A 到 B 的转账记录，和我们的 **write-ahead log** 非常像。不一样的地方在于，当我们收集了很多 **transaction entry** 之后，我们可以变成了 4M 的 **block**，每个 **block** 的头上有一个 **header** 指向前一个 **block**，这样就可以把 **block** 串在一起变成一个 **chain**，一旦有人要篡改就必须把所有都篡改掉。但是它上面的所有信息都是公开的，所以我们可以把数据 **download** 下来，分析钱的流动情况。有很多经济学的研究就是基于比特币的日志，每次在转账的时候都会把数据算一遍。

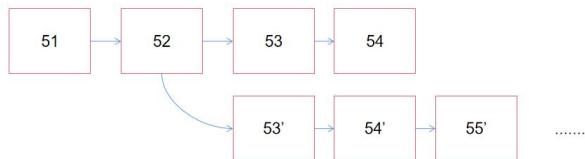
如果防御 **double spending**（一笔钱花两次呢）？如果我们只有一个比特币，在两个 **block** 中分别转给了 A 和 B，这就是 **double-spend**，所以每次做了以后都会 **check** 一下记录。

Proposal 阶段：矿工收集到了很多 **transaction** 之后变成一个 **block**，使得整个 **block** 的哈希前多少位等于 0，然后就 **proposal** 这个 **block**。

Validate 阶段：看看交易是否合法，比如 A 要转给 B，A 的签名对不对，以及 A 有没有

对应的钱，就是把前面的 double-spending 问题都考虑一遍。

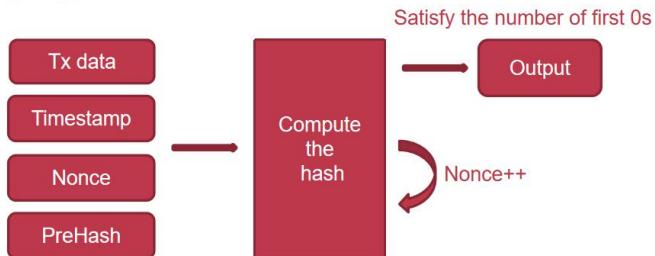
如果说有 51% 的人掌握了算力，这就意味着它们可以构造出新的一条链，强行把某些交易删除。也就是说，每个人都在 blockchain 中有一个自己的日志，然后我们删掉 block 后快速算一个哈希使得 block 是合法的。比如说我们计算出了 $53'$ ，我们的哈希值就变了。我们一直去算这条新的链的值，只要我们新的链比原来的链更长，



一旦 51% 的算力用来做坏事，我们信任的基础就崩溃了。我们花完钱之后，随时可以撤回比特币，那么就没有人再会使用比特币。但是从经济学的角度，一旦掌握了 51% 的算力，投入了那么多，不会使用这种攻击导致比特币的价值归零。

Proof of Work (PoW)

- Brute force to solve a random number (nonce)
- Inefficient, waste of resources



Proof of Work 导致算哈希的时候的算力是白白浪费的，所以比特币的价值是不会比电费低的，所以很多矿场都会搭建在水电站旁边。

这里有一些具体的操作和前面说的是一样的，激励机制（让更多人加入到网络中，矿工可以收一点 transaction fee）。既然比特币最大的拥有者不希望比特币崩溃掉，那么我们就相信它。

有一个 proposal，把所有验证过程交给拥有币最多的人，让它来记账，因为它是最不希望币会跌的人。也可以选择一个 committee，比如说 100 个节点，只要其中 51% 的节点的 honest 的。所以我们需要一种全网都同意的随机算法，只要我们保证每轮都是随机的并且每轮更换的速度很快，那么同时出现 51 个坏人的概率就很低。

Can some one other than the owner spend a coin?

A: 如果没有私钥签名，是不能在比特币上提交 transaction 的。

What if I lose my private key?

A: 没有办法。

Is it possible to have inflation?

A: 有没有可能发生通货膨胀，比特币不会有。但是会有通货紧缩的问题，因为总有人会丢掉自己的 private key。

Is mining really decentralized?

A: 取决于有没有人可以超过 51%。很多研究者做过一些研究过程，认为不能真正的算作去中心化。因为 2019 年大部分的算力在中国。

有了比特币之后，比特币在上传的时候可以加上长度为 100 字节的注释。这个注释引发了很多现象。我们可以让它变成一个脚本，并且要求矿工必须要执行这个脚本。比如说：“我承诺给 D 一个比特币，但是这个条件还没有满足，假如后面 100 个 block 的最后一位是单数，那么我们就给”这其实就是一个赌约，但是没有人可以更改这个赌约，到那个时候所有计算机都会执行这个脚本，只是延迟了。

我们可以把它一个通用的脚本，可以让它变成一个图灵完备的脚本，让全世界挖矿的计算机都在执行这个脚本，就是一个非常大的分布式系统。一下子把比特币从分布式记账变成了分布式计算。底层的脚本，它是发明了一个新的脚本语言。所以就是以太坊。

以太坊提出的脚本就是 **smart contract**，签署合同如果赖账，就需要很麻烦的法律手续，但是如果记在代码中了，我们就不需要法律维权了，这样就改变了生产关系，对信任的追求可以变成对分布式的追求，这件事情很难做到但是非常有吸引力。

一旦有 **smart contract** 之后，有人认为比特币后面 100 个字节就太少了，于是就出现了很多分布式的文件系统可以保存更多的数据。

比如说向分布式文件系统上传一张照片，它会给你一个证明，证明它确实在保存这个照片。

Larger Storage

Store general data, instead of only transaction

Then a node needs to prove it does store the data

- Proof of retrievability, in challenge-response form

Examples: IPFS, STORJ

比特币的第二个问题就是匿名的，会导致非法交易，它是一个 **permission-less chain**。所以我们就需要一个非匿名的 **permissioned chain**。但是这就是让互不信任的人又实名认证了，比较奇怪，许可链（**permissioned chain**）在目前应用不大。

如果有一个人大家都相信它，那么我们就不需要基于 **trust** 的东西了。

Digital Scarcity and NFT

A non-fungible token (NFT) is a unique and non-interchangeable unit of data stored on a digital ledger (blockchain). NFTs can be associated with reproducible digital files such as photos, videos, and audio. NFTs use a digital ledger to provide a public certificate of authenticity or proof of ownership, but it does not restrict the sharing or copying of the underlying digital file. The lack of interchangeability (fungibility) distinguishes NFTs from blockchain cryptocurrencies, such as Bitcoin.



NFT 非常简单，当我们在 blockchain 上产生一个交易，它就拥有了一个唯一的时间戳。有人就可以把收藏品联系在一起。比如画家把画的画扫描成电子版，然后把哈希放到区块链的链上，那么它就是 block 中第一个把哈希放上去的人，这就是一个唯一的数字凭证。我们就可以拍卖在区块链上的第一个 token。

我们刚刚讲的这些都是现在在发生的事情。以及我们看到的去中心化到底是不是去中心化，也需要在检查。

安全性

在进入数字世界的前夜，如果我们没有考虑安全问题，那么我们面对的风险也是更大的。

个人信息偷盗问题；钓鱼问题；电脑被植入后门，成为僵尸网络；Stuxnet（攻击的是伊朗的离心机，通过假证书的方式冒充微软的安全补丁更新进伊朗的机器中）；内部员工盗窃用户数据去卖；跨站脚本攻击；千年虫事件；log4j 漏洞

安全性是 negative goal。Positive goal 比如说读一个文件，很容易测试。一个 negative goal 是证明 Bob 不能读到这个文件，但是这个就很难。

访问 exam.txt 的方法：

1. 修改读写权限
2. 绕过 permission check，直接读磁盘
3. 通过网站 url 找到
4. 电脑关掉之后内存没来得及清零，把电脑拿过来 dump 一下
5. 打开 vi 后，会有一个 backup copy 没有被删除
6. 把网络包截取下来
7. 可以发一个假的 vi，vi 里嵌入了这个木马，或者把 ls 替换成假的 ls
8. 直接到办公室把磁盘拔了
9. 从废纸堆中找到打印了一半的文件

10. 打印机有 cache，有所有打印过的文件
11. 打电话给管理员，重置密码盗取网盘账号
12.

所以我们列这个 list 是永远列不完的，所以 negative goal 是很难达到的。

Why is Security So Hard?

Security sometimes conflicts with other goals

- Security VS. performance
- Security VS. availability
- Security VS. fault tolerance
- Security VS. convenience
- Security VS. simplicity
- ...

安全性经常和别的特性产生冲突。为了容错，比特币钱包有一个私钥，拿纸条抄下来，但是就容易被盗窃。攻击和错误是不一样的，攻击是有针对性地利用一连串的 **fault** 去做坏事，我们为了容错可以放 3 备份，但是不能用 3 备份解决安全问题，只会适得其反。安全没有一个完整的方案，我们不能让一个系统百分百变得安全。所以我们要去想怎么对一个系统从安全的角度去建模，我们怎么去思考风险是什么；第四个就是我们要找到一个 **tradeoff**。

我们要对被保护的目标非常的清晰，要对我们的假设非常权限。

安全分为 CIA（**confidentiality, integrity, availability**），我们可能对于一个场景需要其中之一和之二。

比如一份考卷，CIA 都是需要有的。所以讨论安全的时候我们先要问自己要保护哪个属性。

Threat Model: assumption

攻击者可以是任意人，我们可以假设攻击者可以控制网络中的一部分电脑而不是所有电脑。在比特币中的假设就是全网 51% 的算力的 **honest** 的；攻击者可以控制电脑上的一部分软件；攻击者知道关于你的一部分信息（比如你的密码）；攻击者知道你的软件中的 **bug**。

2021/12/16

计算机安全是一件很难的事情，因为它是一个 negative goal。比如说我们做 lab 要在任何测试用例的测试下都跑 100 分，这就是很难的一件事情。

首先要明确 policy (CIA, 也就是我们的目标)，即 **confidentiality, integrity** 和 **availability**。它不一定要三个同时存在，也有可能是只有一个和只有两个，不同的目标有不同的保护的方

法。

然后我们需要假设，我们相信什么；我们不相信什么；攻击者是谁，在区块链中，我们假设攻击者最多控制了 49% 的算力。有些模型会认为 OS 是不可信的，只相信 Hypervisor。对于银行卡来说，就要考虑 physical attack，里面包含了一拆就会坏掉的装置。有些设备里面有一些酸性物质，一拆就腐蚀掉原数据了。还有一些是社会工程学（social engineering），比如我们接到的诈骗电话。还有一个就是资源问题，比如一个密码一开始需要一万年破解出来，但是计算机的算力也在增加。APT（advanced persistent threat）以国家为后盾的攻击，可以动用很多资源。

Threat model 也不能随便设定的，比如说攻击者在防火墙的范围之外。但是这一点经常不成立，因为从外面到里面并没有这么复杂，并且还有内鬼问题。所以 Google 在五年前提出了概念就是零信任，也就是没有公司的内网和外网之分，员工可以在任何地方连接到公司网，所以要把安全性建立在连到公司网的那一刻，而不是建立一个物理的围墙。

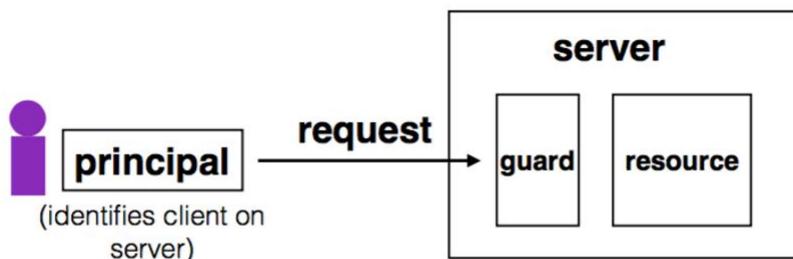
第二个不好的 threat model 就是“攻击者不知道被攻击者的密码”，“软件不开源，所以攻击者不知道我们的软件是怎么 work 的”。其实这些都不太多。

GUARD MODEL

它有两个基本的要素，你要有一扇门，门口要站一个保镖。我们要求这是一个 complete mediation，换句话说，旁边不能有一扇窗导致能够绕过这个保镖。以之前的访问.txt 文件为例，我们能不能把访问它变成一种方法，有一扇门我们就可以更好地保护它。

比如我们的 exam.txt 只有一台服务器保存，我们就可以通过拔网线+放进只有一扇门的机房来把访问它的方式只有一种方法。那么在 OS 中，我们可以视作文件在我们的 OS 服务器上，因为一个应用系统必须通过 OS 的 system call 来访问到这个文件。这样我们就可以认为文件系统是访问文件不可绕过的模块，我们就可以在文件系统做 guard。

如果有 3 种方法访问到一个资源，但是只有两个地方加了 guard，这就不太合理。



Guard typically provides:

- **Authentication**: is the principal who they claim to be?
- **Authorization**: does principal have access to perform request on resource?

在这个过程中要做两件事情，

1. 认证：确认身份
2. 授权：检查是否有访问权限，并且给某个 principal 赋予某种身份的能力。

有了这两者之后，我们举个例子我们的文件系统。我们之前一直问大家的一个问题是：为什么访问文件的时候要通过 `fd` 的方式。

其实核心在于，当我们打开一个文件拿到了一个 `fd` 之后，其他进程是不能访问到那个文件的，所以 `fd` 是和进程、打开方式绑定的。如果我们拿到的是一个 `inode-number`，那么其他进程拿到了 `inode number` 也可以访问文件。所以 `fd` 使得 `os` 在访问文件的过程中是不能被绕开的。

Example: Unix FS

Resource: files, directories.

Server: OS kernel.

Client: process.

Requests: read, write system calls.

Mediation: U/K bit / system call implementation.

Principal: user ID.

Authentication: kernel keeps track of user ID for each process.

Authorization: permission bits & owner uid in each file's inode.

Kernel 会为每一个进程保留一个 `userid`, `process` 的 `userid` 是怎么得到的呢？在登录的时候我们会输入账号密码，然后 `OS` 会判断账号密码，给你这个 `shell`。后面的所有进程都是从登录的 `shell` 进程 `fork` 出来的，所以都有这个 `user id`。

通过对进程的 `userid` 的检查，在加上文件 `inode` 中记录的权限位，就完成了安全的访问控制。

Example: Web Server

Resource:	Wiki pages
Client:	any computer that speaks HTTP
Server:	web application, maybe written in Python
Requests:	read/write wiki pages
Mediation:	server stores data on local disk, accepts only HTTP reqs
Principal:	username
Authentication:	password
Authorization:	list of usernames that can read/write each wiki page

Example: Firewall

Resource:	internal servers
Client:	any computer sending packets
Server:	the entire internal network
Requests:	packets
Mediation:	<ul style="list-style-type: none">– internal network must not be connected to internet in other ways– no open wifi access points on internal network for adversary to use– no internal computers that might be under control of adversary
Principal, authentication:	none
Authorization:	check for IP address & port in table of allowed connections

防火墙建立在公司外网出口的地方，client 就是所有在外网的地方，它有一个端口的表来表明哪些端口是可以通过的。

问题

1. 软件 bug 导致有些数据可以绕开 complete mediation
2. 有些内存可以绕开 OS 防御
3. User 会犯错，我们一定要考虑人在环路（human in the loop，人是系统中的一部分，而最容易出错的地方就是 human）
4. 安全等级
5.

软件的 bug 不可避免，通常 kernel 中每 1000 行有一个 bug。攻击者有可能利用这些 bug 做出一些攻击。解决这个方式就是减少那些最关键的代码，我们需要让内核尽可能变小。这

也就是学术界为什么会有微内核这样的概念：内核越小，bug 越少。学术界对于两个软件是否安全的评价标准只有一个，也就是代码行数。当然可以用形式化验证。

永远不要用 root 登录，因为 root 的权限太高了，我们登录一个计算机之后，最好使用一个最小权限的用户登录。这里有一个 example，paymaxx.com 可以用来计算税单的一个网站，需要用户名和密码登录。但是 url 是明文的，所以任何人只要知道账号就可以下载那个人的税单，这就是没有做到 complete mediation。

第二个例子是 SQL injection，

Example: SQL Injection

username	email	public?
matei	matei@sjtu.edu.cn	yes
mike	mcarbin@sjtu.edu.cn	yes
katrina	lacurts@sjtu.edu.cn	no


```
SELECT username, email FROM users WHERE
username='<username>' AND public='yes'

Let <username> = katrina' OR username='
SELECT username, email FROM users WHERE
username='katrina' OR username=' AND
public='yes'
```

今天我们防止 SQL 注入会使用很长的一个正则表达式，会把那些危险的单双引号过滤掉。字符串解析是和安全最相关的地方。

认证

接下来一个问题就是假设我们已经有了认证和授权步骤，我们接下来分别一个个看来判断到底谁是谁。

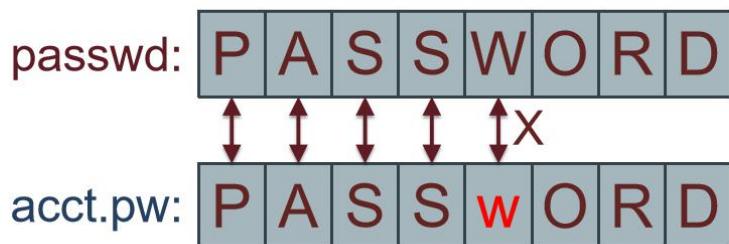
认证步骤，我们需要根据

1. “你知道什么”：密码
2. “你有什么”：门槛和钥匙，可能有很多分
3. “你是什么”：你的指纹、虹膜、步态、声纹。

这里有一个检查 password 的代码，大家看看有什么问题：

An Example: Guessing Password (Tenex)

```
checkpw (user, passwd):
    acct = accounts[user]
    for i in range(0, len(acct.pw)):
        if acct.pw[i] != passwd[i]:
            return False
    return True
```



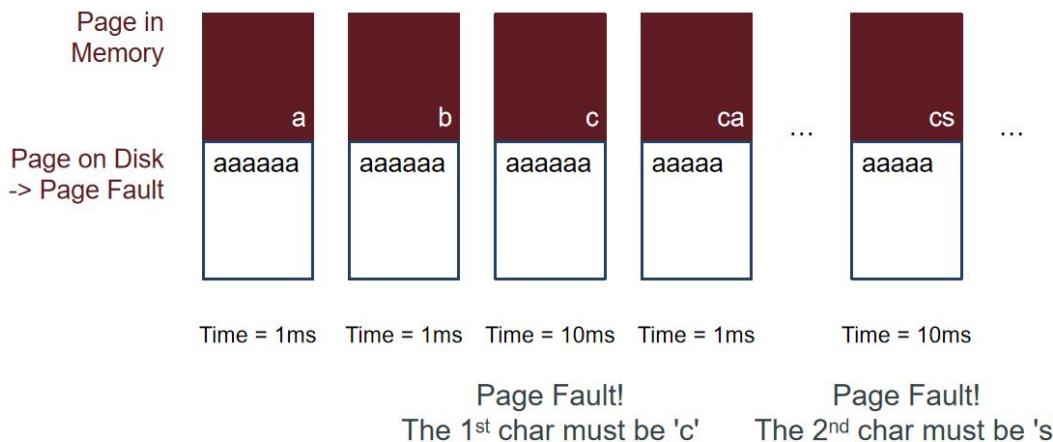
Q: 第一位错和第二位错差了几个 cycle?

A: 一次循环差了 10 个 cycle 左右,但是很难测出来,因为运行的误差都大于十几个 cycle。

Q: 怎么把第一位对和第一位错的边界放大?

A: 在内存里我们做这样一个布局,

Timing Attack: Guess One Character at a Time



我们把密码的第一位放在 4K 交界的边缘, 如果第一位是对的, 会访问第二位, 此时因为第二个 page 不在内存中, 这就会导致一个 swap, 这样正确和错误的时间就会显著的不一样。

怎么做到这一点呢?

攻击者只需要把密码放在文件中, mmap 到内存中。攻击者也可以去 malloc 一块内存, 拼命使用其他内存把这两块先 swap 到硬盘上, 再把一块 swap 到内存里。

这种就叫做 timing attack, 在早期, 很多人认为这存在于理论上, 而最近 timing attack

的攻击越来越多了。

怎么解决这个问题呢？

1. 算哈希值，计算多长的密码，时间都差不多。我们的方法就是存一个哈希值，到时候用哈希值去比较密码，这样做我们最后都不需要保存密码的原文。这样，服务器端就存在用户和密码的哈希值。

username		hash(password)
arya		de5aba604c340e1965bb27d7a4c4ba03f4798ac7
jon		321196d4a6ff137202191489895e58c29475ccab
Sansa		6ea7c2b3e08a3d19fee5766cf9fc51680b267e9f
hodor		c6447b82fbb4b8e7dbcf2d28a4d7372f5dc32687

```
check_password(username, inputted_password):
    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password)
    return stored_hash == inputted_hash
```

那么我们作为攻击者该怎么攻击这个机制呢？

密码和一般的数据具有一些区别，它是具有一定特点的，所以攻击者就构造出了 Rainbow table 的方式。

也就是最常见的密码的哈希表。直接通过撞库攻击把密码撞出来。

Salting

Salting

username	salt	hash(password salt)
arya	5334900209	c5d2a9ffd6052a27e6183d60321c44c58c3c26cc
jon	1128628774	624f0ffa577011e5704bd0760435c6ca69336db
Sansa	8188708254	5ee2b8effce270183ef0f4c7d458b1ed95c0cce5
Hodor	6209415273	f7e17e61376f16ca23560915b578d923d86e0319

```
check_password(username, inputted_password)
    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password | salt)
    return stored_hash == inputted_hash
```

注意偷到 salt 后，我们就可以再把 Rainbow 再算一遍。这就拖慢了攻击者的攻击速度，它在攻击的时候性能就下降了。

这就是“用己方非常小的代价提升攻击者非常大的代价”。

在 buffer overflow lab 中，我们需要把 ASLR (Address space layout randomization) 关掉，否则地址就会变来变去。这就是用己方非常小的代价提升攻击者非常大的代价。

Session Cookies: Strawman

First check username and password, if ok, send:

```
{username, expiration, H(server_key | username |
                           expiration)}
```

Use the tuple to authenticate user for a period of time

- Nice property: no need to store password in memory, or re-enter it often
- Server_key is there to ensure users can't fabricate hash themselves
- Arbitrary secret string on server, can be changed (invalidating cookies)
- Can verify that the username and expiration time is valid by checking hash

Cookie 中包含如上。因为攻击者没有服务器的 server_key，所以不能伪造 cookie 信息。所以在网站登陆完之后，下一次提交请求的时候，服务器就要检查一下这个 cookie 前后是否对得上。如果 cookie 做的不好，合在一起以后可能会被解释成别的。

Session Cookies: Strawman

Problem: the same hash can be used for different username/expiration pairs!

- E.g., "Ben" and "22-May-2012" may also be "Ben2" and "2-May-2012"
- Concatenated string used to compute the hash is same in both cases!
- Can impersonate someone with a similar username

Principle: be explicit and unambiguous when it comes to security

- E.g., use an invertible delimiter scheme for hashing several parts together

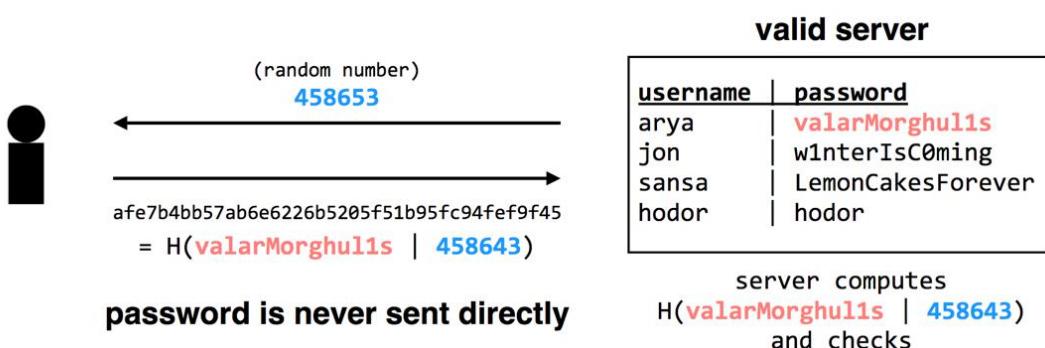
也就是早期的 web cookie 存在二义性，合理的做法就是明确地表示 user=Ben, exoeration=22-May-2012。所以我们一旦做字符串拼接的时候，我们需要考虑是否分开的时候会导致歧义。

1.challenge-response 的协议

用密码最大的一个问题就是，一旦把密码发给服务器之后，它是可以冒充你的。如果服务器上被人安装了一个后门，这样别人就可能冒充你。我们怎么样不把密码发给服务器完成验证呢？

我们可以做一个 challenge-response 的协议，服务器发给你一个 random number R，我们在本地算 $H(R+password)$ ，这样就算我们的 H 中途被人拦截到了，它也不能被使用第二次，因为 R 是随机的。所以这样子就可以保证中间没有人可以拿到你的 password。

Technique 1: challenge-response scheme



Adversary only learns $H(valarMorghulis | 458643)$; can not recover the password from that

2. use passwords to authenticate the server

第二个就是反过来，让服务器证明它知道你的 password: client 选一个 Q 发给服务器，服务器计算 $H(Q+password)$ ，因为只有合法的服务器才知道你的密钥。

但是在这个情况下，存在如下的一种攻击：

1. 尝试登录服务器，得到服务器的 challenge R。
2. 再和服务器说想测试服务器，把服务器返回的 R 发给服务器让它去做验证。
3. 把服务器返回的 $H(R+password)$ 返回给服务器作为 challenge-response scheme 的结果。

这就等于左手倒右手，利用服务器来破解密码，所以这两个机制不能在一起同时使用。

3. Turn offline into online attack

Offline attack: 攻击的时候没有在服务器上留下一些痕迹。中国银行会要求尝试登录的人选择一张图片，然后选出之前上传的图片再输入密码。如果一个攻击者通过遍历用户名攻击的方法把一千个人的图片都拿到手，在这种情况下，中国银行就会发现一个人老是只输入用户名不输入密码，所以银行是看得到这个攻击的。

所以任何一个小小的改进，因为完全安全性很难，所以任何一个改进都是有意义的。

4. 每个网站最好有不同的密码

5. One-time password

One-time Password

- If adversary intercepts password, can keep using it over and over
- Can implement one-time passwords: need to use a different password every time

Design: construct a long chain of hashes.

- Start with password and salt, as before
- Repeatedly apply hash, n times, to get n passwords
- Server stores $x = H(H(H(\dots(H(salt+password)))))) = H^n(salt+password)$

To authenticate, send token= $H^{n-1}(salt+password)$

- Server verifies that $x = H(token)$, then sets $x \leftarrow token$
- User carries a printout of a few hashes, or uses smartphone to compute them.

当我们在服务器端设置一个密码的时候，我们有一个 `password` 和 `salt`，服务器就嵌套算 100 次哈希，保存其结果。

在验证的时候，我们在本地算 99 次哈希。Server 把传过来的结果再算一次哈希，就可以匹配了。然后服务器把这个 `x` 替换原来的 `x`，也就是保存了 99 次哈希的值。下一次我们只需要传算 98 次哈希的值。

最后 100 次用完之后，就再重新设置一个密码，这样就可以控制 `password` 的使用次数，如果有一个攻击者拼命在那里试密码。

Google's App-specific password

Application-specific passwords

Step 2 of 2: Enter the generated application-specific password

You may now enter your new application-specific password into your application.
Note that this password grants complete access to your Google Account. For security reasons, it will not be displayed again:

fmin nnwg bftf dppi

No need to memorize this password.

You should need to enter it only once. Spaces don't matter.

[Done](#)

Reeder on MBA
Chrome on MBA
Gmail on iPhone
Test for CSE

Dec 2, 2012
Dec 2, 2012
Dec 10, 2012
Dec 20, 2012

Dec 19, 2012
Dec 2, 2012
Unavailable
Unavailable

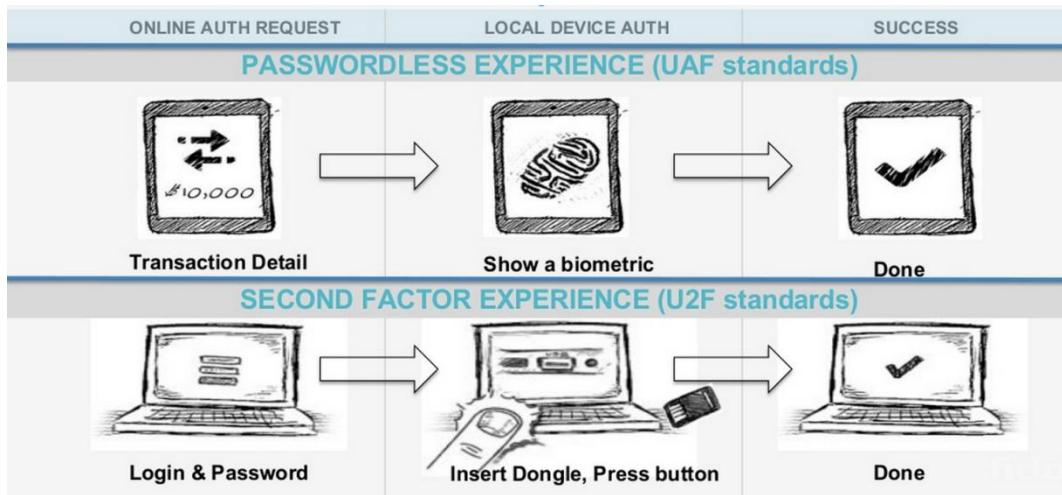
[[Revoke](#)]
[[Revoke](#)]
[[Revoke](#)]
[[Revoke](#)]

6. 我们将 application 和 request authorization 绑定在一块

我们为每个请求都加上一个 `authorization`。所以 `req` 就变成了：`req = { username, "write XX to exam.txt", H(password + "write ..") }`；Server 看到我们要做这个事情，我们就可以通过计算这个哈希来认证。类似于 cookie+session 的机制。

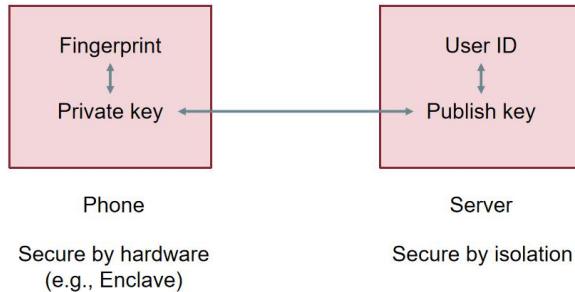
7.FIDO (replace the password)

能不能把 `password` 取消掉。



这个设备保存了你的私钥，当我们登录一个网站的时候，它如果支持 FIDO 协议，它就会检查你电脑上是否插入了这个小设备。

Three Bindings



通过这个小设备实现指纹和私钥的绑定，而在 **server** 端实现用户名和公钥的绑定。通常设备端的安全性会更加重要一些。我们今天的手机在识别指纹和人脸的时候，数据不是放在 iOS 和 Android 里的，我们今天的手机里有一个额外的 OS，叫做 TrustZone，只有当我们需要识别身份的时候，它才会启动。支付宝会放入一个 trust application，要求 TrustZone 做一个认证检查。如果我们不用 TrustZone，而是直接把数据放在 iOS/Android，这就意味着只要有一个应用有 root 权限，它就可以无限模拟支付的流程。

那么在讲密码的时候，就一个很容易被忽视的点。密码修改其实是一个非常危险的事情，今天修改密码已经比以前安全很多了，比如给邮箱发送一个 URL，再点进去，但是如果我们邮箱被人攻击了，那就没有办法了。服务器在生成 reset password url 的过程中，到底是怎么生成这个 url 呢？它里面一定要包含用户名+多长时间过期，如果攻击者知道生成 url 的算法，那它一个自己生成这个 url。所以 url 里面也要放一个随机数，专门有一个论文调查各大网站的重置密码 url 是怎么生成的。里面用到了 /dev/random 里的一个随机数。键盘按了多少次，鼠标产生了多少次中断，有多少个网络包。

2021/12/21

我们继续开始讲系统安全这一块。我们先来复习一下上节课的内容，我们想要保护我们的关键数据，我们可以使用 **guard model**。我们有一个门卫，每当来一个人的时候，他先来

看一下你是谁，然后从他的本子上看你有没有这个权限访问这个资源。这个资源可以是文件、机器设备、OS 自己抽象出来的虚拟对象、token 等。但是在这个过程中，我们必须要做两件事情：

1. 验证你是谁（认证，authentication）：我们可以通过你有什么（钥匙、密码器）、你知道什么（密码）、你是谁（你的一些生物的特征）
2. 判断你是否拥有访问资源的权限（授权，authorization）。

那么今天我们看来换一个角度看一下数据的 secure flow。我们从数据的角度来看一下计算机的运行。计算机的数据流的跟踪有两大类：

1. 把数据看成一个需要保护的对象。
2. 把数据看成可疑的攻击。

这两个方法使用的方法有一定类似的地方，也就是我们在计算机中跟踪数据的流动。我们先看来一下 data flow protection，我们先来看一下攻击者有什么办法来偷到我们的数据。

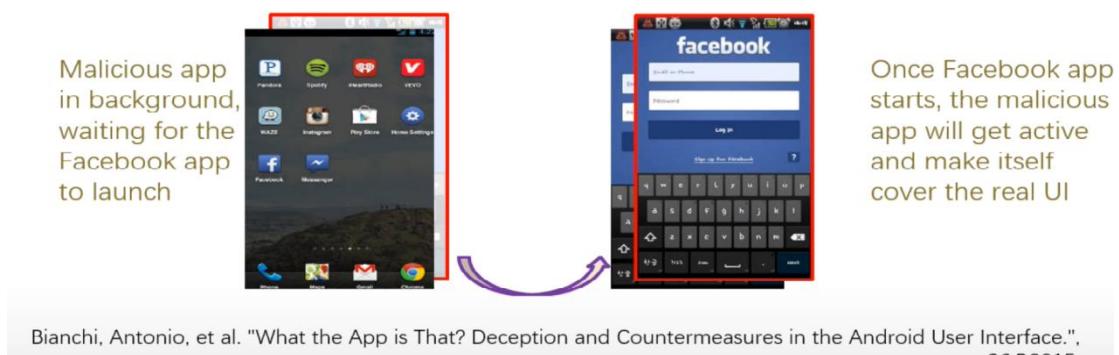
攻击者如何偷走我们的数据

KeyLogger

第一种方法就是 KeyLogger，也就是安装一个能够记录按键的木马，这样我们就可以记录下所有的输入。我们在输入银行卡账号密码的时候都会摸键盘，所以键盘是知道我们的信息的。比如手机推送的广告正好是我刚才说过的东西。

输入法是我们进入手机的入口，我们使用的所有东西它都知道。它对于关键词的准确度的匹配是相当高的，所以它知道我们在各种各样的软件中搜过什么、聊天说过什么。

第二种方法就是 Fishing，在我们笔记本上就是一个错误的 URL 出现一个一模一样的界面来骗取我们的密码，在手机上可以做的非常地隐蔽。



运行完之后隐藏在后台不出来，等我们真正启动 Facebook App 的时候，它通过转到前台的方式切过来，也就是我们刚刚点了 Facebook，然后瞬间通过一个动画弹出来，在这种情况下是很难预防的。并且手机上的另一个特点是全屏的，我们没有办法看到其实有两个界面。

MemScan

第三个方法就是 **MemScan**，就是使用一些具有更高权限（**root**）的软件，对整个内存都做一个扫描。有些手机是有 bug 的，比如三星曾经有一款手机把 **/dev/pmem** 的权限设置为了 777，而 **pmem** 就是 **physical memory**，原先应当是只有内核才能访问的，这样就等于把物理内存完全暴露出来了。

Q：就算有了这么一个物理内存的数据（01 串），我们怎么样把里面的东西偷到呢？

A：注意在交换密钥的时候，需要用到大质数。那么我们在扫描的时候比较容易可以发现这个模式。可能这个质数就是用来做密钥生成的。如果我们还知道用户名等，我们一下子可以在内存中找到用户名字符串所在的位置，紧跟着的这块区域可能就是我们的密码。

(a) Examples of data extracted from RAM / DB.

App	Extracted Cleartext Data
Email	password, email contents, subjects, from/to, contacts
OI Notepad (doc)	document and metadata
KeePass (password mgr)	app password, all stored passwords & descriptions
Pageonce (finance)	password, transactions, bank account information
Facebook (social)	wall posts and messages

(b) Exposure of cleartext sensitive data across all 14 apps.

App	Description	Extracted Cleartext Data			Cleartext Data Hoarding				
		Pass-word	Cont-ents	Meta-data	RAM		Pass-word	Cont-ents	Meta-data
Email	email (default)	Y	Y	Y				Y	Y
GMail	email	Y	Y	Y				Y	Y
YI Mail	email	Y	Y	Y				Y	Y
GDocs	documents							Y	Y
OI Notepad	documents	Y	Y	Y	Y	Y	Y		
Dropbox	documents							Y	Y
KeePass	password mgr	Y	Y	Y	Y	Y	Y		
Keeper	password mgr	Y	Y	Y	Y	Y	Y		
Amazon	commerce								Y
Pageonce	finance	Y	Y	Y	Y	Y	Y		
Min	finance	Y	Y	Y	Y	Y	Y	Y	Y
Google+	social	Y	Y	Y				Y	Y
Facebook	social	Y	Y	Y				Y	Y
LinkedIn	social	Y	Y	Y				Y	Y

(c) Example usage of hoarded data by apps.

图 1 数据泄露问题

由上表我们可以知道，GMail 在登录后会删除在内存中的密码，而帮助我们记住密码的 App：KeePass、Keeper 等，它们的数据库在内存中的明文保存密码的，会直接导致密码的泄露。

因为应用程序开发的时候，默认内存是安全的，过于依赖 OS 提供的进程间的隔离机制，而进程间隔离依赖的是虚拟内存。一旦我们的物理内存泄露了，那么虚拟内存建立的隔离机制就不复存在了。

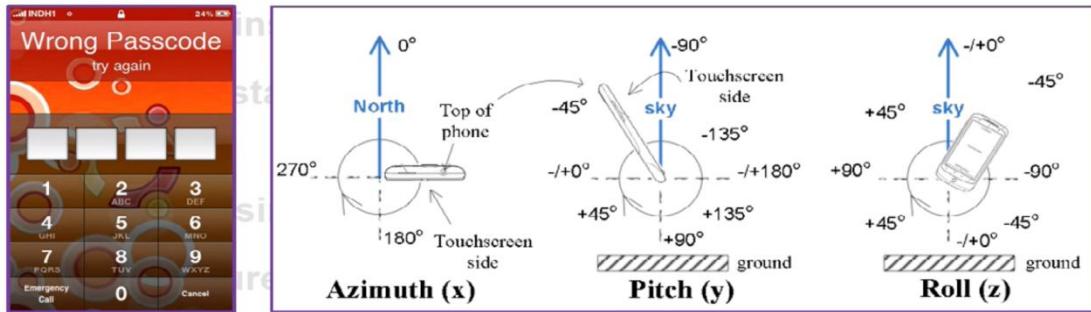
ScreenCapture

最后一类就是截屏，我们每次输入密码的时候都会显示最后一位，但是如果有应用有截屏权限，那么可能就会还原出密码。

Cold-Boot

这是一类物理攻击，利用的是手机内存低温的时候会保留数据这个特点。把手机偷走放在冰柜里冰起来，通过插拔电池对手机做微重启，进入到预先设置好的 OS 里，并且此时原先关机前的内存数据并没有消失，就可以把之前的内存都读出来，找到加密 SSD 的密钥，然后我们就可以读手机存储芯片中的所有数据了。

Side-Channel



根据输入密码的时候手机的陀螺仪的倾斜程度还原出输入的密码。

Taint Tracking

那么既然有这么多种方法可以偷走我们的数据，我们怎么样才能保护它呢，这时候就需要讲一下 Taint Tracking。

它的思路是说，既然要做安全，我们要定义要保护什么数据，并不是所有的数据都不是一样重要的。比如我们装了一个网上下载的 App，需要 200M 空间。可以公开获得的数据和代码是没有保护价值的。而在 App 使用过程中使用的账号密码就是需要保护的。

对于需要保护的 sensitive data，它的生命周期应当最小化。因为它在内存中的时间越长，它就越容易被攻击。因为在内存中可能发生 Swap，因为 OS 并不知道我们的数据是重要的还是不重要的数据，所以 OS 在不知道数据关键等级的情况下，把一个包含密码的内存交换到了磁盘，那么我们只要把磁盘偷走，我们就得到了密钥。除了 swap 之外，还有 hibernation（休眠），当我们休眠的时候，一部分数据会被写到磁盘上，防止断电以后数据就没了。

休眠分为两种模式：

1. 比较快，合上笔记本再打开立马显示出来。这个时候的数据还在内存中。
2. 慢一点，需要等一会儿才能进入到我们合上笔记本之前的状态，此时的数据就需要从硬盘上恢复进内存中。

如果我们可以把关键数据染上颜色，告诉 OS 当我们写数据的时候，看到有颜色的数据就不要 swap 或者加密再写磁盘，这样我们的数据就不会泄露。

比如我们可以把密码染色，当我们在发网络包的时候，我们看到了尝试发送一个染色的 sensitive data，那么这个时候我们就可以禁止住。我们也可以把信用卡号打上标签，禁止一个应用程序把信用卡号发给另一个应用程序。

Q：现在假设我们是写一个恶意应用程序的人，我们也知道了拿到的信用卡号打上了一个标签，我们怎么样才能把这个标签删掉呢？因为颜色本身也是一种数据，肯定有一种方式把颜色编码进来，这个数据就叫做 Paint。data 和 taint 之间就会联系在一起。我们作为攻击者来说，怎么样才能把这个数据安全地发给别人呢？

A：把有标签的数据，打一个压缩包，再拆成十份发送，尝试破坏原先 data+taint 的格式。

但是 taint 之所以叫做 data flow tracking，就是因为当我们打上压缩包并且拆分成 10 个

包以后，这十个压缩包都会被染上颜色。任何一个压缩包出去的时候都会触发警报。所以那么我们怎么样在这个过程中让颜色数据和原先的数据非常紧密地联系在一起，让任何操作都不能拆分开它们。

Dynamic Taint Analysis

```
i = get_input();           // 表格追加 i = 6, taint = True
two = 2;                  // 表格追加 two = 2, taint = True
if (i % 2 == 0) {
    j = i + two;         // 表格追加 j = 8, taint = True (因为是 taint + !taint 产生的数据)
    l = j;                // 表格追加 l = 8, taint = True
}
else {
    k = two * two;
    l = k;
}
jmp l;
```

假如我们现在有一个关键数据 *i*，为了跟踪 *i* 在程序中是怎么流动的，我们需要维护一个表格。在运行第一行的时候，我们就会记录下来第一行：*i = 6, taint = True*。*taint = True* 代表这是一个关键数据被我们记录下来了。

所以通过这个最终，我们就会发现 *jkl* 都是关键数据，当它要做一些发送关键数据的时候，我们是可以最终判断出来。

但是这个方法并没有考虑到控制流的问题，如果 *i* 出现在 *if* 语句的条件中，每次至少都可以判断出关于 *i* 的 1 bit 的信息。

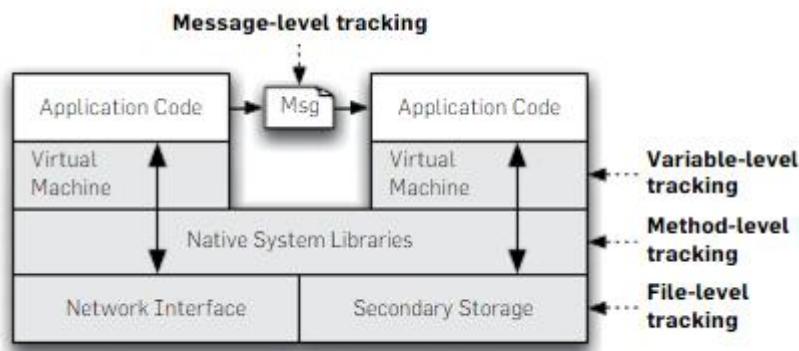
不过正常代码的数据流是比较正常的，可以对绝大部分的应用使用 *taint* 的方案。

Q: 那么到底是谁来做表格的维护呢？当我们去做 *j = i + two;* 的时候，为什么会有表格的更新呢？

A: OS 吗？难道每次运行一行指令的时候要触发一个 *syscall* 吗？

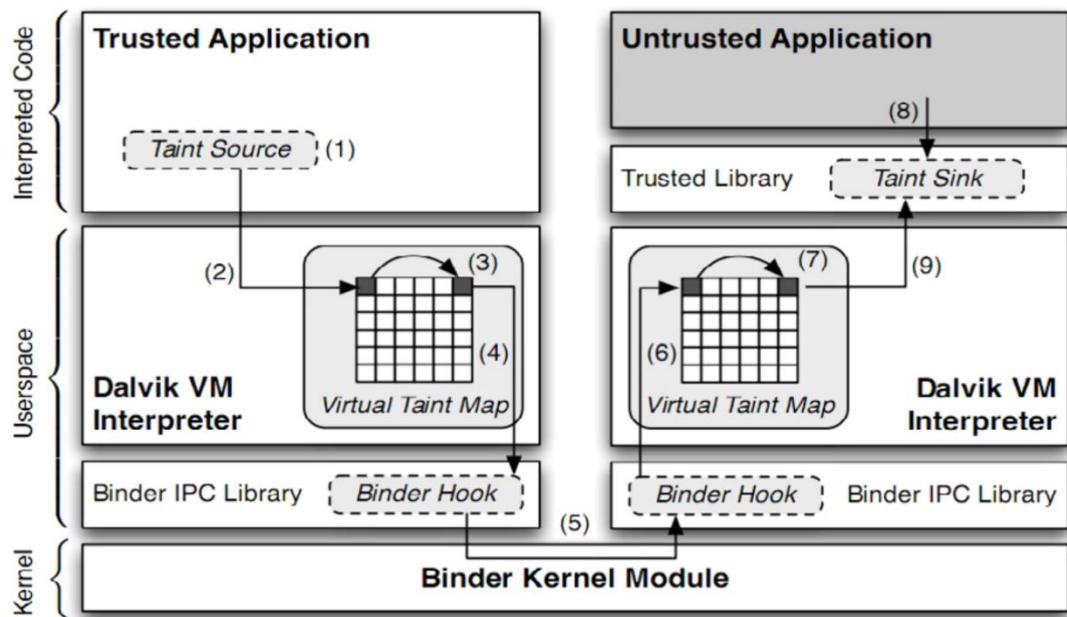
其实 *taint* 的维护和更新是在编译器从源代码生成汇编代码的时候维护的。拿“*j = i + two;*”为例，此时编译器产生汇编指令的时候就不仅仅是要把两个寄存器相加，还需要取出 *i* 和 *two* 各自的 *taint* 做一个 *or* 操作，再放进 *j* 的 *taint* 中。并且为了实现这个算法，必须在变量附近的位置再空出 64 bit 来存放变量对应的 *taint*。所以可想而知生成的汇编代码的效率是很低的，在实际过程中可能会慢 10 倍~50 倍，因为除去额外的汇编行数不说，还可能把原先寄存器中的计算操作变成访存操作。

TaintDroid



有一个程序员为了做这件事情修改了安卓的 OS，使其产生的汇编指令支持 taint tracing（污点追踪）。比如用户在使用 APP 时打入的账号密码、GPS 产生的经纬度数据、照片、IMEI（手机序列号），获取这些数据的 API 都会给对应的关键数据染色。

那么还有一个问题，应用程序下载的程序是怎么样也让它支持染色的呢？如果我们通过反汇编+汇编的方式，成功率可能不到 20%。这个就涉及到安卓的历史了，最开始的安卓程序都是以 Java 字节码发布的，下载到本机后，会通过本机上的 JVM（java 虚拟机）转变成汇编代码。所以其实我们只需要魔改 JVM 的实现，也使其支持染色即可。



我们 application 有一个 taint source，当 java 虚拟机里面同一个变量 copy 到另一个变量的时候，jvm 就会负责帮我们把 taint 一起传递过去。当我们想要从一个应用程序通过 IPC 发到另一个应用程序的时候，它把发送的消息做了一个简化，比如有一个 1K 的消息，这个 1K 的消息，只要有一个 byte 是 taint 的，那么它就认为整个消息是 taint 的。然后进到虚拟机里面继续使用比较细粒度的做一个拆分。并且一个文件只要有一个 byte 是 taint 的，那么它就认为整个文件是 taint 的，这样就可以减少 taint 的数量。这样子虽然会导致精度下降，但也足够使用。

这样去做了以后，比如我们的一个应用程序拿到了一个 GPS 数据，它把 GPS 数据通过

各种加密和压缩放到另一个地方，再通过 IPC 的形式发到另一个应用程序。但是 taint source 还是可以追踪到有应用程序在做坏事。

具体 taint 的流动我们可以简单看一下 binary operation 的例子，

Op Format	Op Semantics	Taint Propagation	Description
<i>const-op</i> $v_A \leftarrow C$	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
<i>move-op</i> $v_A \leftarrow v_B$	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>move-op-R</i> $v_A \leftarrow R$	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
<i>return-op</i> $v_A \leftarrow v_A$	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint (\emptyset if void)
<i>move-op-E</i> $v_A \leftarrow E$	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
<i>throw-op</i> $v_A \leftarrow v_A$	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
<i>unary-op</i> $v_A \leftarrow \otimes v_B$	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>binary-op</i> $v_A \leftarrow v_B \otimes v_C$	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint \cup v_C taint
<i>binary-op</i> $v_A \leftarrow v_A \otimes v_B$	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
<i>binary-op</i> $v_A \leftarrow v_B \otimes C$	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
<i>aput-op</i> $v_A \leftarrow v_B[v_C]$	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[v_C]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
<i>aget-op</i> $v_A \leftarrow v_B[v_C]$	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
<i>sput-op</i> $f_B \leftarrow v_A$	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
<i>sget-op</i> $v_A \leftarrow f_B$	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
<i>iput-op</i> $v_A \leftarrow f_C$	$v_B[f_C] \leftarrow v_A$	$\tau(v_B[f_C]) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
<i>iget-op</i> $v_A \leftarrow f_C$	$v_A \leftarrow f_C$	$\tau(v_A) \leftarrow \tau(v_B[f_C]) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

其实就是除了本身要做的事情，字节码在翻译的时候还要把 taint 的对应操作加上。最终 TaintDroid 的额外性能花销是 30%。就是因为它把大量的 taint 的传递放到了文件级和消息级，这样我们就不用维护 Byte 级别的 taint 改动。

刚才有同学说，这个 Android 太老了，现在大量有 native code，这个编译出来的二进制就已经不是字节码了，怎么办。TaintDroid 想的办法就是它把.so 文件里用到的所有函数列了一个大表，举个例子 `strcpy`： $\text{taint}(dst) = \text{taint}(src)$ ，这样我们就不需要跟踪 `strcpy` 里的每一行，只需要找到里面的 dst 和 src 即可。最终这个表格是 380 多个函数，一个个手动标记。

那么讲完 TaintDroid 之后，我们来讲 malicious input，它和 TaintDroid 的逻辑是反的。TaintDroid 是认为用户的输入是很关键的，需要保护起来。但是用户的输入也有可能是恶意的。

我们以如下一个例子举例。

How does a Hacker Search a Bug?

Step-0: Chose a library (open-source, of course)

- Let's try ffmpeg, which is used in Chrome, VLC, Mplayer...
- There are also rumors that YouTube uses it for conversion

Step-1: List the demuxers of ffmpeg

Step-2: Identify the input data

Step-3: Trace the input data

现在我们选择开源的 `ffmpeg` 作为攻击对象。因为 `ffmpeg` 的应用非常广泛，如果我们要在 Linux 上看视频和看音频，都需要这个多媒体解码库。并且 YouTube 也在使用它，如果我们能找到这个 bug 并且上传到 YouTube，让 YouTube 的服务器在解码的时候触发这个 Bug，那么我们就有可能能够控制 YouTube 的服务器。

所以 step1 是找到 `ffmpeg` 所有编码和解码的文件中。

```
tk@ubuntu:~/BHD/ffmpeg/libavformat$ ls
4xm.c      flic.c      mpjpeg.c      rtp.c
adtsenc.c   flvdec.c   msnw_tcp.c   rtpdec.c
aiff.c      flvenc.c   mtv.c       rtpenc.c
allformats.c flv.h      mvi.c       rtpenc_h264.c
amr.c       framecrcenc.c mxif.c      rtp.h
apc.c       framehook.c  mxifdec.c  rtp_h264.c
ape.c       framehook.h mxifenc.c  rtp_h264.h
ASF.c       gif.c      mxif.h     rtp_internal.h
asfcrypt.c  gxf.c      network.h  rtp_mpv.c
asfcrypt.h  gxfenc.c   nsfdec.c   rtp_mpv.h
asf-enc.c   gxf.h      nut.c      rtpproto.c
asf.h       http.c     nutdec.c   rtsp.c
assdec.c   idcin.c    nutenc.c  rtspcodes.h
assenc.c   idroq.c    nut.h     rtsp.h
au.c       iff.c      nuv.c     sdp.c
```

在这个过程中，他找了 `4xm.c` 这个格式。这是攻击者最喜欢的小众格式，因为小众就有可能有 bug。

`demuxername_read_header()`

- Most demuxers declear a function called it
- Takes a parameter of type AVFormatContext
- This function initializes a pointer:

```
[..]
ByteIOContext *pb = s->pb;
[..]
```

- Many different get_something() are used to extract pb data
 - E.g., get_le32(), get_buffer()
- Conclusion: pb is a pointer to the input data of media files

对于所有的这些文件，都有一个共性的函数，也就是 `_read_header`，也就是给你一个媒体文件。文件肯定有一个 `header`，这个函数就可以根据这个 `header` 来跟踪数据。然后通过 `pb` 这个指针读进来数据。

然后在 `4xm.c` 中发现了这么一行存在隐式的类型转换：

```
166      current_track = AV_RL32(&header[i + 8]);
          int           unsigned int
```

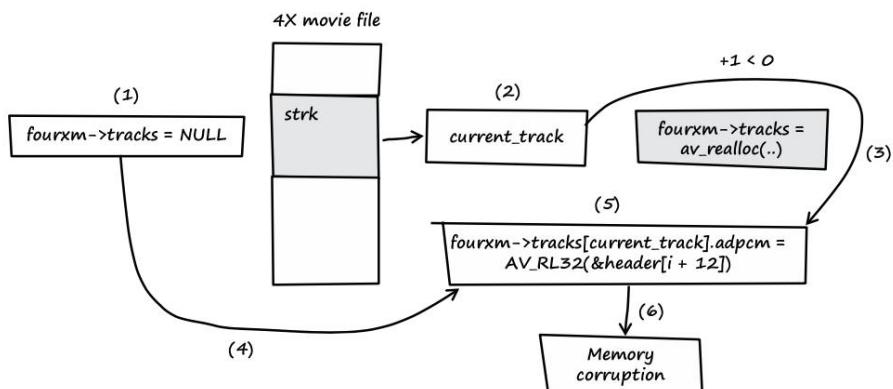
这样，如果我们设置 `header` 的值使其转换完大于 `0x80000000`，那么最终 `current_track` 就是一个负数。

```

167     if (current_track + 1 > fourxm->track_count) {
168         fourxm->track_count = current_track + 1;
169         if((unsigned)fourxm->track_count >= UINT_MAX / sizeof(AudioTrack))
170             return -1;
171         fourxm->tracks = av_realloc(fourxm->tracks,
172                                     fourxm->track_count * sizeof(AudioTrack));
173         if (!fourxm->tracks) {
174             av_free(header);
175             return AVERRORE(NOMEM);
176         }
177     }
178     fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
179     fourxm->tracks[current_track].channels = AV_RL32(&header[i + 36]);
180     fourxm->tracks[current_track].sample_rate = AV_RL32(&header[i + 40]);
181     fourxm->tracks[current_track].bits = AV_RL32(&header[i + 44]);
[...]

```

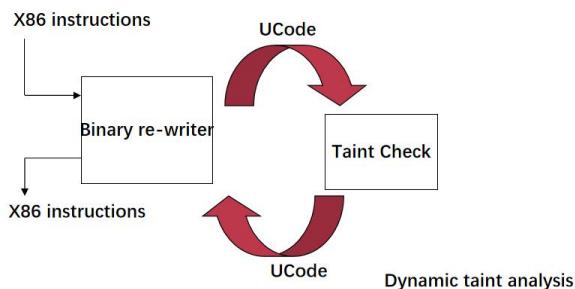
就会跳过 167~177 行，直接尝试往一个数组中的负数范围内去赋值。这个其实已经实现了往任意内存中写任何值，因为 `header` 是我们可以控制的。



TaintCheck Detection Module

我们能否使用 Taint Tracking 技术来找之前的这些 Bug？

我们可以想一个策略，当我们在写一个内存地址的时候，这个地址不能是和传入的数据相关的，否则我们就认为是一个非法的。包括说我们跳过去的一个地址不能是 taint 的数据，否则攻击者就可以根据输入的数据来修改我们的控制流。



这是一个工具，不需要源码，传入的就是 2 进制，在每一位上去做 taint check。这个过程分成三步：

1. 标记 Taint，比如说来自网络的不可信的数据。
2. 跟踪 Taint，在做加减法/copy 的时候。
3. Assert Taint，当我们把 Taint 数据作为函数的跳转地址的时候，就会报警。

TaintCheck Detection Modules

TaintSeed: Mark untrusted data as tainted

TaintTracker: Track each instruction, determine if result is tainted

TaintAssert: Check is tainted data is used dangerously

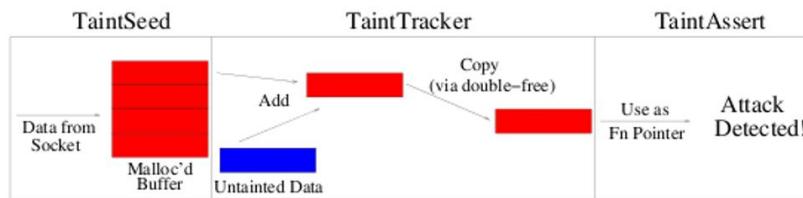


Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).

在我们前面这个例子中，可能就是赋值的时候看见是 Taint 数据就报警。当然，因为它的精度很高，所以会造成约 37 倍的额外开销。

No Data To Protect

我们手机中有关键的数据，可能在内存、硬盘中，并且 Taint 也有可能被洗掉。那么有没有可能我们不在手机里存放关键的数据呢？

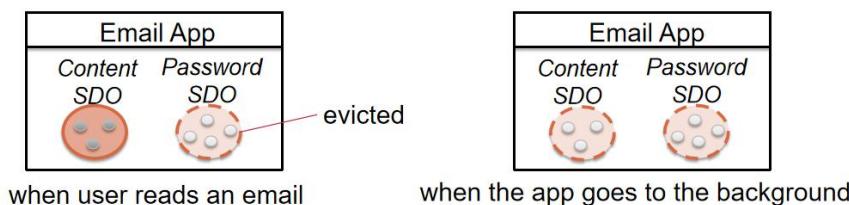
CleanOS [OSDI'12]: New Abstraction: SDO

Sensitive Data Object

Applications **create SDOs** and **add sensitive data to them**

CleanOS implements three **functions** for SDOs:

1. **Tracks** data in SDOs using taint tracking
2. **Evicts** SDOs to a trusted cloud whenever **idle**
3. **Decrypts** SDO data when it is accessed again

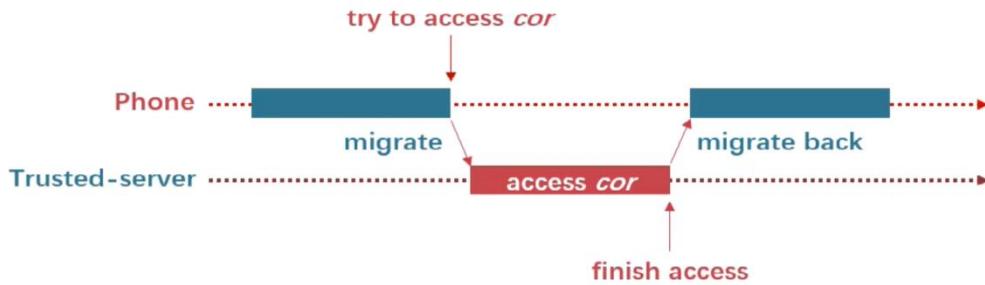


也就是应用在后台的时候，GC 会识别出 sensitive data，并且逐步加密，当我们拿起手机再准备用的时候，数据会解密。这样哪怕有人在我们不用的时候复制出了我们的内存，他也不能破解。

Migrate Confidential Data Access to a Trusted Node

Security-oriented offloading

- An app is migrated to the trusted node when accesses *cor*
- The offloading engine is based on COMET [OSDI'12]



另一种方法是，当我们要登录应用的时候，把我们的 QQ migrate 到家里，登录完了再 migrate 到手机上。所以我们手机丢了都无所谓，因为手机上从来就没有密钥。

安全信道

当数据在端侧，我们有各种各样的办法去保护。当我们要不可避免地传输数据的时候，我怎么该保证传输通道是安全的。所以我们不可避免地讲到加密过程。

```
encrypt(key, message) → ciphertext
decrypt(key, ciphertext) → message
encrypt(34fbcb1, "hello, world") = 0x47348f63a67926cd393d4b93c58f78c
decrypt(34fbcb1, "0x47348f63a67926cd393d4b93c58f78c") = hello, world

property: given the ciphertext, it is (virtually) impossible to
obtain the message without knowing the key

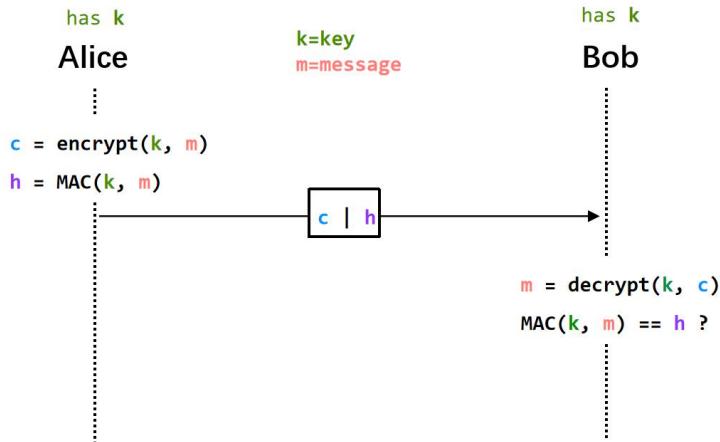


---

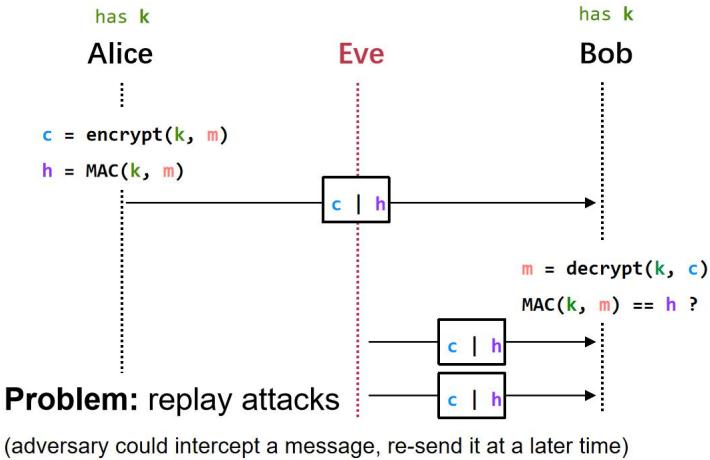

MAC(key, message) → token
MAC(34fbcb1, "hello, world") = 0x59cccc95723737f777e62bc756c8da5c

property: given the message, it is (virtually) impossible to
obtain the token without knowing the key
(it is also impossible to go in the reverse direction)
```

基本上就是给一个 **key** 和 **message**，产生密文。另外一种就是 **MAC**，它和哈希有不太一样的地方，哈希是一一对应的。而 **MAC** 可以认为是一个加密的哈希，得到的 **token** 一定和 **key**、**message** 对应起来。我们可以证明 **token** 包含了 **key** 和 **message**，但是没有办法逆向获得 **key** 和 **message**。

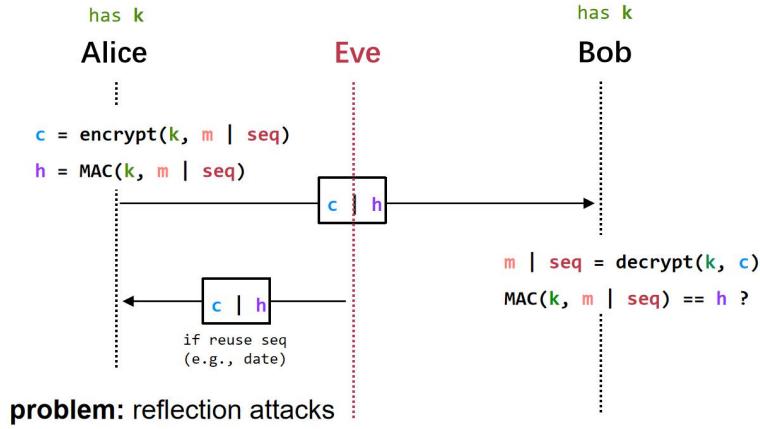


现在我们假设 Alice 要和 Bob 通讯，我们假设它们现在各自拥有一把 key。Alice 要把 message 发给 Bob，她就把得到密文 c 和 MAC h 发过去。Bob 就会重新计算一下 m ，并通过 MAC 来核实 $\text{MAC}(k, m) == h$ 。MAC 就是用来判断密文是否在传输过程中被人改了。



这时候我们有一个中间人 Eve，如果 Alice 发的是登录请求，Eve 就可以截获下来重放。那么 Eve 就可以用截获的东西登录 Alice 的账号。对于 Bob 来说，他不能判断这个消息是来自 Eve 还是 Alice。

为什么会有 Reply Attack？因为我们的包被发了很多次。要解决这个问题，我们需要给包加一个 ID（sequence number）。我们只需要保证 sequence number 是递增的，这样我们就可以发现两个包的 sequence number 是一样的。



(adversary could intercept a message, re-send it later in the opposite direction)

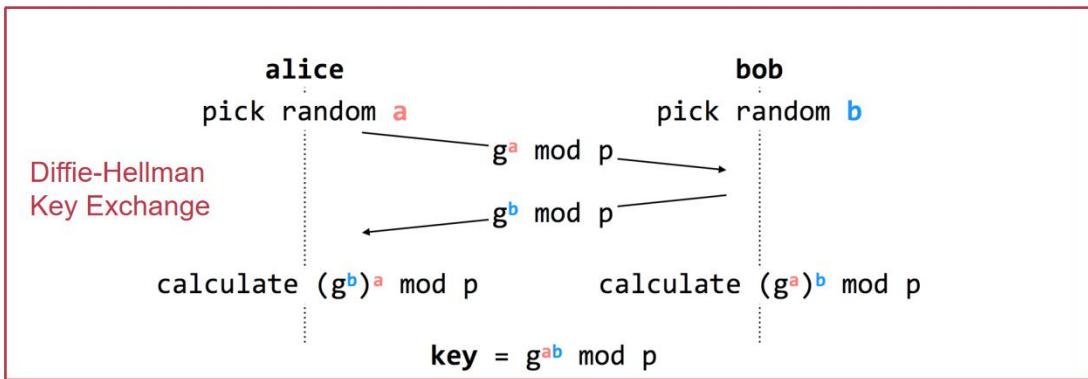
但是 Eve 可以把截获的 Alice 发送的消息还给 Alice，从而得到递增的 sequence number 的消息序列。Alice 可以把消息标记为 from Alice。另一种方式是让 Alice 和 Bob 各自维护一个 Key，当 Alice 向 Bob 发送的时候使用 Key_a ，而 Bob 向 Alice 发的时候就用 Key_b 。这样我们就解决了发和收可能被混在一起的问题。接下来的问题是，我们怎么让 Alice 和 Bob 在一开始的时候交换这个 Key 呢。我们就要来介绍 Diffie-Hellman Key Exchange Algorithm。

DH 密钥交换算法

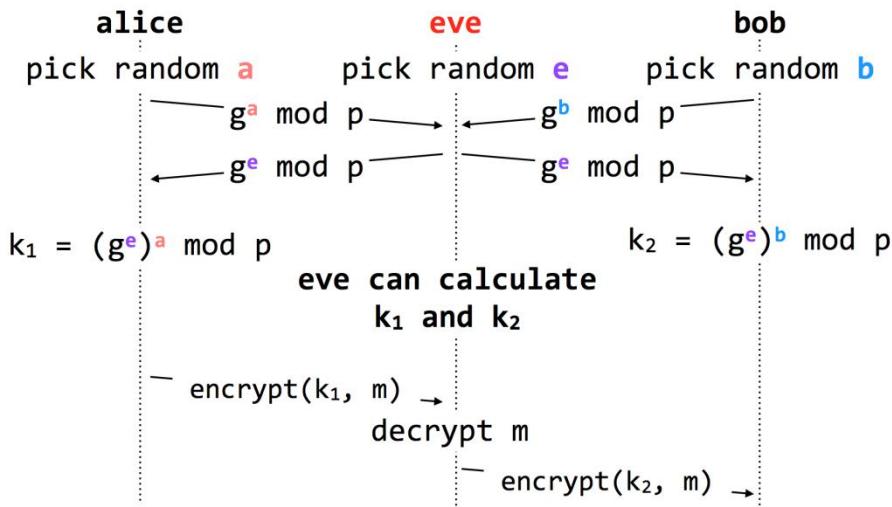
▶ **problem:** how do the parties know the keys?

known: p (prime), g

property: given $g^r \bmod p$, it is (virtually) impossible to determine r even if you know g and p



Alice 和 Bob 可以先通过公开渠道确立质数 p 和数 g 。然后 Alice 随机生成一个数 a ，计算 $g^a \bmod p$ 发送给 Bob，Bob 随机生成一个 b 计算 $g^b \bmod p$ 。这样双方都可以计算出 $key = g^{ab} \bmod p$ 。对于旁人来说，单单从 p 和 g 是很难逆向推出 a 和 b 的。



problem: alice and bob don't know they're not communicating directly

但是 DH 算法不能抵御中间人攻击，因为 Eve 可以骗到 Alice 说他是 Bob，并且骗到 Bob 说他是 Alice。中间人攻击没有别的办法，解决方案只有 RSA（非对称加密）。

RSA（非对称加密算法）

也就是我们有一个公钥和私钥。公钥可以解密私钥加密的内容；私钥可以解密公钥加密的内容。用私钥加密可以产生一个签名，是因为只有我们才能写这个东西。我们用公钥可以去验证这个签名。

举个例子，假设全世界范围内，有一个人我们想和他说一句话，并且我们不希望这句话被别人听到。那么我们就需要把他的公钥拿过来加密，这样全网就只有他才能解密；又比如说，我们现在希望对全网发布一个声明。我们就可以用自己的私钥对发布的消息签一个名，然后放在网上。

所以前者的信息是加密的，而后者的信息是公开的，这是因为应用场景不同。

我们就可以用这种方式让每个人都有公钥私钥对，这样 Alice 就可以先用 Bob 的公钥发给 Bob，哪怕有人截获了也不能破解。然后 Bob 用自己的私钥解密就可以得到这个 Key。

Q: Alice 怎么知道 Bob 的公钥是什么？Alice 拿到一个公钥以后，怎么判断这是 Bob 的合法公钥呢？

A: 需要找一个第三方，把 Bob 和 Bob 的 public key 绑定在一起以后，用机构的私钥做一个签名，这就是所谓的签名。Alice 收到这个以后，用机构的 public key 来验证一下机构的签名。这个机构就是 CA（certificate authority，证书授权中心）。这个机构在全世界非常少，它们的公钥被预装在每一个浏览器中。我们会无条件相信这些机构所签发的所有证书。这就是一切信任的根基。

如果 CA 一不小心签了一个不该签的证书，我们需要有一个把证书回收的方式，使用超时机制就会产生各种各样的问题，所以这不是一个很好的方法。所以我们应该定期地检查一些被召回的证书。

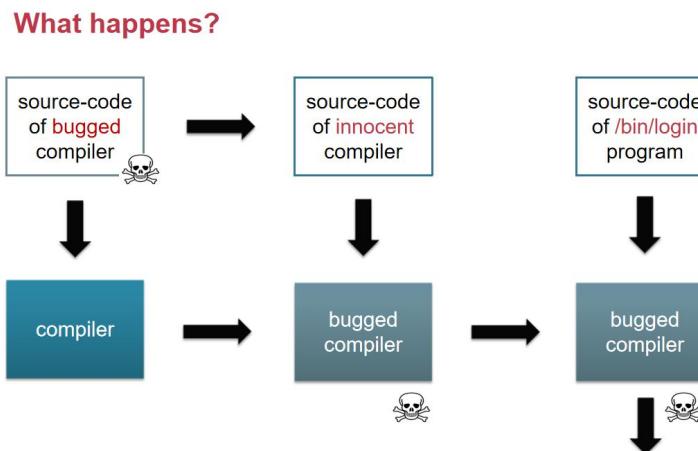
2021/12/23

今天我们继续安全的 topic，我们先来回顾一下上节课的内容，主要从系统的角度讲了数据流的安全，从应用程序的输入开始，到应用程序的内部是如何流动的。我们对数据流动的定义是对它加减乘除，我们希望追踪这一条数据的流动，这样我们就可以追踪这些数据，如果最终会影响到我们函数跳转的地址或者是一些指针，那么我们就认为这个数据是不安全的。

我们最后留下了一点，上节课我们讨论了 **trust** 的问题，在讨论安全性的问题的时候，有两个很重要的方法：确定安全的目标和确立它的 **threat model**（我们相信什么、不相信什么）。

我们需要提一下 Ken Thompson 的图灵奖报告，他创建了 Unix 和 C 语言。他可以登录任何一个用户的 Unix 账号，因为他在 Unix 中放了一个万能密码。修复了以后，他还是可以继续登录，因为编译器也是他写的。修复了以后，大家发现 compiler 还是需要 compiler 编译的。

因为有 bug 的 compiler 编译安全的 compiler，那么生成的 bin 也是有 bug 的。



在 2009 年，整个 Delphi compiler 就被入侵了，2015 年的时候 XCodeGhost 也被人插入了一段会在 APP 里植入广告的代码，把整个改过的 compiler 放到了国内的镜像上。

所以再次提醒，如果有 source code verification 是很难的，我们怎么才能获得 trust 呢？我们今天运行的计算机运行的代码已经超过了几亿行，我们需要设计合理的架构使得对软件和硬件的 trust 降到最低。

如果我们是一个 OS 的开发人员，对于 OS 来说，其他应用程序都是不能够相信的，OS 本身有几千行代码，我们如果要进一步缩小 trust 的范围，那么我们需要相信的是 OS 的核心代码，而不相信 OS 的其他代码，这就是微内核。把其他 OS 代码认为是应用程序。我们总得相信 CPU，但是 CPU 理论上是可以做坏事的。我们在造 CPU 的时候，有人做过 attack，在图纸上修改，使得 CPU 满足某些条件的时候，使得 bit 发生翻转。这意味着，我们让一台 CPU 运行一段特定的代码，然后它的电容就会升高，让权限位翻转，这样我们就得到了计算机的最高权限。

其实这就是如果我们在主板上加入一个组件，可能导致整个主板不可信。这就是我们现在所说的供应链安全，从制造、运输过程中，可能都被人植入一些组件。

所以我们要做到 least privilege，我们只能够给每个 program 最小的权限。

TCB: Trust Computing Base，最小需要相信的 base 软件，再次之外的软件是不需要 trust

的。微内核可能是有 BUG 的，我们可以把它放在虚拟机里，让它只需要相信虚拟机的下一层。如果虚拟机又有问题，我们可以在里面再装一个虚拟机，这就是嵌套虚拟化，我们只需要相信最外面一层的虚拟机的不断减少的代码即可。

盗版软件 KeyGen 也是很容易包含木马的，我们如果要用最好放在虚拟机里，然后把 Key 复制出来，再把虚拟机里。

对于计算机来说，有一个 root of trust，也就是 bootstrap 是信任的基础。我们启动计算机的时候，BIOS 先去启动一个 VMM，然后 VMM 会去验证一个 kernel loader 签名，如果签名对应，那就启动起 kernel loader。然后 kernel loader 加载 kernel，然后 kernel 再去验证其他 application。

TPM 是装在主板上的很小的元器件，里面包含了一个私钥，它固化在硬件里，是不能偷出来的，如果使用物理来强行取的话，它里面就会自动破坏关键数据。TPM 有对应的公钥放在 CA 里。如果我们想让电脑运行一个 OS，那就必须要找到这个 TPM 做一个合法的签名，这样 TPM 就会判断这个是否是一个合法的 OS，如果不合法就去拒绝。

Win11 用了 TPM2.0，导致理论上 win11 的电脑上是不能装其他 OS 的。所以我们只需要相信 TPM，这样后面启动的所有程序都放在了一条信任链里，哪里出了问题我们就可以找到。

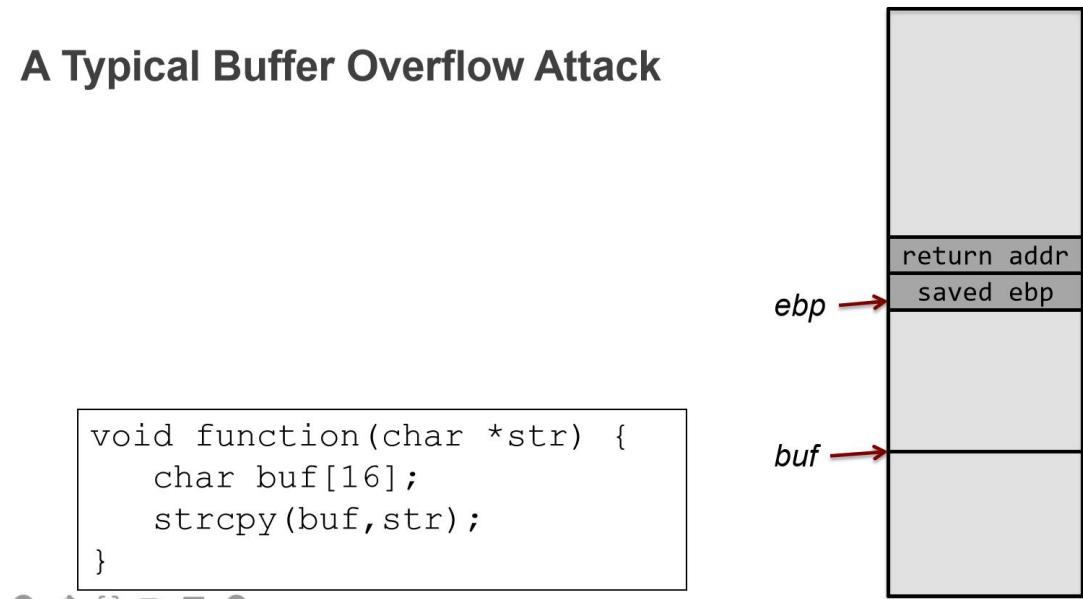
我们相信 TPM 是可信的，也就是 TPM 的私钥没有被人获取过，如果这个假设被打破了，那么整个信任体系就没了。如果我们想验证一台服务器上的 OS，那么我们只需要问服务器上的 TPM 这个 OS 有没有注册过就可以了。TPM 不会被人冒充，因为我们可以用公钥和私钥的形式来验证 TPM。TPM 也存在一些可能被定位的问题，所以现在一些 TPM service 并不对外提供服务。

我们再来看控制流方面的。

Stack Buffer Overflow Attack

Review: Stack Buffer Overflow

A Typical Buffer Overflow Attack

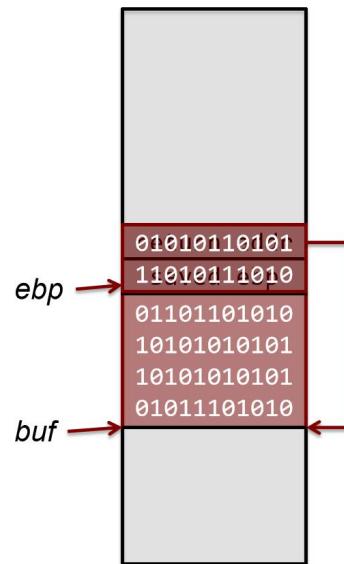


插入一段 shell code 就可以打开 shell。

A Typical Buffer Overflow Attack

- Inject malicious code in buffer
 - E.g., shellcode
- Overwrite return address to buffer
- Once return, the malicious code runs

```
void function(char *str) {  
    char buf[16];  
    strcpy(buf, str);  
}
```



▶ Shellcode Sample to Open "/bin/sh" (21 bytes)

```
int main(){  
  
    char sc[] =      "\x6a\x0b"           // push byte 0xb  
                    "\x58"             // pop eax (now eax=0xb)  
                    "\x99"             // cdq  
                    "\x52"             // push edx (now edx=0)  
                    "\x68\x2f\x2f\x73\x68" // push dword 0x68732f2f  
                    "\x68\x2f\x62\x69\x6e" // push dword 0x6e69622f  
                    "\x89\xe3"          // mov ebx, esp  
                    "\x31\xc9"          // xor ecx, ecx (now ecx=0)  
                    "\xcd\x80";         // int 0x80  
  
    ((void (*)()) sc)();  
}
```

More can be found on <https://www.exploit-db.com>

Name	Registers	Definition
sys_execve	eax 0xb ebx char __user * ecx char __user * __user edx char __user * __user esi struct pt_regs * edi	arch/alpha/kernel/entry.S:925

int 0x80 的意思就是主动触发一个系统调用，CPU 就会跳转到 0x80 编号所对应的 exception handler。（注意这里是一个软件触发的同步的 exception，而不是硬件触发的异步的 interrupt）。Interrupt handler 就会看你为什么会进入到 kernel 里来，也就是看寄存器里的参数。对应的就是 sys_execve，调用这个的目的是调用到 execv("/bin/sh")，push 的两个字符串其实就是 /bin/sh。

passwd 是用来改自己密码的命令，在/etc 下游一个 passwd 和 shadow，记录了所有用户名和密码的哈希值，请问这两个文件的权限应该设置为什么比较合理？肯定是要设置为只有 root 才能访问。如果一个普通用户只希望修改自己的命令，那么此时 passwd 就应该是一个普通用户可以访问的权限，但是它又要修改 root 权限的文件的哈希，所以 passwd 在整个执行过程中，它大部分时间都是 ring3，然后在中间一个很小的时间间隔后，为 ring0，修改完 root 文件后，又回到了 ring3。我们把 passwd 提权的时间段放大，假设以 ring0 执行代码的时候发生了一个 buffer overflow 的 bug，并且我们又执行了上述的代码，并且此时我们又有

`root` 权限，那么我们就得到了一个拥有 `root` 权限的 `shell`。攻击者的最终目标就是利用 `buffer overflow` 执行”`/bin/sh`”。

注意这种会提权的代码有好多个，攻击者就会去找哪些容易利用。注意我们还可以在 exploit-db.com 里找到更多的代码。

防御的办法就是 `data execution prevention`，也就是数据区是可写不可执行的。

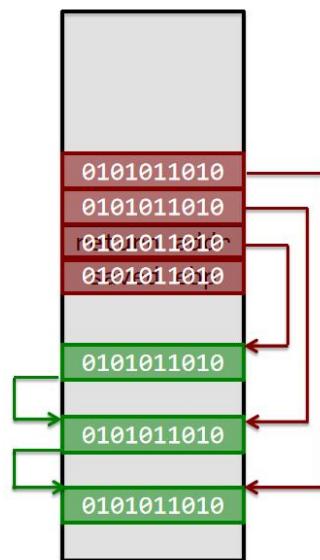
Code Reuse Attack

如果我们现在可写不可执行怎么办，那么我们就需要在代码区找到这个代码。但是好程序不会包含这个代码，我们就要去凑，我们就可以通过一堆的 `return address` 来串到一起。就是要找到语义上等价于上述攻击代码的子代码。

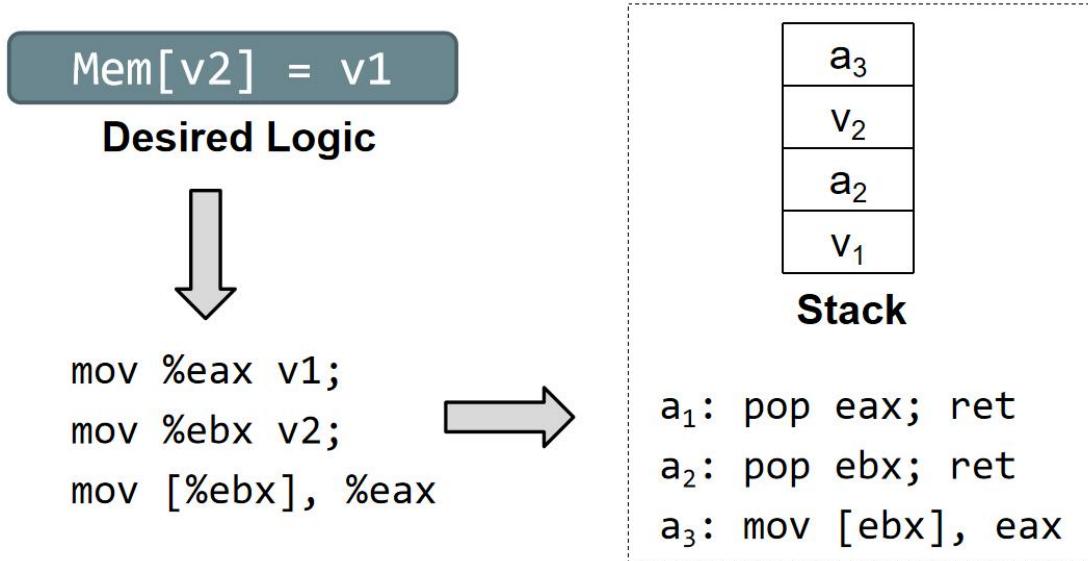
Attack: Code Reuse (instead of inject) Attack

ROP: Return-oriented Programming

- Find code gadgets in existed code base
 - Usually 1-3 instructions, ends with 'ret'
 - In libc and application, intended and unintended
- Push address of gadgets on the stack
- Leverage 'ret' at the end of gadget to connect each code gadgets
- **No code injection**



Code Execution – The ROP Way



对于 eax 和 ebx 的赋值要通过 pop 来完成，这是为什么呢？因为我们要围绕栈来编程，因为我们现在是一个攻击者，如果 v1 是一个特定的值，那么代码里大概率是不会有的。而攻击者现在有的是 stack overflow，所以攻击者就可以在栈中写入 v1, a1, a2, a3。这样就把整个代码通过往栈里 overflow 和 return programming 的方式，实现了往任意内存中的任意写。所以这就是我们说为什么 return-oriented programming 就是把很多字母拼在一起。

怎么防御 return-oriented attack 呢？

我们如果把 binary 藏起来，攻击者就不知道哪里是代码片段。但是怎么藏这个 binary 呢？每次编译的时候，加一些微小的扰动，让每次编译出来的都不一样，比如说-O1, -O2 编译出来的肯定不一样。

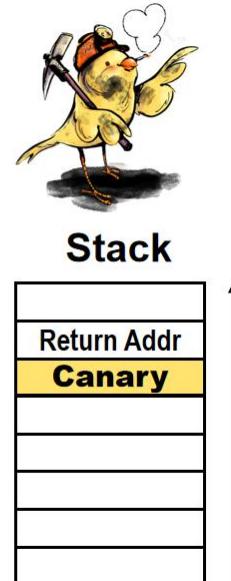
每次运行的时候，我们就使用 ASLR (address space layout randomization)，每次运行的时候加载到的位置都不一样。

还有就是金丝雀（挖矿的时候带一只金丝雀，金丝雀死了就代表有瓦斯）

Canary

Embed "canaries" in stack frames and verify their integrity prior to function return

- Canary is just a random number
- Check canary before return, alert if not equal



StackGuard implemented as a GCC patch

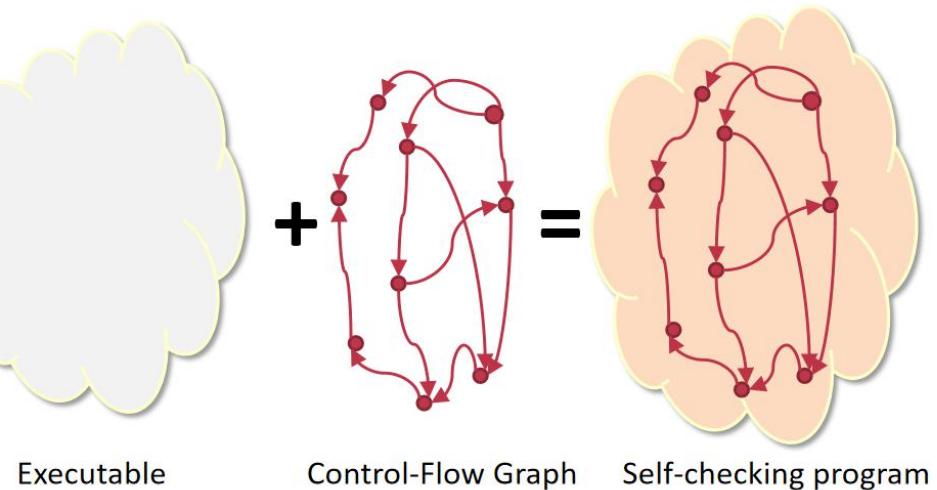
- Program must be recompiled
- Performance overhead: 8% for Apache

每次 return 之前，就检查一下 canary 有没有被修改掉。如果被修改了说明有 return address attack。每次在调用的时候，就要把 canary 放到 return address 的后面，返回之前就要检查，约增加 8% 的 overhead。

其实每一个程序写完以后编译成二进制，function call 都是有规律的，而 return-oriented programming 是在乱跳。这就是我们的 control flow graph，如果我们在编译的时候去构造 control flow graph，在运行的时候去 enforce control flow graph，这样就可以保证控制流的完整性。一旦我们发现控制流不一样了，我们就会发现有人在攻击，但是传统的控制流图并不被包含在 bin 中。

如果我们可以把 cfg 提取出来和 bin 组合在一起运行，这样它就保证了控制流不会乱走，不会出现 return-oriented programming 乱跳的情况。

CFI Idea



怎么才能做到控制流完整性的保护呢？

我们先来看一下程序什么时候会跳转，它分为直接跳转（调到一个固定的位置）和间接跳转（比如我们的 call 和 ret）。Ret 的跳转地址是通过栈来跳转的。

根据统计在 binary 里大部分都是 direct branch，而在 runtime 的时候大部分都是 indirected branch。

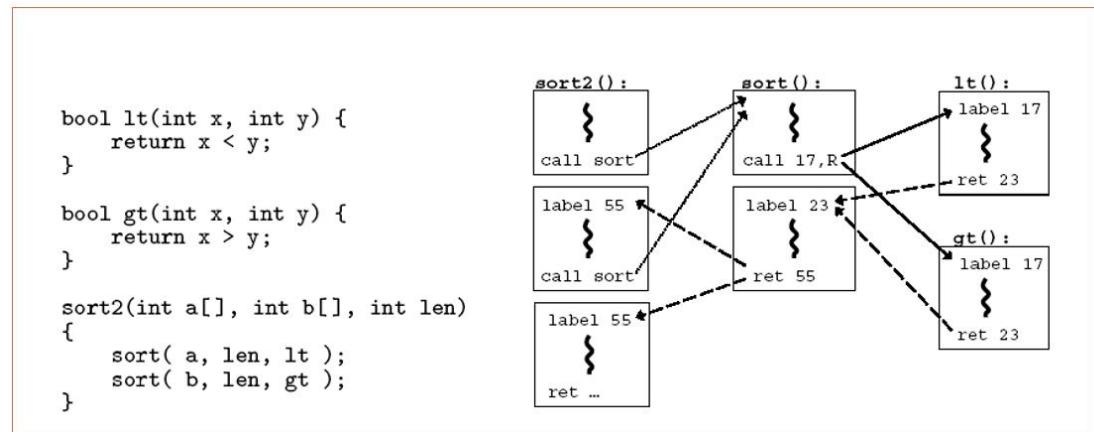
In Apache and its libraries

Types	In Binary	Run-time
Direct call	16.8%	14.5%
Direct jump	74.3%	0.8%
Return	6.3%	16.3%
Indirect call	2.1%	0.2%
Indirect jump	0.5%	68.3%

- has 1 target: 94.7%
- <= 2 targets: 99.3%
- >10 targets: 0.1%

其实原因也很简单，因为我们有循环。对于 indirect call 来说，大概率只会跳到 1~2 个地方。CFI 就说，我们在 call 的时候检查一下我们要 call 的地方是不是我们需要去的地方，它改写了二进制。

CFG Example



Sort2 会调用 sort，里面传了 lt 和 gt 的指针。我们在 sort2 的地方去 call sort。而 sort 最终就是去 call 17。所以就是把跳转的 source 和跳转的 label 用一个办法标记起来。所以我们可以把所有的跳转编号，当我们跳转的时候检查一下。如果我们想跳到另一个地方，就会被解决掉。

而 CFI 就可以通过这种方式做一个最简单的实现，但是它有一个缺点，在 sort 可以调用 lt 和 gt，所以 lt 和 gt 的内部的 label 都是 17。所以如果一个函数可以调用 100 个地方，那么这 100 个地方的 label 都是一样的。

所以 CFI 所定义的控制流是不精确的。那么具体怎么实现呢？

Original	<code>jmp ecx</code>	<code>mov eax, [esp+4] ; dst</code>
Patched	<code>cmp [ecx], 12345678h</code> <code>jne error_label</code> <code>lea ecx, [ecx+4]</code> <code>jmp ecx</code>	<code>; data 12345678h ; ID</code> <code>mov eax, [esp+4] ; dst</code>

也就是在原先的 `jmp ecx` 之前，去先检查一下要跳转过去的目标地址比较一下，如果不一样就跳转到 `error_label`，否则就正常跳转。但这么做有一个问题，如果有一段代码没有 `patch` 过，两个库（一个做过 CFI patch，而另一个没有打过 CFI patch）之间这么调用就会出错的。

我们就可以通过一个小技巧，把它变为 `prefetched` 代码。

CFI: Example of Instrumentation

<code>mov eax, 12345677h ; load ID-1</code>	<code>3E OF 18 05</code>	<code>prefetchnta ; label</code>
<code>inc eax ; add 1 for ID</code>	<code>78 56 34 12</code>	<code>[12345678h] ; ID</code>
<code>cmp [ecx+4], eax ; compare w/dst</code>	<code>8B 44 24 04</code>	<code>mov eax, [esp+4] ; dst</code>
<code>jne error_label ; if != fail</code>	<code>...</code>	
<code>jmp ecx ; jump to label</code>		

Prefetchnta: prefetch memory to cache. Become a `nop` if not available.

`Prefetch` 是一条出错以后也没问题的指令。如果这个地址是一个错误的地址，那么它就是一个 `nop`。所以它能够提供一个兼容性，使得没有打过 `patch` 的代码可以去调用这个打过 `patch` 的库。

假如函数 A 要调用 C，而 B 既可以调用 C 也可以调用 D。那么 CFI 就必须对 C 和 D 使用同样的 `tag`，比如说是 12。那么就等于 A 也可以调用 D 了。一种可能的方法就是 `multiple tag`，对于 B 来说调用 C 和调用 D，必须是 2 个 `call`。

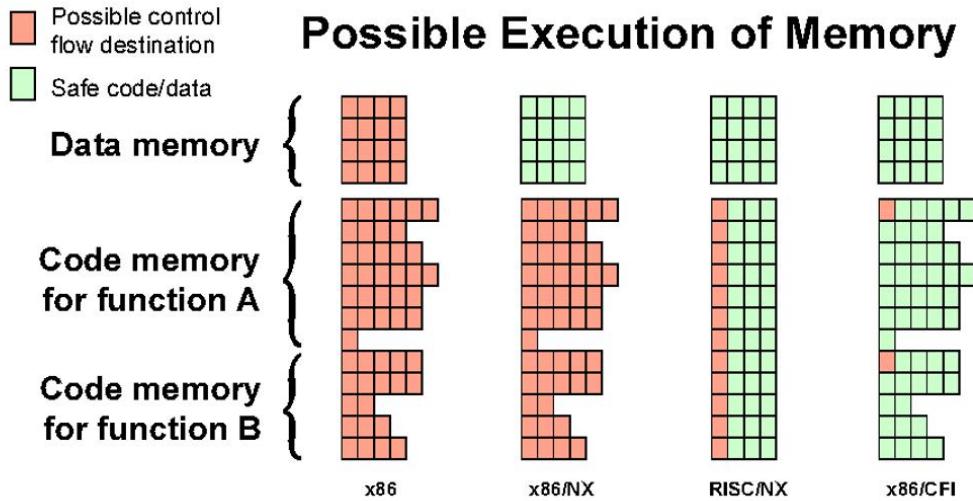
还有一个问题就是，假设有一个 F。先被 A 调用，再被 B 调用，被 B 调用的时候返回到 A 也被 CFI 认为是合法的，要解决这个问题就需要一个 `shadow stack`，它和正常的 `stack` 很像，但是它只存 `return address`。

之所以会有 `return-oriented programming` 是因为我们把数据和控制混在了一个 `frame` 中，导致 `data overflow` 掉了控制流的 `return address`，所以我们就为程序专门构建了一个 `shadow stack`，当我们调用函数的时候，我们同时压栈 `shadow stack` 和原先的 `stack`。

所以有了这之后，A 调用 F 一定会返回 A。所以有了 shadow call stack 就很大程度上解决了这个问题。

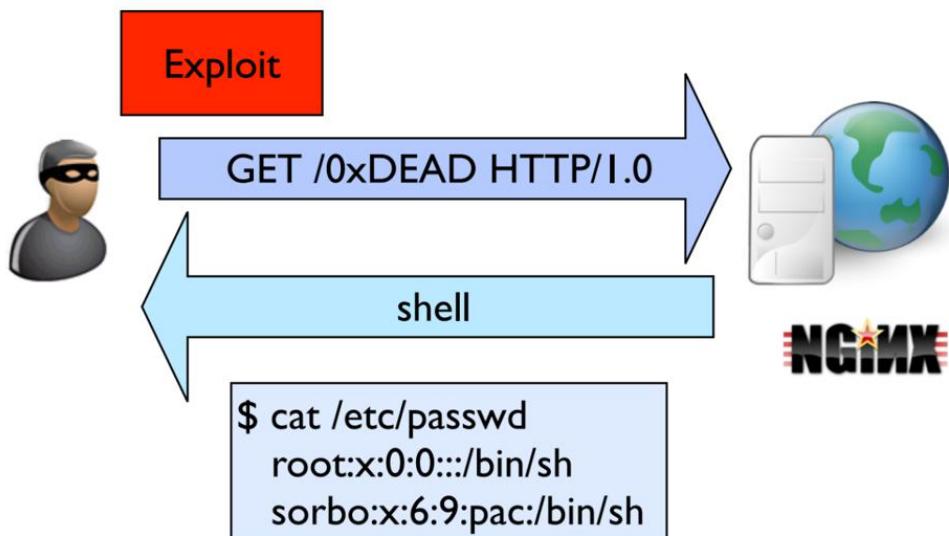
Possible Execution of Memory

[Erlingsson]



如果我们使用定长指令，必须要跳转到 32B padding 的位置。如果我们是 RISC 和 ARM 这种定长指令，只能跳转到头上。而 x86 加上 CFI 之后，只允许跳转到 label 对应的位置。所以 CFI 就是保证了我们的控制流不会来回地跳。

Hacking buffer overflows

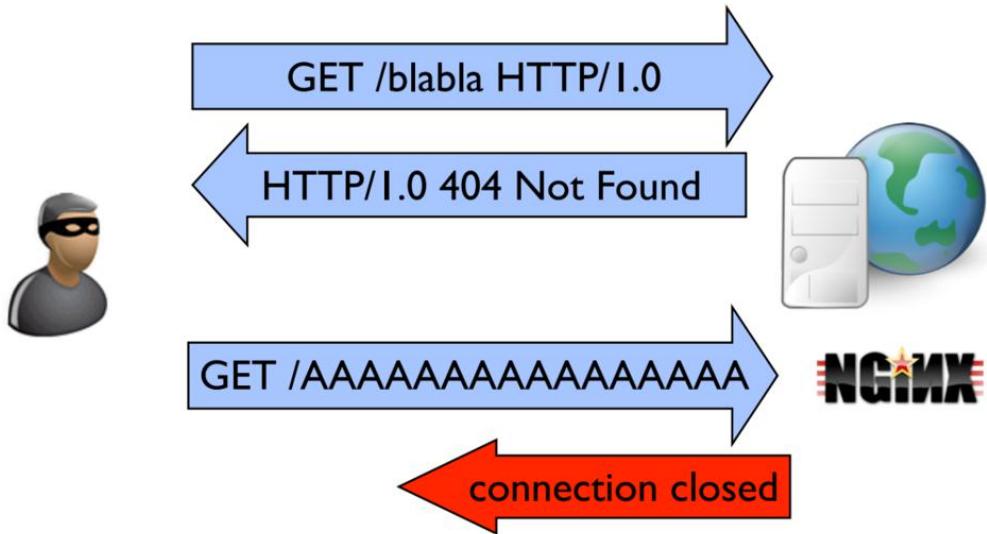


我们来看一个真实的攻击。有一个攻击者和服务器跑着 webserver。攻击者可以通过 get http 得到一个 bitmap，它的目标是通过不断地发送 http，让 nginx 变成一个 shell。这就等于

我们控制了一台服务器。

还有一个假设，nginx 有一个 buffer overflow 的漏洞，让攻击者去干掉 nginx 的栈。但是攻击者没有 nginx 的二进制，所以不知道要写什么地址，并且 nginx 部署了金丝雀和 ASLR。

Crash or no Crash? Enough to build exploit



首先，攻击者 get 一个很长的参数，导致 connection close。攻击者唯一只需要知道存在一个 stack bug 并且知道怎么 trigger。

Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;
```

```
    read(s, &len, sizeof(len));  
    read(s, buf, len);
```

```
    return;  
}
```

Shellcode:

```
dup2(sock, 0);  
dup2(sock, 1);  
execve("/bin/sh", 0, 0);
```

Stack:

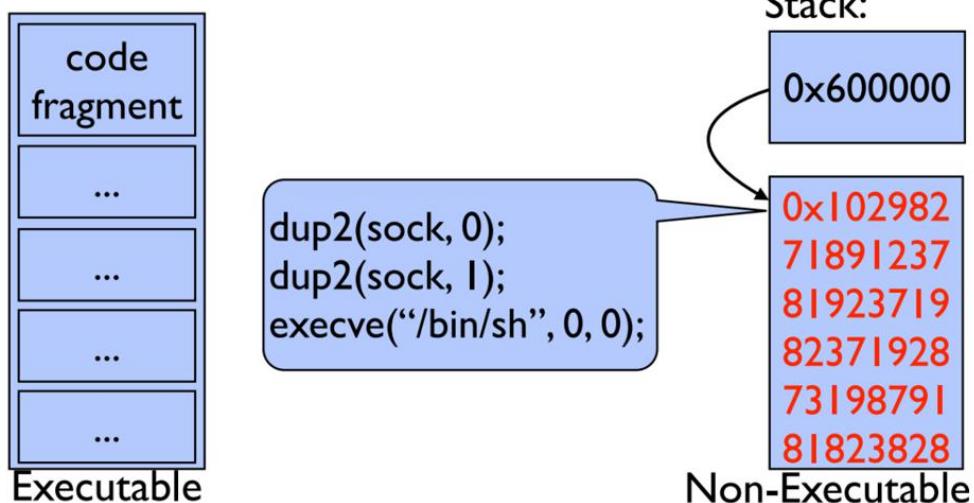
return address 0x600000
0x1029827189
123781923719
823719287319
879181823828

Dup2 其实就把输入重定向到 sock，这样攻击者就可以变成 nginx 的 0 号输入，并且攻

击者就可以看到其输出了。

Return-Oriented Programming (ROP)

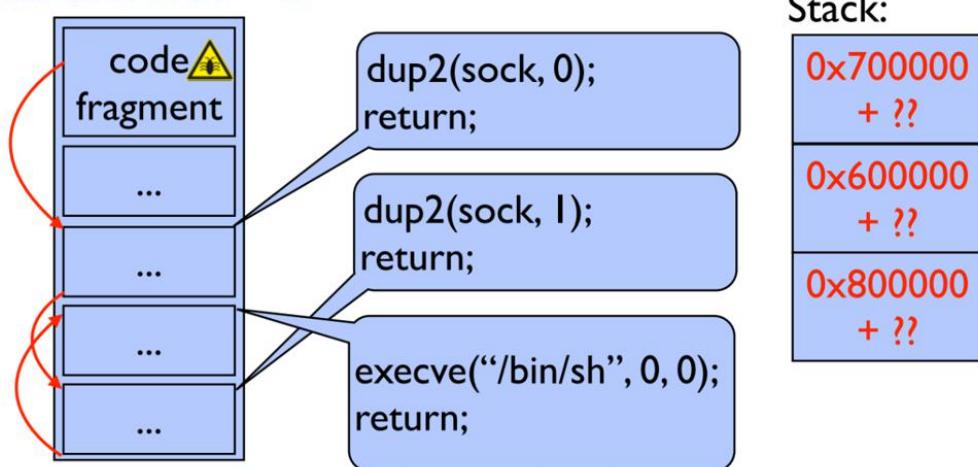
.text:



我们需要从可执行代码中去做 ROP，但是它有 ASLR。ASLR 就会导致它的代码存在一个随机数。

Address Space Layout Randomization (ASLR)

.text: `0x400000 + ??`



注意到 webserver 的 fork 是每个 client 过来了都会 fork 一个 progress 出来，所以每次 fork 出来的地址都是一样的，ASLR 只是保证机器重启的那一刻地址是随机化的。

然后攻击者还需要把 binary 偷到手, 否则不能做 ROP 的 attack。它需要通过 write 的 syscall, 把加载到 memory 中的 binary 传到本机。

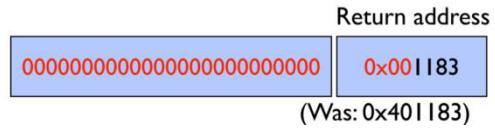
Defeating ASLR: stack reading

- Overwrite a single byte with value X:
 - No crash: stack had value X.
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.



Defeating ASLR: stack reading

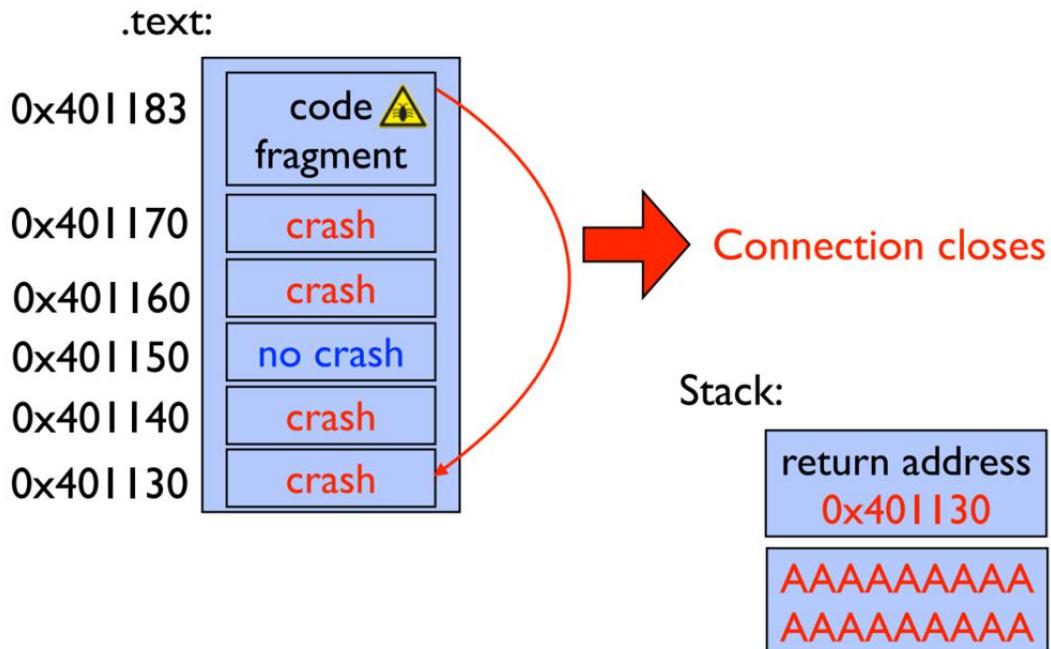
- Overwrite a single byte with value X:
 - No crash: stack had value X.
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.



我们只需要一个个 byte 猜就可以了, 每个 byte 的猜测期望是 128 次。所以只需要猜几百次就可以把 return address 猜出来了, 这样我们就知道代码大概在什么位置了。

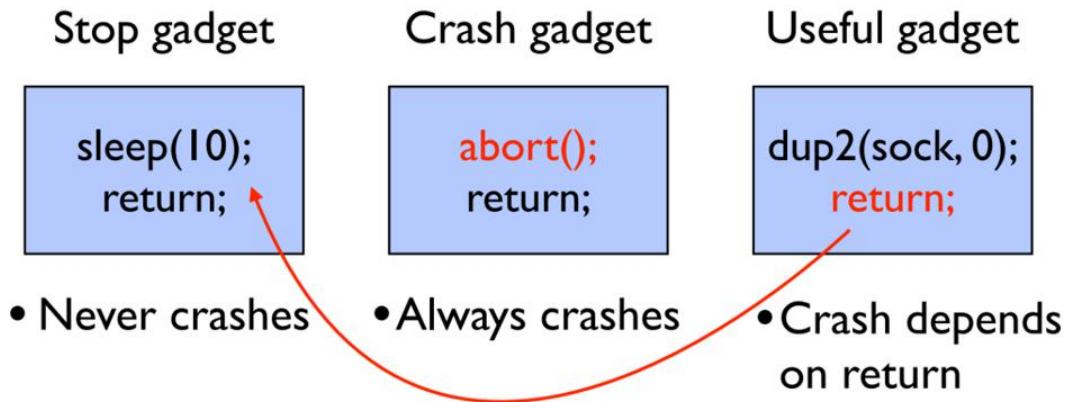
接下来, 假定我们已经知道了这个地址是 0x401183 周围, 接下来我们要去找 gadget。

How to find gadgets?



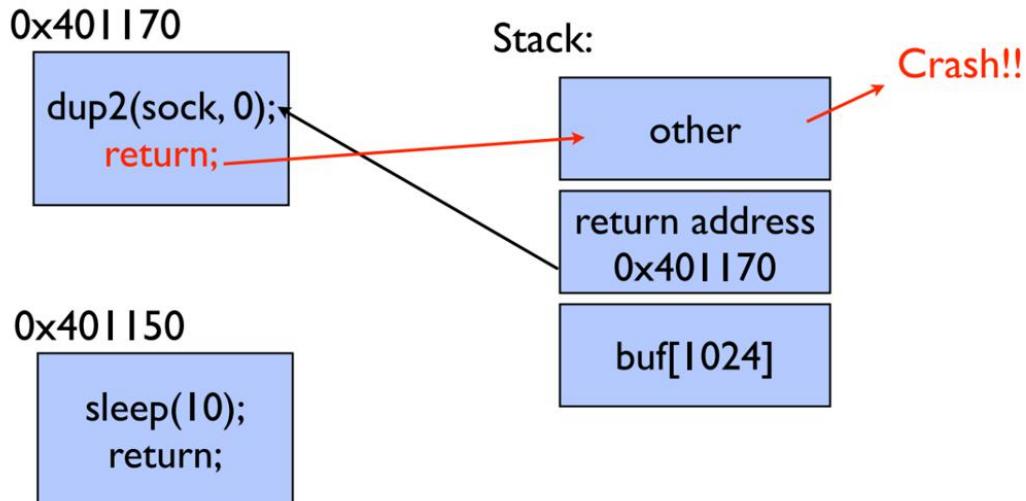
它多跳转几次, 如果跳过去之后 connection close 了, 说明把服务器搞崩了, 如果跳过去不会 crash, 我们就记录下来。

Three types of gadgets



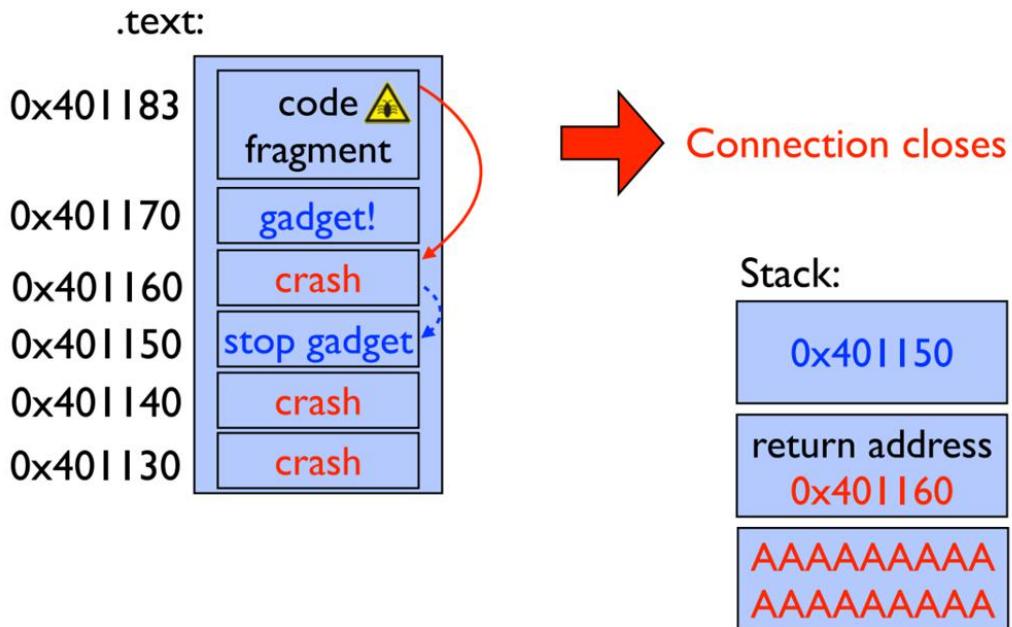
接下来我们就找三种类型的 gadget，跳过去以后发现服务器没响应；发过去以后服务器 crash；跳过去之后，服务器可以正常运行。

Finding useful gadgets



如果我们把 crash 的值改为 not crash，那么我们就知道这里有一个 return。我们就可以通过修改我们的 stack 知道它调用了这个 return。所以当我们改 stack 能够影响服务器的反应，我们就知道它最后一定有一个 return。

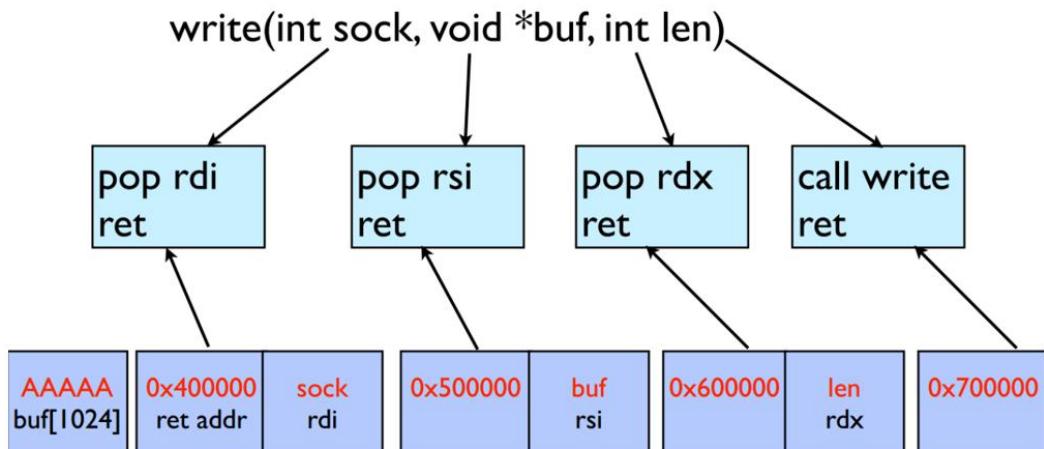
How to find gadgets?



我们可以通过整个 connection 是不是 hang，这样我们就知道是不是一个 return gadget。那么接下来我们要找 write buffer 到 socket 的系统调用，这是为了把二进制偷到手。

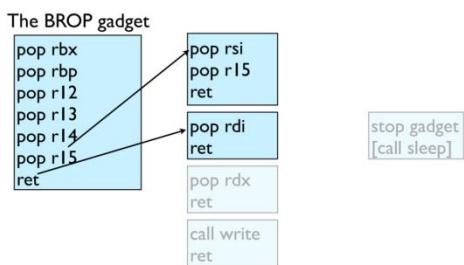
它需要去设置%rdi,%rsi,%rdx这三个参数，最后去调用 int 0x80.

What are we looking for?

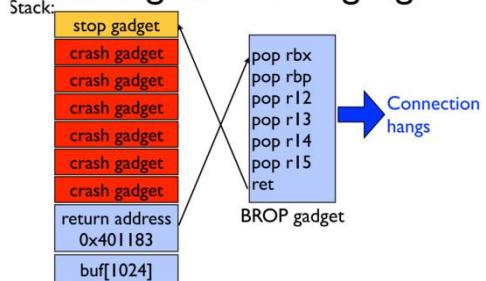


我们怎么样才能找到 pop 这个 return address 呢？根本不难找，在代码中有大量的 callee-saved register，从这段代码中，我们就可以找到我们需要的代码中的两个。

Pieces of the puzzle



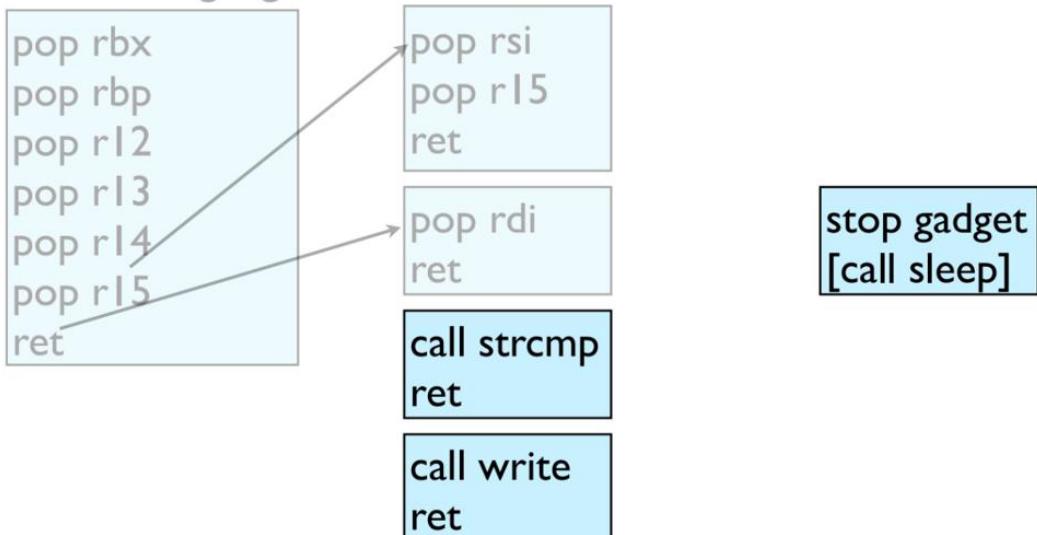
Finding the BROP gadget



如果跳转到了这段代码，我们就会把栈上的 6 个地址都扔掉。我们跳过去，如果最后这个系统没有 crash 并且 hang 住了，那么它一定 pop 了 6 次。所以攻击者只需要一次次试去看有没有 hang 住。怎么去找 pop rdx 呢？

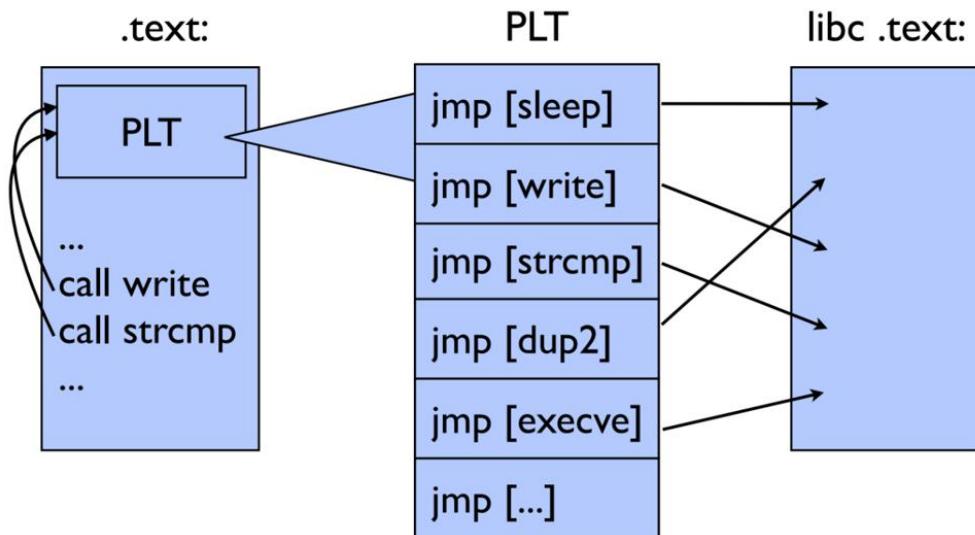
Pieces of the puzzle

The BROP gadget



对于 string compare 来说，rdx 的值会在运行过程中存 string 的长度。所以我们只需要去 call string compare，就会把%rdx。

Procedure Linking Table (PLT)



那么怎么找到 `string_cmp` 的地址呢？此时就需要 PLT 表了。它有一个特点是很有规律，都跳一个函数+return。

攻击者就发现了，如果我们构造一个栈使得我们的栈+4 个 byte 又可以运行，不 crash，说明我们找到了 PLT 表所在的位置。那么我们只需要试几次，就知道哪个是 `write` 和 `strcmp` 了。`strcmp` 有两个参数，第一个参数可读，第二个参数为 null，就会 crash，只有两个参数都可读才不会 crash，所以攻击者就利用这个特征找到了 `strcmp`。攻击者怎么找 `write` 呢？其实他只要跳转到一个 PLT 函数之后，使得攻击者这一端收到数据了，那么我们就知道了 `write` 函数。

Braille

- Fully automated: from first crash to shell.
- 2,000 lines of Ruby.
- Needs function that will trigger overflow:
 - nginx: 68 lines.
 - MySQL: 121 lines.
 - toy proprietary service: 35 lines.

try_exp(data) → true crash
false no crash

Attack complexity

