课程简介 01

通过网络使一群计算机配合完成工作

目的：

parallelism	获取更大的硬盘，更高的并行性
fault tolerance	更好的容错性
physical	不同区域的计算机通信
security/isolation	隔离

infrastructure

1 storage — 如何容错的问题

2 communication — 是分布式系统的工具

3 computation — mr

我们最关心的是存储部分

构建一个外观类似非分布式系统的系统，可以将其分布式的特性屏蔽

分布式的难点：challenge

partial failure	部分计算机出现故障，如何处理
concurrent programming	处理复杂的并发问题
performance	精心设计的系统才能获得足够好的性能

lab

1 mapreduce

2 raft — 容错

3 k/v server

4 shared k/v server

lmpl：构建的工具

1 rpc — to mask the fact that we are communicating through a unreliable network

2 threads — 通过并发操控大量计算机

3 concurrency ctl — lock

performance 服务器的性能的并发负载

scalability — 2x computer -> 2x throughput

但是，单纯的计算资源的堆砌在一定范围内并发量的提升是有用的，但当服务器个数达到一定数目，计算资源已经不再成为瓶颈，要解决数据存储的压力。

fault tolerance

大型的分布式系统中，某个计算机出现故障是十分常见，频繁的。

1 availability — 当部分节点出现错误，整个系统仍然可用

2 recoverability — 当系统从故障中恢复时，可以恢复原来的数据

一个好的系统应该在感知到足够多的错误和故障后，停止响应，当故障得到修复时恢复服务，并恢复到故障之前的状态。 — nv storage 非易失存储（nonvolatile storage）

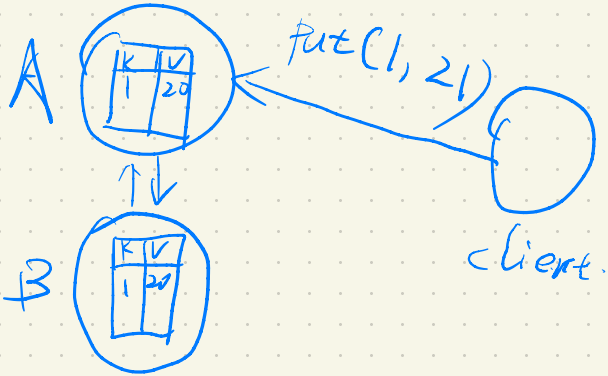
3 replication 使用副本实现容错

topic — consistency 一致性

不同机器上保存的副本不同导致两次查询得到的值不同

client向a更新了一个值，并没有同步到b

若a故障了，启用b时，b中仍然是旧数据



strongly consistency system 每次get都会获取到最新的数据 — 难以实现，开销大（读所有的副本取最新的

weakly consistency system 某些情况下可能会得到老数据

independent replica，两个副本连着同一个电源，那么如果这个电源被切断了，那两个副本就都会丢失

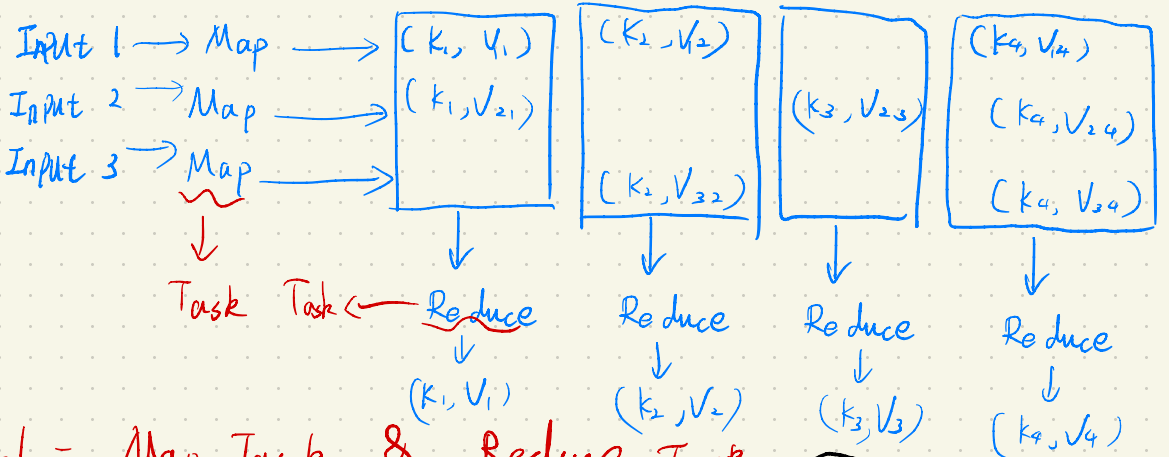
MapReduce：

Google需要一个能便捷的开发分布式应用的框架，而不必在开发时考虑分布式系统的细节。

1 Map函数，从输入文件获取（k，v）

2 reduce函数， 处理整合获取的（k，v）

以词频统计为例



emit:

将结果写入输出文件

存储在GFS: google file system

一个分布式文件系统

Job = Map Task & Reduce Task.

```
Map(k, v):
    split v into words
    for each words w
        emit(w, 1)
```

k: file name

v: content of file.

```
Reduce(k, v):
    emit(sum(v))
```

k: the word

v: a vector of its value produced by each Map

箭头：
箭头处表明的是文件传输，在mapreduce论文在2004年最初发布时，网络问题时一个最大的瓶颈
在设计上尽量避免网络传输，将gfs server部署在mapreduce的worker上
master在分配任务时，优先将任务分配给输入文件所在的worker上

RPC 与多线程

Thread的好处

IO concurrency	进行网络，磁盘读写等操作时可以并行
Paralelism	充分利用CPU的各个核心，做到任务的真正并行
convenience	不必在main中插入代码检查worker的状态（类似thread用detach?

event-driven programming

只有一个线程不断的等待，当某些事件触发了（timer计时，网络请求）则去处理对应事件

1. 并发性不如Thread/goroutine，同时只能处理一个事件
2. CPU多核利用效率不高
3. 编程难度高于Thread
4. 线程少开销更低，每个thread都有一个栈，以及需要系统维护调度thread的数据结构，故thread开销更高

Thread challenge

1. race — 对同一块内存的读写冲突（使用mutex，cas等方法解决

not all the instructions are atomic，不是所有指令都是原子性的，比如inc，inc实际上是load-add-store三个步骤，取决于cpu的具体实现和cpu的字长

coordination（go语言

sync.mutex	— 互斥锁
sync.waitgroup	— 常用于等待一系列线程调用done完成等待
channel	— 用处较广，线程间交换数据，协作等待都可以完成，内部自带mutex
sync.cond	— 条件变量，和一个bool一起使用

```
for u := range aList {
    go func(u TypeofU) {
        //processing variable u
    }(u)
}
```

这里go后面的函数可不可以不传u，直接从外部捕捉u?

不行，因为go启动携程后，for循环遍历到后面的变量了，u也发生了变化。

所以在携程中捕获的变量最好是不变的

go run -race xxx.go

可以帮助检查代码中是否发生了race

线程的创建要在一定范围内，10，100，1000

否则应该提前创建一个线程池，复用线程

线程的创建和销毁存在一定开销

CPU资源有限，频繁的线程切换导致总体效率降低

GFS

plan：

1.storage 总体讲存储的相关知识

2.gfs architecture consistency

storage system

building block for false tolerance system

存储系统是容错系统的重要构件

使apps保存soft state，state less

存储系统保存持久的状态信息

difficulties

需要高性能 -> 共享数据的多个服务器（数据分散在多个服务器

多服务器 -> 服务器崩溃

容错系统 -> 副本

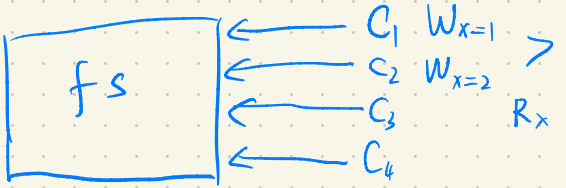
副本 -> 数据不一致

强一致性 -> 性能差

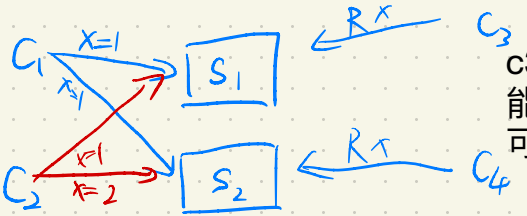
ideal consistency

behave as if single system，困难在于：

1.并发多个client读写同一个值 2. 如何保存副本



同时写，最终结果1/2都可接受



gfs

性能好：副本+容错+一致性

成功：

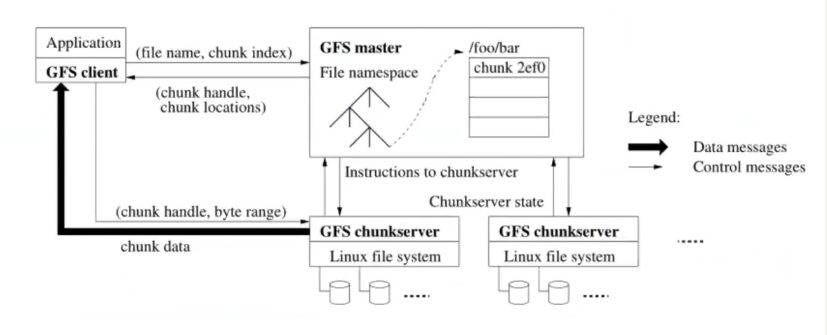
gfs以前，很难构建超过1000个服务器的系统

gfs并不是标准的：

1. 有单个master负责其他节点的一致性
2. 有不一致性

gfs
big：非常庞大的数据集
fast：automatic shareing，自动对数据分割，分布存储
global：所有app看到相同的fs
fault tolerance：自动容错

read a file
1. c 发送 文件名+offset到m
2. m返回chunk句柄，服务器列表，版本号
3. c缓存服务器列表
4. 从最近的服务器读取消息
5. s检查版本号，发送消息



master
文件名 -> chunk handles的数组
chunk handle -> 版本号，持有chunk的server（其中一个primary，其他secondary），释放时间
log + checkpoint
↓
stable storage

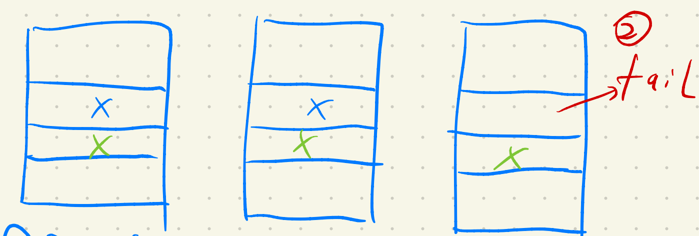
write a file:append → reducers

stable, S和M都存

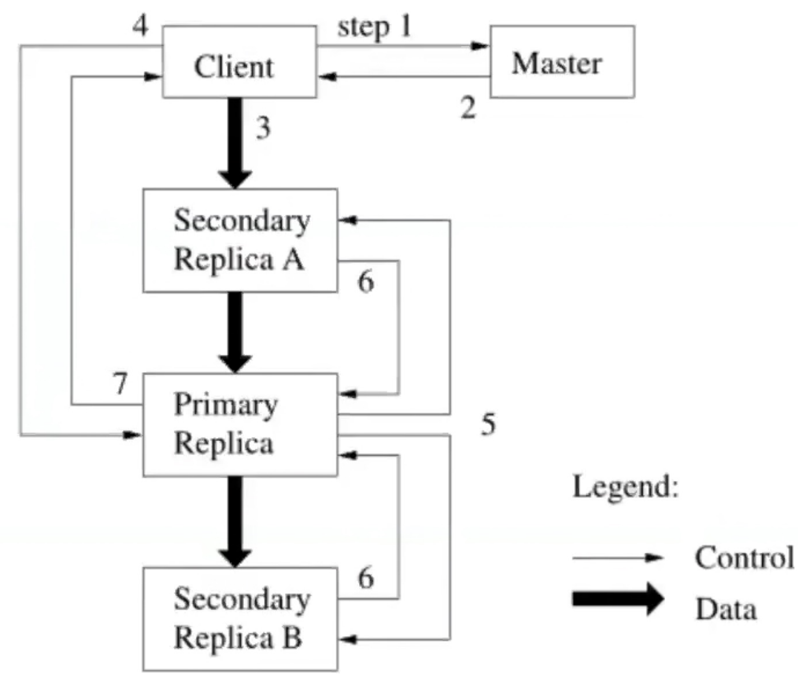
1. c与master交互，获取chunk handle，servers
2. master选出primary，secondary（给primary一个lease，增加版本号）→ 只在选p时增加.
3. 选择最近的s，发送数据，s之间相互传递数据
4. 向p发送append命令，p检查版本号，租约。检查通过，选择一个偏移量写入数据
5. p通知secondary写入，提供offset
6. secondary通知p写入成功
7. p通知c成功与否

若有secondary失败，则返回失败。
若失败，client retry，但primary不会使用相同的offset。

① client write x



③ client retry



retry后，写入的数据重复了（写入的每条数据要有id

primary/backup replication

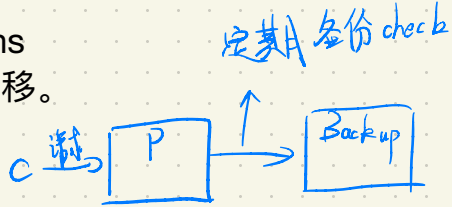
plan:
1. failures
2. challenge
3. 2 approach，状态转移 副本状态机
4. case: vm容错机制

failures:
1. fail-stop failure:能导致机器停机的各种问题，硬件故障等
2. logic bugs: 配置错误，代码错误，恶意攻击
3. 不可抗力: 地震，火灾

challenge
1. primary是否已经故障？有时候因为网络分区（network partition）问题，导致primary无法与内部的一些节点交流，但是仍然可以对外服务。
   - split-brain system，脑裂系统。
2.primary和备份保持同步：当primary崩溃了，可以直接启用backup，从primary崩溃的位置继续工作。让外界看起来就像一台机器一样，但这要求他们始终同步的。
   - apply changes in order，应用各种事务的顺序要一样
   - avoid non-determinism，相同的事务作用在主/备份设备上的作用 效果要相同
3. fail over（故障转移），primary崩溃，启用其他主机
   - 崩溃时primary可能正在发数据包，转移后是否要发这个包?
   - 有多个备份，很多机器都崩溃了，恢复后谁是最新的?

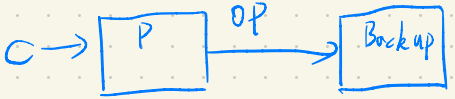
2 approachs

1） 状态转移。



把当前的整个状态复制到备份中，expansive

2）replicate state machine（rsm）复制状态机，将操作复制给备份，节省带宽

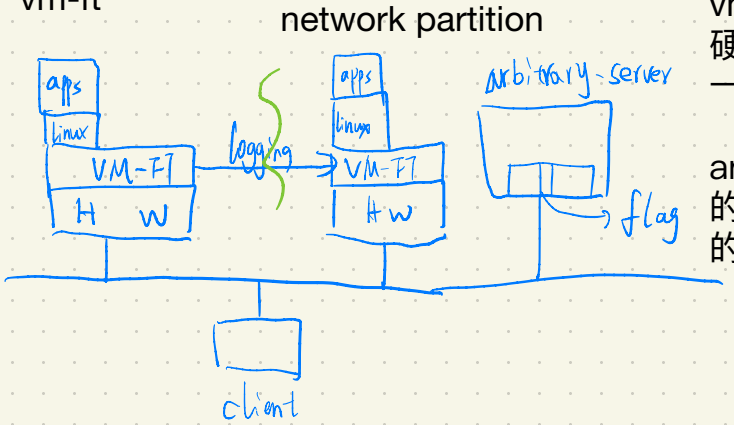


1. 每个操作都是确定性的
2. 两个机器拥有相同的初始状态

复制什么级别的操作?

1. application-level 复制应用程序的操作，append，write等
2. mechine level 复制机器指令，并使用虚拟化技术兼容不同的硬件平台。还可以使应用透明

case：vm-ft



vm-ft建立在硬件上，操作系统使用的是vm模拟出的硬件设备。vm也就有能力捕获每一个网络请求。如果一台机器崩溃，切换后可以继续处理网络请求。

arbitrary server为主机提供存储，会将它挂载到Linux的一个目录下。其中有flag，主机通过test-and-set的方式知道当前谁是primary。

divergence source
primary和backup需要执行相同的指令，差异的来源有
1. non-deterministic 非确定性的指令执行，如中断，获取时间，确定性指令不需要通过logging传输
2. 传输指令的数据包和中断的顺序要相同
3. multicore 两个并行任务的执行结果也要相同

non-deterministic
对于非确定性指令，替换成trap陷阱指令，机器执行到后，交给vmft，vmft模拟该指令的计算结果，并保存下来，发送给备份。

interrupts
当primary运行时出现了中断，就把这个中断，中断断点位置，中断相关的数据发送给backup，backup收到后将其缓存下来，当他执行到相同位置时，读取这些消息并应用更改

output rule
只有primary收到backup对于log数据的确认后，才会对client进行回应。
如果primary的log在传输时丢失，而恰好此时primary宕机，backup接管后将丢失一部分数据

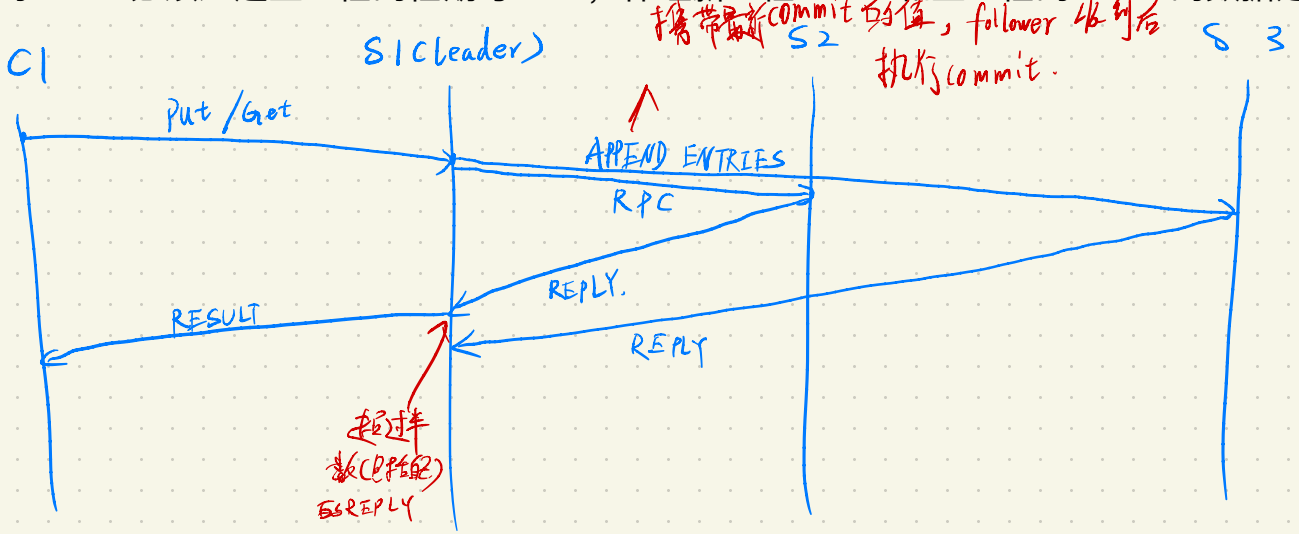
Q&A
1. 备份只有一份，更多备份的情况需要更复杂的算法

raft

single point of failure：
为了防止脑裂split-brain，不能有多个test-and-set server，不能有多个master
因为存在网络分区问题，一旦发生，将会导致数据不一致问题

网络分区，大多数原则。（大多数是所有机器中的大多数
2f+1原则，如果要容忍f个机器出现错误，需要至少2f+1台机器

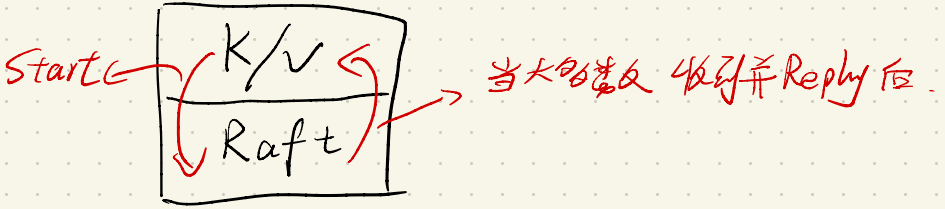
overlap
新一任的leader必须知道上一任的任期号term，保证新一任leader和上一任的leader的数据是连续的，



leader与follower的速度差异
如果follower速度太慢，leader持续的给follower发送日志，follower来不及处理只能保存在内存中，导致内存爆掉。1. 限制follower的速度 或 2.使用checkpoint强制follower应用

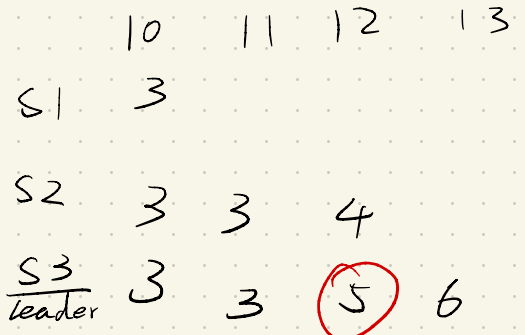
日志的作用
order 日志具有顺序性
retransmission 支持重传，如果有其他节点漏掉了一些命令
persistency 持久化，服务器重启后可以根据log恢复状态
space tentative 需要试探性的空间，发送的指令不确定是否能提交

start(command) index
执行指令，返回执行结果的索引
applyCh，指令放入一个go channel中



election timer
每个节点有一个timer，超过这个时间没有收到leader的心跳，就认为leader挂掉了，要重新选举
timer的时间应该是正常心跳的几倍
为了防止所有节点开始选举，都给自己投票，造成选票僵持的情况，timer应该是一定范围内随机的
timer超时，term++ request vote，请求选票

选举
如果网络不好，发生了partition等问题，导致无法集合半数以上的节点，那么就一直不会选出leader，会一直重复的进行选举

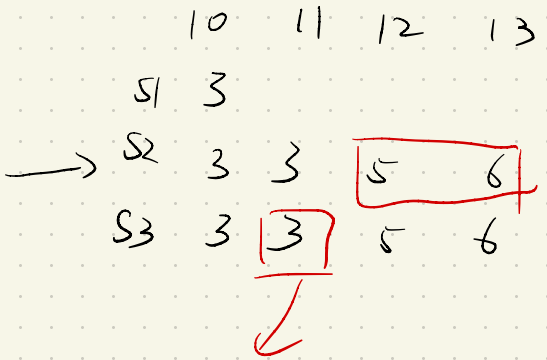


Append Entries

prev Index = 12

prev term = 5

$S_1, S_2 \longrightarrow$ false



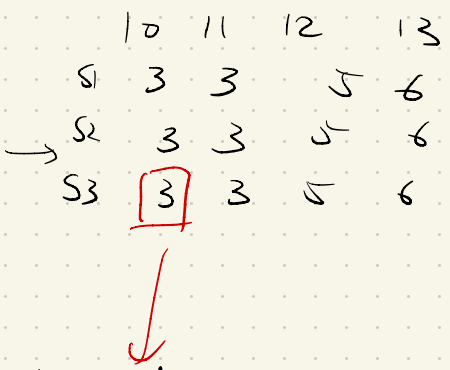
Append Entries

prev Index = 11

prev term = 3

S1 → false

S2 → true



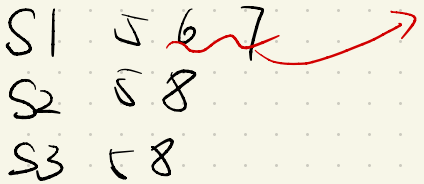
Append Entries

prev Index = 10

prev term = 3

S1 → true

为什么不让log最长的节点当leader?



s1赢得选举，但是发送appendEntries前宕机了，然后快速的恢复后再次当选leader，又在发送rpc时宕机。此时其他节点选举成为term8的leader。