课程 Q&A

LEC 1 Introduction

中文翻译

6.824 2022 第一讲：简介

6.824：分布式系统工程

我所说的"分布式系统"是什么意思：
一组合作提供服务的计算机
这门课主要是关于基础设施服务
例如大型网站的存储、MapReduce、点对点共享
许多重要的基础设施都是分布式的

人们为什么要构建分布式系统？
通过并行处理增加容量
通过复制容忍错误
匹配物理设备（例如传感器）的分布
通过隔离实现安全

但这并不容易：
许多并发部分，复杂的交互
必须应对部分失败
实现性能潜力很棘手

为什么要研究这个主题？
有趣——难题，强大的解决方案
广泛使用——由大型网站的兴起推动
活跃的研究领域——未解决的重要问题
构建具有挑战性——您将在实验室中完成

课程结构

http://pdos.csail.mit.edu/6.824

课程人员：
罗伯特·莫里斯，讲师
塞尔·斯凯格斯，TA
阿塞尔·伊斯莫尔达耶娃，TA
安尼什·阿塔耶，TA

课程组成部分：
讲座
文件
两次考试
实验室
最终项目（可选）

讲座：
大创意、论文讨论、实验室指导
将被录像，可在线观看

文件：
几乎每堂课都会分配一篇论文
研究论文，有些经典，有些新
问题、想法、实施细节、评价
请课前阅读论文！
每篇论文都有一个简短的问题供您回答
我们请您向我们发送有关该论文的问题
在讲座开始前提交答案和问题

考试：
班级期中考试
期末考试在期末周期间进行
主要是关于论文和实验室

实验室：
目标：更深入地理解一些重要的想法
目标：具有分布式编程经验
第一个实验室将于周五起一周内到期
此后一段时间每周一次

Lab 1：分布式大数据框架（如MapReduce）
实验2：使用复制的容错库（Raft）
实验3：一个简单的容错数据库
实验 4：通过分片扩展数据库性能

最后可选的最终项目，以 2 或 3 人为一组。
最终项目替代实验室 4。
您想到一个项目并与我们一起解决。
最后一天的代码、简短的文章、演示。

警告：调试实验室可能非常耗时
尽早开始
在广场上提问
使用助教办公时间

我们使用一组测试对实验室进行评分
我们为您提供所有测试；没有一个是秘密

主要议题

这是一门关于应用程序基础设施的课程。

- 贮存。
- 沟通。
- 计算。

一个大目标：对应用程序隐藏分发的复杂性。

主题：容错
数千台服务器、大型网络 -> 总是有东西坏掉
我们希望对应用程序隐藏这些失败。
“高可用性”：尽管出现故障，服务仍然继续

大创意：复制服务器。
如果一台服务器崩溃，可以继续使用其他服务器。
实验室 2 和 3

主题：一致性
通用基础设施需要明确定义的行为。
例如，“Get(k) 从最近的 Put(k,v) 中生成值。”
获得良好的行为是很难的！
“副本”服务器很难保持相同。

主题：性能
目标：可扩展的吞吐量
Nx 服务器 -> Nx 通过并行 CPU、磁盘、网络的总吞吐量。
随着 N 的增长，扩展变得更加困难：
负载不平衡。
N 中最慢的延迟。
有些事情不会用 N 加速：初始化、交互。
实验室 1、4

主题：权衡
容错性、一致性和性能是敌人。
容错和一致性需要沟通
例如，将数据发送到备份
例如，检查我的数据是否是最新的
通信通常很慢且不可扩展
许多设计仅提供弱一致性以提高速度。
例如，Get() *不会*产生最新的 Put()！
对于应用程序程序员来说这是痛苦的，但可能是一个很好的权衡。
我们将在一致性/性能范围内看到许多设计点。

主题：实施
RPC、线程、并发控制、配置。
实验室...

这种材料在现实世界中经常出现。
所有大型网站和云提供商都是分布式系统方面的专家。
许多大型开源项目都是围绕这些想法构建的。
我们将阅读来自行业的多篇论文。
工业界采纳了学术界的许多想法。

案例研究：MapReduce

我们来谈谈MapReduce（MR）
很好地说明了 6.824 的主要主题
影响力巨大
实验室1的重点

MapReduce概述
上下文：对多 TB 数据集进行多小时计算
例如建立搜索索引，或排序，或分析网络结构
仅适用于数千台计算机
应用程序不是由分布式系统专家编写的
总体目标：对于非专业程序员来说很容易
程序员只需定义Map和Reduce函数

通常相当简单的顺序代码
MR 管理并隐藏分销的各个方面！

MapReduce 作业的抽象视图——字数统计
输入1 -> 地图 -> a,1 b,1
输入2 -> 地图 -> b,1
输入3 -> 地图 -> a,1 c,1
| | |
| | -> 减少 -> c,1
| -----> 减少 -> b,2
---------> 减少 -> a,2
1）输入（已经）被分割成M个文件

2. MR 对每个输入文件调用 Map()，生成一组 k2,v2
“中间”数据
每个 Map() 调用都是一个“任务”
3）当地图打开时，
MR 收集给定 k2 的所有中间 v2，
并将每个键+值传递给Reduce调用
3. 最终输出是来自Reduce()s的<k2,v3>对的集合

字数统计特定代码
地图(k, v)
将 v 拆分成单词
对于每个单词 w
发出（w，“1”）
减少（k，v_set）
发出（len（v_set））

MapReduce 具有良好的扩展性：
N 个“工作”计算机（可能）会为您带来 Nx 吞吐量。
Maps() 可以并行运行，因为它们不交互。
对于Reduce() 来说也是如此。
因此，更多的计算机 -> 更多的吞吐量 - 非常好！

MapReduce 隐藏了许多细节：
将应用程序代码发送到服务器
跟踪哪些任务已完成
将中间数据从 Maps“洗牌”到 Reduces
平衡服务器负载
从失败中恢复

然而，MapReduce 限制了应用程序可以执行的操作：
没有交互或状态（除了通过中间输出）。
无迭代
没有实时或流处理。

输入和输出存储在GFS集群文件系统上
MR需要巨大的并行输入和输出吞吐量。
GFS 将文件以 64 MB 的块的形式分割到许多服务器上
并行读取地图
减少并行写入

GFS 还将每个文件复制到 2 或 3 个服务器上
GFS 是 MapReduce 的一大胜利

一些细节（论文的图1）：
一名协调员，向工人分发任务并记住进度。

1. coordinator将Map任务交给worker，直到所有Map完成
   将写入输出（中间数据）映射到本地磁盘
   将输出按哈希值映射到每个Reduce 任务的一个文件中
   2.所有Map完成后，协调者分发Reduce任务
   每个Reduce从（所有）Map工作线程获取中间输出
   每个Reduce任务在GFS上写入一个单独的输出文件

什么可能会限制性能?
我们关心，因为这是需要优化的事情。
中央处理器？记忆？磁盘？网络?
2004 年，作者受到网络容量的限制。
MR 通过网络发送什么?
地图从 GFS 读取输入。
减少读取 Map 的中间输出。
通常与输入一样大，例如用于排序。
减少写入 GFS 的输出文件。
[图：服务器、网络交换机树]
在MR的all-to-all shuffle中，一半的流量经过根交换机。
Paper 的根交换机：总计 100 至 200 吉比特/秒
1800 台机器，因此 55 兆比特/秒/机器。
55 很小：远低于磁盘或 RAM 的速度。
如今：网络速度更快

MR 如何最大限度地减少网络使用?
协调器尝试在存储其输入的 GFS 服务器上运行每个 Map 任务。
所有计算机都运行 GFS 和 MR 工作线程
因此输入是从本地磁盘（通过 GFS）读取的，而不是通过网络读取的。
中间数据仅通过网络传输一次。
地图工作人员写入本地磁盘。
减少工作人员通过网络从 Map 工作人员磁盘读取的数据。
将其存储在 GFS 中至少需要通过网络两次。
中间数据分为包含许多键的文件。
R远小于键的数量。
大网络传输效率更高。

MR如何获得良好的负载均衡?
如果 N-1 个服务器必须等待 1 个慢速服务器完成，则浪费且缓慢。
但有些任务可能比其他任务需要更长的时间。
解决方案：任务比工人多得多。
协调员向完成先前任务的工人分发新任务。
因此，没有哪个任务大到可以主导完成时间（希望如此）。
因此，速度较快的服务器比较慢的服务器可以完成更多的任务，并且几乎同时完成。

容错能力怎么样?
即，如果工作人员在 MR 工作期间崩溃怎么办?
我们希望向应用程序程序员隐藏失败!
MR 是否必须从头开始重新运行整个作业?

为什么不?
MR 仅重新运行失败的Map() 和Reduce()。
假设 MR 运行 Map 两次，Reduce 看到第一次运行的输出，
另一个Reduce看到第二次运行的输出?
正确性需要重新执行才能产生完全相同的输出。
所以Map和Reduce必须是纯确定性函数:
他们只被允许查看他们的论点/输入。
无状态、无文件 I/O、无交互、无外部通信。
如果您想允许非功能性 Map 或 Reduce 该怎么办?
工人失败将需要重新执行整个工作，
或者您需要回滚到某种全局检查点。

Worker崩溃恢复的详细信息:

- 地图工作者崩溃:
  协调员注意到工作人员不再响应 ping
  协调器知道该工作人员运行了哪些 Map 任务
  这些任务的中间输出现已丢失，必须重新创建
  协调员告诉其他工作人员运行这些任务
  如果所有Reduce都已获取中间数据，则可以省略重新运行
- a 减少worker崩溃:
  已完成的任务没问题——存储在 GFS 中，并带有副本。
  协调员重新启动其他工作人员未完成的任务。

其他故障/问题:

- 如果协调器给两个工作人员相同的 Map() 任务怎么办?
  也许协调员错误地认为一名工人死亡。
  它只会告诉Reduce工人其中之一。
- 如果协调器给两个工作人员相同的Reduce() 任务怎么办?
  他们都会尝试在 GFS 上写入相同的输出文件!
  原子 GFS 重命名可防止混合；将看到一个完整的文件。
- 如果单个工人非常慢——“掉队者”怎么办?
  也许是由于硬件不稳定。
  协调器启动最后几个任务的第二个副本。
- 如果由于硬件或软件损坏，工作人员计算出错误的输出怎么办?
  太糟糕了！ MR 假定“故障停止”CPU 和软件。
- 如果协调器崩溃怎么办?

当前状态?
影响力巨大（Hadoop、Spark 等）。
Google 可能不再使用。
被 Flume / FlumeJava 取代（参见 Chambers 等人的论文）。
GFS 被 Colossus 取代（没有很好的描述），还有 BigTable。

结论
MapReduce 使大集群计算变得流行。

- 不是最有效或最灵活的。

- 扩展性良好。
- 易于编程——隐藏故障和数据移动。
  这些在实践中都是很好的权衡。

我们将在课程后面看到一些更先进的继任者。
祝实验室 1 愉快！

英文原文

6.824 2022 Lecture 1: Introduction

6.824: Distributed Systems Engineering

What I mean by "distributed system":
  a group of computers cooperating to provide a service
  this class is mostly about infrastructure services
    e.g. storage for big web sites, MapReduce, peer-to-peer sharing
  lots of important infrastructure is distributed

Why do people build distributed systems?
  to increase capacity via parallel processing
  to tolerate faults via replication
  to match distribution of physical devices e.g. sensors
  to achieve security via isolation

But it's not easy:
  many concurrent parts, complex interactions
  must cope with partial failure
  tricky to realize performance potential

Why study this topic?
  interesting -- hard problems, powerful solutions
  widely used -- driven by the rise of big Web sites
  active research area -- important unsolved problems
  challenging to build -- you'll do it in the labs

COURSE STRUCTURE

http://pdos.csail.mit.edu/6.824

Course staff:
  Robert Morris, lecturer
  Cel Skeggs, TA
  Assel Ismoldayeva, TA
  Anish Athalye, TA

Course components:
  lectures
  papers
  two exams
  labs
  final project (optional)

Lectures:
  big ideas, paper discussion, lab guidance
  will be video-taped, available online

Papers:
  there's a paper assigned for almost every lecture
  research papers, some classic, some new
  problems, ideas, implementation details, evaluation
  please read papers before class!
  each paper has a short question for you to answer
  and we ask you to send us a question you have about the paper
  submit answer and question before start of lecture

Exams:
  Mid-term exam in class
  Final exam during finals week
  Mostly about papers and labs

Labs:
  goal: deeper understanding of some important ideas
  goal: experience with distributed programming
  first lab is due a week from Friday
  one per week after that for a while

Lab 1: distributed big-data framework (like MapReduce)
Lab 2: fault tolerance library using replication (Raft)
Lab 3: a simple fault-tolerant database
Lab 4: scalable database performance via sharding

Optional final project at the end, in groups of 2 or 3.
  The final project substitutes for Lab 4.
  You think of a project and clear it with us.
  Code, short write-up, demo on last day.

Warning: debugging the labs can be time-consuming
  start early
  ask questions on Piazza
  use the TA office hours

We grade the labs using a set of tests
  we give you all the tests; none are secret

MAIN TOPICS

This is a course about infrastructure for applications.

- Storage.
- Communication.
- Computation.

A big goal: hide the complexity of distribution from applications.

Topic: fault tolerance
  1000s of servers, big network -> always something broken
    We'd like to hide these failures from the application.
    "High availability": service continues despite failures
  Big idea: replicated servers.
    If one server crashes, can proceed using the other(s).
    Labs 2 and 3

Topic: consistency
  General-purpose infrastructure needs well-defined behavior.
    E.g. "Get(k) yields the value from the most recent Put(k,v)."
  Achieving good behavior is hard!
    "Replica" servers are hard to keep identical.

Topic: performance
  The goal: scalable throughput
    Nx servers -> Nx total throughput via parallel CPU, disk, net.
  Scaling gets harder as N grows:
    Load imbalance.
    Slowest-of-N latency.
    Some things don't speed up with N: initialization, interaction.
  Labs 1, 4

Topic: tradeoffs
  Fault-tolerance, consistency, and performance are enemies.
  Fault tolerance and consistency require communication
    e.g., send data to backup
    e.g., check if my data is up-to-date
    communication is often slow and non-scalable
  Many designs provide only weak consistency, to gain speed.
    e.g. Get() does *not* yield the latest Put()!
    Painful for application programmers but may be a good trade-off.
  We'll see many design points in the consistency/performance spectrum.

Topic: implementation
  RPC, threads, concurrency control, configuration.
  The labs...

This material comes up a lot in the real world.
  All big web sites and cloud providers are expert at distributed systems.
  Many big open source projects are built around these ideas.
  We'll read multiple papers from industry.
  And industry has adopted many ideas from academia.

CASE STUDY: MapReduce

Let's talk about MapReduce (MR)
  a good illustration of 6.824's main topics
  hugely influential
  the focus of Lab 1

MapReduce overview
  context: multi-hour computations on multi-terabyte data-sets
    e.g. build search index, or sort, or analyze structure of web
    only practical with 1000s of computers
    applications not written by distributed systems experts
  overall goal: easy for non-specialist programmers
  programmer just defines Map and Reduce functions
    often fairly simple sequential code
  MR manages, and hides, all aspects of distribution!

Abstract view of a MapReduce job -- word count

```
Input1 -> Map -> a,1 b,1
Input2 -> Map ->     b,1
Input3 -> Map -> a,1     c,1
                 |   |   |
                 |   |   -> Reduce -> c,1
                 |   -----> Reduce -> b,2
                 ---------> Reduce -> a,2
```

1. input is (already) split into M files
2. MR calls Map() for each input file, produces set of k2,v2
   "intermediate" data
   each Map() call is a "task"
3. when Maps are don,
   MR gathers all intermediate v2's for a given k2,
   and passes each key + values to a Reduce call
4. final output is set of <k2,v3> pairs from Reduce()s

Word-count-specific code

```
Map(k, v)
  split v into words
  for each word w
    emit(w, "1")
Reduce(k, v_set)
  emit(len(v_set))
```

MapReduce scales well:

N "worker" computers (might) get you Nx throughput.
  Maps()s can run in parallel, since they don't interact.
  Same for Reduce()s.
Thus more computers -> more throughput -- very nice!

MapReduce hides many details:

sending app code to servers
tracking which tasks have finished
"shuffling" intermediate data from Maps to Reduces
balancing load over servers
recovering from failures

However, MapReduce limits what apps can do:

No interaction or state (other than via intermediate output).
No iteration
No real-time or streaming processing.

Input and output are stored on the GFS cluster file system

MR needs huge parallel input and output throughput.
GFS splits files over many servers, in 64 MB chunks
  Maps read in parallel
  Reduces write in parallel
GFS also replicates each file on 2 or 3 servers
GFS is a big win for MapReduce

Some details (paper's Figure 1):
one coordinator, that hands out tasks to workers and remembers progress.

1. coordinator gives Map tasks to workers until all Maps complete
   Maps write output (intermediate data) to local disk
   Maps split output, by hash, into one file per Reduce task
2. after all Maps have finished, coordinator hands out Reduce tasks
   each Reduce fetches its intermediate output from (all) Map workers
   each Reduce task writes a separate output file on GFS

What will likely limit the performance?
We care since that's the thing to optimize.
CPU? memory? disk? network?
In 2004 authors were limited by network capacity.
What does MR send over the network?
Maps read input from GFS.
Reduces read Map intermediate output.
Often as large as input, e.g. for sorting.
Reduces write output files to GFS.
[diagram: servers, tree of network switches]
In MR's all-to-all shuffle, half of traffic goes through root switch.
Paper's root switch: 100 to 200 gigabits/second, total
1800 machines, so 55 megabits/second/machine.
55 is small: much less than disk or RAM speed.
Today: networks are much faster

How does MR minimize network use?
Coordinator tries to run each Map task on GFS server that stores its input.
All computers run both GFS and MR workers
So input is read from local disk (via GFS), not over network.
Intermediate data goes over network just once.
Map worker writes to local disk.
Reduce workers read from Map worker disks over the network.
Storing it in GFS would require at least two trips over the network.
Intermediate data partitioned into files holding many keys.
R is much smaller than the number of keys.
Big network transfers are more efficient.

How does MR get good load balance?
Wasteful and slow if N-1 servers have to wait for 1 slow server to finish.
But some tasks likely take longer than others.
Solution: many more tasks than workers.
Coordinator hands out new tasks to workers who finish previous tasks.
So no task is so big it dominates completion time (hopefully).
So faster servers do more tasks than slower ones, finish abt the same time.

What about fault tolerance?
I.e. what if a worker crashes during a MR job?
We want to hide failures from the application programmer!
Does MR have to re-run the whole job from the beginning?
Why not?
MR re-runs just the failed Map()s and Reduce()s.

Suppose MR runs a Map twice, one Reduce sees first run's output,
another Reduce sees the second run's output?
Correctness requires re-execution to yield exactly the same output.
So Map and Reduce must be pure deterministic functions:
they are only allowed to look at their arguments/input.
no state, no file I/O, no interaction, no external communication.
What if you wanted to allow non-functional Map or Reduce?
Worker failure would require whole job to be re-executed,
or you'd need to roll back to some kind of global checkpoint.

Details of worker crash recovery:

- a Map worker crashes:
  coordinator notices worker no longer responds to pings
  coordinator knows which Map tasks ran on that worker
  those tasks' intermediate output is now lost, must be re-created
  coordinator tells other workers to run those tasks
  can omit re-running if all Reduces have fetched the intermediate data
- a Reduce worker crashes:
  finished tasks are OK -- stored in GFS, with replicas.
  coordinator re-starts worker's unfinished tasks on other workers.

Other failures/problems:

- What if the coordinator gives two workers the same Map() task?
  perhaps the coordinator incorrectly thinks one worker died.
  it will tell Reduce workers about only one of them.
- What if the coordinator gives two workers the same Reduce() task?
  they will both try to write the same output file on GFS!
  atomic GFS rename prevents mixing; one complete file will be visible.
- What if a single worker is very slow -- a "straggler"?
  perhaps due to flakey hardware.
  coordinator starts a second copy of last few tasks.
- What if a worker computes incorrect output, due to broken h/w or s/w?
  too bad! MR assumes "fail-stop" CPUs and software.
- What if the coordinator crashes?

Current status?
Hugely influential (Hadoop, Spark, &c).
Probably no longer in use at Google.
Replaced by Flume / FlumeJava (see paper by Chambers et al).
GFS replaced by Colossus (no good description), and BigTable.

Conclusion
MapReduce made big cluster computation popular.

- Not the most efficient or flexible.

- Scales well.
- Easy to program -- failures and data movement are hidden.
  These were good trade-offs in practice.
  We'll see some more advanced successors later in the course.
  Have fun with Lab 1!

LEC 2 RPC and Threads

中文翻译

6.824 2022 第2讲：基础设施：RPC和线程

今天：
  Go 中的线程和 RPC，着眼于实验室

为什么去?
  对线程的良好支持
  便捷的远程过程调用
  类型和内存安全
  垃圾收集（释放问题后无法使用）
  线程+GC特别有吸引力!
  不太复杂

学习完教程后，请使用 https://golang.org/doc/ effective_go.html

线程数
  一个有用的结构化工具，但可能很棘手
  Go 称它们为 goroutine；其他人都称它们为线程

Thread =“执行线程”
  线程允许一个程序同时做很多事情
  每个线程串行执行，就像普通的非线程程序一样
  线程共享内存
  每个线程都包含一些每个线程的状态：
    程序计数器，寄存器，堆栈，它在等待什么

为什么要线程?
  I/O并发
    客户端并行向许多服务器发送请求并等待回复。
    服务器处理多个客户端请求；每个请求都可能会阻塞。
    在等待磁盘为客户端X读取数据时，
      处理来自客户端 Y 的请求。
  多核性能
    在多个内核上并行执行代码。
  方便
    在后台，每秒检查一次每个工作人员是否还活着。

有没有线程的替代品?
  是的：在单个线程中编写显式交错活动的代码。
    通常称为“事件驱动”。
  保留有关每个活动（例如每个客户端请求）的状态表。
  一个“事件”循环：
    检查每个活动的新输入（例如来自服务器的回复的到达），
    为每项活动执行下一步，
    更新状态。
  事件驱动为您提供 I/O 并发性，
    并消除了线程成本（这可能是巨大的），
    但没有获得多核加速，
    并且编程很痛苦。

线程挑战：
安全共享数据
如果两个线程同时执行 n = n + 1 会怎样？
或者一个线程读取而另一个线程递增？
这是一场“竞赛”——而且通常是一个错误
-> 使用锁（Go 的sync.Mutex）
-> 或避免共享可变数据
线程之间的协调
一个线程正在生成数据，另一个线程正在消耗数据
消费者如何等待（并释放CPU）？
生产者如何唤醒消费者？
-> 使用Go通道或sync.Cond或sync.WaitGroup
僵局
通过锁和/或通信（例如 RPC 或 Go 通道）进行循环

让我们看看本教程的网络爬虫作为线程示例。

什么是网络爬虫？
目标：获取所有网页，例如提供给索引器
你给它一个起始网页
它递归地跟踪所有链接
但不要多次获取给定页面
并且不要陷入循环

爬虫挑战
利用 I/O 并发性
网络延迟比网络容量更受限制
同时获取多个 URL
增加每秒提取的 URL
=> 使用线程进行并发
仅获取每个 URL 一次
避免浪费网络带宽
对远程服务器友善
=> 需要记住访问过哪些 URL
知道什么时候完成

我们将研究三种风格的解决方案 [crawler.go on Schedule page]

串口爬虫：
通过递归串行调用执行深度优先探索
“获取”的地图避免重复，打破循环
单个映射，通过引用传递，调用者可以看到被调用者的更新
但是：一次只获取一页——慢
我们可以在 Serial() 调用前面放一个“go”吗？
让我们尝试一下...发生了什么？

ConcurrentMutex 爬虫：
为每个页面获取创建一个线程
许多并发提取，更高的提取率
“go func”创建一个 goroutine 并开始运行
func...是一个“匿名函数”
线程共享“获取的”地图

因此只有一个线程会获取任何给定的页面
为什么要使用互斥锁（Lock() 和 Unlock()）？
一个理由：
两个线程使用相同的 URL 同时调用 ConcurrentMutex()
由于两个不同的页面包含指向同一 URL 的链接
T1 读取 fetched[url]，T2 读取 fetched[url]
两者都看到 url 尚未被获取（已经 == false）
两者都是fetch，这是错误的
互斥体导致一个等待，而另一个同时进行检查和设置
所以只有一个线程看到already==false
我们说“锁保护数据”
但Go并不强制锁和数据之间有任何关系！
锁定/解锁之间的代码通常称为“临界区”
另一个原因：
从内部来看，map 是一个复杂的数据结构（树？可扩展哈希？）
并发更新/更新可能会破坏内部不变量
并发更新/读取可能会导致读取崩溃
如果我注释掉 Lock() / Unlock() 会怎样？
去运行爬虫.go
为什么它有效？
去跑-racecrawler.go
即使输出正确，也能检测竞争！
ConcurrentMutex 爬虫如何决定它已完成？
同步等待组
Wait() 等待所有 Add() 被 Done() 平衡
即等待所有子线程完成
[图：goroutines 树，覆盖在循环 URL 图上]
树中的每个节点都有一个 WaitGroup
该爬虫可能会创建多少个并发线程？

并发通道爬虫
一个 Go 频道：
通道是一个对象
ch := make(chan int)
通道允许一个线程将对象发送到另一个线程
ch <- x
发送者等待直到某个 goroutine 接收到
y := <- ch
for y := 范围 ch
接收者等待直到某个 goroutine 发送
通道既可以通信又可以同步
多个线程可以在一个通道上发送和接收
渠道便宜
请记住：发送方会阻塞，直到接收方收到为止！
“同步”
注意僵局
并发通道协调器()
coordinator() 创建一个工作协程来获取每个页面
worker() 在通道上发送页面 URL 片段
多个工作人员在单个通道上发送

coordinator() 从通道读取 URL 切片
协调员在哪一行等待?
协调器在等待时是否使用 CPU 时间?
注意：这里没有递归；相反，有一个工作清单。
注意：无需锁定获取的地图，因为它不共享！
协调员如何知道它已经完成?
保存 n 中的工人数量。
每个工作人员在通道上只发送一件物品。

为什么多个线程使用同一个通道是安全的?

工作线程写入 url 切片，协调器读取它，这是一场竞赛吗?
*工作人员仅在发送之前*写入切片

- 协调器仅在*接收后*读取切片
  所以他们不能同时使用切片。

为什么 ConcurrentChannel() 只为“ch <- ...”创建一个 goroutine?
让我们摆脱 goroutine...

何时使用共享和锁，而不是通道?
大多数问题都可以通过两种方式解决
什么最有意义取决于程序员如何思考
状态——共享和锁
沟通——渠道
对于 6.824 实验，我建议状态共享+锁，
和sync.Cond或通道或time.Sleep()用于等待/通知。

远程过程调用 (RPC)
分布式系统机制的关键部分；所有实验室都使用RPC
目标：易于编程的客户端/服务器通信
隐藏网络协议的详细信息
将数据（字符串、数组、映射等）转换为“有线格式”
可移植性/互操作性

RPC消息图：
客户端服务器
请求--->
<---回应

软件结构
客户端应用程序处理程序 fns
存根 FNS 调度程序
RPC 库 RPC 库
网 ---------- 网

Go 示例：日程页面上的 kv.go
一个玩具键/值存储服务器——Put(key,value), Get(key)->value
使用Go的RPC库
常见的：
为每个服务器处理程序声明 Args 和 Reply 结构。
客户：
connect() 的 Dial() 创建到服务器的 TCP 连接
get() 和 put() 是客户端“存根”

Call() 要求 RPC 库执行调用
您指定服务器函数名称、参数、放置回复的位置
库编组参数、发送请求、等待、解组回复
Call() 的返回值指示是否收到回复
通常您还会收到回复。Err 表明服务级别失败
服务器：
Go 要求服务器声明一个带有方法的对象作为 RPC 处理程序
然后服务器向 RPC 库注册该对象
服务器接受 TCP 连接，将它们提供给 RPC 库
RPC 库
读取每个请求
为此请求创建一个新的 goroutine
解组请求
查找命名对象（在 Register() 创建的表中）
调用对象的命名方法（调度）
马歇尔回复
在 TCP 连接上写入回复
服务器的 Get() 和 Put() 处理程序
必须锁定，因为 RPC 库为每个请求创建一个新的 goroutine
读取参数；修改回复

一些细节：
绑定：客户端如何知道要与哪台服务器计算机通信？
对于 Go 的 RPC，服务器名称/端口是 Dial 的参数
大系统有某种名称或配置服务器
编组：将数据格式化为数据包
Go 的 RPC 库可以传递字符串、数组、对象、映射等
Go通过复制指向的数据来传递指针
无法传递通道或函数

RPC问题：失败怎么办？
例如丢包、网络中断、服务器缓慢、服务器崩溃

对于客户端 RPC 库来说，失败是什么样子的？
客户端永远看不到服务器的响应
客户端*不*知道服务器是否看到了请求！
【各点损失图】
也许服务器从未见过该请求
也许服务器已执行，在发送回复之前崩溃
也许服务器已执行，但网络在发送回复之前就死掉了

最简单的故障处理方案：“尽力而为的 RPC”
Call() 等待响应一段时间
如果没有到达，则重新发送请求
这样做几次
然后放弃并返回错误

问：“尽力而为”对于应用程序来说容易应对吗？

一个特别糟糕的情况：
客户端执行
把（“k”，10）；
把（“k”，20）；

都成功了
Get("k") 会产生什么结果?
【图，超时，重发，原件迟到】

问：尽力而为可以吗?
只读操作
如果重复则不执行任何操作的操作
例如数据库检查记录是否已被插入

更好的 RPC 行为："最多一次 RPC"
思路：如果没有应答，客户端重新发送；
服务器 RPC 代码检测重复请求，
返回先前的回复而不是重新运行处理程序
问：如何检测重复请求?
客户端在每个请求中包含唯一 ID (XID)
使用相同的 XID 重新发送
服务器：
如果看到[xid]：
r = 旧[xid]
别的
r = 处理程序()
旧[xid] = r
看到[xid] = true

一些至多一次的复杂性
这将在实验 3 中出现
如果两个客户端使用相同的 XID 怎么办?
大随机数?
如何避免巨大的 saw[xid] 表?
主意：
每个客户端都有一个唯一的 ID（可能是一个很大的随机数）
每个客户端 RPC 序列号
客户端在每个 RPC 中都包含"已查看所有回复 <= X"
很像 TCP 序列 #s 和 ack
那么服务器可以保持 O(# 个客户端) 状态，而不是 O(# XIDs)
服务器最终必须丢弃有关旧 RPC 或旧客户端的信息
什么时候丢弃是安全的?
如何在原始仍在执行时处理 dup req?
服务器还不知道回复
idea：每个正在执行的 RPC 的"pending"标志；等待或忽略

如果最多一次服务器崩溃并重新启动怎么办?
如果内存中最多重复一次信息，服务器将忘记
并在重新启动后接受重复的请求
也许它应该将重复信息写入磁盘
也许副本服务器也应该复制重复的信息

Go RPC 是"最多一次"的简单形式
打开 TCP 连接
向 TCP 连接写入请求
Go RPC 永远不会重新发送请求
所以服务器不会看到重复的请求

如果没有得到回复，Go RPC 代码将返回错误
也许在超时之后（来自 TCP）
也许服务器没有看到请求
也许服务器处理了请求，但服务器/网络在回复返回之前失败

那么“恰好一次”呢?
无限制重试加上重复检测加上容错服务
实验室3

英文原文

6.824 2022 Lecture 2: Infrastructure: RPC and threads

Today:
Threads and RPC in Go, with an eye towards the labs

Why Go?
good support for threads
convenient RPC
type- and memory- safe
garbage-collected (no use after freeing problems)
threads + GC is particularly attractive!
not too complex

After the tutorial, use https://golang.org/doc/effective_go.html

Threads
a useful structuring tool, but can be tricky
Go calls them goroutines; everyone else calls them threads

Thread = "thread of execution"
threads allow one program to do many things at once
each thread executes serially, just like an ordinary non-threaded program
the threads share memory
each thread includes some per-thread state:
program counter, registers, stack, what it's waiting for

Why threads?
I/O concurrency
Client sends requests to many servers in parallel and waits for replies.
Server processes multiple client requests; each request may block.
While waiting for the disk to read data for client X,
process a request from client Y.
Multicore performance
Execute code in parallel on several cores.
Convenience
In background, once per second, check whether each worker is still alive.

Is there an alternative to threads?
Yes: write code that explicitly interleaves activities, in a single thread.
Usually called "event-driven."
Keep a table of state about each activity, e.g. each client request.
One "event" loop that:
checks for new input for each activity (e.g. arrival of reply from server),

  does the next step for each activity,
  updates state.
 Event-driven gets you I/O concurrency,
  and eliminates thread costs (which can be substantial),
  but doesn't get multi-core speedup,
  and is painful to program.

Threading challenges:
 sharing data safely
  what if two threads do n = n + 1 at the same time?
   or one thread reads while another increments?
  this is a "race" -- and is often a bug
  -> use locks (Go's sync.Mutex)
  -> or avoid sharing mutable data
 coordination between threads
  one thread is producing data, another thread is consuming it
   how can the consumer wait (and release the CPU)?
   how can the producer wake up the consumer?
  -> use Go channels or sync.Cond or sync.WaitGroup
 deadlock
  cycles via locks and/or communication (e.g. RPC or Go channels)

Let's look at the tutorial's web crawler as a threading example.

What is a web crawler?
 goal: fetch all web pages, e.g. to feed to an indexer
 you give it a starting web page
 it recursively follows all links
 but don't fetch a given page more than once
  and don't get stuck in cycles

Crawler challenges
 Exploit I/O concurrency
  Network latency is more limiting than network capacity
  Fetch many URLs at the same time
   To increase URLs fetched per second
  => Use threads for concurrency
 Fetch each URL only *once*
  avoid wasting network bandwidth
  be nice to remote servers
  => Need to remember which URLs visited
 Know when finished

We'll look at three styles of solution [crawler.go on schedule page]

Serial crawler:
 performs depth-first exploration via recursive Serial calls
 the "fetched" map avoids repeats, breaks cycles
  a single map, passed by reference, caller sees callee's updates
 but: fetches only one page at a time -- slow
  can we just put a "go" in front of the Serial() call?
  let's try it... what happened?

ConcurrentMutex crawler:
Creates a thread for each page fetch
Many concurrent fetches, higher fetch rate
the "go func" creates a goroutine and starts it running
func... is an "anonymous function"
The threads share the "fetched" map
So only one thread will fetch any given page
Why the Mutex (Lock() and Unlock())?
One reason:
Two threads make simultaneous calls to ConcurrentMutex() with same URL
Due to two different pages containing link to same URL
T1 reads fetched[url], T2 reads fetched[url]
Both see that url hasn't been fetched (already == false)
Both fetch, which is wrong
The mutex causes one to wait while the other does both check and set
So only one thread sees already==false
We say "the lock protects the data"
But not Go does not enforce any relationship between locks and data!
The code between lock/unlock is often called a "critical section"
Another reason:
Internally, map is a complex data structure (tree? expandable hash?)
Concurrent update/update may wreck internal invariants
Concurrent update/read may crash the read
What if I comment out Lock() / Unlock()?
go run crawler.go
Why does it work?
go run -race crawler.go
Detects races even when output is correct!
How does the ConcurrentMutex crawler decide it is done?
sync.WaitGroup
Wait() waits for all Add()s to be balanced by Done()s
i.e. waits for all child threads to finish
[diagram: tree of goroutines, overlaid on cyclic URL graph]
there's a WaitGroup per node in the tree
How many concurrent threads might this crawler create?

ConcurrentChannel crawler
a Go channel:
a channel is an object
ch := make(chan int)
a channel lets one thread send an object to another thread
ch <- x
the sender waits until some goroutine receives
y := <- ch
for y := range ch
a receiver waits until some goroutine sends
channels both communicate and synchronize
several threads can send and receive on a channel
channels are cheap
remember: sender blocks until the receiver receives!

"synchronous"
watch out for deadlock
ConcurrentChannel coordinator()
coordinator() creates a worker goroutine to fetch each page
worker() sends slice of page's URLs on a channel
multiple workers send on the single channel
coordinator() reads URL slices from the channel
At what line does the coordinator wait?
Does the coordinator use CPU time while it waits?
Note: there is no recursion here; instead there's a work list.
Note: no need to lock the fetched map, because it isn't shared!
How does the coordinator know it is done?
Keeps count of workers in n.
Each worker sends exactly one item on channel.

Why is it safe for multiple threads use the same channel?

Worker thread writes url slice, coordinator reads it, is that a race?

- worker only writes slice *before* sending
- coordinator only reads slice *after* receiving
  So they can't use the slice at the same time.

Why does ConcurrentChannel() create a goroutine just for "ch <- ..."?
Let's get rid of the goroutine...

When to use sharing and locks, versus channels?
Most problems can be solved in either style
What makes the most sense depends on how the programmer thinks
state -- sharing and locks
communication -- channels
For the 6.824 labs, I recommend sharing+locks for state,
and sync.Cond or channels or time.Sleep() for waiting/notification.

Remote Procedure Call (RPC)
a key piece of distributed system machinery; all the labs use RPC
goal: easy-to-program client/server communication
hide details of network protocols
convert data (strings, arrays, maps, &c) to "wire format"
portability / interoperability

RPC message diagram:
Client Server
request--->
<---response

Software structure
client app handler fns
stub fns dispatcher
RPC lib RPC lib
net ---------- net

Go example: kv.go on schedule page
  A toy key/value storage server -- Put(key,value), Get(key)->value
  Uses Go's RPC library
  Common:
    Declare Args and Reply struct for each server handler.
  Client:
    connect()'s Dial() creates a TCP connection to the server
    get() and put() are client "stubs"
    Call() asks the RPC library to perform the call
      you specify server function name, arguments, place to put reply
      library marshalls args, sends request, waits, unmarshalls reply
      return value from Call() indicates whether it got a reply
      usually you'll also have a reply.Err indicating service-level failure
  Server:
    Go requires server to declare an object with methods as RPC handlers
    Server then registers that object with the RPC library
    Server accepts TCP connections, gives them to RPC library
    The RPC library
      reads each request
      creates a new goroutine for this request
      unmarshalls request
      looks up the named object (in table create by Register())
      calls the object's named method (dispatch)
      marshalls reply
      writes reply on TCP connection
    The server's Get() and Put() handlers
      Must lock, since RPC library creates a new goroutine for each request
      read args; modify reply

A few details:
  Binding: how does client know what server computer to talk to?
    For Go's RPC, server name/port is an argument to Dial
    Big systems have some kind of name or configuration server
  Marshalling: format data into packets
    Go's RPC library can pass strings, arrays, objects, maps, &c
    Go passes pointers by copying the pointed-to data
    Cannot pass channels or functions

RPC problem: what to do about failures?
  e.g. lost packet, broken network, slow server, crashed server

What does a failure look like to the client RPC library?
  Client never sees a response from the server
  Client does *not* know if the server saw the request!
    [diagram of losses at various points]
    Maybe server never saw the request
    Maybe server executed, crashed just before sending reply
    Maybe server executed, but network died just before delivering reply

Simplest failure-handling scheme: "best-effort RPC"
  Call() waits for response for a while
  If none arrives, re-send the request
  Do this a few times
  Then give up and return an error

Q: is "best effort" easy for applications to cope with?

A particularly bad situation:
  client executes
    Put("k", 10);
    Put("k", 20);
  both succeed
  what will Get("k") yield?
  [diagram, timeout, re-send, original arrives late]

Q: is best effort ever OK?
   read-only operations
   operations that do nothing if repeated
     e.g. DB checks if record has already been inserted

Better RPC behavior: "at-most-once RPC"
  idea: client re-sends if no answer;
    server RPC code detects duplicate requests,
    returns previous reply instead of re-running handler
  Q: how to detect a duplicate request?
  client includes unique ID (XID) with each request
    uses same XID for re-send
  server:
    if seen[xid]:
      r = old[xid]
    else
      r = handler()
      old[xid] = r
      seen[xid] = true

some at-most-once complexities
  this will come up in lab 3
  what if two clients use the same XID?
    big random number?
  how to avoid a huge seen[xid] table?
    idea:
      each client has a unique ID (perhaps a big random number)
      per-client RPC sequence numbers
      client includes "seen all replies <= X" with every RPC
      much like TCP sequence #s and acks
    then server can keep O(# clients) state, rather than O(# XIDs)
  server must eventually discard info about old RPCs or old clients
    when is discard safe?
  how to handle dup req while original is still executing?
    server doesn't know reply yet
    idea: "pending" flag per executing RPC; wait or ignore

What if an at-most-once server crashes and re-starts?
  if at-most-once duplicate info in memory, server will forget
    and accept duplicate requests after re-start
  maybe it should write the duplicate info to disk
  maybe replica server should also replicate duplicate info

Go RPC is a simple form of "at-most-once"
  open TCP connection
  write request to TCP connection
  Go RPC never re-sends a request
    So server won't see duplicate requests
  Go RPC code returns an error if it doesn't get a reply
    perhaps after a timeout (from TCP)
    perhaps server didn't see request
    perhaps server processed request but server/net failed before reply came back

What about "exactly once"?
  unbounded retries plus duplicate detection plus fault-tolerant service
  Lab 3

LEC 3 GFS

中文翻译

6.824 2022年第3讲：政府飞行服务

谷歌文件系统
Sanjay Ghemawat、Howard Gobioff 和梁舜德
2003年国家安全计划

我们为什么要阅读这篇论文?
  分布式存储是一个关键的抽象
    界面/语义应该是什么样的?
    它的内部应该如何运作?
  GFS 论文涉及 6.824 的许多主题
    并行性能、容错、复制、一致性
  很好的系统论文——从应用程序一直到网络的详细信息
  成功的现实世界设计
  有影响力：Hadoop的HDFS

分布式存储为什么难?
  高性能 -> 将数据分片到许多服务器上
  许多服务器 -> 持续出现故障
  容错->复制
  复制 -> 潜在的不一致
  更好的一致性 -> 低性能

我们想要什么来保持一致性?
  理想模型：与单个服务器相同的行为
  理想的服务器一次执行一个客户端操作
    即使多个客户端同时发出操作
  读取反映以前的写入
    即使服务器崩溃并重新启动

所有客户端看到相同的数据
因此：
假设C1和C2同时写入，并且写入之后
完成后，C3和C4读取。他们能看到什么？
C1：宽x1
C2：宽x2
C3：接收？
C4：接收？
答案：1或2，但两者必须看到相同的值。
这是一个“强”一致性模型。
但单台服务器容错能力较差。

容错复制使得强一致性变得棘手。
一个简单但破碎的复制方案：
两个副本服务器，S1 和 S2
客户端并行向两者发送写入
客户端将读取发送到
在我们的示例中，C1 和 C2 的写入消息可能到达
两个副本的顺序不同
如果 C3 读取 S1，它可能会看到 x=1
如果 C4 读取 S2，它可能会看到 x=2
或者如果 S1 收到写入怎么办，但是
客户端在将写入发送到 S2 之前崩溃了？
这不是强一致性！
更好的一致性通常需要沟通
确保副本保持同步——可能会很慢！
性能和一致性之间可能存在很多权衡
今天我们会看到一个

政府财政司司长

语境：
许多 Google 服务需要一个大型、快速的统一存储系统
Mapreduce、爬虫、索引器、日志存储/分析
在多个应用程序之间共享，例如抓取、索引、分析
在许多服务器/磁盘上自动“分片”每个文件
对于许多客户端的并行性能，例如 MapReduce
对于一台服务器来说太大的大文件
故障自动恢复
每次部署只需一个数据中心
仅 Google 应用程序/用户
针对大文件的顺序访问；阅读或追加
即不是小项目的低延迟数据库

2003 年这方面有什么新鲜事吗？他们是如何让 SOSP 论文被接受的？
不是分布、分片、容错的基本思想。
规模巨大。
应用于工业、真实世界的经验。
成功使用弱一致性。

整体结构
客户端（库、RPC——但作为 UNIX FS 不可见）
协调器跟踪文件名
chunkservers 存储 64 MB 的块
大文件在许多块服务器上分割成 64 MB 的块
大块 -> 簿记开销低
每个块复制到 3 个块服务器上
为什么是 3 而不是 2?

协调者状态
RAM 中的表（为了速度，必须较小）：
文件名 -> 块句柄数组 (nv)
块句柄 -> 版本号 (nv)
块服务器列表 (v)
初级（五）
租赁时间 (v)
“nv”状态也写入磁盘，以防崩溃+重启

当客户端C想要读取一个文件时，有哪些步骤?

1. C 将文件名和偏移量发送给协调器（CO）（如果没有缓存）
2. CO 找到该偏移量的块句柄
3. CO 回复 chunkhandles + chunkservers 列表
   仅那些拥有最新版本的
4. C 缓存句柄 + chunkserver 列表
5. C 向最近的 chunkserver 发送请求
   块句柄，偏移量
6. chunk server读取磁盘上的chunk文件，返回给client

客户端只向协调器询问文件名和块句柄列表
客户端缓存名称 -> 块句柄信息
协调器不处理数据，因此（希望）负载不重

协调器如何知道哪些 chunkservers 有给定的 chunk?

当C想要进行“记录追加”时，有哪些步骤?
论文图2

1. C 向 CO 询问文件的最后一个块
2. CO告诉C主要和次要
3. C向所有人发送数据（只是暂时的……），等待所有回复（？）
4. C告诉P追加
5. P 检查租约尚未过期，并且 chunk 有空间
6. P 选择一个偏移量（在块的末尾）
7. P写入自己的chunk文件（一个Linux文件）
8. P 告诉每个辅助的偏移量，告诉附加到块文件
9. P等待所有从节点回复，或者超时
   辅助设备可以回复“错误”，例如磁盘空间不足
10. P告诉C“ok”或“error”
11. C 如果出错则从头开始重试

GFS为客户提供哪些一致性保证?
需要采用告诉应用程序如何使用 GFS 的形式。

这是一种可能性：

如果主节点告诉客户端记录追加成功，那么
随后打开该文件并扫描它的任何读者都会看到
某处附加的记录。

这允许出现很多异常情况：不同的客户端可能会看到以下位置的记录：
不同的订单；他们可能会看到重复的记录；他们可能会看到记录
来自失败的写入。GFS申请必须准备好！

我们如何思考GFS如何履行保障？
看看它对各种故障的处理：
崩溃、崩溃+重启、崩溃+替换、消息丢失、分区。
请问GFS如何保证对各种故障的保障。

- 如果客户端在记录追加序列期间失败怎么办？
- 如果追加的客户端缓存了一个过时的（错误的）主块怎么办？
- 如果读取客户端缓存了一个块的过时服务器列表怎么办？
- 协调器崩溃+重启是否会导致它忘记该文件？
  或者忘记哪些 chunkserver 保存相关的 chunk？
- 两个客户端同时记录追加。
  他们会覆盖彼此的记录吗？
- 假设一个辅助设备没有听到来自主设备的追加命令。
  由于临时网络故障。
  如果读取客户端从该辅助读取怎么办？
- 如果主 S1 处于活动状态并且正在服务客户端请求怎么办？
  但是协调器和S1之间的网络出现故障？
  “网络分区”
  协调员会选择新的主要成员吗？
  现在会有两次初选吗？
  这样附加到一个主数据库，而读取到另一个主数据库？
  从而破坏了一致性保证？
  “裂脑”
- 如果主节点在向所有辅助节点发送附加内容之前崩溃怎么办？
  是否可以选择“没有”看到附加的辅助节点作为新的主节点？
- 具有旧的陈旧块副本的 Chunkserver S4 已脱机。
  主节点和所有活动的辅助节点崩溃。
  S4 恢复生机（在小学和中学之前）。
  协调器会选择 S4（具有陈旧块）作为主节点吗？
  最好是拥有包含旧数据的主数据库，还是根本没有副本？
- 协调器如何为块设置主节点？
  如果客户端想要写入，但没有主租约，或者主租约已过期并失效。
  协调员一直在轮询 chunkservesr 他们拥有哪些块/版本。
  A。如果没有带有最新版本的块服务器#，则错误
  b. 从最新版本中选择主要 P 和辅助 P #
  C。增加版本#，写入磁盘
  d. 告诉P和次要他们是谁，以及新版本#
  e. 副本将新版本#写入磁盘
- 如果辅助服务器总是写入失败，主服务器应该怎么办？
  例如死机、磁盘空间不足或磁盘损坏。
  客户请求不断失败？

或者要求协调员声明一组新的服务器和新版本？
论文没有描述这个过程。

- 如果有一个分区的主服务客户端附加，及其
  租约到期，协调员选择一个新的主节点，新的主节点将
  主分区是否有由分区主分区更新的最新数据？
- 如果协调器完全失败怎么办？
  替代者会知道死去的协调员所知道的一切吗？
  例如每个块的版本号？基本的？租约到期时间？
- 谁/什么决定协调员已经死亡并且必须被替换？
  协调器副本是否可以 ping 协调器，如果没有响应就接管？
- 如果整栋大楼停电怎么办？
  然后电源恢复，所有服务器重新启动。
- 假设协调者想要创建一个新的块副本。
  可能是因为副本太少了。
  假设它是文件中的最后一个块，并且被附加到。
  新副本如何确保它不会错过任何追加？
  毕竟它还不是次要之一。
- 是否有*任何*情况下GFS会违反保证？
  即追加成功，但后续读者看不到该记录。
  所有协调器副本都会永久丢失状态（永久性磁盘故障）。
  可能更糟：结果将是“没有答案”，而不是“数据不正确”。
  “故障停止”
  所有持有该块的块服务器都会永久丢失磁盘内容。
  再次，故障停止；不是最坏的可能结果
  CPU、RAM、网络或磁盘产生不正确的值。
  校验和捕获某些情况，但不是全部
  时间没有正确同步，因此租约不起作用。
  因此，多个初选，也许写入到一个，读取到另一个。

怎样才能没有异常——严格一致性？
即所有客户端看到相同的文件内容。
很难给出真正的答案，但这里有一些问题。

- 所有副本都应完成每次写入，或者不完成。
  也许试探性地写，直到所有人都承诺完成它？
  在所有人都同意执行之前不要公开写入！
- Primary 应检测重复的客户端写入请求。
- 如果主库崩溃，某些副本可能会丢失最后几个操作。
  新的主节点必须与所有副本通信才能找到所有最近的操作，
  并与辅助同步。
- 必须防止客户端读取过时的前二级数据；
  也许二级节点有租约，或者客户知道块版本
  并从协调员那里获得该版本的租约。
  您将在实验 2 和实验 3 中看到解决方案！

性能（图3）
大的读取总吞吐量
16 个客户端总计 94 MB/秒
或每个客户端 6 MB/秒
这样好吗？
一个磁盘的顺序吞吐量约为 30 MB/s

一张网卡大约10MB/s
接近饱和的交换机间链路 125 MB/秒
所以：每个客户端的性能并不大
但多客户端可扩展性很好
哪个更重要?
表 3 报告了生产 GFS 的速度为 500 MB/秒，这是一个很大的数字
写入不同的文件低于可能的最大值
作者归咎于他们的网络堆栈（但没有细节）
并发追加到单个文件
受存储最后一个块的服务器限制
15 年后很难解释，例如磁盘的速度有多快?

值得考虑的随机问题
怎样才能很好地支持小文件呢?
支持数十亿个文件需要什么?
GFS可以用作广域文件系统吗?
在不同城市都有复制品?
一个数据中心中的所有副本的容错能力不是很好!
GFS 需要多长时间才能从故障中恢复?
块服务器的?
协调员的?
GFS 处理慢速 chunkserver 的能力如何?

GFS工程师回顾性采访:
http://queue.acm.org/detail.cfm?id=1594206
文件数量是最大的问题
最终数字增长到表 2 中的 1000 倍!
很难装入协调器 RAM
GC 的所有文件/块的协调器扫描速度很慢
数千个客户端对协调器造成过多的 CPU 负载
应用程序的设计必须能够应对 GFS 语义
和限制
协调器故障转移最初完全手动，需要 10 分钟
BigTable 是解决多小文件问题的一种方法
Colossus 显然将协调器数据分片到许多协调器上

概括
性能、容错性、一致性的案例研究
专门针对 MapReduce 应用程序
好主意:
全局集群文件系统作为通用基础设施
将命名（协调器）与存储（块服务器）分离
并行吞吐量的分片
大文件/块以减少开销
主要到序列写入
租约以防止裂脑 chunkserver 主选
不太好:
单一协调器性能
RAM 和 CPU 耗尽
chunkservers 对于小文件不是很有效

缺乏自动故障转移到协调器副本
也许一致性太放松了

英文原文

6.824 2022 Lecture 3: GFS

The Google File System
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
SOSP 2003

Why are we reading this paper?
  distributed storage is a key abstraction
    what should the interface/semantics look like?
    how should it work internally?
  GFS paper touches on many themes of 6.824
    parallel performance, fault tolerance, replication, consistency
  good systems paper -- details from apps all the way to network
  successful real-world design
  influential: Hadoop's HDFS

Why is distributed storage hard?
  high performance -> shard data over many servers
  many servers -> constant faults
  fault tolerance -> replication
  replication -> potential inconsistencies
  better consistency -> low performance

What would we like for consistency?
  Ideal model: same behavior as a single server
  ideal server executes client operations one at a time
    even if multiple clients issued operations concurrently
  reads reflect previous writes
    even if server crashes and restarts
  all clients see the same data
  thus:
    suppose C1 and C2 write concurrently, and after the writes have
      completed, C3 and C4 read. what can they see?
    C1: Wx1
    C2: Wx2
    C3:   Rx?
    C4:   Rx?
    answer: either 1 or 2, but both have to see the same value.
  This is a "strong" consistency model.
  But a single server has poor fault-tolerance.

Replication for fault-tolerance makes strong consistency tricky.
  a simple but broken replication scheme:
    two replica servers, S1 and S2
    clients send writes to both, in parallel
    clients send reads to either
  in our example, C1's and C2's write messages could arrive in
    different orders at the two replicas

if C3 reads S1, it might see x=1
if C4 reads S2, it might see x=2
or what if S1 receives a write, but
the client crashes before sending the write to S2?
that's not strong consistency!
better consistency usually requires communication to
ensure the replicas stay in sync -- can be slow!
lots of tradeoffs possible between performance and consistency
we'll see one today

GFS

Context:
Many Google services needed a big fast unified storage system
Mapreduce, crawler, indexer, log storage/analysis
Shared among multiple applications e.g. crawl, index, analyze
Automatic "sharding" of each file over many servers/disks
For parallel performance with many clients e.g. MapReduce
For huge files too big for one server
Automatic recovery from failures
Just one data center per deployment
Just Google applications/users
Aimed at sequential access to huge files; read or append
I.e. not a low-latency DB for small items

What was new about this in 2003? How did they get an SOSP paper accepted?
Not the basic ideas of distribution, sharding, fault-tolerance.
Huge scale.
Used in industry, real-world experience.
Successful use of weak consistency.

Overall structure
clients (library, RPC -- but not visible as a UNIX FS)
coordinator tracks file names
chunkservers store 64 MB chunks
big files split into 64 MB chunks, on lots of chunkservers
big chunks -> low book-keeping overhead
each chunk replicated on 3 chunkservers
why 3 rather than 2?

Coordinator state
tables in RAM (for speed, must be smallish):
file name -> array of chunk handles (nv)
chunk handle -> version # (nv)
list of chunkservers (v)
primary (v)
lease time (v)
"nv" state also written to disk, in case crash+reboot

What are the steps when client C wants to read a file?

1. C sends filename and offset to coordinator (CO) (if not cached)
2. CO finds chunk handle for that offset

3. CO replies with list of chunkhandles + chunkservers
   only those with latest version
4. C caches handle + chunkserver list
5. C sends request to nearest chunkserver
   chunk handle, offset
6. chunk server reads from chunk file on disk, returns to client

Clients only ask coordinator about file names and lists of chunk handles
  clients cache name -> chunkhandle info
  coordinator does not handle data, so (hopefully) not heavily loaded

How does the coordinator know what chunkservers have a given chunk?

What are the steps when C wants to do a "record append"?
  paper's Figure 2

1. C asks CO about file's last chunk
2. CO tells C the primary and secondaries
3. C sends data to all (just temporary...), waits for all replies (?)
4. C tells P to append
5. P checks that lease hasn't expired, and chunk has space
6. P picks an offset (at end of chunk)
7. P writes its own chunk file (a Linux file)
8. P tells each secondary the offset, tells to append to chunk file
9. P waits for all secondaries to reply, or timeout
   secondary can reply "error" e.g. out of disk space
10. P tells C "ok" or "error"
11. C retries from start if error

What consistency guarantees does GFS provide to clients?
  Needs to be in a form that tells applications how to use GFS.

Here's a possibility:

  If the primary tells a client that a record append succeeded, then
  any reader that subsequently opens the file and scans it will see
  the appended record somewhere.

That allows lots of anomalies: different clients may see records in
different orders; they may see duplicated records; they may see record
from failed writes. GFS application must be prepared!

How can we think about how GFS fulfils the guarantee?
  Look at its handling of various failures:
    crash, crash+reboot, crash+replacement, message loss, partition.
  Ask how GFS ensures guarantee for each kind of failure.

- What if the client fails during record append sequence?
- What if the appending client has cached a stale (wrong) primary for a chunk?
- What if the reading client has cached a stale server list for a chunk?
- Could a coordinator crash+reboot cause it to forget about the file?
  Or forget what chunkservers hold the relevant chunk?
- Two clients do record append at exactly the same time.
  Will they overwrite each others' records?

- Suppose one secondary doesn't hear the append command from the primary.
  Due to a temporary network failure.
  What if reading client reads from that secondary?
- What if primary S1 is alive and serving client requests,
  but network between coordinator and S1 fails?
  "network partition"
  Will the coordinator pick a new primary?
  Will there now be two primaries?
  So that the append goes to one primary, and the read to the other?
    Thus breaking the consistency guarantee?
    "split brain"
- What if the primary crashes before sending append to all secondaries?
  Could a secondary that *didn't* see the append be chosen as the new primary?
- Chunkserver S4 with an old stale copy of chunk is offline.
  Primary and all live secondaries crash.
  S4 comes back to life (before primary and secondaries).
  Will coordinator choose S4 (with stale chunk) as primary?
  Better to have primary with ancient data, or no replicas at all?
- How does the coordinator set up a primary for a chunk?
  If client wants to write, but no primary, or primary lease expired and dead.
  Coordinator has been polling chunkservesr about what chunks/versions they have.
  a. if no chunkservers w/ latest version #, error
  b. pick primary P and secondaries from those w/ latest version #
  c. increment version #, write to disk
  d. tell P and secondaries who they are, and new version #
  e. replicas write new version # to disk
- What should a primary do if a secondary always fails writes?
  e.g. dead, or out of disk space, or disk has broken.
  Keep failing client requests?
  Or ask coordinator to declare a new set of servers and new version?
    The paper does not describe this process.
- If there's a partitioned primary serving client appends, and its
  lease expires, and the coordinator picks a new primary, will the new
  primary have the latest data as updated by partitioned primary?
- What if the coordinator fails altogether.
  Will the replacement know everything the dead coordinator knew?
  E.g. each chunk's version number? primary? lease expiry time?
- Who/what decides the coordinator is dead, and must be replaced?
  Could the coordinator replicas ping the coordinator, take over if no response?
- What happens if the entire building suffers a power failure?
  And then power is restored, and all servers reboot.
- Suppose the coordinator wants to create a new chunk replica.
  Maybe because too few replicas.
  Suppose it's the last chunk in the file, and being appended to.
  How does the new replica ensure it doesn't miss any appends?
    After all it is not yet one of the secondaries.
- Is there *any* circumstance in which GFS will break the guarantee?
  i.e. append succeeds, but subsequent readers don't see the record.
  All coordinator replicas permanently lose state (permanent disk failure).
    Could be worse: result will be "no answer", not "incorrect data".

"fail-stop"
All chunkservers holding the chunk permanently lose disk content.
again, fail-stop; not the worse possible outcome
CPU, RAM, network, or disk yields an incorrect value.
checksum catches some cases, but not all
Time is not properly synchronized, so leases don't work out.
So multiple primaries, maybe write goes to one, read to the other.

What would it take to have no anomalies -- strict consistency?
I.e. all clients see the same file content.
Too hard to give a real answer, but here are some issues.

- All replicas should complete each write, or none.
  Perhaps tentative writes until all promise to complete it?
  Don't expose writes until all have agreed to perform them!
- Primary should detect duplicate client write requests.
- If primary crashes, some replicas may be missing the last few ops.
  New primary must talk to all replicas to find all recent ops,
  and sync with secondaries.
- Clients must be prevented from reading from stale ex-secondaries;
  perhaps secondaries have leases, or clients know about chunk versions
  and get a lease on that version from coordinator.
  You'll see solutions in Labs 2 and 3!

Performance (Figure 3)
large aggregate throughput for read
94 MB/sec total for 16 clients
or 6 MB/second per client
is that good?
one disk sequential throughput was about 30 MB/s
one NIC was about 10 MB/s
Close to saturating inter-switch link's 125 MB/sec
So: per-client performance is not huge
but multi-client scalability is good
which is more important?
Table 3 reports 500 MB/sec for production GFS, which is a lot
writes to different files lower than possible maximum
authors blame their network stack (but no detail)
concurrent appends to single file
limited by the server that stores last chunk
hard to interpret after 15 years, e.g. how fast were the disks?

Random issues worth considering
What would it take to support small files well?
What would it take to support billions of files?
Could GFS be used as wide-area file system?
With replicas in different cities?
All replicas in one datacenter is not very fault tolerant!
How long does GFS take to recover from a failure?
Of a chunkserver?
Of the coordinator?
How well does GFS cope with slow chunkservers?

Retrospective interview with GFS engineer:
  http://queue.acm.org/detail.cfm?id=1594206
  file count was the biggest problem
    eventual numbers grew to 1000x those in Table 2 !
    hard to fit in coordinator RAM
    coordinator scanning of all files/chunks for GC is slow
  1000s of clients too much CPU load on coordinator
  applications had to be designed to cope with GFS semantics
    and limitations
  coordinator fail-over initially entirely manual, 10s of minutes
  BigTable is one answer to many-small-files problem
  and Colossus apparently shards coordinator data over many coordinators

Summary
  case study of performance, fault-tolerance, consistency
    specialized for MapReduce applications
  good ideas:
    global cluster file system as universal infrastructure
    separation of naming (coordinator) from storage (chunkserver)
    sharding for parallel throughput
    huge files/chunks to reduce overheads
    primary to sequence writes
    leases to prevent split-brain chunkserver primaries
  not so great:
    single coordinator performance
      ran out of RAM and CPU
    chunkservers not very efficient for small files
    lack of automatic fail-over to coordinator replica
    maybe consistency was too relaxed

LEC 4 Primary/Backup Replication

中文翻译

6.824 2022年第4讲：主/备复制

今天
  主/备份复制以实现容错
  VMware FT (2010) 的案例研究，该想法的极端版本

主题（仍然）是容错
  提供可用性
  尽管服务器和网络出现故障
  使用复制

复制可以处理哪些类型的故障?
  复制有利于单个副本的“故障停止”故障
    风扇停止工作，CPU 过热并自行关闭
    有人被复制品的电源线或网线绊倒
    软件注意到磁盘空间不足并停止
  复制可能无助于解决错误或操作员错误
    经常不会出现故障停止

可能是相关的（即某些输入导致所有副本崩溃）
地震或全市停电怎么样？
仅当副本在物理上分离时

两种主要的复制方法：
状态转移
Primary 执行服务
主节点通过网络将状态快照发送到存储系统
失败时：
找到一台备用机器（或者可能有一个专用的备份在等待）
加载软件，加载保存的状态，执行
复制状态机
客户端将操作发送到主节点，
主序列并发送到备份
所有副本执行所有操作
如果启动状态相同，
相同的操作，
相同的订单，
确定性的，
然后相同的最终状态。

状态转移在概念上很简单
但状态可能很大，通过网络传输很慢

复制状态机通常会产生较少的网络流量
与国家相比，运营通常规模较小
但要正确执行很复杂
VM-FT 使用复制状态机，实验室 2/3/4 也是如此

大问题：
状态和操作是什么？
主库是否必须等待备份？
备份如何决定接管？
切换时是否可见异常情况？
如何使替换备份加快速度并恢复复制？

我们希望副本在什么级别上是相同的？
应用程序状态，例如数据库的表？
GFS 就是这样工作的
高效的; 主节点仅将高级操作发送到备份节点
应用程序必须了解容错能力
机器级别，例如寄存器和 RAM 内容？
可能允许我们复制任何现有的应用程序而无需修改！
需要转发机器事件（中断、网络数据包等）
需要“机器”修改来发送/接收事件流...

今天的论文（VMware FT）复制了机器级状态
透明：可以运行任何现有的操作系统和服务器软件！
对客户端来说就像一台服务器

概述
[图：应用程序、操作系统、底层 VM-FT、磁盘服务器、网络、客户端]
字：
hypervisor == 监视器 == VMM（虚拟机监视器）
O/S+app是虚拟机内部运行的“来宾”
两台物理机，主机和备份机

基本思想：
主要和备份最初以相同的内存和寄存器开始
包括相同的软件（操作系统和应用程序）
大多数指令在主设备和备份设备上执行相同
例如 ADD 指令
所以大多数时候，不需要做任何工作就可以使它们保持相同！

主服务器何时必须向备份服务器发送信息？
任何时候发生一些事情都可能导致他们的执行出现分歧。
任何不是执行指令的确定性结果的事情。

FT 必须消除哪些分歧来源？
不是状态函数的指令，例如读取当前时间。
来自外部世界的输入——网络数据包和磁盘读取。
这些显示为 DMA 数据加中断。
中断的计时。
但不是多核竞赛，因为仅限单处理器。

为什么分歧会是一场灾难？
备份上的 b/c 状态与主上的状态不同，
如果主节点失败，客户就会看到不一致的情况。
示例：6.824作业提交服务器
强制实验室午夜截止日期。
硬件计时器在午夜关闭。
让我们用一个“损坏的” FT 来复制提交服务器。
在小学，我的作业数据包中断恰好在计时器关闭之前到达。
Primary会告诉我我的家庭作业获得满分。
在备份时，我的作业紧接着到达，所以备份认为已经晚了。
主要和备份现在具有不同的状态。
目前，没有人注意到，因为主要答复了所有请求。
然后主数据库失败，备份接管，课程工作人员看到
备份的状态，显示我提交晚了！
所以：备份必须看到相同的事件，
以同样的顺序，
在指令流中的相同点。

日志记录通道
主数据库通过网络将所有事件发送到备份数据库
“日志通道”，承载日志条目
中断、传入的网络数据包、从共享磁盘读取的数据
FT 从日志条目中提供备份输入（中断和c）
FT抑制备份的网络输出
如果其中一方无法通过网络与另一方通话
“上线”并提供独家服务
如果主数据库上线，它会停止向备份数据库发送日志条目

每个日志条目：指令号、类型、数据。

FT对定时器中断的处理
目标：主要和备份应该准确地看到中断
指令流中的同一点
基本的：
FT 场定时器中断
FT从CPU读取指令号
FT 在记录通道上发送“指令 # X 处的定时器中断”
FT 将中断传递给主设备并恢复它
（依靠CPU支持将中断直接发送给FT软件）
备份：
忽略它自己的定时器硬件
FT 在备份到达指令 # X 之前*看到日志条目
FT 告诉 CPU 将控制权转移到指令 # X 处的 FT
FT 模仿备份访客看到的计时器中断
（依赖CPU支持在第X条指令后跳转到FT）

FT对网络数据包到达（输入）的处理
基本的：
FT配置NIC将数据包数据写入FT的私有“反弹缓冲区”
在某个时刻，数据包到达，NIC 执行 DMA，然后中断
FT获得中断，从CPU读取指令#
FT 暂停主要
FT 将反弹缓冲区复制到主内存中
FT 在主设备中模拟 NIC 中断
FT将包数据和指令#发送给备份
备份：
FT 从日志流中获取数据和指令#
FT 告诉 CPU 在指令 # X 处中断（到 FT）
FT将数据复制到guest内存，备份中模拟NIC中断

为什么要使用反弹缓冲区？
我们希望数据在完全相同的时间点出现在内存中
主要和备份的执行。
因此，如果他们在中断之前读取数据包内存，他们会看到同样的事情。
否则他们可能会出现分歧。

FT VMM 模拟本地磁盘接口
但实际存储位于网络服务器上——“共享磁盘”
所有文件/目录都在共享存储中；没有本地磁盘
仅与共享磁盘进行主对话
主节点将其读取的块转发到备份节点
备份的 FT 忽略备份应用程序的写入，服务于主数据的读取
共享磁盘可以更快地创建新备份
不必复制主磁盘

备份必须滞后一个日志条目
假设主设备在指令 # X 处获得中断
如果备份已经执行过了X，那就太晚了！
因此，除非至少有一个日志条目正在等待，否则备份 FT 无法执行

然后它只执行该日志条目中的指令 #
并在恢复之前等待下一个日志条目

示例：非确定性指令
即使主/备份具有相同的状态，某些指令也会产生不同的结果
例如读取当前时间或处理器序列#
基本的：
如果主设备执行这样的指令，FT 将 CPU 设置为中断
FT执行指令并记录结果
发送结果和指令#到备份
备份：
FT 读取日志条目，在指令 # 处设置中断
然后 FT 提供主设备获得的值，不执行指令

输出（发送网络数据包、写入共享磁盘）怎么样?
主备都执行输出指令
Primary 的 FT 实际上做输出
Backup的FT丢弃输出

输出示例：数据库服务器
客户端可以发送“增量”请求
DB 增加存储值，并回复新值
所以：
[图表]
假设服务器的值从 10 开始
网络将客户端请求传递给主节点上的 FT
主节点的 FT 通过日志通道发送到备份节点
FT 将请求数据包传送到主备机
Primary执行，将值设置为11，发送“11”回复，FT真正发送回复
backup 执行，将值设置为 11，发送“11”回复，FT 丢弃
正如预期的那样，客户端得到一个“11”响应

可是等等：
假设主发送回复然后崩溃
所以客户端得到“11”回复
并且日志记录通道丢弃带有客户端请求的日志条目
Primary 已失效，因此不会重新发送
备份上线
但它在内存中的值是“10”!
现在客户端发送另一个增量请求
它将再次得到“11”，而不是“12”
哎呀

解决方案：输出规则（第 2.2 节）
在主发送输出之前（例如发送到客户端或共享磁盘），
必须等待备份确认所有先前的日志条目

再次，使用输出规则：
[图表]
基本的：
接收客户端“增量”请求
在日志通道上发送客户端请求
即将发送“11”回复给客户

首先等待备份确认先前的日志条目
然后发送“11”回复给客户端
假设主服务器在此序列中的某个时刻崩溃
如果在主设备收到备份设备的确认之前
也许备份没有看到客户端的请求，并且没有增加
但primary也不会回复
如果主设备收到备份设备的确认后
那么客户端可能会看到“11”回复
但备份保证已收到带有客户端请求的日志条目
所以备份将增加到 11

输出规则很重要
在最强一致的复制系统中以某种形式出现
通常称为“同步复制”b/c 主数据库必须等待
严重制约性能
一个针对特定应用的聪明才智的领域
例如。主节点在回复只读操作之前可能不需要等待
FT没有应用级知识，必须保守

问：如果主服务器在收到备份确认后立即崩溃怎么办？
但在初级发出输出之前？
这是否意味着永远不会生成输出？

答：备份在以下指令之前或之后生效
将回复数据包发送给客户端。
如果之前，它将发送回复数据包。
如果之后，FT 将丢弃该数据包。但备份的
TCP 会认为它发送了它，并且会期待一个 TCP ACK 数据包，并且
如果没有收到ACK就会重新发送。

问：但是如果主控制器在发出输出后崩溃了怎么办？
备份会再次发出输出吗？

答： 可能吧！
对于 TCP 来说没问题，因为接收方会忽略重复的序列号。
可以写入共享磁盘，
因为备份会将相同的数据写入相同的块#。

在复制系统中，切换时的重复输出很常见
客户端需要保持足够的状态来忽略重复项
或者进行设计以确保重复无害

问：FT 是否能够应对网络分区——它会遭受脑裂吗？
例如，如果主设备和备份设备都认为对方已关闭。
他们俩都会上线吗？

答：共享磁盘服务器打破了平局。
磁盘服务器支持原子测试和设置。
如果主数据库或备份数据库认为其他数据库已失效，则尝试进行测试和设置。
如果只有一个活着，它将赢得测试并设置并上线。
如果双方都尝试，其中一方会失败并停止。

共享磁盘服务器需要可靠！
 如果磁盘服务器宕机，服务也会宕机
 他们想要一个昂贵的容错磁盘服务器

问：为什么他们不支持多核？

性能（表1）
 FT/非 FT：令人印象深刻！
  慢一点
 记录带宽
  直接反映磁盘读取速率+网络输入速率
  最大 18 Mbit/s
 对我来说，记录通道流量似乎很低
  应用程序可以以每秒 100 兆位的速度读取磁盘
  因此他们的应用程序可能不会占用大量磁盘空间

FT什么时候会有吸引力？
 关键但低强度的服务，例如名称服务器。
 软件不方便修改的服务。

高吞吐量服务的复制怎么样？
 人们将应用程序级复制状态机用于数据库等。
  状态只是DB，而不是内存+磁盘的全部。
  这些事件是 DB 命令（put 或 get），而不是数据包和中断。
  可以有例如只读操作的快捷方式。
 结果：日志流量减少，输出规则暂停减少。
 GFS 使用应用程序级复制，实验室 2 和 c 也是如此

概括：
 主备复制
  VM-FT：干净的示例
 如何应对分区无单点故障？
  下一讲
 如何获得更好的性能？
  应用程序级复制状态机

VMware KB (#1013428) 讨论了多 CPU 支持。VM-FT 可能已切换
从复制状态机方法到状态转移方法，但是
不清楚这是否属实。

http://www.wooditwork.com/2014/08/26/whats-new-vsphere-6-0-fault-tolerance/

http://www-mount.ece.umn.edu/~jjyi/MoBS/2007/program/01C-Xu.pdf

http://web.eecs.umich.edu/~nsatish/abstracts/ASPLOS-10-Respec.html

英文原文

6.824 2022 Lecture 4: Primary/Backup Replication

Today
 Primary/Backup Replication for Fault Tolerance
 Case study of VMware FT (2010), an extreme version of the idea

The topic is (still) fault tolerance
- to provide availability
- despite server and network failures
- using replication

What kinds of failures can replication deal with?
- Replication is good for "fail-stop" failure of a single replica
  - fan stops working, CPU overheats and shuts itself down
  - someone trips over replica's power cord or network cable
  - software notices it is out of disk space and stops
- Replication may not help with bugs or operator error
  - Often not fail-stop
  - May be correlated (i.e. some input causes all replicas to crash)
- How about earthquake or city-wide power failure?
  - Only if replicas are physically separated

Two main replication approaches:
- State transfer
  - Primary executes the service
  - Primary sends state snapshots over network to a storage system
  - On failure:
    - Find a spare machine (or maybe there's a dedicated backup waiting)
    - Load software, load saved state, execute
- Replicated state machine
  - Clients send operations to primary,
    - primary sequences and sends to backups
  - All replicas execute all operations
  - If same start state,
    - same operations,
    - same order,
    - deterministic,
    - then same end state.

State transfer is conceptually simple
- But state may be large, slow to transfer over network

Replicated state machine often generates less network traffic
- Operations are often small compared to state
- But complex to get right
- VM-FT uses replicated state machine, as do Labs 2/3/4

Big Questions:
- What are the state and operations?
- Does primary have to wait for backup?
- How does backup decide to take over?
- Are anomalies visible at cut-over?
- How to bring a replacement backup up to speed and resume replication?

At what level do we want replicas to be identical?
- Application state, e.g. a database's tables?
  - GFS works this way
  - Efficient; primary only sends high-level operations to backup
  - Application must understand fault tolerance

Machine level, e.g. registers and RAM content?
  might allow us to replicate any existing application w/o modification!
  requires forwarding of machine events (interrupts, network packets, &c)
  requires "machine" modifications to send/recv event stream...

Today's paper (VMware FT) replicates machine-level state
  Transparent: can run any existing O/S and server software!
  Appears like a single server to clients

Overview
  [diagram: app, O/S, VM-FT underneath, disk server, network, clients]
  words:
    hypervisor == monitor == VMM (virtual machine monitor)
    O/S+app is the "guest" running inside a virtual machine
  two physical machines, primary and backup

The basic idea:
  Primary and backup initially start with identical memory and registers
    Including identical software (O/S and app)
  Most instructions execute identically on primary and backup
    e.g. an ADD instruction
  So most of the time, no work is required to cause them to remain identical!

When does the primary have to send information to the backup?
  Any time something happens that might cause their executions to diverge.
  Anything that's not a deterministic consequence of executing instructions.

What sources of divergence must FT eliminate?
  Instructions that aren't functions of state, such as reading current time.
  Inputs from external world -- network packets and disk reads.
    These appear as DMA'd data plus an interrupt.
  Timing of interrupts.
  But not multi-core races, since uniprocessor only.

Why would divergence be a disaster?
  b/c state on backup would differ from state on primary,
    and if primary then failed, clients would see inconsistency.
  Example: the 6.824 homework submission server
    Enforces midnight deadline for labs.
    A hardware timer goes off at midnight.
    Let's replicate submission server with a *broken* FT.
    On primary, my homework packet interrupt arrives just *before* timer goes off.
      Primary will tell me I get full credit for homework.
    On backup, my homework arrives just after, so backup thinks it is late.
      Primary and backup now have divergent state.
      For now, no-one notices, since the primary answers all requests.
    Then primary fails, backup takes over, and course staff see
      backup's state, which says I submitted late!
  So: backup must see same events,
    in same order,
    at same points in instruction stream.

The logging channel
  primary sends all events to backup over network
    "logging channel", carrying log entries
    interrupts, incoming network packets, data read from shared disk
  FT provides backup's input (interrupts &c) from log entries
  FT suppresses backup's network output
  if either stops being able to talk to the other over the network
    "goes live" and provides sole service
    if primary goes live, it stops sending log entries to the backup

Each log entry: instruction #, type, data.

FT's handling of timer interrupts
  Goal: primary and backup should see interrupt at exactly
      the same point in the instruction stream
  Primary:
    FT fields the timer interrupt
    FT reads instruction number from CPU
    FT sends "timer interrupt at instruction # X" on logging channel
    FT delivers interrupt to primary, and resumes it
    (relies on CPU support to direct interrupts to FT software)
  Backup:
    ignores its own timer hardware
    FT sees log entry *before* backup gets to instruction # X
    FT tells CPU to transfer control to FT at instruction # X
    FT mimics a timer interrupt that backup guest sees
    (relies on CPU support to jump to FT after the X'th instruction)

FT's handling of network packet arrival (input)
  Primary:
    FT configures NIC to write packet data into FT's private "bounce buffer"
    At some point a packet arrives, NIC does DMA, then interrupts
    FT gets the interrupt, reads instruction # from CPU
    FT pauses the primary
    FT copies the bounce buffer into the primary's memory
    FT simulates a NIC interrupt in primary
    FT sends the packet data and the instruction # to the backup
  Backup:
    FT gets data and instruction # from log stream
    FT tells CPU to interrupt (to FT) at instruction # X
    FT copies the data to guest memory, simulates NIC interrupt in backup

Why the bounce buffer?
  We want the data to appear in memory at exactly the same point in
    execution of the primary and backup.
  So they see the same thing if they read packet memory before interrupt.
  Otherwise they may diverge.

FT VMM emulates a local disk interface
  but actual storage is on a network server -- the "shared disk"
  all files/directories are in the shared storage; no local disks
  only primary talks to the shared disk

primary forwards blocks it reads to the backup
backup's FT ignores backup app's writes, serves reads from primary's data
shared disk makes creating a new backup much faster
don't have to copy primary's disk

The backup must lag by one log entry
Suppose primary gets an interrupt at instruction # X
If backup has already executed past X, it is too late!
So backup FT can't execute unless at least one log entry is waiting
Then it executes just to the instruction # in that log entry
And waits for the next log entry before resuming

Example: non-deterministic instructions
some instructions yield different results even if primary/backup have same state
e.g. reading the current time or processor serial #
Primary:
FT sets up the CPU to interrupt if primary executes such an instruction
FT executes the instruction and records the result
sends result and instruction # to backup
Backup:
FT reads log entry, sets up for interrupt at instruction #
FT then supplies value that the primary got, does not execute instruction

What about output (sending network packets, writing the shared disk)?
Primary and backup both execute instructions for output
Primary's FT actually does the output
Backup's FT discards the output

Output example: DB server
clients can send "increment" request
DB increments stored value, replies with new value
so:
[diagram]
suppose the server's value starts out at 10
network delivers client request to FT on primary
primary's FT sends on logging channel to backup
FTs deliver request packet to primary and backup
primary executes, sets value to 11, sends "11" reply, FT really sends reply
backup executes, sets value to 11, sends "11" reply, and FT discards
the client gets one "11" response, as expected

But wait:
suppose primary sends reply and then crashes
so client gets the "11" reply
AND the logging channel discards the log entry w/ client request
primary is dead, so it won't re-send
backup goes live
but it has value "10" in its memory!
now a client sends another increment request
it will get "11" again, not "12"
oops

Solution: the Output Rule (Section 2.2)
  before primary sends output (e.g. to a client, or shared disk),
  must wait for backup to acknowledge all previous log entries

Again, with output rule:
  [diagram]
  primary:
    receives client "increment" request
    sends client request on logging channel
    about to send "11" reply to client
    first waits for backup to acknowledge previous log entry
    then sends "11" reply to client
  suppose the primary crashes at some point in this sequence
  if before primary receives acknowledgement from backup
    maybe backup didn't see client's request, and didn't increment
    but also primary won't have replied
  if after primary receives acknowledgement from backup
    then client may see "11" reply
    but backup guaranteed to have received log entry w/ client's request
    so backup will increment to 11

The Output Rule is a big deal
  Occurs in some form in most strongly consistent replication systems
    Often called "synchronous replication" b/c primary must wait
  A serious constraint on performance
  An area for application-specific cleverness
    Eg. maybe no need for primary to wait before replying to read-only operation
  FT has no application-level knowledge, must be conservative

Q: What if the primary crashes just after getting acknowledgement from backup,
   but before the primary emits the output?
   Does this mean that the output won't ever be generated?

A: The backup goes live either before or after the instruction that
     sends the reply packet to the client.
   If before, it will send the reply packet.
   If after, FT will have discarded the packet. But the backup's
     TCP will think it sent it, and will expect a TCP ACK packet, and
     will re-send if it doesn't get the ACK.

Q: But what if the primary crashed *after* emitting the output?
   Will the backup emit the output a *second* time?

A: It might!
   OK for TCP, since receivers ignore duplicate sequence numbers.
   OK for writes to shared disk,
     since backup will write same data to same block #.

Duplicate output at cut-over is pretty common in replication systems
  Clients need to keep enough state to ignore duplicates
  Or be designed so that duplicates are harmless

Q: Does FT cope with network partition -- could it suffer from split brain?
  E.g. if primary and backup both think the other is down.
  Will they both go live?

A: The shared disk server breaks the tie.
  Disk server supports atomic test-and-set.
  If primary or backup thinks other is dead, attempts test-and-set.
  If only one is alive, it will win test-and-set and go live.
  If both try, one will lose, and halt.

The shared disk server needs to be reliable!
 If disk server is down, service is down
 They have in mind an expensive fault-tolerant disk server

Q: Why don't they support multi-core?

Performance (table 1)
 FT/Non-FT: impressive!
  little slow down
 Logging bandwidth
  Directly reflects disk read rate + network input rate
  18 Mbit/s is the max
 The logging channel traffic numbers seem low to me
  Applications can read a disk at a few 100 megabits/second
  So their applications may not be very disk-intensive

When might FT be attractive?
 Critical but low-intensity services, e.g. name server.
 Services whose software is not convenient to modify.

What about replication for high-throughput services?
 People use application-level replicated state machines for e.g. databases.
  The state is just the DB, not all of memory+disk.
  The events are DB commands (put or get), not packets and interrupts.
  Can have short-cuts for e.g. read-only operations.
 Result: less logging traffic, fewer Output Rule pauses.
 GFS use application-level replication, as do Lab 2 &c

Summary:
 Primary-backup replication
  VM-FT: clean example
 How to cope with partition without single point of failure?
  Next lecture
 How to get better performance?
  Application-level replicated state machines

VMware KB (#1013428) talks about multi-CPU support. VM-FT may have switched from a replicated state machine approach to the state transfer approach, but unclear whether that is true or not.

http://www.wooditwork.com/2014/08/26/whats-new-vsphere-6-0-fault-tolerance/

http://www-mount.ece.umn.edu/~jjyi/MoBS/2007/program/01C-Xu.pdf

http://web.eecs.umich.edu/~nsatish/abstracts/ASPLOS-10-Respec.html

LEC 5 Fault Tolerance：Raft（1）

中文翻译

6.824 2022年第五讲：筏（1）

本次讲座
今天：Raft 选举和日志处理（实验 2A、2B）
下一页：Raft 持久性、客户端行为、快照（实验室 2C、2D）

我们见过的容错系统中的一种模式

- MR复制计算但依赖于单个master来组织
- GFS复制数据但依赖master来选择primary
- VMware FT 复制服务，但依靠测试和设置来选择主要服务
  全部依靠单一实体做出关键决策
  很好：单个实体的决策避免了脑裂
  很容易让一个实体始终与自己达成一致

我们如何制作容错测试和设置服务？
我们需要复制
两台服务器 S1 和 S2 怎么样
如果两者均已启动，则 S1 负责，将决策转发给 S2
如果 S2 发现 S1 已关闭，S2 将接管作为唯一的测试和设置服务器
会出什么问题吗？
网络分区！脑裂！

问题：计算机无法区分“服务器崩溃”和“网络损坏”
症状相同：通过网络查询没有响应
这个困难在很长一段时间内似乎无法克服
似乎需要外部代理（人）来决定何时切换服务器
我们更喜欢自动化方案！

应对分区的重要见解：多数投票
服务器数量为奇数，例如 3
做任何事情都需要得到多数人的同意——三分之二
如果没有多数，则等待
为什么多数有助于避免脑裂？
  最多一个分区可以拥有多数
  打破了我们在只有两台服务器的情况下看到的对称性
注意：大多数是在所有服务器中，而不仅仅是在活动服务器中
注：获得多数后继续
  不要再等了，因为他们可能已经死了
更一般地说，2f+1 可以容忍 f 个失败的服务器
  因为剩余的 f+1 是 2f+1 的大多数
  如果超过 f 失败（或无法联系），则没有进展
通常称为“法定人数”系统

多数的一个关键属性是任意两个相交
路口的服务器可以传达有关先前决策的信息
例如，另一位 Raft 领导人已经当选为本任期

1990 年左右发明了两种分区容忍复制方案，
Paxos 和视图标记复制
称为“共识”或“协议”协议
在过去 15 年里，这项技术在现实世界中得到了广泛应用
筏纸是对现代技术的很好的介绍

*** 主题：Raft 概述

以 Raft 状态机复制——实验 3 为例：
[图：客户端，3个副本，k/v层+状态，raft层+日志]
Raft 是每个副本中包含的库

一个客户端命令的时间图
[C、L、F1、F2]
客户端向领导者中的 k/v 层发送 Put/Get“命令”
Leader的Raft层添加命令到日志
领导者向追随者发送 AppendEntries RPC
关注者添加命令到日志
领导者等待绝大多数人（包括它自己）的回复
如果大多数人将其放入日志中，则条目被“提交”
承诺意味着即使失败也不会被遗忘
大多数 -> 将由下一个领导者的投票请求看到
Leader执行命令，回复client
领导者“piggybacks”在下一个 AppendEntries 中提交信息
一旦领导者表示已承诺，追随者就会执行条目

为什么是日志?
该服务保留状态机状态，例如键/值数据库
日志是相同信息的替代表示!
为什么两者都有?
日志对命令进行排序
帮助副本就单个执行顺序达成一致
帮助领导者确保追随者拥有相同的日志
日志存储暂定命令直到提交
日志存储命令，以防领导者必须重新发送给追随者
日志持久存储命令，以便在重新启动后重播

服务器的日志是否彼此完全相同?
否：某些副本可能会滞后
否：我们会看到他们可以暂时有不同的条目
好消息：
它们最终会收敛到相同
提交机制确保服务器只执行稳定的条目

实验2 Raft接口
rf.Start(命令)（索引、术语、isleader）
Lab 3 k/v 服务器的 Put()/Get() RPC 处理程序调用 Start()
Start() 仅对领导者有意义
在新的日志条目上启动 Raft 协议
追加到领导日志
领导者发出 AppendEntries RPC
Start() 返回不等待 RPC 回复
k/v层的Put()/Get()必须等待commit，在applyCh上

如果服务器在提交之前失去领导权，协议可能会失败
那么命令可能会丢失，客户端必须重新发送
isleader: false 如果该服务器不是领导者，客户端应该尝试另一个
term：currentTerm，帮助调用者检测领导者是否后来被降级
索引：要查看的日志条目以查看命令是否已提交
ApplyMsg，带有索引和命令
每个对等方在 applyCh 上为每个提交的条目发送一条 ApplyMsg
按日志顺序
每个对等点的本地服务代码执行，更新本地副本状态
领导者向等待的客户端发送 RPC 回复

Raft 的设计有两个主要部分：
选举新领导人
尽管出现故障，仍确保日志相同

*** 主题：领导者选举（实验室 2A）

为什么是领导者？
确保所有副本以相同的顺序执行相同的命令
（有些设计，例如 Paxos，没有领导者）

Raft 对领导者的序列进行编号
新领导->新任期
一个任期最多有一个领导者；可能没有领导者
编号有助于服务器跟随最新的领导者，而不是被取代的领导者

Raft Peer何时开始领导者选举？
当它没有收到现任领导人关于“选举超时”的消息时
增加本地 currentTerm，尝试收集选票
注意：这可能导致不必要的选举；这很慢但很安全
注：老领导可能还活着并认为是领导

如何保证一个任期内最多有一位领导者？
（图2 RequestVote RPC和服务器规则）
领导者必须获得大多数服务器的“赞成”票
每个服务器每个学期只能投一票
如果是候选人，则为自己投票
如果不是候选人，则投票给第一个提出要求的候选人（在图 2 规则内）
对于给定的任期，至多一台服务器可以获得多数票
-> 即使网络分区最多也只有一位领导者
-> 即使某些服务器失败，选举也能成功
注意：同样，大多数是在所有服务器中（不仅仅是实时服务器）

服务器如何了解新当选的领导者？
领导者发出 AppendEntries 心跳
与新的更高的术语编号
只有领导者发送AppendEntries
每届只有一名领导人
因此，如果您看到带有术语 T 的 AppendEntries，您就知道 T 的领导者是谁
心跳抑制任何新的选举
领导者必须比选举超时更频繁地发送心跳

选举可能因以下两个原因而失败：

- 少于大多数服务器可访问
  *同时候选人平分选票，没有人获得多数票

如果选举不成功怎么办?
没有心跳 -> 再次超时 -> 新任期的新选举
较高任期优先，较旧任期候选人退出

如果没有特别注意，选举往往会因分裂选票而失败
所有选举计时器可能会在大约同一时间响起
每个候选人都为自己投票
所以没有人会投票给其他人!
所以每个人都会得到一票，没有人会获得多数票

Raft 如何避免分裂选票?
每个服务器为其选举超时时间添加一些随机性
[服务器超时到期时间图]
随机性打破了服务器之间的对称性
人们会选择最低的随机延迟
希望在下次超时到期之前有足够的时间进行选择
其他人将看到新领导者的 AppendEntries 心跳并且
不成为候选人
随机延迟是网络协议中的常见模式

如何选择选举超时时间?

- 至少几个心跳间隔（以防网络丢失心跳）
  避免不必要的选举，浪费时间
  *随机部分足够长，足以让一个候选人在下一次开始之前成功
- 足够短，可以对失败做出快速反应，避免长时间停顿
- 足够短，可以在测试人员感到不安之前进行几次重试
  测试人员要求选举在 5 秒或更短时间内完成

如果旧领导者不知道新领导者当选怎么办?
也许老领导没有看到选举信息
也许旧的领导者处于少数网络分区中
新的领导者意味着大多数服务器已经增加了 currentTerm
任何一位旧领导人都会在 AppendEntries 回复中看到新术语并下台
否则老领导将无法得到大多数回复
所以旧的领导者不会提交或执行任何新的日志条目
因此没有裂脑
但少数人可能接受旧服务器的 AppendEntries
因此日志可能会在旧术语结束时出现分歧

Raft 与 Paxos：https://dl.acm.org/doi/10.1145/3293611.3331595

英文原文

6.824 2022 Lecture 5: Raft (1)

this lecture
today: Raft elections and log handling (Lab 2A, 2B)
next: Raft persistence, client behavior, snapshots (Lab 2C, 2D)

a pattern in the fault-tolerant systems we've seen

- MR replicates computation but relies on a single master to organize
- GFS replicates data but relies on the master to pick primaries
- VMware FT replicates service but relies on test-and-set to pick primary
  all rely on a single entity to make critical decisions
  nice: decisions by a single entity avoid split brain
  it's easy to make a single entity always agree with itself

how can we make e.g. a fault-tolerant test-and-set service?
  we need replication
  how about two servers, S1 and S2
  if both are up, S1 is in charge, forwards decisions to S2
  if S2 sees S1 is down, S2 takes over as sole test-and-set server
  what could go wrong?
  network partition! split brain!

the problem: computers cannot distinguish "server crashed" from "network broken"
  the symptom is the same: no response to a query over the network
  this difficulty seemed insurmountable for a long time
  seemed to require outside agent (a human) to decide when to switch servers
  we'd prefer an automated scheme!

the big insight for coping w/ partition: majority vote
  have an odd number of servers, e.g. 3
  agreement from a majority is required to do anything -- 2 out of 3
  if no majority, wait
  why does majority help avoid split brain?
    at most one partition can have a majority
    breaks the symmetry we saw with just two servers
  note: majority is out of all servers, not just out of live ones
  note: proceed after acquiring majority
    don't wait for more since they may be dead
  more generally 2f+1 can tolerate f failed servers
    since the remaining f+1 is a majority of 2f+1
    if more than f fail (or can't be contacted), no progress
  often called "quorum" systems

a key property of majorities is that any two intersect
  servers in the intersection can convey information about previous decisions
  e.g. another Raft leader has already been elected for this term

Two partition-tolerant replication schemes were invented around 1990,
  Paxos and View-Stamped Replication
  called "consensus" or "agreement" protocols
  in the last 15 years this technology has seen a lot of real-world use
  the Raft paper is a good introduction to modern techniques

*** topic: Raft overview

state machine replication with Raft -- Lab 3 as example:
  [diagram: clients, 3 replicas, k/v layer + state, raft layer + logs]
  Raft is a library included in each replica

time diagram of one client command
  [C, L, F1, F2]
  client sends Put/Get "command" to k/v layer in leader
  leader's Raft layer adds command to log
  leader sends AppendEntries RPCs to followers
  followers add command to log
  leader waits for replies from a bare majority (including itself)
  entry is "committed" if a majority put it in their logs
    committed means won't be forgotten even if failures
    majority -> will be seen by the next leader's vote requests
  leader executes command, replies to client
  leader "piggybacks" commit info in next AppendEntries
  followers execute entry once leader says it's committed

why the logs?
  the service keeps the state machine state, e.g. key/value DB
    the log is an alternate representation of the same information!
    why both?
  the log orders the commands
    to help replicas agree on a single execution order
    to help the leader ensure followers have identical logs
  the log stores tentative commands until committed
  the log stores commands in case leader must re-send to followers
  the log stores commands persistently for replay after reboot

are the servers' logs exact replicas of each other?
  no: some replicas may lag
  no: we'll see that they can temporarily have different entries
  the good news:
    they'll eventually converge to be identical
    the commit mechanism ensures servers only execute stable entries

lab 2 Raft interface
  rf.Start(command) (index, term, isleader)
    Lab 3 k/v server's Put()/Get() RPC handlers call Start()
    Start() only makes sense on the leader
    starts Raft agreement on a new log entry
      append to leader's log
      leader sends out AppendEntries RPCs
      Start() returns w/o waiting for RPC replies
      k/v layer's Put()/Get() must wait for commit, on applyCh
    agreement might fail if server loses leadership before committing
      then the command is likely lost, client must re-send
    isleader: false if this server isn't the leader, client should try another
    term: currentTerm, to help caller detect if leader is later demoted
    index: log entry to watch to see if the command was committed
  ApplyMsg, with Index and Command
    each peer sends an ApplyMsg on applyCh for each committed entry
    in log order
    each peer's local service code executes, updates local replica state
    leader sends RPC reply to waiting client

there are two main parts to Raft's design:
  electing a new leader
  ensuring identical logs despite failures

*** topic: leader election (Lab 2A)

why a leader?
  ensures all replicas execute the same commands, in the same order
  (some designs, e.g. Paxos, don't have a leader)

Raft numbers the sequence of leaders
  new leader -> new term
  a term has at most one leader; might have no leader
  the numbering helps servers follow latest leader, not superseded leader

when does a Raft peer start a leader election?
  when it doesn't hear from current leader for an "election timeout"
  increments local currentTerm, tries to collect votes
  note: this can lead to un-needed elections; that's slow but safe
  note: old leader may still be alive and think it is the leader

how to ensure at most one leader in a term?
  (Figure 2 RequestVote RPC and Rules for Servers)
  leader must get "yes" votes from a majority of servers
  each server can cast only one vote per term
    if candidate, votes for itself
    if not a candidate, votes for first that asks (within Figure 2 rules)
  at most one server can get majority of votes for a given term
    -> at most one leader even if network partition
    -> election can succeed even if some servers have failed
  note: again, majority is out of all servers (not just the live servers)

how does a server learn about a newly elected leader?
  the leader sends out AppendEntries heart-beats
    with the new higher term number
  only the leader sends AppendEntries
    only one leader per term
    so if you see AppendEntries with term T, you know who the leader for T is
  the heart-beats suppress any new election
    leader must send heart-beats more often than the election timeout

an election may not succeed for two reasons:

- less than a majority of servers are reachable
- simultaneous candidates split the vote, none gets majority

what happens if an election doesn't succeed?
  no heartbeats -> another timeout -> a new election for a new term
  higher term takes precedence, candidates for older terms quit

without special care, elections will often fail due to split vote
  all election timers likely to go off at around the same time
  every candidate votes for itself
  so no-one will vote for anyone else!
  so everyone will get exactly one vote, no-one will have a majority

how does Raft avoid split votes?
  each server adds some randomness to its election timeout period
  [diagram of times at which servers' timeouts expire]
  randomness breaks symmetry among the servers
    one will choose lowest random delay
  hopefully enough time to elect before next timeout expires
  others will see new leader's AppendEntries heartbeats and
    not become candidates
  randomized delays are a common pattern in network protocols

how to choose the election timeout?

- at least a few heartbeat intervals (in case network drops a heartbeat)
  to avoid needless elections, which waste time
- random part long enough to let one candidate succeed before next starts
- short enough to react quickly to failure, avoid long pauses
- short enough to allow a few re-tries before tester gets upset
  tester requires election to complete in 5 seconds or less

what if old leader isn't aware a new leader is elected?
  perhaps old leader didn't see election messages
  perhaps old leader is in a minority network partition
  new leader means a majority of servers have incremented currentTerm
  either old leader will see new term in a AppendEntries reply and step down
  or old leader won't be able to get a majority of replies
    so old leader won't commit or execute any new log entries
  thus no split brain
  but a minority may accept old server's AppendEntries
    so logs may diverge at end of old term

Raft vs. Paxos: https://dl.acm.org/doi/10.1145/3293611.3331595

LEC 6 Debugging

LEC 7 Fault Tolerance：Raft（2）

中文翻译

6.824 2022年第7讲：筏（2）

*** 主题：Raft 日志（实验 2B）

只要领导者不睡觉：
  客户只与领导者互动
  客户端看不到关注者状态或日志

更换领导者时事情会变得有趣
  例如在旧领导失败后
  如何在不出现异常的情况下更换领导者？
    副本不同、操作缺失、操作重复等

我们要确保什么?
如果任何服务器执行日志条目中的给定命令，
然后没有服务器对该日志条目执行其他操作
(图3的状态机安全)
为什么？如果服务器不同意操作，那么
领导者的变更可能会改变客户端可见的状态，
这违反了我们模仿单个服务器的目标。
例子:
S1：放置（k1，1）| 把(k1,2)
S2：放置（k1，1）| 把(k2,3)
不能允许两者都执行他们的第二个日志条目!

日志怎么能不同意呢?
领导者在向所有人发送 AppendEntries 之前崩溃
S1：3
S2：3 3
S3：3 3
(3 是日志条目中的术语编号)
更糟糕的是：日志在同一条目中可能有不同的命令!
在一系列领导者崩溃之后，例如
10 11 12 13 <- 日志条目#
S1：3
S2：3 3 4
S3：3 3 5

Raft 通过让追随者采用新领导者的日志来强制达成协议
例子:
S3 被选为第 6 届新领导人
S3想要在索引13处追加一个新的日志条目
S3 向所有对象发送 AppendEntries RPC
上一页日志索引=12
上一个日志项=5
S2 回复 false（AppendEntries 步骤 2）
S3 将 nextIndex[S2] 递减至 12
S3 发送带有条目 12+13、prevLogIndex=11、prevLogTerm=3 的 AppendEntries
S2 删除其条目 12（AppendEntries 步骤 3）
并附加新条目 12+13
S1 的情况类似，但 S3 必须再备份一个

回滚的结果:
每个活动追随者都会删除与领导者不同的日志尾部
然后每个实时追随者在该点之后接受领导者的条目
现在追随者的日志与领导者的日志相同

问：为什么可以忘记 S2 的 index=12 term=4 条目?

新领导者可以回滚上一任期结束时的“已提交”条目吗?
即，新领导者的日志中是否会丢失已提交的条目?
这将是一场灾难——老领导可能已经对客户说“是”
所以：Raft 需要确保当选的领导者拥有所有已提交的日志条目

为什么不选举日志最长的服务器作为领导者?
希望保证提交的条目永远不会回滚
例子:
S1:5 6 7
S2:5 8
S3:5 8
首先,这种情况会发生吗?如何?
S1 第 6 学期的领导者;崩溃+重启;第 7 学期的领导者;崩溃并保持低位
两次它只附加到自己的日志后就崩溃了
Q:S1在第7学期崩溃后,为什么S2/S3不选择6作为下学期?
下一学期将为 8,因为 S2/S3 中至少有一个在投票时获悉 7
第 8 学期 S2 领先,只有 S2+S3 活着,然后崩溃
所有对等点重新启动
谁应该成为下一个领导者?
S1 具有最长的日志,但条目 8 可能已提交!
所以新的领导者只能是S2或S3之一
所以规则不能简单地是"最长的日志"

5.4.1末尾解释了"选举限制"
RequestVote 处理程序仅投票给"至少是最新的"候选人:
候选者在最后一个日志条目中具有更高的术语,或者
候选人具有相同的上学期和相同长度或更长的日志
所以:
S2和S3不会投票给S1
S2和S3将互相投票
所以只有S2或S3可以成为领导者,将迫使S1丢弃6,7
好的,因为 6,7 不属于多数 -> 未提交 -> 回复从未发送给客户
-> 客户端将重新发送丢弃的命令

要点:
"至少是最新的"规则确保新领导者的日志包含
所有可能提交的条目
所以新的领导者不会回滚任何已提交的操作

问题(来自上一讲)
图7,顶级服务器挂掉;哪些可以当选?

谁能成为图7中的领导者?(顶级服务器已死)
需要 4 票才能成为领导者
a: 是 -- a、b、e、f
b: 否 -- b, f
e 的最后一项相同,但它的对数更长
c:是——a、b、c、e、f
d:是 -- a、b、c、d、e、f
e: 否 -- b, f
f: 否 -- f

为什么 d 不阻止 a 成为领导者?
毕竟 d 的对数比 a 的对数有更高的项
a 不需要 d 的投票即可获得多数票
a甚至不需要等待d的投票

为什么图 7 分析很重要?
领导者的选择决定了哪些条目被保留或被丢弃
关键：如果服务对客户做出积极回应，
这是有希望的，不要忘记!
必须保守：如果客户*可以*看到"是"，
领导者变更*必须*保留该日志条目。
选举限制通过多数交集来实现这一点。
为什么可以丢弃 e 的最后 4,4?
为什么可以（也许）保留 c 的最后 6 个?
客户能看到他们的"是"吗?

"承诺的操作"在 6.824 中有两个含义：

1. 即使由于（允许的）故障，操作也不会丢失。
   在 Raft 中：当大多数服务器将其保留在日志中时。
   这是"提交点"（参见图 8）。
2. 系统知道操作已提交。
   在 Raft 中：领导者看到了多数。

再次：
我们无法在提交之前回复客户"是"。
我们不能忘记可能已经实施的一项行动。

如何快速回滚
图 2 的设计为每个 RPC 备份一个条目 —— 慢!
实验室测试仪可能需要更快的回滚
论文概述了第 5.3 节末尾的计划
没有细节；这是我的猜测；更好的方案是可能的
案例1 案例2 案例3
S1: 4 5 5 4 4 4 4
S2：4 6 6 6 或 4 6 6 6 或 4 6 6 6
S2 是第 6 个学期的领导者，S1 复活，S2 在最后 6 个学期发送 AE
AE 的 prevLogTerm=6
S1 的拒绝包括：
XTerm：冲突条目中的术语（如果有）
XIndex：包含该术语的第一个条目的索引（如果有）
XLen：日志长度
情况 1（领导者没有 XTerm）：
下一个索引 = X索引
案例2（领导者有XTerm）：
nextIndex = 领导者在 XTerm 中的最后一个条目
情况3（关注者日志太短）：
下一个索引 = XLen

*** 主题：持久性（实验室 2C）

服务器崩溃后我们希望发生什么?
Raft 可以在缺少一台服务器的情况下继续运行
但出现故障的服务器必须尽快修复，以避免低于多数
两种修复策略：

- 替换为新的（空的）服务器
  需要将整个日志（或快照）传输到新服务器（慢）

我们必须支持这一点，以防失败是永久性的

- 或重新启动崩溃的服务器，以完好无损的状态重新加入，赶上
  需要在崩溃时持续存在的状态
  我们必须支持这一点，因为同时断电
  我们来谈谈第二个策略——坚持

如果服务器崩溃并重新启动，Raft 必须记住什么？
图2列出了“持久状态”：
log[]、当前术语、已投票
Raft 服务器只有在这些完好无损的情况下才能在重新启动后重新加入
因此它必须将它们保存到非易失性存储中
非易失性 = 磁盘、SSD、电池供电 RAM 等
在代码中更改非易失性状态的每个点之后保存
或者在发送任何 RPC 或 RPC 回复之前
为什么要记录[]？
如果服务器在提交条目时处于领导者多数，
尽管重新启动也必须记住条目，所以下一个领导者
投票多数包括条目，因此选举限制确保
新领导也有入场。
为什么投赞成票？
防止客户端投票给一位候选人，然后重新启动，
然后在同一任期内投票给不同的候选人
可能导致两位领导人同时任职
为什么是当前期限？
避免追随被取代的领导者。
避免在被取代的选举中投票。

一些 Raft 状态是不稳定的
commitIndex、lastApplied、下一个/matchIndex[]
为什么不保存这些就可以了？

持久性往往是性能的瓶颈
硬盘写入需要10毫秒，SSD写入需要0.1毫秒
所以持久性将我们限制在 100 到 10,000 次操作/秒
（另一个潜在的瓶颈是 RPC，在 LAN 上需要 << 1 毫秒）
有很多技巧可以应对持久性缓慢的问题：
每次磁盘写入批量处理许多新日志条目
保留到电池供电的 RAM，而不是磁盘
懒惰可能会丢失最后几次提交的更新

服务（例如 k/v 服务器）如何在崩溃+重启后恢复其状态？
简单方法：从空状态开始，重播 Raft 的整个持久日志
lastApplied 是易失性的并且从零开始，因此您可能不需要额外的代码！
这就是图 2 所做的
但对于长期存在的系统来说，重播会太慢
更快：使用 Raft 快照并仅重放日志的尾部

*** 主题：日志压缩和快照（实验室 2D）

问题：
日志将变得巨大——比状态机状态大得多！
重新启动或发送到新服务器时需要很长时间才能重新播放

幸运的是：
服务器不需要*完整的日志*和*服务状态
日志的执行部分被捕获在状态中
客户端只能看到状态，看不到日志
服务状态通常要小得多，所以让我们保持这一点

服务器*不能*丢弃哪些日志条目？
已提交但尚未执行
尚不清楚是否已犯

解决方案：服务定期创建持久的“快照”
[图：服务状态、磁盘上的快照、raft 日志（在内存中、在磁盘上）]
执行特定日志条目时的服务状态副本。
例如 k/v 表。
服务将快照传递给 Raft，其中包含最后包含的日志索引。
Raft 保留其状态和快照。
然后 Raft 会丢弃快照索引之前的日志。
每个服务器快照（不仅仅是领导者）。

崩溃+重启时会发生什么？
服务从磁盘读取快照
Raft 从磁盘读取持久化日志
Raft 将lastApplied 设置为快照的最后包含索引
以避免重新应用已经应用的日志条目

问题：如果追随者的日志在领导者的日志开始之前结束怎么办？
因为跟随者离线并且领导者丢弃了日志的早期部分
nextIndex[i] 将备份到领导者日志的开头
因此领导者无法使用 AppendEntries RPC 修复该追随者
因此InstallSnapshot RPC

哲学注释：
状态往往等同于操作历史
其中之一可能更适合存储或通信
我们将在课程后面看到这种二元性的例子

实用注意事项：
如果状态较小，Raft 的快照方案是合理的
对于大数据库，例如复制千兆字节的数据，不太好
创建整个数据库并将其写入磁盘的速度很慢
也许服务数据应该以 B 树形式存在于磁盘上
无需显式快照，因为已经在磁盘上
不过，处理滞后的副本很困难
领导者应该保存日志一段时间
或者记住状态的哪些部分已更新

*** 线性化能力

客户端程序员需要知道系统如何运行
实施者需要知道哪些设计/优化是可以的
我们需要一个“一致性契约”

到目前为止，我已经非正式地呼吁“像单个服务器一样行事”
这对于一个客户端、一次一个操作来说是可以的
归结为“如果服务器说它做了一个操作，那么
随后的客户阅读应该会看到它。”
但多个并发客户端意味着“后续”是
没有明确定义。
例如，如果两个客户端向同一个客户端写入不同的值
DB同时记录，哪个值应该是
后续阅读看到了吗？

“线性化”是强一致性的常见版本
类似于单个服务器的预期行为
可以在分布式系统中实现（需要一些费用）
您的实验 3 键/值存储将是线性化的

线性化定义：
执行历史是可线性化的如果
人们可以找到所有操作的总顺序，
匹配实时（对于非重叠操作），并且
其中每次读取都会看到来自的值
按顺序写在它前面。

历史记录是客户端操作的记录，每个操作都有
参数、返回值、开始时间、完成时间

历史示例1：
```
|-Wx1-| |-Wx2-|
  |---Rx2---|
    |-Rx1-|
```
x 轴是实时的
“Wx1”表示“写入值1来记录x”
“Rx1”表示“读取记录 x 产生值 1”
客户端发送这些操作（无论是一个还是多个客户端）
|- 是客户端发送RPC请求的时间
-| 是客户得到“是”答复的时间
对于 Raft，这些是第一次调用操作的 Start() 时间，
以及 Raft 将提交的 op 发送到 applyCh 的时间

这段历史可以线性化吗？
我们能否找到满足规则的操作总顺序？
绘制约束箭头：
值约束 (W -> R)
实时约束（Wx1 -> Wx2）
该订单满足约束条件：
宽x1 接收x1 宽x2 接收x2
因为原始并发历史是有全序的
满足规则，历史是线性化的。

注意：该定义基于外部行为
这样我们就可以应用它而不必知道服务如何运作
注意：历史明确地以以下形式包含并发性：
重叠操作（操作不会在某个时间点发生），因此很好
匹配分布式系统的运行方式。

历史示例2：

```
 |-Wx1-| |-Wx2-|
   |--Rx2--|
           |-Rx1-|
```

绘制约束箭头：

Wx1 先于 Wx2（时间）

Rx2 之前的 Wx2（值）

Rx2 先于 Rx1（时间）

Rx1 先于 Wx2（值）

存在一个循环——所以它不能变成线性顺序。所以这
历史是不可线性化的。

即使 Rx1 与 Wx2 重叠，它也可以在没有 Rx2 的情况下线性化。

上次讲座回顾：

线性化是您将看到的最严格的一致性。

它形式化了单个一次性服务器可能的行为。

它通常应用于客户端/服务器存储系统。

线性化是我们在实验 3 中要求您实现的行为。

线性化是根据具体的客户可见的"历史"来定义的。

要显示历史是可线性化的，请找到总计（一次一个）

遵循任何之前/之后和写入->读取值的顺序。

要显示不可线性化，请显示一个循环（两者都不能先行），

或枚举所有订单可能性并显示没有一个是合法的。

历史示例3：

```
|--Wx0--| |--Wx1--|
          |--Wx2--|
      |-Rx2-| |-Rx1-|
```

这可能看起来是非线性的，因为 Wx2 看起来

就像它出现在 Wx1 之后，但 Rx2 出现在 Rx1 之前。

但这个顺序表明它是可线性化的：Wx0 Wx2 Rx2 Wx1 Rx1

所以：

该服务可以选择任一顺序进行并发写入。

例如，将并发操作交给 Raft.Start() 的键/值服务器

线性顺序可以与开始时间顺序不同！

历史示例 4：

```
|--Wx0--| |--Wx1--|
          |--Wx2--|
C1：|-Rx2-| |-Rx1-|
C2：|-Rx1-| |-Rx2-|
```

可以有总订单吗？

C1需要Wx2 Rx2 Wx1 Rx1

C2需要Wx1 Rx1 Wx2 Rx2

我们不能同时拥有 Wx2 在 Wx1 之前和 Wx2 在 Wx1 之后。

所以不可线性化。

所以：

服务可以选择并发写入的任一顺序

但所有客户端必须以相同的顺序看到写入

当我们有副本或缓存时，这一点很重要

他们必须就操作发生的顺序达成一致

历史示例 5：
|-Wx1-|
　　|-Wx2-|
　　　　|-Rx1-|
订单可以包含 Wx2 Rx1 吗？
　no：读取没有看到最新写入的值
订单可以包含 Rx1 Wx2 吗？
　否：如果一个操作在另一个操作开始之前结束，则顺序必须保持顺序
没有顺序是可能的——不可线性化
所以：
　读取必须返回新数据：陈旧值不可线性化
　即使读者不知道写作
　　时间规则要求读取以产生最新数据
　线性化禁止许多情况：
　　裂脑（两个活跃的领导者）
　　重启后忘记提交的写入
　　从滞后副本中读取

历史示例 6：
假设客户在没有得到答复的情况下重新发送请求
如果是响应丢失了：
　领导者记得它已经看到的客户请求
　如果看到重复，则使用第一次执行时保存的响应进行回复
但这可能会产生一个现在已经过时的保存值！
线性化能力是什么意思？
C1：|-Wx3-| |-Wx4-|
C2: |-Rx3----------|
顺序：Wx3 Rx3 Wx4
所以：返回旧的保存值3是正确的
　　返回 4 也是正确的
此示例可能会帮助您推理实验 3。

如果你想避免看到，线性化是理想的
　过时的数据，或者如果您想要原子读取-修改-写入操作
　就像追加一样。

当我们查看系统时，您会更加欣赏细节
　具有不太严格的一致性。

您可能会发现此页面很有用：
https://www.anishathalye.com/2017/06/04/testing-distributed-systems-for-线性化/

*** 重复 RPC 检测（实验室 3）

如果 Put 或 Get RPC 超时，客户端该怎么办？
　即 Call() 返回 false
　如果服务器死机，或者请求被丢弃：重新发送
　如果服务器执行，但回复丢失：重新发送是危险的

问题：
　这两种情况在客户看来是一样的（没有回复）
　如果已经执行，客户端仍然需要结果

想法：重复RPC检测
让 k/v 服务检测重复的客户端请求
客户端为每个请求选择一个唯一的 ID，并在 RPC 中发送
同一 RPC 重发时的 ID 相同
k/v 服务维护一个按 ID 索引的“重复表”
为每个 RPC 创建一个表条目
执行后，将回复内容记录在复制表中
如果第二个 RPC 使用相同的 ID 到达，则它是重复的
根据表中的值生成回复

新的领导者如何获得重复的表?
将 ID 放入记录的操作中并交给 Raft
所有副本都应在执行时更新其重复表
所以如果他们成为领导者，信息就已经存在了

如果服务器崩溃，它如何恢复其表?
如果没有快照，重播日志将填充表
如果是快照，快照必须包含表的副本

如果重复请求在原始请求执行之前到达怎么办?
可以再次调用 Start()
它可能会在日志中出现两次（相同的客户端 ID，相同的 seq #）
当cmd出现在applyCh上时，如果表说已经看到，则不执行

保持重复表较小的想法
每个客户端一个表条目，而不是每个 RPC 一个表条目
每个客户端一次只有一个 RPC 未完成
每个客户端按顺序对 RPC 进行编号
当服务器收到客户端 RPC #10 时，
它可以忘记客户端的较低条目
因为这意味着客户端永远不会重新发送旧的 RPC

一些细节：
每个客户端都需要一个唯一的客户端 ID——也许是一个 64 位随机数
客户端在每个 RPC 中发送客户端 ID 和 seq #
如果重新发送则重复 seq #
k/v 服务中按客户端 ID 索引的重复表
仅包含 seq # 和值（如果已执行）
RPC 处理程序首先检查表，如果 seq # > 表条目，则仅 Start()s
每个日志条目必须包含客户端 ID、seq #
当applyCh上出现操作时
更新客户端表条目中的 seq # 和值
唤醒等待的 RPC 处理程序（如果有）

可是等等!
k/v 服务器现在从重复表返回旧值
如果表中的回复值不再是最新的怎么办?
这可以吗?

例子：
C1 C2

把(x,10)
第一次发送 get(x)，reply(10) 被丢弃
把(x,20)
重新发送 get(x)，服务器从表中获取 10，而不是 20

线性化能力是什么意思?
C1：|-Wx10-| |-Wx20-|
C2：|-Rx10----------|
订单：Wx10 Rx10 Wx20
所以：返回记住的值10是正确的

*** 只读操作（第 8 节末尾）

问：Raft Leader 是否必须提交只读操作
回复之前的日志？例如获取（密钥）？

也就是说，领导者是否可以立即响应 Get() 使用
其键/值表的当前内容?

答：不，图 2 中的方案或实验室中不会。
假设 S1 认为自己是领导者，并收到 Get(k)。
它可能最近输掉了一次选举，但没有意识到，
由于网络数据包丢失。
新的领导者，比如 S2，可能已经处理了密钥的 Put()，
因此 S1 的键/值表中的值已过时。
提供陈旧数据是不可线性化的；这是裂脑。

所以：图 2 要求将 Get() 提交到日志中。
如果领导者能够提交 Get()，那么（此时
在日志中）它仍然是领导者。在S1的情况下
上面，在不知不觉中失去了领导地位，它不会
能够获得大多数积极的 AppendEntries 回复
需要提交 Get()，因此它不会回复客户端。

但是：许多应用程序的读取量很大。提交 Get()
需要时间。有什么办法可以避免提交
用于只读操作？这是一个巨大的考虑因素
实用的系统。

想法：租赁
修改Raft协议如下
定义租用期限，例如5秒
每次领导者获得 AppendEntries 多数之后，
它有权响应只读请求
不添加只读请求的租用期
到日志，即不发送 AppendEntries。
新领导者在前一个租约期之前无法执行 Put()
已过期
因此关注者可以跟踪他们上次回复的时间
到 AppendEntries，并告诉新的领导者（在
请求投票回复）。
结果：只读操作更快，仍然可线性化。

注意：对于实验室，您应该将 Get() 提交到日志中；
不实施租赁。

在实践中，人们常常（但并非总是）愿意忍受陈旧的事物
数据以换取更高的性能

英文原文

6.824 2022 Lecture 7: Raft (2)

*** topic: the Raft log (Lab 2B)

as long as the leader stays up:
clients only interact with the leader
clients don't see follower states or logs

things get interesting when changing leaders
e.g. after the old leader fails
how to change leaders without anomalies?
diverging replicas, missing operations, repeated operations, &c

what do we want to ensure?
if any server executes a given command in a log entry,
then no server executes something else for that log entry
(Figure 3's State Machine Safety)
why? if the servers disagree on the operations, then a
change of leader might change the client-visible state,
which violates our goal of mimicing a single server.
example:
S1: put(k1,1) | put(k1,2)
S2: put(k1,1) | put(k2,3)
can't allow both to execute their 2nd log entries!

how can logs disagree?
a leader crashes before sending AppendEntries to all
S1: 3
S2: 3 3
S3: 3 3
(the 3s are the term number in the log entry)
worse: logs might have different commands in same entry!
after a series of leader crashes, e.g.
10 11 12 13 <- log entry #
S1: 3
S2: 3 3 4
S3: 3 3 5

Raft forces agreement by having followers adopt new leader's log
example:
S3 is chosen as new leader for term 6
S3 wants to append a new log entry at index 13
S3 sends an AppendEntries RPC to all
prevLogIndex=12
prevLogTerm=5
S2 replies false (AppendEntries step 2)

S3 decrements nextIndex[S2] to 12
S3 sends AppendEntries w/ entries 12+13, prevLogIndex=11, prevLogTerm=3
S2 deletes its entry 12 (AppendEntries step 3)
and appends new entries 12+13
similar story for S1, but S3 has to back up one farther

the result of roll-back:
each live follower deletes tail of log that differs from leader
then each live follower accepts leader's entries after that point
now followers' logs are identical to leader's log

Q: why was it OK to forget about S2's index=12 term=4 entry?

could new leader roll back *committed* entries from end of previous term?
i.e. could a committed entry be missing from the new leader's log?
this would be a disaster -- old leader might have already said "yes" to a client
so: Raft needs to ensure elected leader has all committed log entries

why not elect the server with the longest log as leader?
in the hope of guaranteeing that a committed entry is never rolled back
example:
S1: 5 6 7
S2: 5 8
S3: 5 8
first, could this scenario happen? how?
S1 leader in term 6; crash+reboot; leader in term 7; crash and stay down
both times it crashed after only appending to its own log
Q: after S1 crashes in term 7, why won't S2/S3 choose 6 as next term?
next term will be 8, since at least one of S2/S3 learned of 7 while voting
S2 leader in term 8, only S2+S3 alive, then crash
all peers reboot
who should be next leader?
S1 has longest log, but entry 8 could have committed !!!
so new leader can only be one of S2 or S3
so the rule cannot be simply "longest log"

end of 5.4.1 explains the "election restriction"
RequestVote handler only votes for candidate who is "at least as up to date":
candidate has higher term in last log entry, or
candidate has same last term and same length or longer log
so:
S2 and S3 won't vote for S1
S2 and S3 will vote for each other
so only S2 or S3 can be leader, will force S1 to discard 6,7
ok since 6,7 not on majority -> not committed -> reply never sent to clients
-> clients will resend the discarded commands

the point:
"at least as up to date" rule ensures new leader's log contains
all potentially committed entries
so new leader won't roll back any committed operation

The Question (from last lecture)
  figure 7, top server is dead; which can be elected?

who could become leader in figure 7? (with top server dead)
  need 4 votes to become leader
  a: yes -- a, b, e, f
  b: no -- b, f
    e has same last term, but its log is longer
  c: yes -- a, b, c, e, f
  d: yes -- a, b, c, d, e, f
  e: no -- b, f
  f: no -- f

why won't d prevent a from becoming leader?
  after all, d's log has higher term than a's log
  a does not need d's vote in order to get a majority
  a does not even need to wait for d's vote

why is Figure 7 analysis important?
  choice of leader determines which entries are preserved vs discarded
  critical: if service responded positively to a client,
    it is promising not to forget!
  must be conservative: if client *could* have seen a "yes",
    leader change *must* preserve that log entry.
    Election Restriction does this via majority intersection.
  why OK to discard e's last 4,4?
  why OK to (perhaps) preserve c's last 6?
    could client have seen a "yes" for them?

"a committed operation" has two meanings in 6.824:

1. the op cannot be lost, even due to (allowable) failures.
   in Raft: when a majority of servers persist it in their logs.
   this is the "commit point" (though see Figure 8).
2. the system knows the op is committed.
   in Raft: leader saw a majority.

again:
  we cannot reply "yes" to client before commit.
  we cannot forget an operation that may have been committed.

how to roll back quickly
  the Figure 2 design backs up one entry per RPC -- slow!
  lab tester may require faster roll-back
  paper outlines a scheme towards end of Section 5.3
    no details; here's my guess; better schemes are possible
      Case 1   Case 2   Case 3
  S1: 4 5 5   4 4 4   4
  S2: 4 6 6 6 or 4 6 6 6 or 4 6 6 6
  S2 is leader for term 6, S1 comes back to life, S2 sends AE for last 6
    AE has prevLogTerm=6
  rejection from S1 includes:
    XTerm: term in the conflicting entry (if any)

XIndex: index of first entry with that term (if any)
XLen: log length
Case 1 (leader doesn't have XTerm):
nextIndex = XIndex
Case 2 (leader has XTerm):
nextIndex = leader's last entry for XTerm
Case 3 (follower's log is too short):
nextIndex = XLen

*** topic: persistence (Lab 2C)

what would we like to happen after a server crashes?
Raft can continue with one missing server
but failed server must be repaired soon to avoid dipping below a majority
two repair strategies:

- replace with a fresh (empty) server
  requires transfer of entire log (or snapshot) to new server (slow)
  we must support this, in case failure is permanent
- or reboot crashed server, re-join with state intact, catch up
  requires state that persists across crashes
  we must support this, for simultaneous power failure
  let's talk about the second strategy -- persistence

if a server crashes and restarts, what must Raft remember?
Figure 2 lists "persistent state":
log[], currentTerm, votedFor
a Raft server can only re-join after restart if these are intact
thus it must save them to non-volatile storage
non-volatile = disk, SSD, battery-backed RAM, &c
save after each point in code that changes non-volatile state
or before sending any RPC or RPC reply
why log[]?
if a server was in leader's majority for committing an entry,
must remember entry despite reboot, so next leader's
vote majority includes the entry, so Election Restriction ensures
new leader also has the entry.
why votedFor?
to prevent a client from voting for one candidate, then reboot,
then vote for a different candidate in the same term
could lead to two leaders for the same term
why currentTerm?
avoid following a superseded leader.
avoid voting in a superseded election.

some Raft state is volatile
commitIndex, lastApplied, next/matchIndex[]
why is it OK not to save these?

persistence is often the bottleneck for performance
a hard disk write takes 10 ms, SSD write takes 0.1 ms
so persistence limits us to 100 to 10,000 ops/second
(the other potential bottleneck is RPC, which takes << 1 ms on a LAN)

lots of tricks to cope with slowness of persistence:
batch many new log entries per disk write
persist to battery-backed RAM, not disk
be lazy and risk loss of last few committed updates

how does the service (e.g. k/v server) recover its state after a crash+reboot?
easy approach: start with empty state, re-play Raft's entire persisted log
lastApplied is volatile and starts at zero, so you may need no extra code!
this is what Figure 2 does
but re-play will be too slow for a long-lived system
faster: use Raft snapshot and replay just the tail of the log

*** topic: log compaction and Snapshots (Lab 2D)

problem:
log will get to be huge -- much larger than state-machine state!
will take a long time to re-play on reboot or send to a new server

luckily:
a server doesn't need *both* the complete log *and* the service state
the executed part of the log is captured in the state
clients only see the state, not the log
service state usually much smaller, so let's keep just that

what log entries *can't* a server discard?
committed but not yet executed
not yet known if committed

solution: service periodically creates persistent "snapshot"
[diagram: service state, snapshot on disk, raft log (in mem, on disk)]
copy of service state as of execution of a specific log entry.
e.g. k/v table.
service hands snapshot to Raft, with last included log index.
Raft persists its state and the snapshot.
Raft then discards log before snapshot index.
every server snapshots (not just the leader).

what happens on crash+restart?
service reads snapshot from disk
Raft reads persisted log from disk
Raft sets lastApplied to snapshot's last included index
to avoid re-applying already-applied log entries

problem: what if follower's log ends before leader's log starts?
because follower was offline and leader discarded early part of log
nextIndex[i] will back up to start of leader's log
so leader can't repair that follower with AppendEntries RPCs
thus the InstallSnapshot RPC

philosophical note:
state is often equivalent to operation history
one or the other may be better to store or communicate
we'll see examples of this duality later in the course

practical notes:
  Raft's snapshot scheme is reasonable if the state is small
  for a big DB, e.g. if replicating gigabytes of data, not so good
    slow to create and write entire DB to disk
  perhaps service data should live on disk in a B-Tree
    no need to explicitly snapshot, since on disk already
  dealing with lagging replicas is hard, though
    leader should save the log for a while
    or remember which parts of state have been updated

*** linearizability

client programmers need to know how the system can behave
  implementers need to know what designs/optimizations are OK
  we need a "consistency contract"

so far I've appealed informally to "act like a single server"
  that's OK for one client, one-at-a-time operations
  boils down to "if the server says it did an operation, then
    a subsequent client read should see it."
  but multiple concurrent clients mean "subsequent" is
    not well defined.
  e.g. if two clients write different values to the same
    DB record at the same time, which value should a
    subsequent read see?

"linearizability" is a common version of strong consistency
  similar to behavior expected of a single server
  can be implemented in a distributed system (at some expense)
  your Lab 3 key/value store will be linearizable

linearizability definition:
  an execution history is linearizable if
    one can find a total order of all operations,
    that matches real-time (for non-overlapping ops), and
    in which each read sees the value from the
    write preceding it in the order.

a history is a record of client operations, each with
  arguments, return value, time of start, time completed

example history 1:
  |-Wx1-| |-Wx2-|
    |---Rx2---|
      |-Rx1-|
the x-axis is real time
"Wx1" means "write value 1 to record x"
"Rx1" means "a read of record x yielded value 1"
clients send these operations (it doesn't matter if one or many clients)
  the |- is the time at which the client sends the request RPC
  the -| is the time at which the client gets a "yes" reply
  for Raft, these are the time of the first Start() call for an op,
    and the time at which Raft sends the committed op up the applyCh

is this history linearizable?
  can we find a total order of operations which satisifies the rules?
  draw the constraint arrows:
    value constraints (W -> R)
    real-time constraint (Wx1 -> Wx2)
  this order satisfies the constraints:
    Wx1 Rx1 Wx2 Rx2
  since there's a total order of the original concurrent history
    that satisfies the rules, the history is linearizable.

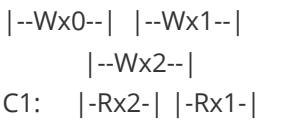
note: the definition is based on external behavior
  so we can apply it without having to know how service works
note: histories explicitly incorporate concurrency in the form of
  overlapping operations (ops don't occur at a point in time), thus good
  match for how distributed systems operate.

example history 2:
  |-Wx1-| |-Wx2-|
    |--Rx2--|
          |-Rx1-|
draw the constraint arrows:
  Wx1 before Wx2 (time)
  Wx2 before Rx2 (value)
  Rx2 before Rx1 (time)
  Rx1 before Wx2 (value)
there's a cycle -- so it cannot be turned into a linear order. so this
history is not linearizable.

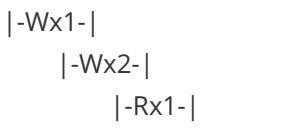
it would be linearizable w/o Rx2, even though Rx1 overlaps with Wx2.

review from last lecture:
  linearizability is the most strict consistency you'll see.
    it formalizes the behavior possible with a single one-at-a-time server.
    it's usually applied to client/server storage systems.
  linearizability is the behavior we're asking you for in Lab 3.
  linearizability is defined on concrete client-visible "histories".
  to show a history is linearizable, find a total (one-at-a-time)
    order that obeys any before/after and write->read values.
  to show not linearizable, show a cycle (neither can go first),
    or enumerate all order possibilities and show none are legal.

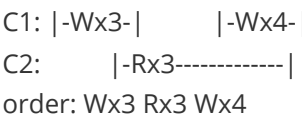
example history 3:
|--Wx0--| |--Wx1--|
          |--Wx2--|
       |-Rx2-| |-Rx1-|
this may look non-linearizable because Wx2 looks
  like it comes after Wx1, but Rx2 comes before Rx1.
but this order shows it's linearizable: Wx0 Wx2 Rx2 Wx1 Rx1
so:
  the service can pick either order for concurrent writes.
  e.g. a key/value server handing concurrent ops to Raft.Start()
  the linear order can be different from start-time order!

example history 4:


can there be a total order?
C1 needs Wx2 Rx2 Wx1 Rx1
C2 needs Wx1 Rx1 Wx2 Rx2
we can't have both Wx2 before Wx1, and Wx2 after Wx1.
so not linearizable.
so:
service can choose either order for concurrent writes
but all clients must see the writes in the same order
this is important when we have replicas or caches
they have to all agree on the order in which operations occur

example history 5:


can order include Wx2 Rx1?
no: the read doesn't see the latest written value
can order include Rx1 Wx2?
no: the order has to preserve order if one op ends before another starts
no order is possible -- not linearizable
so:
reads must return fresh data: stale values aren't linearizable
even if the reader doesn't know about the write
the time rule requires reads to yield the latest data
linearizability forbids many situations:
split brain (two active leaders)
forgetting committed writes after a reboot
reading from lagging replicas

example history 6:
suppose clients re-send requests if they don't get a reply
in case it was the response that was lost:
leader remembers client requests it has already seen
if sees duplicate, replies with saved response from first execution
but this may yield a saved value that is now out of date!
what does linearizabilty say?


order: Wx3 Rx3 Wx4
so: returning the old saved value 3 is correct
returning 4 would also be correct
this example might help you reason about Lab 3.

linearizability is desirable if you want to avoid ever seeing
out-of-date data, or if you want atomic read-modify-write operations
like append.

you'll appreciate the details more when we look at systems
  with less strict consistency.

You may find this page useful:
https://www.anishathalye.com/2017/06/04/testing-distributed-systems-for-linearizability/

*** duplicate RPC detection (Lab 3)

What should a client do if a Put or Get RPC times out?
  i.e. Call() returns false
  if server is dead, or request was dropped: re-send
  if server executed, but reply was lost: re-send is dangerous

problem:
  these two cases look the same to the client (no reply)
  if already executed, client still needs the result

idea: duplicate RPC detection
  let's have the k/v service detect duplicate client requests
  client picks a unique ID for each request, sends in RPC
    same ID in re-sends of same RPC
  k/v service maintains a "duplicate table" indexed by ID
  makes a table entry for each RPC
    after executing, record reply content in duplicate table
  if 2nd RPC arrives with the same ID, it's a duplicate
    generate reply from the value in the table

how does a new leader get the duplicate table?
  put ID in logged operations handed to Raft
  all replicas should update their duplicate tables as they execute
  so the information is already there if they become leader

if server crashes how does it restore its table?
  if no snapshots, replay of log will populate the table
  if snapshots, snapshot must contain a copy of the table

what if a duplicate request arrives before the original executes?
  could just call Start() (again)
  it will probably appear twice in the log (same client ID, same seq #)
  when cmd appears on applyCh, don't execute if table says already seen

idea to keep the duplicate table small
  one table entry per client, rather than one per RPC
  each client has only one RPC outstanding at a time
  each client numbers RPCs sequentially
  when server receives client RPC #10,
    it can forget about client's lower entries
    since this means client won't ever re-send older RPCs

some details:
  each client needs a unique client ID -- perhaps a 64-bit random number
  client sends client ID and seq # in every RPC
    repeats seq # if it re-sends
  duplicate table in k/v service indexed by client ID
    contains just seq #, and value if already executed

RPC handler first checks table, only Start()s if seq # > table entry
each log entry must include client ID, seq #
when operation appears on applyCh
  update the seq # and value in the client's table entry
  wake up the waiting RPC handler (if any)

but wait!
the k/v server is now returning old values from the duplicate table
what if the reply value in the table is no longer up to date?
is that OK?

example:
C1 C2

---

put(x,10)
  first send of get(x), reply(10) dropped
put(x,20)
  re-sends get(x), server gets 10 from table, not 20

what does linearizabilty say?
C1: |-Wx10-| |-Wx20-|
C2: |-Rx10----------|
order: Wx10 Rx10 Wx20
so: returning the remembered value 10 is correct

*** read-only operations (end of Section 8)

Q: does the Raft leader have to commit read-only operations in
  the log before replying? e.g. Get(key)?

that is, could the leader respond immediately to a Get() using
the current content of its key/value table?

A: no, not with the scheme in Figure 2 or in the labs.
  suppose S1 thinks it is the leader, and receives a Get(k).
  it might have recently lost an election, but not realize,
  due to lost network packets.
  the new leader, say S2, might have processed Put()s for the key,
  so that the value in S1's key/value table is stale.
  serving stale data is not linearizable; it's split-brain.

so: Figure 2 requires Get()s to be committed into the log.
  if the leader is able to commit a Get(), then (at that point
  in the log) it is still the leader. in the case of S1
  above, which unknowingly lost leadership, it won't be
  able to get the majority of positive AppendEntries replies
  required to commit the Get(), so it won't reply to the client.

but: many applications are read-heavy. committing Get()s
takes time. is there any way to avoid commit
for read-only operations? this is a huge consideration in
practical systems.

idea: leases
  modify the Raft protocol as follows
  define a lease period, e.g. 5 seconds
  after each time the leader gets an AppendEntries majority,
    it is entitled to respond to read-only requests for
    a lease period without adding read-only requests
    to the log, i.e. without sending AppendEntries.
  a new leader cannot execute Put()s until previous lease period
    has expired
  so followers keep track of the last time they responded
    to an AppendEntries, and tell the new leader (in the
    RequestVote reply).
  result: faster read-only operations, still linearizable.

note: for the Labs, you should commit Get()s into the log;
      don't implement leases.

in practice, people are often (but not always) willing to live with stale
  data in return for higher performance

LEC 8 Q&A Lab2 A+B

LEC 9 Zookeeper

中文翻译

6.824 2022年第9讲：Zookeeper案例研究

阅读：“ZooKeeper：互联网规模系统的无等待协调”，Patrick
亨特、马哈德夫·科纳尔、弗拉维奥·P·容凯拉、本杰明·里德。2010年会议论文集
USENIX 年度技术会议。

公告：
  下周四期中考试，课堂上，面对面
  项目提案将于下周五提交

今天的讲座使用ZooKeeper有两种方式：

- 构建容错服务的更简单方法。
- 基于 Raft 式复制构建的现实服务中的高性能。

如果我们想做一个像 MR coordinator 这样的容错服务，
  我们很想用 Raft 进行复制，
  那就没问题了！

但直接在 Raft 上构建很复杂
  现有的 raft 库有帮助，但仍然很复杂
  复制状态机通常会影响应用程序结构
  有没有更简单的方法？

您可以将状态机复制（Raft）视为复制
  计算；状态作为副作用被复制。

只复制状态怎么样?
容错存储系统中的状态
计算读/写容错状态
就像使用数据库或文件一样——比状态机更容易
如果计算失败，状态仍然会被保留。
可以从该状态重新开始计算。
因此不需要重复计算。

MapReduce 会是什么样子?
[ZK、MR坐标、工人、GFS]

MR coord 可以在 ZK 中存储什么?
坐标 IP 地址、作业集、任务状态、工作人员集、分配

ZK通常充当通用"配置服务"

如果 MR 坐标失败怎么办?
我们没有 MR 坐标服务器副本!
但我们不需要!
只需选择任何服务器，在其上运行 MR coord 软件，
让它从 ZK 读取其状态。
新坐标可以从失败的坐标处继续。

挑战
即使坐标崩溃，也以安全的方式更新多项目状态
新坐标需要能够恢复/修复状态
选择 MR 坐标（一次一个！）
如果旧坐标没有意识到它已被替换怎么办
ZK中还能读/写状态吗?
或者在其他存储中?
读取性能

Zookeeper数据模型（图1）
状态：类似文件系统的 znode 树
文件名、文件内容、目录、路径名
每个znode都有一个版本号
znode 的类型：
常规的
短暂的
顺序：名称+序列号

人们如何使用 ZK znode?
znodes 可能包含例如 MR 协调器的 IP 地址，
工人的 IP 地址、MR 职位描述、
分配/完成了哪些 MR 任务
znode 的存在可能表明存在活动的 MR 协调器，
一名工人正在为其存在做广告
分层名称方便允许不同的应用程序
共享单个 ZK 服务器，并组织每个
应用程序的不同类型的数据（工作人员的一个目录
广告存在，一组作业的另一个目录，&c）

znode 上的操作（第 2.2 节）
创建（路径、数据、标志）
独占——只有第一次创建才表明成功
删除（路径，版本）
如果 znode.version = version，则删除
存在（路径，观察）
watch=true 表示如果稍后创建/删除路径也会发送通知
getData（路径，手表）
setData(路径、数据、版本)
如果 znode.version = 版本，则更新
getChildren（路径，手表）
同步（）
先同步后读取确保同步之前的写入对同一客户端的读取可见
客户端可以改为提交写入

ZooKeeper API 针对并发和同步进行了良好调整：

- 独家文件创建；恰好有一个并发创建返回成功
- getData()/setData(x, version) 支持迷你交易
- 当客户端失败时，会话会自动执行操作（例如，在失败时释放锁定）
- 顺序文件在多个客户端之间创建顺序
- 手表避免轮询

示例：MapReduce 协调员选举
股票代码（）：
虽然正确：
如果创建（“/ mr / c”，短暂= true）
充当主人...
否则如果存在("/mr/c", watch=true)
等待观看事件
别的
再试一次

退休（）：
删除(“/mr/c”)

笔记：
独家创造
如果多个客户端同时尝试，只有一个会成功
临时znode
协调器故障自动让新的协调器被选举出来
手表
潜在的替代协调员可以等待而不进行轮询

如果当选的服务协调员失败怎么办？
请注意，尽管看起来像独占锁，
失败的可能性使情况
与 Gosync.Mutex 等非常不同
客户端失败 -> 客户端停止向 ZK 发送 keep-alive 消息
没有 keep-alives -> ZK 使会话超时
会话超时 -> ZK 自动删除客户端的临时文件

如果协调器在 ZK 中更新状态时崩溃怎么办?
特别是如果有多个 znode 包含状态数据
例如，每个 MR 任务的 znode 指示待处理/活动/完成，在哪里可以找到输出
也许是结构化数据，因此协调器永远不需要更新多个 znode
单独的 setData() 调用是原子的（全有或全无 vs 失败）
可能有一个包含当前状态 znode 名称的摘要 znode
创建具有新状态的*新* znode
setData("sum", 当前状态 znode 的名称)
ZK 保证顺序，因此“sum”更新将不可见，除非
所有之前的写入都是可见的
“单一提交写入”

如果协调员还活着并且认为它仍然是协调员怎么办?
但 ZK 已经决定它已经死了并删除了它的临时 /mr/c 文件?
可能会选出一位新的协调员。
两台计算机会认为它们是协调者吗?
这可能会发生。
旧协调器可以修改ZK中的状态吗?
这不可能发生!
当 ZK 使客户端会话超时时，会自动发生两件事：
ZK 删除客户端临时节点。
ZK 停止监听会话。
所以旧的协调器不能再向ZK发送请求。
一旦旧协调员意识到会话已死，它
可以创建一个新的协调员，但现在它知道自己不是协调员。

如果当选的协调员将状态存储在 ZK 之外，例如在 GFS 中，
新协调员必须告诉 GFS（或其他什么）忽略
来自被罢免的协调员的请求。
也许使用纪元号。
称为“击剑”。

如果您不想有协调员，但想要怎么办
多个并发 ZK 客户端更新共享状态?

示例：将 ZooKeeper znode 中存储的数字加 1
虽然正确：
x, v := getData("f")
如果设置数据（x + 1，版本=v）：
休息
这是一笔“小型交易”
效果是原子读-修改-写
许多变体，例如 VMware-FT 的测试和设置

我们可以使用 ZK 进行容错通信以及状态存储
工作人员通过创建临时 znode 进行广告
容错：worker 崩溃 -> znode 消失
MR 客户提交工作
容错：如果客户端崩溃，作业请求仍然存在!
重新启动的客户端可以检查 ZK 以查看工作是否/何时完成。

ZK 客户端界面是如何设计的以获得良好的性能?

主要关注点是读取性能

1）许多ZK追随者，客户端分布在他们身上以实现并行性

客户端将所有操作发送给该 ZK follower

ZK follower 在本地从其 ZK 数据副本执行读取

避免加载 ZK 领导者

ZK follower 将写入转发给 ZK Leader

2）观看，而不是民意调查

ZK 追随者（不是 ZK 领导者）完成工作

3）ZK客户端发起异步操作

即发送请求；完成通知单独到达

与 RPC 不同

客户端可以启动多次写入而无需等待

ZK批量高效处理它们；更少的消息、磁盘写入

客户端库对它们进行编号，ZK 按顺序执行它们

例如更新一堆 znode，然后创建“就绪”znode

读取可能看不到最新完成的写入!

因为客户的追随者可能有点落后（不是写的大多数）

所以读取不是线性化的

通过性能/可扩展性证明合理性

可以发送写入（或同步）以强制等待最近的写入

但是，写入时按日志顺序（“zxid”顺序）一次执行一个

您需要这个来进行小型交易，例如独占 create()

并写入以 zxid 顺序出现在所有 ZK 关注者和客户端上

因此，如果 ZK 客户端看到“就绪”znode，它将看到之前的更新

ZK保证什么?

所有写入的单一顺序——ZK领导者的日志顺序。

“zxid”

所有客户端看到的写入都按 zxid 顺序出现。

包括其他客户端的写入。

客户端的读取可以看到客户端之前的所有写入。

客户的 ZK follower 可能需要等到客户的

最新写入由 ZK Leader 提交。

客户端保证在看到事件之前看到观看事件

该事件之后写入的值。

读/写保证如何发挥作用的示例

假设我们在 ZK 中有协调器写入的配置数据，

但许多其他 ZK 客户端需要阅读

数据由一堆 zknode 组成

论文建议：

协调器删除“就绪”zknode

协调器更新配置 zknodes

协调器创建“就绪”zknode

客户端在读取之前等待直到准备好的 zknode 存在

但是如果客户端在协调器删除它之前看到“就绪”怎么办?

写入顺序： 读取顺序：

存在（“准备好”，手表=真）

读f1

删除（“准备好”）
写f1
写f2
读取 f2
创建（“准备好”）

效率取决于客户如何使用ZK！
简单锁有什么问题？（第 6 页）
假设有数百个客户端正在等待锁？
更好：没有羊群效应的锁
1.创建一个“顺序”文件

2. 列出文件
3. 如果没有更低的编号，则获取锁！
4. 如果存在（下一个较低编号，watch=true）
5.等待活动...
5. 转到2

问：可以在步骤 2 和 3 之间创建编号较小的文件吗？
问：轮到客户之前可以看火吗？
答：是的
lock-10 <- 当前锁持有者
lock-11 <- 下一个
lock-12 <- 我的请求

```
如果创建 lock-11 的客户端在获得锁之前死亡，则
手表会响，但还没轮到我。
```

ZK 实现的目标是高性能：
数据必须适合内存，因此读取速度很快（无需读取磁盘）。
所以你不能在ZooKeeper中存储大量数据。
写入（日志条目）必须写入磁盘并等待。
因此，已提交的更新不会因崩溃或电源故障而丢失。
损害延迟；批处理可以提高吞吐量。
定期将完整的快照写入磁盘。
模糊技术意味着快照写入与请求处理是并发的。

表现如何？

图 5——吞吐量。
为什么线条向右移动时会上升？
为什么x=0的性能会随着服务器数量的增加而下降？
为什么“3 个服务器”行更改在 100% 读取时最差？
每秒 20,000 次操作的 x=0 吞吐量是好是坏？
是什么决定了这 20,000 人？为什么不是20万？
每个操作都是 1000 字节的写入...

延迟怎么样？
表 2/第 5.2 节表示 1.2 毫秒。
对于在每个写入请求后等待的单个工作人员（客户端）。
1.2 毫秒从何而来？
磁盘写入？沟通？计算？

恢复时间怎么样?
图8
追随者失败 -> 只是总吞吐量下降。
领导者失败 -> 超时和选举暂停。
从视觉上看，大约是一秒。

ZooKeeper 是一个成功的设计。
请参阅 ZooKeeper 的维基百科页面以获取使用它的项目列表
通常用作一种容错名称服务
当前协调器的 IP 地址是什么？存在哪些工人？
可用于简化整体容错策略
将所有状态存储在ZK中，例如MR作业队列，任务状态
那么服务服务器本身不需要复制

参考：
https://zookeeper.apache.org/doc/r3.4.8/api/org/apache/zookeeper/ZooKeeper.html
ZAB：http://dl.acm.org/itation.cfm?id=2056409
https://zookeeper.apache.org/
https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf（免费等待，通用物体等）

英文原文

6.824 2022 Lecture 9: Zookeeper Case Study

Reading: "ZooKeeper: wait-free coordination for internet-scale systems", Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, Benjamin Reed. Proceedings of the 2010 USENIX Annual Technical Conference.

announcements:
mid-term next thursday, in class, in person
project proposal due next friday

today's lecture uses ZooKeeper in two ways:

- a simpler way to structure fault-tolerant services.
- high-performance in a real-life service built on Raft-like replication.

if we wanted to make a fault-tolerant service like MR coordinator,
we'd be tempted to replicate with Raft,
and that would be OK!

but building directly on Raft is complex
an existing raft library wd help, but still complex
replicated state machine usually affects application structure
is there a simpler way?

you can think of state machine replication (Raft) as replicating
the computation; the state is replicated as a side-effect.

how about replicating just the state?
state in fault-tolerant storage system
computation reads/writes fault-tolerant state
like using a DB, or files -- easier than state machine
if computation fails, state is nevertheless preserved.

can re-start computation from that state.
thus no need to replicate computation.

what would it look like for MapReduce?
[ZK, MR coord, workers, GFS]

what might MR coord store in ZK?
coord IP addr, set of jobs, status of tasks, set of workers, assignments

ZK often acts as a general-purpose "configuration service"

what if MR coord fails?
we don't have a replica MR coord server!
but we don't need one!
just pick any server, run MR coord s/w on it,
have it read its state from ZK.
new coord can pick up where failed one left off.

challenges
update multi-item state in a way that's safe even if coord crashes
new coord needs to be able to recover/repair state
elect MR coord (one at a time!)
what if old coord doesn't realize it's been replaced
can it still read/write state in ZK?
or in other storage?
read performance

Zookeeper data model (Figure 1)
the state: a file-system-like tree of znodes
file names, file content, directories, path names
each znode has a version number
types of znodes:
regular
ephemeral
sequential: name + seqno

how do people use ZK znodes?
znodes might contain e.g. MR coordinator's IP address,
workers' IP addresses, MR job description,
which MR tasks are assigned/completed
presence of znode might indicate that an active MR coordinator exists,
that a worker is advertisiting its existence
hierarchical names convenient to allow different apps
to share a single ZK server, and to organize each
app's different kind of data (one dir for workers to
advertise existence, another dir for set of jobs, &c)

Operations on znodes (Section 2.2)
create(path, data, flags)
exclusive -- only first create indicates success
delete(path, version)
if znode.version = version, then delete
exists(path, watch)
watch=true means also send notification if path is later created/deleted

```
  getData(path, watch)
  setData(path, data, version)
    if znode.version = version, then update
  getChildren(path, watch)
  sync()
    sync then read ensures writes before sync are visible to same client's read
    client could instead submit a write
```

ZooKeeper API well tuned for concurrency and synchronization:

- exclusive file creation; exactly one concurrent create returns success
- getData()/setData(x, version) supports mini-transactions
- sessions automate actions when clients fail (e.g. release lock on failure)
- sequential files create order among multiple clients
- watches avoid polling

Example: MapReduce coordinator election

```
  ticker():
    while true:
      if create("/mr/c", ephemeral=true)
        act as master...
      else if exists("/mr/c", watch=true)
        wait for watch event
      else
        try again

  retire():
    delete("/mr/c")
```

note:

```
  exclusive create
    if multiple clients concurrently attempt, only one will succeed
  ephemeral znode
    coordinator failure automatically lets new coordinator be elected
  watch
    potential replacement coordinators can wait w/o polling
```

what if the elected service coordinator fails?

```
  note that even though looks like an exclusive lock,
    the possibility of failure makes the situation
    very different from e.g. Go sync.Mutex
  client failure -> client stops sending keep-alive messages to ZK
  no keep-alives -> ZK times out the session
  session timeout -> ZK automatically deletes client's ephemeral files
```

what if the coordinator crashes while updating state in ZK?

```
  particularly if there are multiple znodes containing state data
    e.g. znode per MR task indicating pending/active/done, where to find output
  maybe structure data so coordinator never needs to update more than one znode
    individual setData() calls are atomic (all or nothing vs failure)
  maybe have a summary znode containing names of current state znodes
    create new znodes with new state
    setData("sum", names of current state znodes)
```

ZK guarantees order, so "sum" update won't be visible unless
all preceding writes are visible
"single committing write"

what if the coordinator is alive and thinks it is still coordinator,
but ZK has decided it is dead and deleted its ephemeral /mr/c file?
a new coordinator will likely be elected.
will two computers think they are the coordinator?
this could happen.
can the old coordinator modify state in ZK?
this cannnot happen!
when ZK times out a client's session, two things happen atomically:
ZK deletes the clients ephemeral nodes.
ZK stops listening to the session.
so old coordinator can no longer send requests to ZK.
once old coordinator realizes the session is dead, it
can create a new one, but now it knows it isn't coordinator.

if elected coordinators store state outside of ZK, e.g. in GFS,
new coordinator must tell GFS (or whatever) to ignore
requests from deposed coordinator.
perhaps using epoch numbers.
called "fencing".

what if you don't want to have a coordinator, but want
multiple concurrent ZK clients to update shared state?

Example: add one to a number stored in a ZooKeeper znode
while true:
x, v := getData("f")
if setData(x + 1, version=v):
break
this is a "mini-transaction"
effect is atomic read-modify-write
lots of variants, e.g. test-and-set for VMware-FT

we can use ZK for fault-tolerant communication as well as state storage
worker advertises by creating an ephemeral znode
fault-tolerant: worker crash -> znode disappears
MR clients submitting jobs
fault-tolerant: if client crashes, job request still there!
a re-started client can check in ZK to see if/when job is done.

how is the ZK client interface designed for good performance?
main focus is on read performance

1. many ZK followers, clients are spread over them for parallelism
   client sends all operations to that ZK follower
   ZK follower executes reads locally, from its replica of ZK data
   to avoid loading the ZK leader
   ZK follower forwards writes to ZK leader
2. watch, not poll
   and the ZK follower (not the ZK leader) does the work

3. clients of ZK launch async operations
i.e. send request; completion notification arrives separately
unlike RPC
a client can launch many writes without waiting
ZK processes them efficiently in a batch; fewer msgs, disk writes
client library numbers them, ZK executes them in that order
e.g. to update a bunch of znodes then create "ready" znode

a read may not see latest completed writes!
since client's follower may be a little behind (not in write's majority)
so reads aren't linearizable
justified by performance/scalability
can send a write (or sync) to force wait for recent writes
writes, however, execute one at a time in log order ("zxid" order)
you need this for mini-transactions like exclusive create()
and writes appear in zxid order at all ZK followers and clients
so if ZK client sees "ready" znode, it will see updates that preceded it

what does ZK guarantee?
a single order for all writes -- ZK leader's log order.
"zxid"
all clients see writes appear in zxid order.
including writes by other clients.
a client's read sees all of client's preceding writes.
client's ZK follower may need to wait until client's
latest write is committed by ZK leader.
client guaranteed to see watch event before it sees
values from writes after that event.

an example of how read/write guarantees play out
suppose we have configuration data in ZK that coordinator writes,
but that many other ZK clients need to read
and the data consists of a bunch of zknodes
paper suggests:
coordinator deletes "ready" zknode
coordinator updates the configuration zknodes
coordinator creates "ready" zknode
clients wait until ready zknode exists before reading
but what if client sees "ready" just before coordinator deletes it?

Write order:	Read order:
	exists("ready", watch=true)
	read f1
delete("ready")	
write f1	
write f2	
	read f2
create("ready")	

Efficiency depends on how clients use ZK!
what's wrong with Simple Locks? (page 6)
suppose 100s of clients are waiting for the lock?
better: Locks without Herd Effect

1. create a "sequential" file
2. list files
3. if no lower-numbered, lock is acquired!
4. if exists(next-lower-numbered, watch=true)
5. wait for event...
6. goto 2

Q: could a lower-numbered file be created between steps 2 and 3?
Q: can watch fire before it is the client's turn?
A: yes
lock-10 <- current lock holder
lock-11 <- next one
lock-12 <- my request

```
if client that created lock-11 dies before it gets the lock, the
watch will fire but it isn't my turn yet.
```

The ZK implementation aims at high performance:
Data must fit in memory, so reads are fast (no need to read disk).
So you can't store huge quantities of data in ZooKeeper.
Writes (log entries) must be written to disk, and waited for.
So committed updates aren't lost in a crash or power failure.
Hurts latency; batching can help throughput.
Periodically complete snapshots are written to disk.
Fuzzy technique means snapshot write is concurrent with request processing.

How is the performance?

Figure 5 -- throughput.
Why do the lines go up as they move to the right?
Why does the x=0 performance go down as the number of servers increases?
Why is the "3 servers" line change to be worst at 100% reads?
Is the x=0 throughput of 20,000 ops/second good or bad?
What might determine the 20,000? Why not 200,000?
Each op is a 1000-byte write...

What about latency?
Table 2 / Section 5.2 says 1.2 milliseconds.
For a single worker (client) waiting after each write request.
Where might the 1.2 milliseconds come from?
Disk writes? Communication? Computation?

What about recovery time?
Figure 8
Follower failure -> just a decrease in total throughput.
Leader failure -> a pause for timeout and election.
Visually, on the order of a second.

ZooKeeper is a successful design.
see ZooKeeper's Wikipedia page for a list of projects that use it
often used as a kind of fault-tolerant name service
what's the current coordinator's IP address? what workers exist?
can be used to simplify overall fault-tolerance strategy

store all state in ZK e.g. MR queue of jobs, status of tasks
then service servers needn't themselves replicate

References:
https://zookeeper.apache.org/doc/r3.4.8/api/org/apache/zookeeper/ZooKeeper.html
ZAB: http://dl.acm.org/citation.cfm?id=2056409
https://zookeeper.apache.org/
https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf (wait free, universal objects, etc.)

LEC 10 Chain Replication

中文翻译

6.824 2022第10讲：链式复制

链式复制支持高吞吐量和可用性
(OSDI 2004) 作者：Renesse 和 Schneider

今天的几个话题：
重新访问主/备份复制
链复制作为更好的主/备份
比较市盈率、链与法定人数
分片系统的结构

关于 Raft/Paxos/Zab（“quorum”）属性的一些想法
例如，使用 5 台服务器，您只能承受 2 次故障
因此您可能有两个完整的副本但无法继续！
协议很复杂——图2

还有更简单的方法！
有不同的权衡

主/备份（论文第 4 节）
古老，以各种形式广泛使用
GFS 对块使用（非严格）p/b 方案
CR纸是一种改进的p/b纸

基本的主/备安排
[图：CFG、客户端、主数据库、两个备份、状态 = 键/值表]
主要数字操作，如果并发客户端操作到达则强制执行顺序
主节点将每个节点转发到备份节点（并行）
主服务器等待所有备份的响应，回复客户端
主数据库可以响应读取而不发送到备份
如果主数据库发生故障，其中一个备份将成为新的主数据库
即使只有一个副本幸存也可以继续
所以 N 个副本可以承受 N-1 次故障——比 Raft 更好
单独的配置服务 (CFG) 管理故障转移
配置 = 当前主设备和备份设备的身份
通常构建在 Paxos 或 Raft 或 ZooKeeper 上
对所有服务器执行 ping 操作以检测故障
CFG 必须至少包含一个副本
任何新配置中的先前配置。

如果备份失败，CFG 必须将其从配置中删除，
b/c 主数据库必须等待所有备份

有时称为 ROWA（读一写全部），而不是法定人数
“全部写入”使我们能够容忍 N-1 次失败
因为每个副本都有所有提交的值，与仲裁不同

我们需要检查任何强复制方案的一些关键属性：

1. 绝不泄露未提交的数据
即可能无法承受可容忍故障的数据
2.故障恢复后，所有副本必须同意！
如果其中一个看到来自主数据库的最后一条消息，但另一个没有看到
3.网络分区无脑裂
只有一个新的主要；旧的小学必须停止

主/备份#1？
不良情况示例：
put(x, 99) 到达初级
get(x) 紧随其后
在主节点获得备份对 put(x,99) 的响应之前
我们说主数据库可以在不与备份通信的情况下提供读取服务
但返回99并不安全！
答案：要么用旧值响应，
或阻止读取，直到所有备份响应 put(x,99)。

#2 主/备份？
新的主节点可以合并所有副本的最后几个操作。
或者，选择接收最高 # op 的备份作为主要备份。
新的主节点必须首先强制备份节点同意它

#3 主/备份？
如果从主分区分区，如何防止备份接管？
并且primary还活着。
*没有*让备份做出这个决定！
只有CFG可以声明新的主+备份。
基于 CFG 对其可以到达的服务器的意见。
如果旧的主节点实际上还活着，但 CFG 无法与其通信怎么办？
如何防止旧的主要服务请求？
更新：
旧的主节点无法响应客户端请求，直到所有
其配置回复中的副本
新配置中必须至少有一个副本
因此副本必须小心，不要回复旧的主副本！
用于读取：
主数据库不必与备份进行读取操作
因此 CFG 必须向主要项目授予租约

Chain Replication 发表的时候只有少数人
了解强一致性复制的详细设计；
它很有影响力，并且构建了相当多的现实世界系统
在上面。

链复制旨在解决哪些 p/b 问题?

1.初级要做很多工作

2. Primary要发送大量网络数据

3. 主故障后重新同步很复杂，因为备份可能不同

链式复制的基本思想：

[客户端，S1=头，S2，S3=尾，CFG（=主）]

（可以有更多副本）

客户端向 head 发送更新请求

头部选择订单（分配序列号）

head 更新本地副本，发送到 S1

S1更新本地副本，发送到S2

S2更新本地副本，发送到S3

S3更新本地副本，向客户端发送响应

更新按顺序沿着链移动：

在每台服务器上，较早的更新先于较晚的更新交付

客户端向 tail 发送读请求

tail读取本地副本并响应客户端

好处：头发送的网络数据比主节点少

好处：客户交互工作分为头部和尾部

问题：假设链复制回复了更新请求

从头开始，一旦下一个链服务器说它收到了

转发更新，而不是从尾部响应。解释一下如何

这可能会导致链式复制产生不正常的结果

可线性化。

如果头部出现故障怎么办?

CFG（主站）负责从故障中恢复

CFG 任命第二链服务器为新负责人

并告诉客户新任负责人是谁

所有（剩余的）服务器仍然是精确的副本吗?

他们很快就会的!

按顺序交付意味着每个服务器都是相同的

与之前的相比，只是缺少最后几次更新

所以每个服务器都需要与后继服务器进行比较

只需发送最后几次更新

一些客户端更新请求可能会丢失

如果只有那个失败的头脑知道他们就好了

客户不会收到回复

并最终会重新发送到新的头

如果尾巴失败怎么办?

CFG 告诉倒数第二个服务器是新尾部

并告诉客户，对于读取请求

倒数第二个至少与旧尾部一样最新

对于新尾部收到但旧尾部未收到的更新，

系统不会向客户端发送响应。

客户端将超时并重新发送

第 2 条规定客户有责任检查

超时操作是否实际上已经
被处决（通常比听起来更难）。

如果中间服务器出现故障怎么办？
CFG 告诉上一个/下一个服务器相互通信
以前的服务器可能必须重新发送一些更新
它已经发送到发生故障的服务器

请注意，服务器即使在转发后也需要记住更新
如果失败需要他们重新发送
什么时候免费？
tail 在收到更新时将 ACK 发送回链
当服务器收到 ACK 时，它可以通过该操作释放所有内容

CR 不会透露未提交的更新的论点是什么？
即客户端可以读取一个值，
但它随后会因为可容忍的失败而消失？
或者客户能否得到更新的“是”响应，
但失败后它就不存在了？
读取来自尾部
尾部仅在其他所有服务器看到更新后才看到更新
所以在发生故障后，每个服务器仍然有该更新
update到达尾部后才响应
此时每个服务器都有它

如何添加新服务器？（“延长链条”）
您需要执行此操作以在发生故障后恢复复制级别。
再次，CFG 管理这个
新服务器添加在尾部
缓慢的可能性：
告诉旧尾停止处理更新
告诉旧尾部将其数据的完整副本发送到新尾部
告诉旧尾巴开始充当中间服务器，
转发到新尾部
告诉新尾巴开始充当尾巴
告诉客户新尾巴
慢 b/c 我们将暂停所有更新几分钟或几小时
更好的是提前传输状态快照
然后冻结系统足够长的时间来发送
新尾巴的最后几次更新
然后重新配置并取消冻结
也许使用 ZooKeeper 的模糊快照想法

分区情况与 p/b 中的情况很相似
CFG 做出所有决定
它将选择一个新的头&c
基于其对服务器活跃度的看法——即仅在 CFG 的分区中
新头是旧的第二服务器，它应该忽略
来自旧头脑的更新，以应对“如果老了怎么办”
头还活着，但 CFG 认为它已经失败了”
CFG 需要授予 tail 租约来服务客户端读取，
并且在租约到期之前不指定新尾部

p/b 与链式复制?
p/b 可能具有较低的延迟（对于小请求）
链头的网络负载比主链少
如果数据项很大（如 GFS），则很重要
链条在头部和尾部之间分配工作
Primary 完成了这一切，也许更多是一个瓶颈
链有更简单的故事，如果头发生故障，服务器应该接管，
以及如何确保服务器恢复同步

链（或 p/b）与 Raft/Paxos/Zab（法定人数）？
p/b 可以容忍 N 次故障中的 N-1 次，法定人数仅 N/2
p/b 更简单，也许比法定人数更快
p/b 需要单独的 CFG，仲裁自包含
p/b 必须在失败后等待重新配置，仲裁继续进行
即使一台服务器慢，p/b 也会慢，法定人数可以容忍少数暂时慢的服务器
p/b CFG 的服务器故障检测器很难调整：
任何失败的服务器都会停止运行，所以要尽快宣布失败！
但过于急切的故障检测器会浪费时间将数据复制到新服务器。
仲裁系统更优雅地处理短暂/不明确的故障

长期以来，p/b（和链）主导的数据复制
Paxos 被认为对于高性能数据库而言过于复杂且缓慢
最近法定人数系统已经取得进展
由于对暂时缓慢/片状副本的良好容忍度

如果您的数据太多而无法放入单个副本组怎么办?
例如数百万个对象
您需要在许多“副本组”中“分片”

分片图：
[CFG、G1、G2、..、Gn、客户]
GFS 看起来像这样
现代系统可能使用 ZK 进行 CFG，使用链或 p/b 或 Raft 进行数据

如何在大型分片设置中在服务器上布置链?
论文的5.2 / 5.3 / 5.4

不太好的链或 p/b 分片安排：
每组三台服务器服务于一个分片/链
分片A：S1 S2 S3
分片 B：S4 S5 S6
问题：某些服务器的负载会比其他服务器高
每个组中的主节点都会很慢，而其他组则有闲置容量
头部和尾部的负载会比中间的负载更多
负载不足的服务器浪费金钱！
问题：更换失败的副本需要很长时间！
新服务器必须通过网络获取整个磁盘的数据
来自剩余副本之一
1 TB 以千兆位/秒传输需要两个小时！
剩余副本在完成之前失败的重大风险！

更好的计划（第 5.4 节中的“rndpar”）：
- 将数据分割成比服务器更多的分片
  - （所以每个分片比之前的排列要小得多）
- 每个服务器都是许多分片组中的副本
  - 分片A：S1 S2 S3
  - 分片 B：S2 S3 S1
  - 分片 C：S3 S1 S2
  - （这是有规律的排列，但一般都是随机的）
- 对于p/b，服务器在某些组中是主服务器，在其他组中是备份服务器
- 对于链来说，服务器在一些链中处于头部，在另一些链中处于尾部，在另一些链中处于中间
- 现在请求处理工作可能会更加平衡

rndpar 的修复速度如何？
- 假设一台服务器发生故障。
- 假设它参与了 M 个副本组、M 个分片。
- 让我们不要指定单个替换服务器
  - 选择 M 个替换服务器，每个分片都有一个不同的服务器。
  - 这些是现有的服务器，我们赋予它们新的责任。
- 现在M个碎片的修复可以并行进行了！
  - 不再需要几个小时，而是花费 1/M 的时间。

如果三个随机服务器出现故障，rndpar 会如何处理？
- 随着每台服务器上的分片数量增加，它会变得更多
  - 可能*某些*分片上有其三个副本
  - 三个随机出现故障的服务器。
- 这并不理想。
- rndpar 为我们提供了快速修复，但它有所削弱
  - 少数失败会导致全部失败的可能性更大
  - 某些分片的副本（图 7）。

结论
- 链复制是对 ROWA 方案最清晰的描述之一
- 它在平衡工作方面做得很好
- 它有一种在失败后重新同步副本的简单方法
- 有影响的：用于EBS、Ceph、Parameter Server、COPS、FAWN。
- 它是具有不同属性的多种设计（p/b、quorums）之一

5.1要点：
- 链*吞吐量*高达 p/bb/c，受头/主 CPU 限制
- 对于 0% 更新，链仅受 tail 限制，p/b 仅受 prim 限制
- 对于 100% 更新，链仅受 head 限制，p/b 仅受 prim 限制
- 只有中间有区别 b/c 链拆分工作
  - 头和尾之间
- 但网络通信假设是免费的；在真实生活中
  - p/b 有问题 b/c prim 必须发送给所有备份
- 但写入延迟在现实世界中是一个问题，因为
  - 客户经常等待“承诺”回复

5.2 要点：
假设有数千条链和数十台服务器。
想法：将你的数据分割到许多链上，并且链
超过一定数量的服务器，以便每台服务器
存在于许多链中，有的为头，有的为尾，有的为链
在中间
这平衡了头部和尾部与中部的负载。
图5好像没说太多，也许只是说如果
你只有 25 个客户，那么就没啥意义了
拥有超过 25 台服务器。

5.3 要点：
假设有数千条链和数十台服务器。
修复时间很重要，因为单个服务器可以存储
数据量如此之大，需要数小时才能通过网络传输
单一网络链接。
当*单个*服务器发生故障时，需要分散责任
对于它通过"多个"其他服务器复制的数据，
获得快速并行修复。

5.4 要点：
假设有数千条链和数十台服务器。
最好的并行恢复速度是如果没有限制并且
随机放置，以便源和目的地
的恢复流量在单个
失败。
但是如果链随机分布在服务器上，那么*任意*
三个（如果 chainlen=3）随机服务器故障的组合
有很大机会销毁*某些*链的所有副本。
环形拓扑是一种妥协：你不能扩散
恢复负载非常广泛，但有一些随机服务器故障
不太可能销毁任何一条链的所有副本。
对于他们的设置，重建速度似乎更快
比同时发生故障而毁掉一条链条更重要。
然而，他们假设单个服务器的 MTBF 为 24 小时，
这确实意味着当修复可以时，快速修复至关重要
需要几个小时，但 24 小时似乎短得不切实际。

英文原文

6.824 2022 Lecture 10: Chain Replication

Chain replication for supporting high throughput and availability
(OSDI 2004) by Renesse and Schneider

a couple of topics today:
revisit primary/backup replication
chain replication as a better primary/backup
compare p/b vs chain vs quorum
structure of sharded systems

some thoughts about Raft/Paxos/Zab ("quorum") properties
  with e.g. 5 servers you can only survive 2 failures
    so you might have two intact replicas but not be able to proceed!
  protocol is complex -- Figure 2

there are simpler approaches!
  with different tradeoffs

primary/backup (paper's Section 4)
  old, widely used in various forms
  GFS uses a (non-strict) p/b scheme for chunks
  CR paper is an improved p/b

the basic primary/backup arrangement
  [diagram: CFG, clients, primary, two backups, state = key/value table]
  primary numbers ops, to impose order if concurrent client ops arrive
  primary forwards each to backups (in parallel)
  primary waits for response from ALL backups, replies to client
  primary can respond to reads w/o sending to backups
  if primary fails, one of the backups becomes new primary
    can proceed if even a single replica survives
    so N replicas can survive N-1 failures -- better than Raft
  a separate configuration service (CFG) manages failover
    configuration = identity of current primary and backups
    usually built on Paxos or Raft or ZooKeeper
    pings all servers to detect failures
    CFG must include at least one replica from
      previous configuration in any new configuration.
  if a backup fails, CFG must remove it from the configuration,
    b/c the primary has to wait for all backups

sometimes called ROWA (Read One Write All), as opposed to quorum
  the "write all" is what allows us to tolerate N-1 failures
    since every replica has all committed values, unlike quorums

a few key properties we need to check for any strong replication scheme:

1. never reveal uncommitted data
   i.e. data that might not survive a tolerated failure
2. after failure recovery, all replicas must agree!
   if one saw last msg from primary, but the other did not
3. no split brain if network partitions
   only one new primary; old primary must stop

#1 for primary/backup?
  example bad situation:
    put(x, 99) arrives at primary
    get(x) immediately follows it
      before primary gets backups' responses to put(x,99)
    we said primary can serve reads w/o talking to backups
    but it's not safe to return 99!
  answer: either respond with old value,
    or block read until all backups respond to put(x,99).

#2 for primary/backup?
  new primary could merge all replicas' last few operations.
  or, elect as primary the backup that received the highest # op.
  and new primary must start by forcing backups to agree with it

#3 for primary/backup?
  how to prevent backup from taking over if partitioned from primary?
    and primary is still alive.
    don't have backups make this decision!
    only the CFG can declare a new primary + backups.
      based on CFG's opinion of which servers it can reach.
  what if the old primary is actually alive, but CFG can't talk to it?
    how to prevent old primary from serving requests?
    for updates:
      old primary cannot respond to client requests until all
        replicas in its configuration reply
      at least one replica must be part of the new configuration
      so replicas must be careful not to reply to an old primary!
    for reads:
      primary doesn't have to talk to backups for reads
      so CFG must grant lease to primary

Chain Replication was published at a time when only a few people
  understood the detailed design of strongly consistent replication;
  it's been influential, and a fair number of real-world systems build
  on it.

what p/b problems does Chain Replication aim to fix?

    1. primary has to do a lot of work
    2. primary has to send a lot of network data
    3. re-sync after primary failure is complex, since backups may differ

the basic Chain Replication idea:
  [clients, S1=head, S2, S3=tail, CFG (= master)]
  (can be more replicas)
  clients send updates requests to head
    head picks an order (assigns sequence numbers)
    head updates local replica, sends to S1
    S1 updates local replica, sends to S2
    S2 updates local replica, sends to S3
    S3 updates local replica, sends response to client
    updates move along the chain in order:
      at each server, earlier updates delivered before later ones
  clients send read requests to tail
    tail reads local replica and responds to client

benefit: head sends less network data than a primary

benefit: client interaction work is split between head and tail

The Question: Suppose Chain Replication replied to update requests from the head, as soon as the next chain server said it received the forwarded update, instead of responding from the tail. Explain how that could cause Chain Replication to produce results that are not linearizable.

what if the head fails?
- the CFG (master) is in charge of recovering from failures
- CFG tells 2nd chain server to be new head
  - and tells clients who the new head is
- will all (remaining) servers still be exact replicas?
  - they will soon!
  - in-order delivery means each server is identical
    - to the one before it, just missing last few updates
  - so each server needs to compare notes with successor and
    - just send those last few updates
- some client update requests may be lost
  - if only the failed head knew about them
  - clients won't receive responses
  - and will eventually re-send to new head

what if the tail fails?
- CFG tells next-to-last server to be new tail
  - and tells clients, for read requests
- next-to-last is at least as up to date as the old tail
- for updates that new tail received but old tail didn't,
  - system won't send responses to clients.
  - clients will time out and re-send
  - Section 2 says clients are responsible for checking
    - whether timed-out operations have actually already
    - been executed (often harder than it sounds).

what if an intermediate server fails?
- CFG tells previous/next servers to talk to each other
- previous server may have to re-send some updates that
  - it had already sent to failed server

note that servers need to remember updates even after forwarding
- in case a failure requires them to re-send
- when to free?
- tail sends ACKs back up the chain as it receives updates
  - when a server gets an ACK, it can free all through that op

what's the argument that CR won't reveal an uncommitted update?
- i.e. could client read a value,
  - but it then disappears due to a tolerated failure?
- or could a client get a "yes" response to an update,
  - but then it's not there after a failure?
- reads come from the tail
  - the tail only sees an update after every other server sees it
  - so after a failure, every server still has that update

update is responded to after it gets to the tail
at which point every server has it

how to add a new server? ("extend the chain")
you need to do this to restore replication level after a failure.
again, CFG manages this
new server is added at the tail
a slow possibility:
tell the old tail to stop processing updates
tell the old tail to send a complete copy of its data to the new tail
tell the old tail to start acting as an intermediate server,
forwarding to the new tail
tell the new tail to start acting as the tail
tell clients about new tail
slow b/c we're pausing all updates for minutes or hours
better is to transfer a snapshot of the state in advance
then freeze the system just long enough to send the
last few updates to the new tail
then reconfigure and un-freeze
perhaps use ZooKeeper's fuzzy snapshot idea

partition situation is much as in p/b
CFG makes all decisions
it will pick a single new head &c
based on its view of server liveness -- i.e. just in CFG's partition
new head is old 2nd server, it should ignore
updates from the old head, to cope with "what if old
head is alive but CFG thinks it has failed"
CFG needs to grant tail a lease to serve client reads,
and not designate a new tail until lease has expired

p/b versus chain replication?
p/b may have lower latency (for small requests)
chain head has less network load than primary
important if data items are big (as with GFS)
chain splits work between head and tail
primary does it all, maybe more of a bottleneck
chain has simpler story for which server should take over if head fails,
and how ensure servers get back in sync

chain (or p/b) versus Raft/Paxos/Zab (quorum)?
p/b can tolerate N-1 of N failures, quorum only N/2
p/b simpler, maybe faster than quorum
p/b requires separate CFG, quorum self-contained
p/b must wait for reconfig after failure, quorum keeps going
p/b slow if even one server slow, quorum tolerates temporary slow minority
p/b CFG's server failure detector hard to tune:
any failed server stalls p/b, so want to declare failed quickly!
but over-eager failure detector will waste time copying data to new server.
quorum system handles short / unclear failures more gracefully

for a long time p/b (and chain) dominated data replication
  Paxos was viewed as too complex and slow for high-performance DBs
  recently quorum systems have been gaining ground
    due to good toleration of temporarily slow/flaky replicas

what if you have too much data to fit on a single replica group?
  e.g. millions of objects
  you need to "shard" across many "replica groups"

sharding diagram:
  [CFG, G1, G2, .., Gn, clients]
  GFS looked like this
  modern system might use ZK for CFG, chain or p/b or Raft for data

how to lay out chains on server in a big sharding setup?
  the paper's 5.2 / 5.3 / 5.4

a not-so-great chain or p/b sharding arrangement:
  each set of three servers serves a single shard / chain
    shard A: S1 S2 S3
    shard B: S4 S5 S6
  problem: some servers will be more loaded than others
    the primary in each group will be slow while the others have idle capacity
    the head and tail will be more loaded than the middle
    the under-loaded servers waste money!
  problem: replacing a failed replica takes a long time!
    the new server must fetch a whole disk of data over the network
      from one of the remaining replicas
    a terabyte at a gigabit/second takes two hours!
    significant risk of remaining replicas failing before completion!

a better plan ("rndpar" in Section 5.4):
  split data into many more shards than servers
    (so each shard is much smaller than in previous arrangement)
  each server is a replica in many shard groups
    shard A: S1 S2 S3
    shard B: S2 S3 S1
    shard C: S3 S1 S2
    (this is a regular arrangement, but in general would be random)
  for p/b, a server is primary in some groups, backup in other
  for chain, a server is head in some, tail in others, middle in others
  now request processing work is likely to be more balanced

how does rndpar do for repair speed?
  suppose one server fails.
  say it participated in M replica groups, for M shards.
  instead of designating a single replacement server, let's
    choose M replacement servers, a different one for each shard.
    these are existing servers, which we're giving a new responsibility.
  now repair of the M shards can go on in parallel!
    instead of taking a few hours, it will take 1/M'th that time.

how does rndpar do if three random servers fail?
as the number of shards on each server increases, it gets more
likely that *some* shard had its three replicas on
the three random servers that failed.
this is not ideal.
rndpar gives us fast repair, but it's somewhat undermined
by higher probability that a few failures wipes out all
replicas for some shard (Figure 7).

conclusion
Chain Replication is one of the clearest descriptions of a ROWA scheme
it does a good job of balancing work
it has a simple approach to re-syncing replicas after a failure
influential: used in EBS, Ceph, Parameter Server, COPS, FAWN.
it's one of a number of designs (p/b, quorums) with different properties

---

5.1 take-away:
chain *throughput* as high as p/b b/c limited by head/primary CPU
for 0% updates, chain limited by tail alone, and p/b limited by prim alone
for 100% updates, chain limited by head alone, and p/b limited by prim alone
only in the middle is there a difference b/c chain splits work
between head and tail
BUT network communication assumed to be free; in real life
p/b has a problem b/c prim must send to all backups
BUT write latency is a problem in the real world, since
client often waiting for "committed" reply

5.2 take-away:
assume 1000s of chains and dozens of servers.
idea: split your data over many chains, and the chains
over a modest number of servers, so that every server
is in many chains, some as head, some as tail, some
in the middle
this balances the load of being head and tail vs middle.
Figure 5 doesn't seem to say much, maybe just that if
you have only 25 clients, then there's not much point
in having more than about 25 servers.

5.3 take-away:
assume 1000s of chains and dozens of servers.
repair time is a big deal, since a single server can store
so much data that it takes hours to transfer over a
single network link.
when a *single* server fails, need to spread responsibility
for the data it replicated over *multiple* other servers,
to get fast parallel repair.

5.4 take-away:
assume 1000s of chains and dozens of servers.
best for parallel recovery speed is if no constraints and
random placement, so that both sources and destinations

of recovery traffic are evenly-ish spread after a single failure.
BUT if chains are randomly spread over servers, then *any* combination of three (if chainlen=3) random server failures has a good chance of destroying all replicas of *some* chain.
the ring topologies are a compromise: you can't spread recovery load very widely, but a few random server failures are less likely to destroy all of any one chain's replicas.
for their setup, speed of reconstruction seems to be more important than simultaneous failures wiping out a chain.
however, they assume MTBF of a single server of 24 hours, which does mean fast repair is crucial when repair can take hours, but 24 hours seems unrealistically short.

LEC 11 Distributed Transactions

中文翻译

6.824 2022第11讲：分布式事务

主题：
分布式事务=并发控制+原子提交

我们在课程中处于什么位置？
到目前为止，主要是为了容错而进行分布
多台服务器试图看起来像一台可靠的服务器
现在，很多服务器都是为了性能
将数据分割（分片）到多个服务器上，以实现并行性
只要客户一次使用一个数据项就可以
如果应用操作涉及到不同分片的记录怎么办？
失败？原子性？
重要的问题，我们会看到很多方法

有什么问题？
大量数据记录，分片在多个服务器上，大量客户端
[图：客户端、服务器、按键分片的数据]
客户端应用程序操作通常涉及多次读取和写入
银行转账：借记卡和贷记卡
vote：检查是否已经投票，记录投票，增加计数
在社交图中安装双向链接
我们希望对应用程序编写者隐藏交错和失败

这个问题首先出现在数据库中
传统解决方案：交易
程序员将代码序列的开始/结束标记为事务
系统自动提供良好的行为

交易示例
x 和 y 是银行余额——数据库表中的记录
在不同的服务器上（也许在不同的银行）
两者起价均为 10 美元
T1和T2是交易
T1：将 1 美元从 x 转移到 y

T2：审计，检查没有钱损失
T1： T2：
开始-X 开始-X
添加(x, 1) tmp1 = 获取(x)
添加（y，-1） tmp2 = 获取（y）
END-X 打印 tmp1, tmp2
结束-X

什么是交易的正确行为?
通常称为“酸”
原子性——无论失败，都写入或不写入
一致——遵守特定于应用程序的不变量
隔离——xactions之间没有干扰——可序列化
持久——提交的写入是永久性的
ACID 事务很神奇！
程序员编写简单的串行代码
系统自动添加正确锁定！
系统自动添加容错！
我们对分布式事务的 ACID 感兴趣
数据分布在多个服务器上

可序列化是什么意思?
您执行一些并发事务，从而产生结果
“结果”意味着数据库中的输出和更改
如果满足以下条件，结果是可序列化的：
交易存在串行执行顺序
产生与实际执行相同的结果
（串行意味着一次一个——没有并行执行）
（这个定义应该让你想起线性化）

您可以通过以下方式测试执行结果是否可序列化
寻找产生相同结果的订单。
对于我们的示例，可能的序列订单是
T1；T2
T2；T1
所以正确的（可序列化的）结果是：
T1；T2：x=11 y=9“11,9”
T2；T1：x=11 y=9“10,10”
两者的结果不同；都可以
没有其他结果是好的
该实现可能并行执行 T1 和 T2
但它仍然必须产生结果，就像按顺序排列一样

如果 T1 的操作完全在 T2 的两个 get() 之间运行怎么办?
结果可以序列化吗?
T2 将打印 10,9
但 10,9 不是两个可序列化结果之一！
如果T2完全在T1的两个add()之间运行怎么办?
T2 将打印 11,10
但 11,10 不是两个可序列化结果之一！
如果 x 的服务器执行增量但 y 的服务器失败怎么办?
x=11 y=10 不是可序列化的结果之一！

为什么可串行化很受欢迎
- 程序员的简单模型
  - 他们可以编写复杂的事务而忽略并发性
- 它允许在不同记录上并行执行事务

如果出现问题，交易可以“中止”
- 中止取消任何记录修改
- 交易可能会主动中止，
  - 例如，如果帐户不存在，或者 y 的余额 <= 0
- 系统可能会强制中止，例如打破锁定死锁
- 服务器故障可能导致中止
- 中止的结果应该就像从未执行过一样！
  - 必须撤消或不应用更新
- 应用程序可能（或可能不会）再次尝试该事务

分布式事务有两个主要组成部分：
- 并发控制（以提供隔离/可串行性）
- 原子提交（尽管失败但仍提供原子性）

一、并发控制
- 并发事务的正确执行

事务的两类并发控制：
- 悲观：
  - 使用前锁定记录
  - 冲突导致延迟（等待锁）
- 乐观的：
  - 使用记录而不加锁
  - 提交检查读/写是否可序列化
  - 冲突导致中止+重试
  - 称为乐观并发控制（OCC）
- 如果冲突频繁，悲观情绪会更快
- 如果冲突很少，乐观会更快

今天：悲观并发控制
几周后：乐观并发控制（FaRM）

“两阶段锁定”是实现可串行性的一种方法
- 2PL 规则：
  - 事务在使用记录之前必须获取记录的锁
  - 事务必须保持其锁定，直到*之后*提交或中止

以 2PL 为例
- 假设T1和T2同时开始
- 交易系统根据需要自动获取锁
- 所以 T1/T2 中第一个使用 x 的将获得锁
- 另一个等待第一个完全完成
- 这禁止不可串行化的交错

细节：
- 每个数据库记录都有一个锁
- 如果是分布式的，锁通常存储在记录的服务器上
  - [图：客户端、服务器、记录、锁]
  - （但是两阶段锁定受分布影响不大）

正在执行的事务在第一次使用时根据需要获取锁
add() 和 get() 隐式获取记录的锁
END-X() 释放所有锁
所有锁都是独占的（对于本次讨论，没有读取器/写入器锁）
全称是"强严格两阶段锁定"
与线程锁定相关（例如Go的Mutex），但是：
程序员没有显式锁定
编程器供应 BEGIN-X/END-X
第一次使用每条记录时数据库自动锁定
交易结束时数据库自动解锁
DB 可以自动中止以解决死锁

为什么要持有锁直到提交/中止之后?
为什么不在唱片完成后立即发行呢?
由此产生的问题的示例：
假设 T2 在 get(x) 之后释放 x 的锁
然后 T1 可以在 T2 的 get() 之间执行
T2 将打印 10,9
不是可序列化的执行：既不是 T1;T2 也不是 T2;T1

两阶段锁定会产生死锁，例如
T1 T2
获取（x） 获取（y）
得到(y) 得到(x)
系统必须检测（周期？超时？）并中止事务

2PL 会禁止正确的（可序列化的）执行吗?
是的; 例子：
T1 T2
得到(x)
得到(x)
把(x,2)
把(x,1)
锁定将禁止这种交错
但结果 (x=1) 是可序列化的（与 T2;T1 相同）

问题：描述两相锁定产生的情况
比简单锁定更高的性能。简单锁定：锁定*every*
在*任何*使用之前记录；中止/提交后释放。

下一主题： 分布式事务与故障

分布式事务如何应对故障?
假设，对于我们的示例，x 和 y 位于不同的"工作"服务器上
假设 x 的服务器加 1，但 y 的服务器在减法之前崩溃了?
或者x的服务器加1，但y的服务器意识到该帐户不存在?
或者 x 和 y 都可以发挥自己的作用，但不确定另一个是否会?
这是一个难题!

我们想要"原子提交"：
一群计算机正在合作执行某项任务
每台计算机都有不同的角色
我们想要原子性：全部执行，或者不执行

挑战：失败、绩效

我们将研究一个称为“两阶段提交”的协议
由分布式数据库用于多服务器事务

那个设定
数据在多个服务器之间分片
事务在“事务协调器”（TC）上运行
对于每次读/写，TC都会向相关分片服务器发送RPC
每个人都是“参与者”
每个参与者管理其数据分片的锁
可能有很多并发事务，很多TC
TC为每笔交易分配唯一的交易ID（TID）
每条消息、每一条 xaction 状态都标记有 TID
为避免混淆

无失败的两阶段提交：
[时序图：TC、A、B]
TC 向 A、B 发送 put()、get()、&c RPC
A和B锁定记录。
修改是暂定的，在副本上，仅在提交时安装。
TC 到达事务结束。
TC向A和B发送PREPARE消息。
如果A能够承诺，
A 回答是。
那么A就处于“准备”状态。
否则，A 回答“否”。
B 也一样。
如果 A 和 B 都说 YES，则 TC 向 A 和 B 发送 COMMIT 消息。
如果 A 或 B 说“否”，TC 会发送 ABORT 消息。
如果 A/B 从 TC 收到 COMMIT 消息，则提交。
即他们将暂定记录复制到真实数据库。
并释放交易对其记录的锁定。
A/B 确认 COMMIT 消息。

为什么到目前为止这是正确的？
除非双方都同意，否则 A 或 B 都不能承诺。

如果 B 崩溃并重新启动怎么办？
如果B在崩溃之前发送YES，B必须记住（尽管崩溃）！
因为 A 可能已经收到 COMMIT 并提交了。
因此，即使在重新启动后，B 也必须能够提交（或不提交）。

因此参与者必须写入持久（磁盘上）状态：
B在说YES之前必须记住磁盘上的数据，包括修改过的数据。
如果 B 重新启动，并且磁盘显示 YES 但没有收到来自 TC 的 COMMIT，
B必须询问TC，或者等待TC重新发送。
同时，B 必须继续持有该事务的锁。
如果TC说COMMIT，B将修改的数据复制到真实数据。

如果 TC 崩溃并重新启动怎么办?
如果 TC 可能在崩溃前发送了 COMMIT，那么 TC 必须记住!
因为一名工人可能已经承诺了。
因此TC在发送COMMIT消息之前必须将COMMIT写入磁盘。
如果崩溃并重新启动，则重复 COMMIT，
或者如果参与者询问（即如果 A/B 没有收到 COMMIT 消息）。
参与者必须过滤掉重复的 COMMIT（使用 TID）。

如果 TC 从未从 B 处得到“是/否”怎么办?
也许B坠毁了并且没有恢复；也许网络坏了。
TC 可能会超时并中止（因为尚未发送任何 COMMIT 消息）。
好：允许服务器释放锁。

如果 B 在等待 TC 的 PREPARE 期间超时或崩溃怎么办?
B 尚未响应 PREPARE，因此 TC 无法决定提交
这样B就可以单方面中止，并释放锁
对未来准备回答“否”

如果 B 对 PREPARE 回答 YES，但没有收到 COMMIT 或 ABORT，该怎么办?
B可以单方面决定中止吗?
不！TC 可能两者都得到了“是”，
并向 A 发送 COMMIT，但在发送给 B 之前崩溃了。
那么 A 将提交，B 将中止：不正确。
B 也不能单方面承诺：
A 可能会投反对票。

所以：如果 B 投了赞成票，它必须“阻止”：等待 TC 的决定。

笔记：
提交/中止决定由单个实体（TC）做出。
这使得两阶段提交相对简单。
惩罚是 A/B 在投赞成票后必须等待 TC。

TC什么时候可以完全忘记已提交的事务?
如果它看到每个参与者对 COMMIT 的确认。
那么参与者就不需要再询问了。

参与者什么时候可以完全忘记已提交的交易?
在它确认 TC 的 COMMIT 消息后。
如果它再次提交，并且没有事务记录，
它必须已经承诺并忘记了，并且可以（再次）承认。

两阶段提交视角
当事务使用多个分片上的数据时，在分片数据库中使用
但它的名声却很不好：
慢：多轮消息
慢：磁盘写入
准备/提交交换上持有锁；阻止其他 xaction
TC崩溃会导致无限期阻塞，并持有锁
因此通常仅在单个小域中使用
例如，不在银行之间、不在航空公司之间、不在大范围内
更快的分布式事务是一个活跃的研究领域。

Raft 和两阶段提交解决不同的问题！
使用 Raft 通过复制获得高可用性
即在某些服务器崩溃时能够运行
服务器都做*相同的*事情
当每个参与者做不同的事情时使用 2PC
*所有人*都必须尽自己的一份力量
2PC 对可用性没有帮助
因为所有服务器都必须启动才能完成任务
Raft 并不能确保所有服务器都做某事
因为只要大多数人活着

如果您想要高可用性*和*原子提交怎么办？
这是一个计划。
[图表]
TC和服务器都应该使用Raft进行复制
在复制的服务之间运行两阶段提交
那么你就能容忍失败并仍然取得进步
您将在实验 4 中构建类似的东西来传输分片
Spanner 使用这种安排（我们将在几周后阅读）

---

http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html

英文原文

6.824 2022 Lecture 11: Distributed Transactions

Topics:
distributed transactions = concurrency control + atomic commit

where are we in the course?
so far, mostly distribution for fault tolerance
multiple servers trying to look like one reliable server
now, many servers for performance
split the data up (shard) over multiple servers, for parallelism
fine as long as clients use data items one at a time
what if application operation involves records in different shards?
failures? atomicity?
important problem, we'll see a number of approaches

what's the problem?
lots of data records, sharded on multiple servers, lots of clients
[diagram: clients, servers, data sharded by key]
client application actions often involve multiple reads and writes
bank transfer: debit and credit
vote: check if already voted, record vote, increment count
install bi-directional links in a social graph
we'd like to hide interleaving and failure from application writers

this problem first came up in databases
the traditional solution: transactions
programmer marks beginning/end of sequences of code as transactions
the system automatically provides good behavior

example transactions
  x and y are bank balances -- records in database tables
    on different servers (maybe at different banks)
    both start out as $10 T1 and T2 are transactions T1 : transfer$1 from x to y
    T2: audit, to check that no money is lost
  T1:   T2:
  BEGIN-X   BEGIN-X
    add(x, 1)   tmp1 = get(x)
    add(y, -1)   tmp2 = get(y)
  END-X   print tmp1, tmp2
                END-X

what is correct behavior for a transaction?
  usually called "ACID"
    Atomic -- all writes or none, despite failures
    Consistent -- obeys application-specific invariants
    Isolated -- no interference between xactions -- serializable
    Durable -- committed writes are permanent
  ACID transactions are magic!
    programmer writes straightforward serial code
    system automatically adds correct locking!
    system automatically adds fault tolerance!
  we're interested in ACID for distributed transactions
    with data sharded over multiple servers

What does serializable mean?
  you execute some concurrent transactions, which yield results
    "results" means both output and changes in the DB
  the results are serializable if:
    there exists a serial execution order of the transactions
    that yields the same results as the actual execution
  (serial means one at a time -- no parallel execution)
  (this definition should remind you of linearizability)

You can test whether an execution's result is serializable by
  looking for an order that yields the same results.
  for our example, the possible serial orders are
    T1; T2
    T2; T1
  so the correct (serializable) results are:
    T1; T2 : x=11 y=9 "11,9"
    T2; T1 : x=11 y=9 "10,10"
  the results for the two differ; either is OK
  no other result is OK
  the implementation might have executed T1 and T2 in parallel
    but it must still yield results as if in a serial order

what if T1's operations run entirely between T2's two get()s?
  would the result be serializable?
  T2 would print 10,9
  but 10,9 is not one of the two serializable results!
what if T2 runs entirely between T1's two adds()s?

T2 would print 11,10
but 11,10 is not one of the two serializable results!
what if x's server does the increment but y's server fails?
x=11 y=10 is not one of the serializable results!

Why serializability is popular
An easy model for programmers
They can write complex transactions while ignoring concurrency
It allows parallel execution of transactions on different records

a transaction can "abort" if something goes wrong
an abort un-does any record modifications
the transaction might voluntarily abort,
e.g. if the account doesn't exist, or y's balance is <= 0
the system may force an abort, e.g. to break a locking deadlock
server failure can result in abort
result of abort should be as if never executed!!!
must un-do, or not apply, updates
the application might (or might not) try the transaction again

distributed transactions have two big components:
concurrency control (to provide isolation/serializability)
atomic commit (to provide atomicity despite failure)

first, concurrency control
correct execution of concurrent transactions

two classes of concurrency control for transactions:
pessimistic:
lock records before use
conflicts cause delays (waiting for locks)
optimistic:
use records without locking
commit checks if reads/writes were serializable
conflict causes abort+retry
called Optimistic Concurrency Control (OCC)
pessimistic is faster if conflicts are frequent
optimistic is faster if conflicts are rare

today: pessimistic concurrency control
in a few weeks: optimistic concurrency control (FaRM)

"Two-phase locking" is one way to implement serializability
2PL rules:
a transaction must acquire a record's lock before using it
a transaction must hold its locks until *after* commit or abort

2PL for our example
suppose T1 and T2 start at the same time
the transaction system automatically acquires locks as needed
so first of T1/T2 to use x will get the lock
the other waits until the first completely finishes
this prohibits the non-serializable interleavings

details:
  each database record has a lock
  if distributed, the lock is typically stored at the record's server
    [diagram: clients, servers, records, locks]
    (but two-phase locking isn't affected much by distribution)
  an executing transaction acquires locks as needed, at the first use
    add() and get() implicitly acquires record's lock
    END-X() releases all locks
  all locks are exclusive (for this discussion, no reader/writer locks)
  the full name is "strong strict two-phase locking"
  related to thread locking (e.g. Go's Mutex), but:
    programmer doesn't explicitly lock
    programmer supplies BEGIN-X/END-X
    DB locks automatically, on first use of each record
    DB unlocks automatically, at transaction end
    DB may automatically abort to cure deadlock

Why hold locks until after commit/abort?
  why not release as soon as done with the record?
  example of a resulting problem:
    suppose T2 releases x's lock after get(x)
    T1 could then execute between T2's get()s
    T2 would print 10,9
    not a serializable execution: neither T1;T2 nor T2;T1

Two-phase locking can produce deadlock, e.g.
  T1   T2
  get(x) get(y)
  get(y) get(x)
The system must detect (cycles? timeout?) and abort a transaction

Could 2PL ever forbid a correct (serializable) execution?
  yes; example:
    T1   T2
    get(x)
              get(x)
              put(x,2)
    put(x,1)
  locking would forbid this interleaving
  but the result (x=1) is serializable (same as T2;T1)

The Question: describe a situation where Two-Phase Locking yields higher performance than Simple Locking. Simple locking: lock *every* record before *any* use; release after abort/commit.

Next topic: distributed transactions versus failures

how can distributed transactions cope with failures?
  suppose, for our example, x and y are on different "worker" servers
  suppose x's server adds 1, but y's crashes before subtracting?
  or x's server adds 1, but y's realizes the account doesn't exist?
  or x and y both can do their part, but aren't sure if the other will?
  it's a hard problem!

We want "atomic commit":
A bunch of computers are cooperating on some task
Each computer has a different role
We want atomicity: all execute, or none execute
Challenges: failures, performance

We're going to look at a protocol called "two-phase commit"
Used by distributed databases for multi-server transactions

The setting
Data is sharded among multiple servers
Transactions run on "transaction coordinators" (TCs)
For each read/write, TC sends RPC to relevant shard server
Each is a "participant"
Each participant manages locks for its shard of the data
There may be many concurrent transactions, many TCs
TC assigns unique transaction ID (TID) to each transaction
Every message, every piece of xaction state tagged with TID
To avoid confusion

Two-phase commit without failures:
[time diagram: TC, A, B]
TC sends put(), get(), &c RPCs to A, B
A and B lock records.
Modifications are tentative, on a copy, only installed if commit.
TC gets to the end of the transaction.
TC sends PREPARE messages to A and B.
If A is able to commit,
A responds YES.
then A is in "prepared" state.
otherwise, A responds NO.
Same for B.
If both A and B say YES, TC sends COMMIT messages to A and B.
If either A or B says NO, TC sends ABORT messages.
A/B commit if they get a COMMIT message from the TC.
I.e. they copy tentative records to the real DB.
And release the transaction's locks on their records.
A/B acknowledge COMMIT message.

Why is this correct so far?
Neither A or B can commit unless they both agreed.

What if B crashes and restarts?
If B sent YES before crash, B must remember (despite crash)!
Because A might have received a COMMIT and committed.
So B must be able to commit (or not) even after a reboot.

Thus participants must write persistent (on-disk) state:
B must remember on disk before saying YES, including modified data.
If B reboots, and disk says YES but didn't receive COMMIT from TC,
B must ask TC, or wait for TC to re-send.
And meanwhile, B must continue to hold the transaction's locks.
If TC says COMMIT, B copies modified data to real data.

What if TC crashes and restarts?
  If TC might have sent COMMIT before crash, TC must remember!
    Since one worker may already have committed.
  Thus TC must write COMMIT to disk before sending COMMIT msgs.
  And repeat COMMIT if it crashes and reboots,
    or if a participant asks (i.e. if A/B didn't get COMMIT msg).
  Participants must filter out duplicate COMMITs (using TID).

What if TC never gets a YES/NO from B?
  Perhaps B crashed and didn't recover; perhaps network is broken.
  TC can time out, and abort (since has not sent any COMMIT msgs).
  Good: allows servers to release locks.

What if B times out or crashes while waiting for PREPARE from TC?
  B has not yet responded to PREPARE, so TC can't have decided commit
  so B can unilaterally abort, and release locks
  respond NO to future PREPARE

What if B replied YES to PREPARE, but doesn't receive COMMIT or ABORT?
  Can B unilaterally decide to abort?
    No! TC might have gotten YES from both,
    and sent out COMMIT to A, but crashed before sending to B.
    So then A would commit and B would abort: incorrect.
  B can't unilaterally commit, either:
    A might have voted NO.

So: if B voted YES, it must "block": wait for TC decision.

Note:
  The commit/abort decision is made by a single entity -- the TC.
  This makes two-phase commit relatively straightforward.
  The penalty is that A/B, after voting YES, must wait for the TC.

When can TC completely forget about a committed transaction?
  If it sees an acknowledgement from every participant for the COMMIT.
  Then no participant will ever need to ask again.

When can participant completely forget about a committed transaction?
  After it acknowledges the TC's COMMIT message.
  If it gets another COMMIT, and has no record of the transaction,
    it must have already committed and forgotten, and can acknowledge (again).

Two-phase commit perspective
  Used in sharded DBs when a transaction uses data on multiple shards
  But it has a bad reputation:
    slow: multiple rounds of messages
    slow: disk writes
    locks are held over the prepare/commit exchanges; blocks other xactions
    TC crash can cause indefinite blocking, with locks held
  Thus usually used only in a single small domain
    E.g. not between banks, not between airlines, not over wide area
  Faster distributed transactions are an active research area.

Raft and two-phase commit solve different problems!
  Use Raft to get high availability by replicating
    i.e. to be able to operate when some servers are crashed
    the servers all do the *same* thing
  Use 2PC when each participant does something different
    And *all* of them must do their part
  2PC does not help availability
    since all servers must be up to get anything done
  Raft does not ensure that all servers do something
    since only a majority have to be alive

What if you want high availability *and* atomic commit?
  Here's one plan.
  [diagram]
  The TC and servers should each be replicated with Raft
  Run two-phase commit among the replicated services
  Then you can tolerate failures and still make progress
  You'll build something like this to transfer shards in Lab 4
  Spanner uses this arrangement (we'll read in a few weeks)

---

http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html

LEC 12 Cache Consistency：Frangipani

中文翻译

6.824 2022第12讲：鸡蛋花

Frangipani：可扩展的分布式文件系统
曼·李·泰卡斯
特殊安全计划 1997

我们为什么要读这篇论文?
  缓存一致性
  分布式事务
  分布式崩溃恢复
  以及三者之间的互动

总体设计是什么?
  网络文件系统
    与现有应用程序（文本编辑器等）透明地协作
    很像雅典娜的AFS
  [用户；工作站+鸡蛋花；网络; 花瓣]
  Petal：块存储服务；复制；条带+分片以提高性能
  鸡蛋花在 Petal 卖什么?
    目录、索引节点、文件内容块、空闲位图
    就像普通的硬盘文件系统一样
  Frangipani：去中心化文件服务；缓存以提高性能

预期用途是什么？

- 环境：具有合作工程师的单一实验室
  - 作者的研究实验室
  - 编程、文本处理、电子邮件等
- 办公室的工作站
- 大多数文件访问是用户自己的文件
- 需要在任何工作站之间共享任何文件
  - 用户/用户协作
  - 一个用户登录多个工作站
- 所以：
  - 常见情况是独占访问；想要快点
  - 但文件有时需要共享；希望这是正确的
- 这是撰写论文时的常见情况

为什么 Frangipani 的设计符合预期用途？

- 强一致性，这是人类对文件系统的期望
- 每个工作站中的缓存——回写
  - 所有更新最初仅应用在工作站的缓存中——速度很快
  - 包括例如创建文件、创建目录、重命名等
  - 如果所有内容都已缓存，则更新将在没有任何 RPC 的情况下继续进行
  - 因此文件系统代码必须驻留在工作站中，而不是服务器中
- 最复杂的是客户端，而不是共享的 Petal 服务器
  - 更多客户端 -> 更多 CPU 能力
  - 复杂的服务器是以前系统（例如 NFS、AFS）的严重瓶颈

Frangipani 工作站缓存中有什么？

- 如果 WS1 想要创建并写入 /grades 该怎么办？
- 从 Petal 读取 / 信息到 WS1 的缓存中
- 在缓存中添加“等级”条目
- 不要立即给花瓣回信！
  - 如果 WS1 想要做更多修改

挑战

- WS2 运行“ls /”或“cat /grades”
  - WS2 会看到 WS1 的写入吗？
  - 回写式缓存，因此 WS 的写入不在 Petal 中
  - 缓存使陈旧读取成为严重威胁
  - “连贯性”
- WS1 和 WS2 同时尝试创建 /a 和 /b
  - 在同一目录下
  - 他们会覆盖彼此对 / 的更改吗？
  - 没有中央文件服务器来解决这个问题！
  - “原子性”
- 创建文件时 WS1 崩溃
  - 许多步骤：分配i节点，初始化i节点，更新目录
  - 如果不完整就会留下一团糟
  - 如何确保其他工作站不会看到混乱的情况？如何清理？
  - “崩溃恢复”

“缓存一致性”解决了“读看到写”问题
目标是线性化和缓存
许多系统使用“缓存一致性协议”
多核、文件服务器、分布式共享内存

Frangipani 的一致性协议（简化）：
锁服务器（LS），每个文件/目录一个锁
文件所有者
---
xWS1
y WS1
工作站（WS）鸡蛋花缓存：
文件/目录锁定内容
---
x 忙...
你闲着...
如果 WS 持有锁，
忙碌：正在使用数据
空闲：持有锁但现在不使用缓存的数据
工作站规则：
除非你持有锁，否则不要缓存
获取锁，然后从 Petal 读取
写入 Petal，然后释放锁
一致性协议消息：
请求（WS -> LS）
授予（LS -> WS）
撤销（LS -> WS）
发布（WS -> LS）

锁由文件/目录命名（实际上是 i 编号），
虽然锁服务器实际上不理解任何东西
关于文件系统或 Petal。

示例：WS1 更改文件 z，然后 WS2 读取 z

WS1 LS WS2
读z
--请求(z)-->
所有者(z)=WS1
<--授予(z)---
(从 Petal 读取+缓存 z 数据)
(本地修改z)
(完成后，缓存锁处于状态)
读z
<--请求(z)--
<--撤销(z)--
(将修改后的 z 写入 Petal)
--释放(z)-->
所有者(z)=WS2
--授予(z)-->
(从 Petal 读取 z)

要点：
锁和规则强制读取查看上次写入
锁确保“最后写入”是明确定义的

笔记：
在工作站释放文件锁定之前，
它必须将修改后的文件内容以及元数据写入Petal
因此工作站获得文件内容和元数据的一致性
ie 文件读取查看最新内容

一致性优化
“空闲”状态已经是一种优化
Frangipani 具有共享读锁，以及独占写锁

下一个挑战：原子性
如果两个工作站尝试同时创建相同的文件怎么办？
部分完成的多重写入操作是否可见？
例如 file create 初始化 i-node，添加目录条目
例如重命名（两个名字都可见？都不可见？）

Frangipani 实现事务性文件系统操作：
操作对应于系统调用（创建文件、删除文件、重命名等）
WS 获取它将修改的所有文件系统数据的锁
在持有所有锁的情况下执行操作
仅在完成后释放
仅在整个操作完成后响应“撤销”
因此其他 WS 无法看到部分完成的操作
并且没有其他 WS 可以同时执行冲突的更新

注意 Frangipani 的锁正在做两种不同的事情：
缓存一致性（显示写入）
原子事务（隐藏写入）

下一个挑战：崩溃恢复

如果 Frangipani 工作站在持有锁时挂掉怎么办？
其他工作站将希望继续运行...
我们可以撤销死 WS 的锁吗？
如果失效的 WS 修改了其缓存中的数据怎么办？
如果死亡的 WS 开始将修改后的数据写回 Petal 会怎样？
例如 WS 将新的目录条目写入 Petal，但未初始化 i-node
这是令人不安的案例

是否可以等待崩溃的工作站重新启动？

Frangipani 使用预写日志记录进行崩溃恢复
在将任何操作的缓存块写入 Petal 之前，首先将日志写入 Petal
因此，如果崩溃的工作站已完成一些 Petal 写入操作，
但不是全部，写入可以从Petal中的日志完成
非常传统——但是.....
1）Frangipani每个工作站都有单独的日志
这避免了日志记录瓶颈，简化了去中心化
但将给定文件的更新分散到许多日志中
2）Frangipani的日志位于共享Petal存储中，而不是本地磁盘中

WS2 可以读取 WS1 的日志以从 WS1 崩溃中恢复
单独的日志是一种有趣且不寻常的安排

日志里有什么?
日志条目:
(这是一些猜测，论文并不明确)
日志序列号
更新数组:
块#，新版本#，地址，新字节
仅包含元数据更新，不包含文件内容更新
示例 -- 创建文件 d/f 生成一个日志条目:
两个条目的更新数组:
将"f"条目添加到 d 的内容块，并使用新的 i 编号
初始化 f 的 i 节点
最初日志条目位于 WS 本地内存中（还不是 Petal）

当 WS 从 LS 获取已修改目录的锁撤销时:
1）将其整个日志强制到Petal，然后
2）将缓存的更新块发送到Petal，然后
3）释放LS的锁

为什么WS在更新之前必须将日志写入Petal
Petal 中的 i 节点和目录 &c?

为什么要延迟写入日志直到LS撤销锁?

当 WS1 在持有锁时崩溃时会发生什么?
不多，直到 WS2 请求 WS1 持有的锁
LS 向 WS1 发送撤销，未得到响应
LS 超时，告诉 WS2 从 Petal 中的日志恢复 WS1
WS2 如何从 WS1 的日志中恢复?
从 Petal 读取 WS1 的日志
执行记录操作描述的花瓣写入
告诉LS已经完成，这样LS就可以释放WS1的锁

请注意，至关重要的是每个 WS 日志都位于 Petal 中，这样它就可以
可由任何 WS 读取以进行恢复。

如果 WS1 在将最近的日志写入 Petal 之前崩溃怎么办?
如果 WS1 崩溃，则 WS1 最近的操作可能会完全丢失。
但 Petal 中的文件系统将在内部保持一致。

尽管交错，为什么仅重放一个日志是安全的
其他工作站对相同文件的操作?
例子:
WS1：删除（d/f）崩溃
WS2：创建（d/f)
WS3：恢复WS1
WS3 正在恢复 WS1 的日志 — 但它不查看 WS2 的日志
恢复会重播删除吗?
这就是问题
否——被"版本号"机制阻止
Petal 中每个元数据块（i 节点）中的版本号
每个记录的操作中的版本号是块的版本加一

仅当操作版本 > 块版本时恢复才会重播
即仅当该块尚未被该操作更新时
WS3 是否需要获取 d 或 d/f 锁?
否：如果版本号与操作之前相同，则 WS1 没有
进行写入，因此它无法释放锁，
所以没有其他人可以拥有锁，所以
在 Petal 中更新 WS3 是安全的

日志不包含文件*内容*——后果是什么?
工作站可以按照不同的顺序向 Petal 发送内容写入
从应用程序调用 write 的顺序开始。
如果没有崩溃，这会被锁隐藏，所以没关系。
如果崩溃，Petal 可能会留下最近写入的随机子集。
对于许多应用程序来说，这并不重要，因为不太可能发生崩溃
无论如何，让程序输出保持有用的形式。
细心的程序（例如文本编辑器、数据库）使用fsync()。
此行为模仿具有本地磁盘的 Linux。

如果什么：
WS1 持有锁
网络分区
WS2 判定 WS1 已死亡，恢复并释放 WS1 的锁
如果 WS1 实际上还活着，它随后是否可以尝试写入锁覆盖的数据?
锁有租约!
锁拥有者不能使用超过租用期限的锁
LS 直到租约到期后才开始恢复

Paxos 或 Raft 是否隐藏在这里?
是——基于 Paxos 的配置管理器，用于锁服务器、Petal 服务器
尽管存在分区，但仍确保单个锁定服务器
确保每个 Petal 分片都有一个主分片

表现?
很难判断 1997 年的数字
它们是否达到硬件限制？磁盘黑白、净黑白
总吞吐量会随着硬件的增加而增加吗?
每个添加的 Frangipani 工作站是否贡献更多
容量以及更多负载?
我们可能关心哪些场景?
读/写大量小文件（例如阅读我的电子邮件）
读/写大文件

小文件性能——图5
X 轴是活动工作站的数量
每个工作站运行文件密集型基准测试
工作站使用不同的文件和目录
Y轴是单个工作站的完成时间
平坦意味着良好的扩展==没有明显的共享瓶颈
大概每个工作站都只使用自己的缓存
也许 Petal 的许多磁盘也能产生并行性能

大文件性能
每个磁盘：6 MB/秒
花瓣条纹不仅如此
7 个 Petal 服务器，每个 Petal 服务器 9 个磁盘
原始磁盘黑白为 336 MB/s，但通过 Petal 仅为 100 MB/s
单个 Frangipani 工作站，表 3
写入：15 MB/s——受网络链接限制
读取：10 MB/s——受到弱预取（？）的限制，可能是 15
许多鸡蛋花工作站
图 6——更多机器的读取规模良好
图 7 — 写入达到 Petal 的硬件限制（复制为 2 倍）

对于哪些工作负载，Frangipani 可能表现不佳?
大量读/写共享?
很多小文件?

花瓣细节
Petal 提供带有容错存储的 Frangipani
所以值得讨论
块读/写接口
与现有文件系统兼容
看起来像单个巨大的磁盘，但实际上有很多服务器和很多磁盘
大、高性能
条带状 64 KB 块
虚拟：64位稀疏地址空间，写入时分配
地址转换映射
主/备（一台备份服务器）
主数据库将每次写入发送到备份数据库
使用 Paxos 就每个 virt addr 范围的主地址达成一致
崩溃后恢复怎么样?
假设对是 S1+S2
S1 发生故障，S2 现在是唯一的服务器
S1 重启，但错过了很多更新
S2 会记住它写入的每个块的列表!
所以S1只需要读取这些块，而不是整个磁盘
记录
virt->phys 映射和漏写信息

局限性
对于程序员工作站等最有用，否则就没那么有用了
对于许多应用程序（例如网站）来说，文件系统并不是一个很好的 API
Frangipani 强制执行权限，因此工作站必须可信
因此 Athena 无法在 Athena 工作站上运行 Frangipani
鸡蛋花/花瓣分裂有点尴尬
两层日志
Petal 可能会接受来自“down”Frangipani 工作站的更新
比简单文件服务器更多的 RPC 消息

需要记住的想法
复杂的客户端共享简单的存储——也许是可扩展的
缓存一致性
分布式事务

分布式崩溃恢复
上述之间的相互作用

英文原文

6.824 2022 Lecture 12: Frangipani

Frangipani: A Scalable Distributed File System
Thekkath, Mann, Lee
SOSP 1997

why are we reading this paper?
  cache coherence
  distributed transactions
  distributed crash recovery
  and the interaction among the three

what's the overall design?
  a network file system
    works transparently with existing apps (text editors &c)
    much like Athena's AFS
  [users; workstations + Frangipani; network; petal]
  Petal: block storage service; replicated; striped+sharded for performance
  What does Frangipani store in Petal?
    directories, i-nodes, file content blocks, free bitmaps
    just like an ordinary hard disk file system
  Frangipani: decentralized file service; cache for performance

what's the intended use?
  environment: single lab with collaborating engineers
    the authors' research lab
    programming, text processing, e-mail, &c
  workstations in offices
  most file access is to user's own files
  need to potentially share any file among any workstations
    user/user collaboration
    one user logging into multiple workstations
  so:
    common case is exclusive access; want that to be fast
    but files sometimes need to be shared; want that to be correct
  this was a common scenario when the paper was written

why is Frangipani's design good for the intended use?
  strong consistency, which humans expect from a file system
  caching in each workstation -- write-back
    all updates initially applied just in workstation's cache -- fast
    including e.g. creating files, creating directories, rename, &c
    updates proceed without any RPCs if everything already cached
    so file system code must reside in the workstation, not server
  most complexity is in clients, not the shared Petal servers
    more clients -> more CPU power
    complex servers were a serious bottleneck in previous systems e.g. NFS, AFS

what's in the Frangipani workstation cache?
  what if WS1 wants to create and write /grades?
  read / information from Petal into WS1's cache
  add entry for "grades" just in the cache
  don't immediately write back to Petal!
    in case WS1 wants to do more modifications

challenges
  WS2 runs "ls /" or "cat /grades"
    will WS2 see WS1's writes?
    write-back cache, so WS's writes aren't in Petal
    caches make stale reads a serious threat
    "coherence"
  WS1 and WS2 concurrently try to create /a and /b
    in the same directory
    will they overwrite each others' changes to /?
    there's no central file server to sort this out!
    "atomicity"
  WS1 crashes while creating a file
    many steps: allocate i-node, initialze i-node, update directory
    leaves a mess if incomplete
    how to ensure other workstations don't see the mess? how to clean up?
    "crash recovery"

"cache coherence" solves the "read sees write" problem
  the goal is linearizability AND caching
  many systems use "cache coherence protocols"
    multi-core, file servers, distributed shared memory

Frangipani's coherence protocol (simplified):
  lock server (LS), with one lock per file/directory
    file owner
    ----
    x   WS1
    y   WS1
  workstation (WS) Frangipani cache:
    file/dir lock content
    ----
    x   busy ...
    y   idle ...
  if WS holds lock,
    busy: using data right now
    idle: holds lock but not using the cached data right now
  workstation rules:
    don't cache unless you hold the lock
    acquire lock, then read from Petal
    write to Petal, then release lock
  coherence protocol messages:
    request (WS -> LS)
    grant (LS -> WS)

revoke (LS -> WS)
release (WS -> LS)

the locks are named by files/directories (really i-numbers),
though the lock server doesn't actually understand anything
about file systems or Petal.

example: WS1 changes file z, then WS2 reads z

```
WS1                  LS          WS2
read z
 --request(z)-->
                  owner(z)=WS1
 <--grant(z)---
(read+cache z data from Petal)
(modify z locally)
(when done, cached lock in state)
                              read z
                   <--request(z)--
  <--revoke(z)--
(write modified z to Petal)
  --release(z)-->
                  owner(z)=WS2
                   --grant(z)-->
                           (read z from Petal)
```

the point:
locks and rules force reads to see last write
locks ensure that "last write" is well-defined

note:
before a workstation releases a lock on a file,
it must write modified file content to Petal, as well as meta-data
so workstations get coherence for file content, as well as meta-data
i.e. file reads see most recent content

coherence optimizations
the "idle" state is already an optimization
Frangipani has shared read locks, as well as exclusive write locks

next challenge: atomicity
what if two workstations try to create the same file at the same time?
are partially complete multi-write operations visible?
e.g. file create initializes i-node, adds directory entry
e.g. rename (both names visible? neither?)

Frangipani implements transactional file-system operations:
operation corresponds to a system call (create file, remove file, rename, &c)
WS acquires locks on all file system data that it will modify
performs operation with all locks held
only releases when finished
only responds to a "revoke" after entire operation is complete
thus no other WS can see partially-completed operations
and no other WS can simultaneously perform conflicting updates

note Frangipani's locks are doing two different things:
  cache coherence (revealing writes)
  atomic transactions (concealing writes)

next challenge: crash recovery

What if a Frangipani workstation dies while holding locks?
  other workstations will want to continue operating...
  can we just revoke dead WS's locks?
  what if dead WS had modified data in its cache?
  what if dead WS had started to write back modified data to Petal?
    e.g. WS wrote new directory entry to Petal, but not initialized i-node
    this is the troubling case

Is it OK to just wait until a crashed workstation reboots?

Frangipani uses write-ahead logging for crash recovery
  Before writing any of op's cached blocks to Petal, first write log to Petal
  So if a crashed workstation has done some Petal writes for an operation,
    but not all, the writes can be completed from the log in Petal
  Very traditional -- but...

    1. Frangipani has a separate log for each workstation
       this avoids a logging bottleneck, eases decentralization
       but scatters updates to a given file over many logs
    2. Frangipani's logs are in shared Petal storage, not local disk
       WS2 can read WS1's log to recover from WS1 crashing
         Separate logs is an interesting and unusual arrangement

What's in the log?
  log entry:
    (this is a bit of guess-work, paper isn't explicit)
    log sequence number
    array of updates:
      block #, new version #, addr, new bytes
    just contains meta-data updates, not file content updates
  example -- create file d/f produces a log entry:
    a two-entry update array:
      add an "f" entry to d's content block, with new i-number
      initialize the i-node for f
  initially the log entry is in WS local memory (not yet Petal)

When WS gets lock revocation on modified directory from LS:

    1. force its entire log to Petal, then
    2. send the cached updated blocks to Petal, then
    3. release the locks to the LS

Why must WS write log to Petal before updating
  i-node and directory &c in Petal?

Why delay writing the log until LS revokes locks?

What happens when WS1 crashes while holding locks?
  Not much, until WS2 requests a lock that WS1 holds
    LS sends revoke to WS1, gets no response
    LS times out, tells WS2 to recover WS1 from its log in Petal
  What does WS2 do to recover from WS1's log?
    Read WS1's log from Petal
    Perform Petal writes described by logged operations
    Tell LS it is done, so LS can release WS1's locks

Note it's crucal that each WS log is in Petal so that it can
  be read by any WS for recovery.

What if WS1 crashes before it writes its recent log to Petal?
  WS1's recent operations may be totally lost if WS1 crashes.
  But the file system in Petal will be internally consistent.

Why is it safe to replay just one log, despite interleaved
  operations on same files by other workstations?
Example:
  WS1: delete(d/f)   crash
  WS2:   create(d/f)
  WS3:   recover WS1
  WS3 is recovering WS1's log -- but it doesn't look at WS2's log
  Will recovery re-play the delete?
    This is The Question
    No -- prevented by "version number" mechanism
    Version number in each meta-data block (i-node) in Petal
    Version number(s) in each logged op is block's version plus one
    Recovery replays only if op's version > block version
      i.e. only if the block hasn't yet been updated by this op
  Does WS3 need to aquire the d or d/f lock?
    No: if version number same as before operation, WS1 didn't
        do the write, so it couldn't have released the lock,
        so no-one else could have the lock, so
        it's safe for WS3 safe to update in Petal

The log doesn't hold file *content* -- what are the consequences?
  Workstations may send content writes to Petal in an order different
    from the order in which application called write.
  If no crash, this is hidden by locks, so it doesn't matter.
  If crash, Petal may be left with a random subset of recent writes.
  For many apps, this doesn't matter, since a crash is unlikely
    to leave program output in a useful form anyway.
  Careful programs (e.g. text editor, database) use fsync().
  This behavior mimics e.g. Linux with a local disk.

What if:
  WS1 holds a lock
  Network partition
  WS2 decides WS1 is dead, recovers, releases WS1's locks
  If WS1 is actually alive, could it subsequently try to write data covered by the lock?
  Locks have leases!

Lock owner can't use a lock past its lease period
LS doesn't start recovery until after lease expires

Is Paxos or Raft hidden somewhere here?
Yes -- Paxos-based configuration managers for lock servers, Petal servers
ensures a single lock server, despite partition
ensures a single primary for each Petal shard

Performance?
hard to judge numbers from 1997
do they hit hardware limits? disk b/w, net b/w
does total throughput increase with more hardware?
does each added Frangipani workstation contribute more
capacity as well as more load?
what scenarios might we care about?
read/write lots of little files (e.g. reading my e-mail)
read/write huge files

Small file performance -- Figure 5
X axis is number of active workstations
each workstation runs a file-intensive benchmark
workstations use different files and directories
Y axis is completion time for a single workstation
flat implies good scaling == no significant shared bottleneck
presumably each workstation is just using its own cache
possibly Petal's many disks also yield parallel performance

Big file performance
each disk: 6 MB / sec
Petal stripes to get more than that
7 Petal servers, 9 disks per Petal server
336 MB/s raw disk b/w, but only 100 MB/s via Petal
a single Frangipani workstation, Table 3
write: 15 MB/s -- limited by network link
read: 10 MB/s -- limited by weak pre-fetch (?), could be 15
lots of Frangipani workstations
Figure 6 -- reads scale well with more machines
Figure 7 -- writes hit hardware limits of Petal (2x for replication)

For what workloads is Frangipani likely to have poor performance?
lots of read/write sharing?
lots of small files?

Petal details
Petal provides Frangipani w/ fault-tolerant storage
so it's worth discussing
block read/write interface
compatible with existing file systems
looks like single huge disk, but many servers and many many disks
big, high performance
striped, 64-KB blocks
virtual: 64-bit sparse address space, allocate on write
address translation map

primary/backup (one backup server)
primary sends each write to the backup
uses Paxos to agree on primary for each virt addr range
what about recovery after crash?
suppose pair is S1+S2
S1 fails, S2 is now sole server
S1 restarts, but has missed lots of updates
S2 remembers a list of every block it wrote!
so S1 only has to read those blocks, not entire disk
logging
virt->phys map and missed-write info

Limitations
Most useful for e.g. programmer workstations, not so much otherwise
A file system is not a great API for many applications, e.g. web site
Frangipani enforces permissions, so workstations must be trusted
so Athena couldn't run Frangipani on Athena workstations
Frangipani/Petal split is a little awkward
both layers log
Petal may accept updates from "down" Frangipani workstations
more RPC messages than a simple file server

Ideas to remember
complex clients sharing simple storage -- maybe scalable
cache coherence
distributed transactions
distributed crash recovery
interactions among the above

LEC 13 Spanner

中文翻译

6.824 2022第13讲：扳手

为什么要写这篇论文（Google Spanner，OSDI 2012）？
广域分布式事务的罕见例子。
非常理想。
但两阶段提交被认为太慢并且容易阻塞。
广域同步复制的罕见示例。
简洁的想法：
Paxos 上的两阶段提交。
快速 R/O 事务的同步时间。
Google 内部经常使用。

激励用例是什么？
Google F1 广告数据库（第 5.4 节）。
之前对许多 MySQL 和 BigTable DB 进行了分片；尴尬的。
需要：
更好的（同步）复制。
更灵活的分片。
跨分片交易。

工作负载以只读事务为主（表 6）。
需要强一致性。
外部一致性/线性化/串行化。

基本组织：
数据中心A：
“客户端”是网络服务器，例如 Gmail
数据被分片到多个服务器上：
是
新西兰
数据中心B：
有自己的本地客户
以及它自己的数据分片副本
是
新西兰
数据中心C：
相同的设置

由 Paxos 管理的复制；每个分片一个 Paxos 组。
副本位于不同的数据中心。
与 Raft 类似——每个 Paxos 组都有一个领导者。
与在实验室中一样，Paxos 复制操作日志。

为什么这样安排？
分片通过并行性允许巨大的总吞吐量。
不同城市的数据中心独立发生故障。
客户端可以快速读取本地副本！
这就是驱动设计的时间戳和 TrueTime 的原因。
可以将复制品放置在相关客户附近。
Paxos 仅需要多数——容忍慢速/远程副本。

面临哪些挑战？
本地副本上的数据可能不是最新的。
本地副本可能无法反映最新的 Paxos 写入！
一笔交易可能涉及多个分片 -> 多个 Paxos 组。
读取多条记录的事务必须是可序列化的。
但本地分片可能反映已提交事务的不同子集！

Spanner 以不同的方式处理读/写和读/只读事务。

首先，读/写事务。

读/写事务示例（银行转账）：
开始
x = x + 1
y = y - 1
结尾
我们不希望在两个操作之间偷偷地读取或写入 x 或 y。
提交后，所有读取都应该看到我们的更新。

摘要：带有 Paxos 复制参与者的两阶段提交 (2pc)。
（暂时省略时间戳。）
（这是针对读/写事务，而不是读/写事务。）
客户端选择一个唯一的交易 ID (TID)。

客户端将每次读取发送给相关分片的 Paxos 领导者 (2.1)。
每个分片首先获取相关记录的锁。
可能还得等。
每个分片有单独的锁表，位于分片领导者中。
读锁不通过 Paxos 复制，因此领导者失败 -> 中止。
客户端在提交之前将写入保持私有。
当客户端提交时（4.2.1）：
选择一个 Paxos 组作为 2pc 事务协调器 (TC)。
向相关分片领导发送写入。
每个书面分片领导者：
获取写入记录的锁。
通过 Paxos 记录"准备"记录，以复制锁和新值。
告诉TC它已经准备好了。
或者如果崩溃并因此丢失锁表，请告诉 TC"不"。
交易协调员：
决定提交或中止。
通过 Paxos 将决定记录给其小组。
将结果告诉参与者领导和客户。
各参会领导：
通过 Paxos 记录 TC 的决定。
执行其写入。
释放事务的锁。

到目前为止有关设计的一些要点（仅读/写事务）。
锁定（两阶段锁定）确保可串行化。
2pc 被广泛讨厌，因为如果 TC 失败，它就会阻塞并持有锁。
用Paxos复制TC就解决了这个问题！
r/w 事务需要很长时间。
许多数据中心间消息。
表 6 建议跨美国读/写事务大约需要 100 毫秒。
跨城市的情况则少得多（表 3）。
但是很多并行性：很多客户端，很多分片。
因此，如果繁忙，总吞吐量可能会很高。

从现在开始，我将主要将每个 Paxos 组视为一个实体。
复制分片数据。
复制两阶段提交状态。

现在用于只读 (r/o) 事务。
这些涉及多次读取，可能来自多个分片。
我们希望 r/o xactions 比 r/w xactions 快得多！

Spanner 消除了 R/O 交易的两大成本：
从本地副本读取，以避免 Paxos 和跨数据中心消息。
但请注意，本地副本可能不是最新的！
没有锁，没有两阶段提交，没有事务管理器。
再次避免向 Paxos Leader 发送跨数据中心消息。
并避免减慢读写事务。
表 3 和表 6 显示延迟改善了 10 倍！
这是一个大问题。
如何将其与正确性相平方？

r/o 事务的正确性约束：
可序列化：
与一笔一笔地执行交易的结果相同。
即使它们实际上可能同时执行。
即，r/o xaction 必须基本上适合 r/w xactions。
查看之前事务的所有写入，但不查看后续事务的所有写入。
即使与 r/w xactions *并发*！而且不锁！
外部一致：
如果 T1 在 T2 开始之前完成，则 T2 必须看到 T1 的写入。
“之前”指的是真实（挂钟）时间。
与线性化类似。
排除读取陈旧数据的可能性。

为什么不让 r/o 事务只读取最新提交的值？
假设我们有两笔银行转账，以及一个读取两者的交易。
T1：Wx Wy C
T2：Wx Wy C
T3：接收 Ry
结果与任何序列顺序都不匹配！
不是T1、T2、T3。
不是T1、T3、T2。
我们希望 T3 能够看到 T2 的两个写入，或者一个也看不到。
我们希望 T3 的读取*全部*发生在相对于 T1/T2 的*相同*点。

想法：快照隔离（SI）：
同步所有计算机的时钟（与真实的挂钟时间）。
为每笔交易分配一个时间戳。
r/w：提交时间。
r/o：开始时间。
我们希望结果就像按时间戳顺序一次一个一样。
即使实际读取以不同的顺序发生。
每个副本存储每个记录的多个时间戳版本。
所有 ar/w 事务的写入都具有相同的时间戳。
r/o 事务的读取会看到截至 xaction 时间戳的版本。
具有小于 xaction 的最高时间戳的记录版本。
称为快照隔离。

我们的快照隔离示例：
x@10=9 x@20=8
y@10=11 y@20=12
T1 @ 10：Wx Wy C
T2 @ 20：Wx Wy C
T3 @ 15：Rx Ry
“@10”表示时间戳。
现在 T3 的读取都将从 @10 版本提供。
即使 T3 对 y 的读取发生在 T2 之后，T3 也不会看到 T2 的写入。
现在结果可序列化：T1 T2 T3
序列顺序与时间戳顺序相同。

为什么 T3 可以读取 y 的*旧*值，即使有更新的值？
T2 和 T3 是并发的，因此外部一致性允许任一顺序。
请记住：r/o 事务需要读取值
截至其时间戳，*不*见稍后的写入。

问题：如果 T3 从未见过 T1 写入的副本中读取 x 会怎样？
因为副本不在 Paxos 中占多数？

解决方案：复制“安全时间”。
Paxos 领导者按时间戳顺序发送写入。
在时间 20 提供读取服务之前，副本必须看到 Paxos 写入时间 > 20。
所以它知道它已经看到所有写入 < 20。
如果已准备但未提交的事务也必须延迟（第 4.1.3 节）。
因此：r/o 事务可以从本地副本读取——通常速度很快。

问题：如果时钟不完全同步怎么办？

如果时钟不完全同步会出现什么问题？
对于使用锁的读/写事务没有问题。
如果 r/o 事务的 TS 太大：
它的 TS 将高于副本安全时间，并且读取会阻塞。
正确但缓慢——延迟因时钟误差而增加。
如果 r/o 事务的 TS 太小：
它将错过 r/o xaction 开始之前提交的写入。
由于它的低 TS 会导致它使用旧版本的记录。
这违反了外部一致性。

如果 r/o xaction 的 TS 太小，问题示例：
r/w T0 @ 0：Wx1 C
读/写 T1 @ 10：Wx2 C
r/o T2 @ 5：接收？
（C 代表提交）
这将导致 T2 在时间 0 读取 x 的版本，即 1。
但 T2 在 T1 提交后开始（实时），
因此外部一致性要求 T2 看到 x=2。
所以我们需要一种方法来处理不正确的时钟！

[计算机]时钟出了什么问题？
它们会漂移——它们不会以每秒一秒的速度前进。
你可以用更好的时钟来设置它们，
但由于通信延迟的变化，这只是近似值。

Google 的时间参考系统（第 5.3 节）
[UTC、GPS卫星、主站、服务器、TT间隔]
每个数据中心有几个主服务器。
每个时间主机都有一个 GPS 接收器或一个“原子钟”。
GPS 接收器的精确度通常优于一微秒。
该论文没有说明原子钟的含义。
可能与 GPS 同步，但在没有 GPS 的情况下也能保持一段时间的准确性。
如果惯性滑行，错误会累积，可能每周会累积微秒。
其他服务器与附近的一些时间大师交谈。
由于网络延迟、检查之间的漂移而产生的不确定性。

真实时间
 时间服务产生 TTinterval = [最早，最晚]。
 保证正确的时间位于该间隔内的某个位置。
 根据测量的网络延迟计算出的间隔宽度，
  时钟漂移假设、GPS 规格。
 图 6：间隔通常为微秒，但有时为 10+ 毫秒。
 所以：服务器时钟并不完全同步，而是 TrueTime
  限制服务器时钟的错误程度。

Spanner 如何确保如果 r/w T1 在 r/o T2 开始之前完成，则 TS1 < TS2。
 即 r/o 事务时间戳不会太小，
  以便他们遵守外部一致性。
 两条规则（4.1.2）：
 启动规则：
  xaction TS = TT.now().latest
   对于 r/o，在开始时
   对于读/写，提交开始时
 提交等待，用于 r/w xaction：
  在完成提交之前，延迟直到 TS < TS.now().earliest
  保证TS已经通过。

使用间隔和提交等待更新的示例：
 场景是T1提交，然后T2启动，T2必须看到T1的写入。
 即我们需要 TS1 < TS2。
 r/w T0 @ 1：Wx1 C
             |1----------10| |11----------20|
 读/写 T1 @ 10：Wx2 PC
                      |10--------12|
 r/o T2 @ 12：接收?
 （P代表T1的Prepare，C代表T1的完成Commit）
在 P 处，T1 选择 TS1 = TT.now().latest = 10
提交等待强制 C 在 TS1 之后发生。
假设 T2 在 C 之后开始，因此在时间 10 之后开始。
T2选择TT.now().latest，它在当前时间之后，即10点之后。
所以 TS2 > TS1。
所以T2的Rx看到T1的Wx。

为什么这为 r/o 事务提供了外部一致性：
 鉴于 T1 在 T2 开始之前完成。
 提交等待意味着 TS1 肯定会成为过去。
 TS2 = TT.now().latest 保证 >= 正确时间
  因此 >= 任何先前提交的事务的 TS（由于其提交等待）

更普遍：
 快照隔离为您提供可序列化的 r/o 事务。
  时间戳设定了顺序。
  快照版本（和安全时间）在时间戳上实现一致的读取。
  Xaction 看到来自较低 TS xaction 的所有写入，没有看到来自较高 TS xaction 的写入。
  如果您不关心外部一致性，任何数字都可以用于 TS。
 同步时间戳产生外部一致性。
  即使在不同数据中心的交易之间也是如此。
  即使从本地副本读取可能会滞后。

为什么这一切都有用?
 快速滚转交易:
  从本地副本读取!
   即使副本已过时也要正确。
  没有锁!
  因此，表 3 和表 6 中的延迟改善了 10 倍。
 虽然:
  r/o 事务读取可能会由于安全时间而阻塞，以赶上。
  r/w 事务提交可能会在提交等待中阻塞。
  准确的（小间隔）时间可以最大限度地减少这些延迟。

概括:
 很少看到已部署的系统提供分布式事务
  地理分布的数据。
 Spanner 令人惊讶地证明了它的实用性。
 时间戳方案是最有趣的方面。
 在Google内部广泛使用；商业 Google 服务；有影响。

Spanner：成为 SQL 系统，2017 年，https://research.google/pubs/pub46103.pdf

英文原文

6.824 2022 Lecture 13: Spanner

Why this paper (Google Spanner, OSDI 2012)?
 A rare example of wide-area distributed transactions.
  Very desirable.
  But two-phase commit viewed as too slow and prone to blocking.
 A rare example of wide-area synchronous replication.
 Neat ideas:
  Two-phase commit over Paxos.
  Synchronized time for fast r/o transactions.
 Used a lot inside Google.

What was the motivating use case?
 Google F1 advertising database (Section 5.4).
 Previously sharded over many MySQL and BigTable DBs; awkward.
 Needed:
  Better (synchronous) replication.
  More flexible sharding.
  Cross-shard transactions.
 Workload is dominated by read-only transactions (Table 6).
 Strong consistency is required.
  External consistency / linearizability / serializability.

The basic organization:
 Datacenter A:
  "clients" are web servers e.g. for gmail
  data is sharded over multiple servers:
   a-m
   n-z
 Datacenter B:

  has its own local clients
  and its own copy of the data shards
    a-m
    n-z
 Datacenter C:
  same setup

Replication managed by Paxos; one Paxos group per shard.
 Replicas are in different data centers.
 Similar to Raft -- each Paxos group has a leader.
 As in the labs, Paxos replicates a log of operations.

Why this arrangement?
 Sharding allows huge total throughput via parallelism.
 Datacenters fail independently -- different cities.
 Clients can read local replica -- fast!
   This is what's driving the design's timestamps and TrueTime.
 Can place replicas near relevant customers.
 Paxos requires only a majority -- tolerate slow/distant replicas.

What are the challenges?
 Data on local replica may not be fresh.
   Local replica may not reflect latest Paxos writes!
 A transaction may involve multiple shards -> multiple Paxos groups.
 Transactions that read multiple records must be serializable.
   But local shards may reflect different subsets of committed transactions!

Spanner treats read/write and read/only transactions differently.

First, read/write transactions.

Example read/write transaction (bank transfer):
 BEGIN
   x = x + 1
   y = y - 1
 END
We don't want any read or write of x or y sneaking between our two ops.
After commit, all reads should see our updates.

Summary: two-phase commit (2pc) with Paxos-replicated participants.
 (Omitting timestamps for now.)
 (This is for r/w transactions, not r/o.)
 Client picks a unique transaction id (TID).
 Client sends each read to Paxos leader of relevant shard (2.1).
   Each shard first acquires a lock on the relevant record.
     May have to wait.
   Separate lock table per shard, in shard leader.
   Read locks are not replicated via Paxos, so leader failure -> abort.
 Client keeps writes private until commit.
 When client commits (4.2.1):
   Chooses a Paxos group to act as 2pc Transaction Coordinator (TC).
   Sends writes to relevant shard leaders.
   Each written shard leader:

Acquires lock(s) on the written record(s).
Log a "prepare" record via Paxos, to replicate lock and new value.
Tell TC it is prepared.
Or tell TC "no" if crashed and thus lost lock table.
Transaction Coordinator:
Decides commit or abort.
Logs the decision to its group via Paxos.
Tell participant leaders and client the result.
Each participant leader:
Log the TC's decision via Paxos.
Perform its writes.
Release the transaction's locks.

Some points about the design so far (just read/write transactions).
Locking (two-phase locking) ensures serializability.
2pc widely hated b/c it blocks with locks held if TC fails.
Replicating the TC with Paxos solves this problem!
r/w transactions take a long time.
Many inter-data-center messages.
Table 6 suggests about 100 ms for cross-USA r/w transaction.
Much less for cross-city (Table 3).
But lots of parallelism: many clients, many shards.
So total throughput could be high if busy.

From now on I'll mostly view each Paxos group as a single entity.
Replicates shard data.
Replicates two-phase commit state.

Now for read-only (r/o) transactions.
These involve multiple reads, perhaps from multiple shards.
We'd like r/o xactions to be much faster than r/w xactions!

Spanner eliminates two big costs for r/o transactions:
Read from local replicas, to avoid Paxos and cross-datacenter msgs.
But note local replica may not be up to date!
No locks, no two-phase commit, no transaction manager.
Again to avoid cross-data center msg to Paxos leader.
And to avoid slowing down r/w transactions.
Tables 3 and 6 show a 10x latency improvement as a result!
This is a big deal.
How to square this with correctness?

Correctness constraints on r/o transactions:
Serializable:
Same results as if transactions executed one-by-one.
Even though they may actually execute concurrently.
I.e. an r/o xaction must essentially fit between r/w xactions.
See all writes from prior transactions, nothing from subsequent.
Even though *concurrent* with r/w xactions! And not locking!
Externally consistent:
If T1 completes before T2 starts, T2 must see T1's writes.
"Before" refers to real (wall-clock) time.

Similar to linearizable.
Rules out reading stale data.

Why not have r/o transactions just read the latest committed values?
Suppose we have two bank transfers, and a transaction that reads both.
T1: Wx Wy C
T2: Wx Wy C
T3: Rx Ry
The results won't match any serial order!
Not T1, T2, T3.
Not T1, T3, T2.
We want T3 to see both of T2's writes, or none.
We want T3's reads to *all* occur at the *same* point relative to T1/T2.

Idea: Snapshot Isolation (SI):
Synchronize all computers' clocks (to real wall-clock time).
Assign every transaction a time-stamp.
r/w: commit time.
r/o: start time.
We want results as if one-at-a-time in time-stamp order.
Even if actual reads occur in different order.
Each replica stores multiple time-stamped versions of each record.
All of a r/w transactions's writes get the same time-stamp.
An r/o transaction's reads see version as of xaction's time-stamp.
The record version with the highest time-stamp less than the xaction's.
Called Snapshot Isolation.

Our example with Snapshot Isolation:
x@10=9 x@20=8
y@10=11 y@20=12
T1 @ 10: Wx Wy C
T2 @ 20: Wx Wy C
T3 @ 15: Rx Ry
"@ 10" indicates the time-stamp.
Now T3's reads will both be served from the @10 versions.
T3 won't see T2's write even though T3's read of y occurs after T2.
Now the results are serializable: T1 T2 T3
The serial order is the same as time-stamp order.

Why OK for T3 to read the *old* value of y even though there's a newer value?
T2 and T3 are concurrent, so external consistency allows either order.
Remember: r/o transactions need to read values
as of their timestamp, and *not* see later writes.

Problem: what if T3 reads x from a replica that hasn't seen T1's write?
Because the replica wasn't in the Paxos majority?

Solution: replica "safe time".
Paxos leaders send writes in timestamp order.
Before serving a read at time 20, replica must see Paxos write for time > 20.
So it knows it has seen all writes < 20.
Must also delay if prepared but uncommitted transactions (Section 4.1.3).
Thus: r/o transactions can read from local replica -- usually fast.

Problem: what if clocks are not perfectly synchronized?

What goes wrong if clocks aren't synchronized exactly?
  No problem for r/w transactions, which use locks.
  If an r/o transaction's TS is too large:
    Its TS will be higher than replica safe times, and reads will block.
    Correct but slow -- delay increased by amount of clock error.
  If an r/o transaction's TS is too small:
    It will miss writes that committed before the r/o xaction started.
    Since its low TS will cause it to use old versions of records.
    This violates external consistency.

Example of problem if r/o xaction's TS is too small:
  r/w T0 @ 0: Wx1 C
  r/w T1 @ 10:   Wx2 C
  r/o T2 @ 5:   Rx?
(C for commit)
This would cause T2 to read the version of x at time 0, which was 1.
But T2 started after T1 committed (in real time),
  so external consistency requires that T2 see x=2.
So we need a way to deal with incorrect clocks!

What goes wrong with [computer] clocks?
  They drift -- they don't advance at exactly one second per second.
  You can set them from a better clock,
    but only approximately, due to variable communication delays.

Google's time reference system (Section 5.3)
  [UTC, GPS satellites, masters, servers, TTinterval]
  A few time master servers per data center.
  Each time master has either a GPS receiver or an "atomic clock".
  GPS receivers are typically accurate to better than a microsecond.
  The paper doesn't say what it means by an atomic clock.
    Probably synced to GPS, but accurate for a while w/o GPS.
    If coasting, error accumulates, maybe microseconds per week.
  Other servers talk to a few nearby time masters.
    Uncertainty due to network delays, drift between checks.

TrueTime
  Time service yields a TTinterval = [earliest, latest].
  The correct time is guaranteed to be somewhere in the interval.
  Interval width computed from measured network delays,
    clock drift assumptions, GPS specs.
  Figure 6: intervals are usually microseconds, but sometimes 10+ milliseconds.
  So: server clocks aren't exactly synchronized, but TrueTime
    bounds how wrong a server's clock can be.

How Spanner ensures that if r/w T1 finishes before r/o T2 starts, TS1 < TS2.
  I.e. that r/o transaction timestamps are not too small,
    so that they obey external consistency.
  Two rules (4.1.2):
  Start rule:
    xaction TS = TT.now().latest

for r/o, at start time
for r/w, when commit begins
Commit wait, for r/w xaction:
Before completing commit, delay until TS < TS.now().earliest
Guarantees that TS has passed.

Example updated with intervals and commit wait:
The scenario is T1 commits, then T2 starts, T2 must see T1's writes.
I.e. we need TS1 < TS2.

```
r/w T0 @  1: Wx1 C
             |1----------10| |11----------20|
r/w T1 @ 10:       Wx2 P        C
                   |10--------12|
r/o T2 @ 12:                 Rx?
```

(P for T1's Prepare, C for T1 finishing Commit)
At P, T1 chooses TS1 = TT.now().latest = 10
Commit Wait forces C to occur after TS1.
T2 starts after C by assumption, and thus after time 10.
T2 chooses TT.now().latest, which is after current time, which is after 10.
So TS2 > TS1.
So T2's Rx sees T1's Wx.

Why this provides external consistency for r/o transactions:
Given that T1 finishes before T2 starts.
Commit wait means TS1 is guaranteed to be in the past.
TS2 = TT.now().latest is guaranteed to be >= correct time
thus >= TS of any previous committed transaction (due to its commit wait)

More generally:
Snapshot Isolation gives you serializable r/o transactions.
Timestamps set an order.
Snapshot versions (and safe time) implement consistent reads at a timestamp.
Xaction sees all writes from lower-TS xactions, none from higher.
Any number will do for TS if you don't care about external consistency.
Synchronized timestamps yield external consistency.
Even among transactions at different data centers.
Even though reading from local replicas that might lag.

Why is all this useful?
Fast r/o transactions:
Read from local replicas!
Correct even if replica is stale.
No locks!
Thus the 10x latency improvement in Tables 3 and 6.
Although:
r/o transaction reads may block due to safe time, to catch up.
r/w transaction commits may block in Commit Wait.
Accurate (small interval) time minimizes these delays.

Summary:
Rare to see deployed systems offer distributed transactions
over geographically distributed data.
Spanner was a surprising demonstration that it can be practical.
Timestamping scheme is the most interesting aspect.
Widely used within Google; a commercial Google service; influential.

Spanner: Becoming a SQL System, 2017, https://research.google/pubs/pub46103.pdf

LEC 14 Optimistic Concurrency Control

中文翻译

6.824 2022年第14讲：FaRM，乐观并发控制

我们为什么要阅读 FaRM?
交易+复制+分片的另一种方式
这仍然是一个开放的研究领域!
乐观并发控制
充分利用 RDMA NIC 的巨大性能潜力

整体设置
全部集中在一个数据中心
配置管理器使用 ZooKeeper 选择主备/备份
带主/备份复制的分片
P1 B1
P2 B2
...
只要每个分片至少有一个副本就可以恢复
即 f+1 个副本可以容忍 f 个故障
事务客户端（它们在服务器中运行）
事务代码充当两阶段提交事务协调器（TC）

目标:
每秒数百万个分布式事务
所以时间预算是几十微秒
非常具有挑战性!

他们如何获得高性能?
在许多服务器上进行分片（评估中为 90）
数据必须适合总 RAM（因此不会读取磁盘）
非易失性 RAM（因此不会写入磁盘）
单侧 RDMA（快速跨网络访问 RAM）
快速用户级访问 NIC
利用单边 RDMA 的事务+复制协议

NVRAM（非易失性 RAM）
FaRM 写入到 RAM，而不是磁盘——消除了巨大的瓶颈
RAM写入需要200 ns，硬盘写入需要10 ms，SSD写入需要100 us
ns = 纳秒、ms = 毫秒、us = 微秒
但RAM在断电时会丢失内容！本身并不持久。
为什么不直接写入 f+1 台机器的 RAM，以容忍 f 次故障?

如果失败总是独立的，可能就足够了
但电源故障并不是独立的——可能会影响 100% 的机器！
所以：
每个机架中都有电池，可以运行机器几分钟
当主电源出现故障时，电源硬件通知软件
软件停止所有事务处理
s/w 将 FaRM 的 RAM 写入 SSD；可能需要几分钟
然后机器关闭
重新启动时，FaRM 从 SSD 读取保存的内存映像
“非易失性 RAM”
如果崩溃阻止软件写入 SSD 怎么办？
例如 FaRM 或内核中的错误，或 cpu/内存/硬件错误
FaRM 通过复制应对单机崩溃
崩溃（除了电源故障）必须是独立的！
概括：
NVRAM 消除持久性写入瓶颈
网络和 CPU 成为剩余瓶颈

为什么网络常常成为性能瓶颈？
FaRM 假设单一数据中心，因此光速延迟较低
但网络数据处理的CPU消耗往往很大！
LAN 上的 RPC over TCP 的常用设置：
应用程序应用程序
--- ---
套接字缓冲区 缓冲区
TCP TCP
网卡驱动程序
网卡 ---------- 网卡
许多昂贵的 CPU 操作：
系统调用
复制消息
中断
上下文切换
慢的：
构建 RPC 比每秒交付超过几十万个任务更困难
有线黑白（例如 10 GB/秒）很少是短 RPC 的限制
每个数据包的 CPU 成本传统上限制了小消息的性能

FaRM 使用两种网络思想：
内核绕过
RDMA

内核绕过
[图：FaRM 用户程序、CPU 内核、DMA 队列、NIC]
应用程序直接与 NIC 交互——无系统调用，无内核
NIC DMA 进/出用户 RAM
FaRM 软件轮询 DMA 区域以检查传入消息
NIC 轮询 DMA 区域以检查传出消息

RDMA（远程直接内存访问）
[src主机、网卡、交换机、网卡、目标内存、目标CPU]
远程网卡直接读写内存
发送者提供内存地址
不涉及远程CPU！
这是“单方面的RDMA”
以原子方式读取整个缓存行
RDMA NIC 使用可靠的协议，带有 ACK
一台服务器吞吐量：10+百万/秒（图2）
延迟：5 微秒（来自他们的 NSDI 2014 论文）

如果客户端可以直接访问，性能将是惊人的
通过单侧 RDMA 将 DB 记录存储在服务器上！

如何将单边 RDMA 与复制和事务结合起来？
到目前为止我们看到的协议都需要服务器的主动参与。
例如检查和设置锁，
检查租约，
指示何时安全持续。
不能立即与单向 RDMA 兼容。

事务的两类并发控制：
悲观（两阶段锁定）：
等待第一次使用对象时的锁定；保持直到提交/中止
冲突导致延误
乐观的：
不加锁读取对象
在提交之前不要安装写入
提交“验证”以查看其他 xaction 是否冲突
有效：提交写入
无效：中止
称为乐观并发控制（OCC）

FaRM 使用 OCC
原因：
OCC 让 FaRM 使用单侧 RDMA 读取进行读取
所以服务器不需要主动参与读取
FaRM 的 OCC 如何验证？稍后我们将查看图 4。

FaRM交易API（简化）：
txCreate()
o = txRead(oid) -- RDMA
+= 1 的
txWrite(oid, o) -- 纯本地
ok = txCommit()——图 4

什么是 oid？
<区域#，地址>
Region # 将映射索引到 [primary, backup1, ...]
目标RDMA NIC使用地址直接读取RAM

服务器内存布局
区域，每个区域都是一个对象数组
对象布局
带有版本号的标头，以及版本号高位的锁定标志
对于每个其他服务器
传入日志
传入消息队列
（发送者通过 RDMA 写入，本地 FaRM 通过轮询读取）
所有这些都在非易失性 RAM 中（即在电源故障时写入 SSD）

图4：事务执行/提交协议
让我们逐一考虑图 4 中的步骤
关注并发控制（而不是容错）

执行阶段
TC（客户端）从服务器读取它需要的对象
包括它将写入的记录
使用单侧 RDMA 读取
不带锁
这就是乐观并发控制中的乐观
TC 记住版本号
TC 缓冲区写入本地

现在 TC 承诺；两大目标：
原子分布式提交——全部写入或无写入
可序列化——就好像完全在每个其他事务之前或之后

LOCK（提交协议中的第一条消息）
TC 发送到每个写入对象的主节点
TC 使用 RDMA 附加到每个主节点的日志中
LOCK 记录包含 oid、版本 # xaction 读取、新值
LOCK 现在已记录在主设备的 NVRAM 中
能在停电时幸存下来
LOCK 消息既是预读日志条目，
以及对主节点的 RPC 请求

Primary 收到 LOCK 后会做什么？
FaRM 软件轮询 RAM 中的传入日志，查看我们的 LOCK
如果对象被锁定，或者版本！= xaction 读取的内容，
向 TC 发送“否”回复
否则设置对象的锁定标志并回复“是”
但还不要修改数据！
锁检查、版本检查和锁设置都是原子的
使用原子比较和交换指令
“锁定”标志是对象版本号中的高位
如果其他 CPU 也在处理 LOCK，或者客户端正在使用 RDMA 读取

TC等待所有LOCK回复消息
如果有任何“否”，则中止
将 ABORT 附加到主节点的日志中，以便它们可以释放锁定
从 txCommit() 返回“否”

现在让我们忽略 VALIDATE 和 COMMIT BACKUP

此时初选需要知道 TC 的决定

TC 将 COMMIT-PRIMARY 附加到主节点的日志中
 TC只等待RDMA硬件确认（ack）
  不等待主进程处理日志条目
  硬件确认意味着主 NVRAM 中是安全的
 TC 从 txCommit() 返回“yes”

当主程序在其日志中处理 COMMIT-PRIMARY 时：
 将新值复制到对象的内存中
 增加对象的版本#
 清除对象的锁定标志

提交点是第一个 COMMIT-PRIMARY 被写入时
 因为那时可以透露交易结果

例子：
 T1 和 T2 都想增加 x
  x = x + 1
 可串行化允许什么结果？
  即如果一次运行一个可能会出现什么结果？
  x = 2，两个客户都告诉“成功”
  x = 1，一名客户告知“成功”，其他客户告知“中止”
  x = 0，两个客户端都被告知“已中止”

如果 T1 和 T2 完全同步怎么办？
 T1：Rx0 Lx Cx
 T2：Rx0 Lx Cx
 会发生什么？

或者
 T1：Rx0 Lx Cx
 T2：Rx0 Lx Cx

或者
 T1：Rx0 Lx Cx
 T2：Rx0 Lx Cx

关于 FaRM 的 OCC 提供可序列化性的原因的直觉：
 即检查“执行是否与一次执行相同？”
 如果没有冲突的交易：
  版本不会改变
 如果存在冲突交易：
  一个或另一个将看到锁定或更改的版本#

图 4 中的 VALIDATE 怎么样？
 它是对事务刚刚读取的对象的优化
 VALIDATE = 单边 RDMA 读取以重新获取对象的版本 # 和锁定标志
 如果锁定设置，或版本号自读取后发生更改，TC 将中止
 不设置锁，因此比 LOCK+COMMIT 更快

验证示例：
x 和 y 最初为零
T1：
 如果 x == 0：

  y = 1
T2：
 如果 y == 0：
  x = 1
（这是一个经典的交易测试示例）
T1,T2 产生 y=1,x=0
T2,T1 产生 x=1,y=0
中止可能会导致 x=0,y=0
但可串行化禁止 x=1,y=1

假设同时：
 T1：Rx Ly Vx Cy
 T2：Ry Lx Vy Cx
 会发生什么？
 LOCK 都会成功！
 VALIDATE 都会失败，因为锁定位都已设置
 所以两者都会中止——这没关系

怎么样：
 T1：Rx Ly Vx Cy
 T2：Ry Lx Vy Cx
 T1 提交
 T2 中止，因为 T2 的 Vy 看到 T1 的锁定或更高版本
但我们不能在其他 L 之前有*两个* V
所以 VALIDATE 在这个例子中似乎是正确的
 快速：单方面 VALIDATE 读取而不是 LOCK+COMMIT 写入

纯只读 FaRM 事务仅使用单侧 RDMA 读取
 无写入、无日志记录、无锁定
 非常快！

容错能力怎么样？
 假设某些计算机崩溃并且不重新启动
 最有趣的是 TC 和一些初选崩溃
 但我们假设每个分片的一个备份仍然存在

关键问题：
 如果交易因失败而中断，
  并且客户可能被告知已提交交易，
  或者提交的值可能已被另一个 xaction 读取，
 那么事务必须在恢复期间保留并完成。

看图4。
一旦提交的写入可能会被揭示
 第一个 COMMIT-PRIMARY 被发送（因为主写入和解锁）。
所以到那时，所有事务的写入都必须在所有
 所有相关分片的 f+1 个副本。
好消息：LOCK 和 COMMIT-BACKUP 可以实现这一点。
 LOCK 告诉所有原色新值。
 COMMIT-BACKUP 告诉所有备份新值。
 在所有 LOCK 和 COMMIT-BACKUPS 完成之前，TC 不会发送 COMMIT-PRIMARY。
 备份可能没有处理 COMMIT-BACKUP，但在 NVRAM 日志中。

类似地，TC 不会返回给客户端，直到至少一个
COMMIT-PRIMARY 在主日志中是安全的。
如果没有 COMMIT-PRIMARY，危险的情况是：
TC 在 COMMIT-BACKUPS 之后（在 COMMIT-PRIMARY 之前）对应用程序回复“yes”。
TC 和所有备份都会失败。
现在剩下的唯一证据就是 LOCK 记录。
但即使是完整的 LOCK 记录集也无法告诉我们 TC 是否已提交
也许 TC 由于验证失败而中止！
编写 COMMIT-PRIMARY 可以处理风险，因为 TC 的
决策将在任何分片失败的情况下继续存在。
因为有一个分片具有一整套 COMMIT-BACKUP 和 COMMIT-PRIMARY。
其中任何一个都是初选决定承诺的证据。

FaRM 非常令人印象深刻；它是否不够完美？

- 由于 OCC，如果冲突很少，效果最好。
- 数据必须适合总 RAM。
- 仅在数据中心内进行复制（无地理分布）。
- 数据模型是低级的；需要例如 SQL 库。
- 需要一些不寻常的 RDMA 和 NVRAM 硬件。

FaRM 与 Spanner 有何不同？
都对事务进行分片、复制和使用两阶段提交 (2pc)
扳手：
专注于应对由于地理复制而导致的网络延迟
Paxos 容忍延迟
TrueTime 让他们从本地副本读取
性能：r/w xaction 需要 10 到 100 ms（表 3 和表 6）
农场
专注于降低CPU成本
RDMA、直接 NIC 访问、NVRAM 以避免磁盘写入
RDMA 引导他们实现乐观并发控制 (OCC)
性能：简单事务 58 微秒（6.3，图 7）
即比 Spanner 快 100 倍

概括
超高速分布式事务
硬件很奇特（NVRAM 和 RDMA），但可能很快就会普及
使用 OCC 提高速度并允许快速单侧 RDMA 读取

英文原文

6.824 2022 Lecture 14: FaRM, Optimistic Concurrency Control

why are we reading about FaRM?
another take on transactions+replication+sharding
this is still an open research area!
optimistic concurrency control
exploiting huge performance potential of RDMA NICs

the overall setup
  all in one data center
  configuration manager, using ZooKeeper, chooses primaries/backups
  sharded w/ primary/backup replication
    P1 B1
    P2 B2
    ...
    can recover as long as at least one replica of each shard
    i.e. f+1 replicas tolerate f failures
  transaction clients (which they run in the servers)
  transaction code acts as two-phase-commit Transaction Coordinator (TC)

the goal:
  millions of distributed transactions per second
  so the time budget is tens of microseconds
  very challenging!

how do they get high performance?
  sharding over many servers (90 in the evaluation)
  data must fit in total RAM (so no disk reads)
  non-volatile RAM (so no disk writes)
  one-sided RDMA (fast cross-network access to RAM)
  fast user-level access to NIC
  transaction+replication protocol that exploits one-sided RDMA

NVRAM (non-volatile RAM)
  FaRM writes go to RAM, not disk -- eliminates a huge bottleneck
  RAM write takes 200 ns, hard drive write takes 10 ms, SSD write 100 us
    ns = nanosecond, ms = millisecond, us = microsecond
  but RAM loses content in power failure! not persistent by itself.
  why not just write to RAM of f+1 machines, to tolerate f failures?
    might be enough if failures were always independent
    but power failure is not independent -- may strike 100% of machines!
  so:
    batteries in every rack, can run machines for a few minutes
    power h/w notifies s/w when main power fails
    s/w halts all transaction processing
    s/w writes FaRM's RAM to SSD; may take a few minutes
    then machine shuts down
    on re-start, FaRM reads saved memory image from SSD
    "non-volatile RAM"
  what if crash prevents s/w from writing SSD?
    e.g bug in FaRM or kernel, or cpu/memory/hardware error
    FaRM copes with single-machine crashes with replication
    crashes (other than power failure) must be independent!
  summary:
    NVRAM eliminates persistence write bottleneck
    leaving network and CPU as remaining bottlenecks

why is the network often a performance bottleneck?
  FaRM assumes single data-center, so low speed-of-light delay
  but CPU cost of network data handling is often large!
  the usual setup for RPC over TCP over LAN:
    app   app
    ---   ---
    socket buffers   buffers
    TCP   TCP
    NIC driver   driver
    NIC ---------- NIC
  lots of expensive CPU operations:
    system calls
    copy messages
    interrupts
    context switches
  slow:
    hard to build RPC than can deliver more than a few 100,000 / second
    wire b/w (e.g. 10 gigabits/second) is rarely the limit for short RPC
    per-packet CPU costs traditionally limit performance for small messages

FaRM uses two networking ideas:
  Kernel bypass
  RDMA

Kernel bypass
  [diagram: FaRM user program, CPU cores, DMA queues, NIC]
  application directly interacts with NIC -- no system calls, no kernel
  NIC DMAs into/out of user RAM
  FaRM s/w polls DMA areas to check for incoming messages
  NIC polls DMA areas to check for outgoing messages

RDMA (remote direct memory access)
  [src host, NIC, switch, NIC, target memory, target CPU]
  remote NIC directly reads/writes memory
    Sender provides memory address
    Remote CPU is not involved!
    This is "one-sided RDMA"
    Reads an entire cache line, atomically
  RDMA NICs use reliable protocol, with ACKs
  one server's throughput: 10+ million/second (Figure 2)
  latency: 5 microseconds (from their NSDI 2014 paper)

Performance would be amazing if clients could directly access
  DB records on servers via one-sided RDMA!

How to combine one-sided RDMA with replication and transactions?
  The protocols we've seen so far require active server participation.
  e.g. to check and set locks,
       to check lease,
       to indicate when safely persisted.
  Not immediately compatible with one-sided RDMA.

two classes of concurrency control for transactions:
  pessimistic (two-phase locking):
    wait for lock on first use of object; hold until commit/abort
    conflicts cause delays
  optimistic:
    read objects without locking
    don't install writes until commit
    commit "validates" to see if other xactions conflicted
    valid: commit the writes
    invalid: abort
    called Optimistic Concurrency Control (OCC)

FaRM uses OCC
  the reason:
    OCC lets FaRM read using one-sided RDMA reads
    so server needn't actively participate in reads
  how does FaRM's OCC validate? we'll look at Figure 4 in a minute.

FaRM transaction API (simplified):
  txCreate()
  o = txRead(oid) -- RDMA
  o.f += 1
  txWrite(oid, o) -- purely local
  ok = txCommit() -- Figure 4

what's an oid?
  <region #, address>
  region # indexes a mapping to [primary, backup1, ...]
  target RDMA NIC uses address directly to read RAM

server memory layout
  regions, each an array of objects
  object layout
    header with version #, and lock flag in high bit of version #
  for each other server
    incoming log
    incoming message queue
    (senders write via RDMA, local FaRM reads via polling)
  all this in non-volatile RAM (i.e. written to SSD on power failure)

Figure 4: transaction execution / commit protocol
  let's consider steps in Figure 4 one by one
  focus on concurrency control (not fault tolerance)

Execute phase
  TC (the client) reads the objects it needs from servers
    including records that it will write
    using one-sided RDMA reads
    without locking
    this is the optimism in Optimistic Concurrency Control
  TC remembers the version numbers
  TC buffers writes locally

now TC commits; two big goals:
  atomic distributed commit -- all writes or none
  serializability -- as if entirely before or after every other transaction

LOCK (first message in commit protocol)
  TC sends to primary of each written object
  TC uses RDMA to append to its log at each primary
  LOCK record contains oid, version # xaction read, new value
  LOCK is now logged in primary's NVRAM
    will survive a power failure
  LOCK message is both a read-ahead log entry,
    and an RPC request to the primary

what does primary do on receipt of LOCK?
  FaRM s/w polls incoming logs in RAM, sees our LOCK
  if object locked, or version != what xaction read,
    send "no" reply to TC
  otherwise set the object's lock flag and reply "yes"
    but don't yet modify the data!
  lock check, version check, and lock set are atomic
    using atomic compare-and-swap instruction
    "locked" flag is high-order bit in object's version number
    in case other CPU also processing a LOCK, or a client is reading w/ RDMA

TC waits for all LOCK reply messages
  if any "no", abort
    append ABORT to primaries' logs so they can release locks
    returns "no" from txCommit()

let's ignore VALIDATE and COMMIT BACKUP for now

at this point primaries need to know TC's decision

TC appends COMMIT-PRIMARY to primaries' logs
  TC only waits for RDMA hardware acknowledgement (ack)
    does not wait for primary to process log entry
    hardware ack means safe in primary's NVRAM
  TC returns "yes" from txCommit()

when primary processes COMMIT-PRIMARY in its log:
  copy new value to object's memory
  increment object's version #
  clear object's lock flag

the commit point is when the first COMMIT-PRIMARY is written
  since at that point the transactions results can be revealed

example:
  T1 and T2 both want to increment x
    x = x + 1
  what results does serializability allow?
    i.e. what outcomes are possible if run one at a time?
    x = 2, both clients told "success"

x = 1, one client told "success", other "aborted"
x = 0, both clients told "aborted"

what if T1 and T2 are exactly in step?
T1: Rx0 Lx Cx
T2: Rx0 Lx Cx
what will happen?

or
T1: Rx0 Lx Cx
T2: Rx0 Lx Cx

or
T1: Rx0 Lx Cx
T2: Rx0 Lx Cx

intuition for why FaRM's OCC provides serializability:
i.e. checks "was execution same as one at a time?"
if there was no conflicting transaction:
the versions won't have changed
if there was a conflicting transaction:
one or the other will see a lock or changed version #

what about VALIDATE in Figure 4?
it is an optimization for objects that are just read by a transaction
VALIDATE = one-sided RDMA read to re-fetch object's version # and lock flag
if lock set, or version # changed since read, TC aborts
does not set the lock, thus faster than LOCK+COMMIT

VALIDATE example:
x and y initially zero
T1:
if x == 0:
y = 1
T2:
if y == 0:
x = 1
(this is a classic test example for transactions)
T1,T2 yields y=1,x=0
T2,T1 yields x=1,y=0
aborts could leave x=0,y=0
but serializability forbids x=1,y=1

suppose simultaneous:
T1: Rx Ly Vx Cy
T2: Ry Lx Vy Cx
what will happen?
the LOCKs will both succeed!
the VALIDATEs will both fail, since lock bits are both set
so both will abort -- which is OK

how about:
  T1: Rx Ly Vx   Cy
  T2: Ry   Lx Vy Cx
  T1 commits
  T2 aborts since T2's Vy sees T1's lock or higher version
but we can't have *both* V's before the other L's
so VALIDATE seems correct in this example
  and fast: one-sided VALIDATE read rather than LOCK+COMMIT writes

a purely read-only FaRM transaction uses only one-sided RDMA reads
  no writes, no log records, no locking
  very fast!

what about fault tolerance?
  suppose some computers crash and don't reboot
  most interesting if TC and some primaries crash
  but we assume one backup from each shard survives

the critical issue:
  if a transaction was interrupted by a failure,
    and a client could have been told a transaction committed,
    or a committed value could have been read by another xaction,
  then the transaction must be preserved and completed during recovery.

look at Figure 4.
a committed write might be revealed as soon the
  first COMMIT-PRIMARY is sent (since primary writes and unlocks).
so by then, all of the transaction's writes must be on all
  f+1 replicas of all relevant shards.
the good news: LOCK and COMMIT-BACKUP achieve this.
  LOCK tells all primaries the new value(s).
  COMMIT-BACKUP tells all backups the new value(s).
  TC doesn't send COMMIT-PRIMARY until all LOCKs and COMMIT-BACKUPS complete.
  backups may not have processed COMMIT-BACKUPs, but in NVRAM logs.

similarly, TC doesn't return to client until at least one
 COMMIT-PRIMARY is safe in primary log.
  without the COMMIT-PRIMARY, the risky case is:
    TC replies "yes" to app after COMMIT-BACKUPS (before COMMIT-PRIMARY).
    TC and all backups then fail.
    now the only evidence left is the LOCK records.
    but even a complete set of LOCK records doesn't tell us if TC committed
      maybe TC aborted due to failed VALIDATE!
  writing the COMMIT-PRIMARY handles the risk, because the TC's
     decision will survive f failures of any shard.
  since there's one shard with a full set of COMMIT-BACKUP and COMMIT-PRIMARY.
  any of which is evidence that the primary decided to commit.

FaRM is very impressive; does it fall short of perfection?

- works best if few conflicts, due to OCC.
- data must fit in total RAM.
- replication only within a datacenter (no geographic distribution).

- the data model is low-level; would need e.g. SQL library.
- requires somewhat unusual RDMA and NVRAM hardware.

how does FaRM differ from Spanner?
  both shard, replicate, and use two-phase commit (2pc) for transactions
  Spanner:
    focuses on coping with network delay due to geographic replication
    Paxos tolerates delay
    TrueTime lets them read from local replicas
    performance: r/w xaction takes 10 to 100 ms (Tables 3 and 6)
  FaRM
    focuses on reducing CPU costs
    RDMA, direct NIC access, NVRAM to avoid disk writes
    RDMA leads them to Optimistic Concurrency Control (OCC)
    performance: 58 microseconds for simple transactions (6.3, Figure 7)
      i.e. 100 times faster than Spanner

summary
  super high speed distributed transactions
  hardware is exotic (NVRAM and RDMA) but may be common soon
  use of OCC for speed and to allow fast one-sided RDMA reads

LEC 15 Big Data：Spark

中文翻译

6.824 2022年第15讲：火花

弹性分布式数据集：容错抽象
内存集群计算 Zaharia 等人，NSDI 2012

今天我们将话题从存储转向计算

背景：计算机网络上的并行程序
  这是一个古老的梦想：通过廉价的普通计算机获得巨大的计算能力
  如何使它们易于编程？
  之前的一个热门想法：分布式共享内存（DSM）
  人们知道如何为多核编写线程并行代码
    [图表：核心、RAM]
    线程，共享数据结构，锁，就像Go编程一样
  为什么不在服务器集群上运行这些相同的程序呢？
    使用所有服务器的内存，就像使用单个大内存一样
    透明地拦截对远程地址的引用
  很酷！
  两个杀手级问题：
    远程访问速度太慢，现有并行算法太慢
      你真的无法掩盖本地和远程数据之间的差异
      你需要编写代码并知道它会被分发
    很难从一台服务器的故障中恢复
      例如 8 个服务器不是问题；稀有的; 只是从头开始
      对于数百台服务器来说是一个大问题
  Spark（和MapReduce）是对DSM中这些缺陷的部分反应

我们为什么要关注 Spark?
广泛用于数据中心计算
将 MapReduce 概括为数据流
比 MapReduce 更好地支持迭代应用程序

三个主要主题:
编程模型
执行策略
容错

让我们看看页面排名
这是 Spark 源存储库中的 SparkPageRank.scala
就像第 3.2.2 节中的代码一样，有更多细节

```
1  vallines = Spark.read.textFile("in").rdd
2  val links1 =lines.map{ s =>
3  val 部分 = s.split("\\s+")
4（零件（0），零件（1））
5  }
6  val links2 = links1.distinct()
7  val links3 = links2.groupByKey()
8  val links4 = links3.cache()
9  var 排名 = links4.mapValues(v => 1.0)
10
11 为 (i <- 1 到 10) {
12 val jj = links4.join(ranks)
13 val contribs = jj.values.flatMap{
14 案例（网址、排名）=>
15 urls.map(url => (url, 排名 / urls.size))
16}
17 排名 = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
18}
19
20 val 输出=ranks.collect()
21 output.foreach(tup => println(s"${tup._1} 的排名：${tup._2} ."))
```

page-rank是一个经典的大数据算法例子
由谷歌发明来估计网页的重要性，
基于网络链接
Google 使用页面排名对搜索结果进行排序

页面排名输入每个链接一行，从大型网络爬行中提取
从 url 到 url
输入量巨大!
存储在 HDFS（如 GFS）中，分区（分割）为许多块

页面排名算法
迭代，本质上模拟多轮用户点击链接
有 85% 的机会关注当前页面中的链接之一
15% 的机会访问随机页面
在每次迭代中沿着链接推动概率
概率（“等级”）逐渐收敛
MapReduce 中的多次迭代有点尴尬且效率低下

我的示例输入——文件“in”：
[图：链接图]
u1 u3
u1 u1
u2 u3
u2 u2
u3 u1

我将在 Spark（本地计算机，而不是集群）中运行页面排名：
./bin/run-example SparkPageRank 为 10
u2 的排名： 0.2610116705534049 。
u3 的排名： 0.9999999999999998 。
u1 的排名： 1.7389883294465944 。

显然 u1 是最重要的页面。

让我们在 Spark/Scala 解释器中运行一些页面排名代码
./bin/spark-shell

```
vallines = Spark.read.textFile("in").rdd
  -- “in”是 HDFS 上的文件（如 GFS）
  ——什么是线？它包含文件“in”的内容吗？
线.collect()
  --lines 产生一个字符串列表，每行输入一个字符串
  -- 如果我们再次运行lines.collect()，它会重新读取文件“in”
val links1 =lines.map{ s => val parts = s.split("\\s+"); （部分（0），部分（1））}
links1.collect()
  -- 需要与字符串分开
  -- map, split, tuple -- 依次作用于每一行
  -- 将每个字符串“x y”解析为元组(“x”,“y”)
  --map 对每个输入分区并行运行
  -- 一次只查看一条记录（行）
  -- 【图：分区RDD，并行map执行】
val links2 = links1.distinct()
  -- 需要消除配对中的重复项
  --distinct() 排序或散列将重复项放在一起
val links3 = links2.groupByKey()
  -- 需要一起考虑每个页面的所有链接
  -- groupByKey() 排序或散列以将每个键的实例放在一起
val links4 = links3.cache()
  -- 我们将在每次迭代中使用链接
  -- 默认是每次使用 RDD 时（重新）计算它
  --cache()==保留在内存中
varrank = links4.mapValues(v => 1.0)
  -- 这将是输出
  -- 迭代更新，这些是起始值
  -- 我们将在每次迭代中计算新版本的排名

-- 现在进行第一次循环迭代
val jj = links4.join(ranks)
  -- 连接将每个页面的链接列表和当前排名结合在一起
  -- 使用处理一页所需的信息生成一条记录
  -- 页面的 url、传出 url、页面的当前排名
  -- 为什么我们必须显式地构建这个记录？
```

```
      -- 为什么不在排名和链接表中根据需要查找信息？
      -- 我们无法查看此模型中的全局表！
      --该模型限制我们一次只能考虑一条记录。
      --所以我们需要构建包含我们需要的所有信息的单个记录。
val contribs = jj.values.flatMap{ case (urls,rank) => urls.map(url =>
(url,rank / urls.size)) }
  -- 对于每个链接，“来自”页面的排名除以其链接数量
  -- 关键是“to”url（而不是“from”url）
排名 = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
  -- 总结通向每个页面的链接
  --reduceByKey 随机排列以汇集对每个 url 的所有贡献
     和总和
  -- mapValues 只是修改每个总和

-- 第二次循环迭代
val jj2 = links4.join(ranks)
val contribs2 = jj2.values.flatMap{ case (urls, 排名) => urls.map(url =>
(url, 排名 / urls.size)) }
排名 = contribs2.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)

-- 我们在这里生成一个新的排名 RDD，而不是修改排名。

-- 循环 &c 只是创建一个谱系图。
--它没有做任何实际的工作。

val 输出=ranks.collect()
  --collect() 是一个动作。
  --它会导致整个计算执行！
output.foreach(tup => println(s"${tup._1} 的排名：${tup._2} ."))
```

对于多步计算，比MapReduce更方便
 我们要说的是阶段是什么！
 何时进行映射、何时进行洗牌、何时进行缩减

直到最后的collect()，这段代码只是创建了一个谱系图
 它不处理数据
 “懒惰的”

谱系图是什么样的？
 图3
 这是一个转换阶段图——数据流图
 这是一个完整的计算方法
 请注意，添加到图中的循环——实际上并没有循环
  每个循环迭代都有一个*新的*排名/贡献
 注意链接 RDD 的重用

为什么这么整齐？

- 简单的程序控制谱系图的构建
- 一个简单的程序可以驱动巨大的计算
- 程序员不必担心分发细节
- 指导程序员使用高效的结构
- 语言和集群计算之间的良好集成

- 映射将闭包和程序数据注入集群计算中
- RDD 使用语言数据类型，如数组
- 轻松从集群计算中获取结果

执行是什么样的?
[图：驱动程序、分区输入文件、工作人员] -- 图 2

- Scala 代码在“驱动程序”机器中运行
- 驱动程序使用谱系图来告诉工人该做什么
- 输入HDFS（如GFS）
- 输入数据文件已经在许多存储服务器上“分区”
  一个分区中的前 1,000,000 行，另一个分区中的下一行，等等。
- 分区比机器多，用于负载平衡
- 每台worker机器都有一个分区，按顺序应用谱系图

狭隘的依赖性
不同分区上独立的计算
像地图（）
工人可以“管道化”一系列狭窄的转换
出色的并行加速！就像MapReduce的Map阶段一样。

那么distinct()呢？groupByKey()? 加入（）？减少按键（）？
这些需要查看“所有”分区的数据，而不仅仅是一个
因为具有给定键的所有记录必须一起考虑
这些是论文的“广泛”依赖性

宽依赖是如何实现的?
[图表]
驱动程序知道广泛的依赖关系在哪里
例如，页面排名中的map()和distinct()之间
上游改造、下游改造
数据必须被“洗牌”到新的分区中
例如将所有给定的密钥放在一起
上游改造后：
按洗牌标准分割输出（通常是某个键）
在内存中排列成桶，每个下游分区一个
下游改造前：
（等到上游转换完成——驱动程序管理这个）
每个工作人员从每个上游工作人员获取其存储桶
现在数据以不同的方式分区
就像Map->Reduce shuffle一样
宽就是贵！
所有数据都通过网络移动
这是一个障碍——所有工人都必须等到一切都完成

如果重新使用随机播放输出怎么办?
例如我们页面排名中的 links4
默认情况下，必须从原始输入开始重新计算
persist() 和cache() 导致链接保存在内存中以供重复使用

与 MapReduce 相比，重用持久数据是一个很大的优势

Spark 可以使用整个谱系图的视图进行优化
流记录，一次一个，通过一系列狭窄的转换
增加局部性，有利于 CPU 数据缓存
避免将整个记录分区存储在内存中
当输入已经以相同方式分区时消除洗牌
可以做到这一点，因为它在开始计算之前延迟生成了谱系图

容错能力怎么样？
数千个工人 -> 工人频繁故障！
故障恢复策略对于普通性能很重要
驱动 RDD 不可变/确定性设计选择
主要技巧：重新计算而不是复制

如果一名工人摔倒了会出现什么问题？
到目前为止，我们丢失了工作人员已完成工作的分区（RDD）
[图：狭窄的依赖关系]
驱动程序在其他机器上崩溃机器的分区上重新运行转换
通常每台机器负责许多分区
这样工作就可以分散
因此重新计算非常快
对于窄依赖关系，只有丢失的分区必须重新执行

当存在广泛的依赖性时失败怎么办？
[图：广泛依赖]
重新计算一个失败的分区需要来自*所有*分区的信息
所以*所有*分区可能需要从头开始重新执行！
即使他们没有失败
Spark 支持 HDFS 检查点来解决这个问题
驱动程序只需沿着最新检查点的沿袭重新计算
对于页面排名，也许检查点每 10 次迭代排名一次

那么评测部分呢？
我们期待什么？该文件承诺什么？
在内存之外操作，而不是从 HDFS/GFS
优化复杂图
但请注意，很难评估程序员的便利性或灵活性

图 7 / 第 6.1 节
他们在比较什么
Hadoop
HadoopBinMem
火花
为什么他们将第一次迭代与后来的迭代分开？
它包括从 HDFS 的初始读取——速度慢，而且所有读取都相同
为什么后来的迭代 Spark 速度快了 20 倍？
与 Hadoop 相比？
Hadoop 在每次迭代中读取/写入 HDFS
与 HadoopBinMem 相比？
Hadoop 甚至要在内存中获取 HDFS 也会产生大量开销
Spark 将数据作为本机 Java 对象保存在内存中
为什么 Spark 对 K-Means 的优势不如对 Logistic 回归的优势？
Spark 的获胜主要是通过避免 Hadoop 的 HDFS 读/写

如果与 HDFS 操作相比计算时间较长，则获胜的空间较小
要点：将数据保存在内存中比保存在磁盘上更快

图 10 / 第 6.2 节
这是PageRank
比 Hadoop 快 2 到 3 倍，可能是由于内存与磁盘的关系
为什么不是 20 倍？我们不知道。
也许时间被洗牌所主导，而这对两者来说都是相同的？
Spark + 受控分区
通过指定分区消除一次洗牌
2 倍改进
可能 2x b/c 每次迭代都有两次洗牌：join 和 reduceByKey
也许随机播放主导了性能

限制？
为海量数据的批量处理做好准备
所有记录都以相同的方式处理——数据并行
对于网站交互、购物车等没有用处
没有读取或更新单个数据项的概念
对于稳定的数据流没有用（但 Spark 随后得到了发展）

概括
与 MapReduce 相比，Spark 提高了表达能力和性能
成功的关键是什么？
应用程序开始描述数据流图
Spark 可以使用数据流图的先验知识进行优化
在转换之间将数据保留在内存中，而不是写入 GFS 然后读取
Spark成功，广泛应用

英文原文

6.824 2022 Lecture 15: Spark

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for
In-Memory Cluster Computing Zaharia et al., NSDI 2012

today we're switching topics from storage to computation

background: parallel programs on networks of computers
it's an old dream: huge compute power from inexpensive ordinary computers
how to make them easy to program?
a previous hot idea: Distributed Shared Memory (DSM)
people knew how to write threaded parallel code for multi-core
[diagram: cores, RAM]
threads, shared data structures, locks, just like Go programming
why not run those same programs on a cluster of servers?
use memory of all the servers as if a single big memory
transparently intercept references to remote addresses
very cool!
two killer problems:
remote access so much slower that existing parallel algorithms were too slow
you really can't conceal the difference between local and remote data
you need to write code knowing it will be distributed

hard to recover from the failure of one server
not a problem with e.g. 8 servers; rare; just re-start from beginning
a huge problem for 100s of servers
Spark (and MapReduce) are partially reactions to these defects in DSM

why are we looking at Spark?
widely-used for datacenter computations
generalizes MapReduce into dataflow
supports iterative applications better than MapReduce

three main topics:
programming model
execution strategy
fault tolerance

let's look at page-rank
here's SparkPageRank.scala from the Spark source repository
like the code in Section 3.2.2, with more detail

```
1  1      val lines = spark.read.textFile("in").rdd
2  2      val links1 = lines.map{ s =>
3  3        val parts = s.split("\\s+")
4  4        (parts(0), parts(1))
5  5      }
6  6      val links2 = links1.distinct()
7  7      val links3 = links2.groupByKey()
8  8      val links4 = links3.cache()
9  9      var ranks = links4.mapValues(v => 1.0)
10 10
11 11     for (i <- 1 to 10) {
12 12       val jj = links4.join(ranks)
13 13       val contribs = jj.values.flatMap{
14 14         case (urls, rank) =>
15 15           urls.map(url => (url, rank / urls.size))
16 16       }
17 17       ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
18 18     }
19 19
20 20     val output = ranks.collect()
21 21     output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
```

page-rank is a classic big-data algorithm example
invented by Google to estimate importance of web pages,
based on web links
Google uses page-rank to sort search results

page-rank input has one line per link, extracted from a big web crawl
from-url to-url
the input is huge!
stored in HDFS (like GFS), partitioned (split) into many chunks

page-rank algorithm
  iterative, essentially simulates multiple rounds of users clicking links
    85% chance of following one of the links from current page
    15% chance of visiting a random page
  pushes probabilities along links on each iteration
  probabilities ("rank") gradually converge
  the multiple iterations are a bit awkward and inefficient in MapReduce

my example input -- file "in":
  [diagram: the link graph]
  u1 u3
  u1 u1
  u2 u3
  u2 u2
  u3 u1

I'll run page-rank in Spark (local machine, not a cluster):
  ./bin/run-example SparkPageRank in 10
  u2 has rank: 0.2610116705534049 .
  u3 has rank: 0.9999999999999998 .
  u1 has rank: 1.7389883294465944 .

apparently u1 is the most important page.

let's run some of the page-rank code in the Spark/Scala interpreter
    ./bin/spark-shell

```
val lines = spark.read.textFile("in").rdd
  -- "in" is a file on HDFS (like GFS)
  -- what is lines? does it contain the content of file "in"?
lines.collect()
  -- lines yields a list of strings, one per line of input
  -- if we run lines.collect() again, it re-reads file "in"
val links1 = lines.map{ s => val parts = s.split("\\s+"); (parts(0),
parts(1)) }
links1.collect()
  -- need to separate from and to strings
  -- map, split, tuple -- acts on each line in turn
  -- parses each string "x y" into tuple ( "x", "y" )
  -- map runs in parallel for each input partition
  -- looks at just one record (line) at a time
  -- [diagram: partitioned RDD, parallel map execution]
val links2 = links1.distinct()
  -- need to eliminate duplicate from/to pairs
  -- distinct() sorts or hashes to bring duplicates together
val links3 = links2.groupByKey()
  -- need to consider all links out of each page together
  -- groupByKey() sorts or hashes to bring instances of each key together
val links4 = links3.cache()
  -- we're going to use links in each iteration
  -- the default is to (re-)compute an RDD each time we use it
  -- cache() == persist in memory
var ranks = links4.mapValues(v => 1.0)
  -- this will be the output
```

```
  -- iteratively updated, these are starting values
  -- we'll compute a new version of ranks on each iteration

-- now for first loop iteration
val jj = links4.join(ranks)
  -- the join brings each page's link list and current rank together
  -- to generate a single record with the info needed to process one page
  -- page's url, outgoing urls, page's current rank
  -- why do we have to explicitly construct this record?
     -- why not look info up as needed in the ranks and links4 tables?
     -- we cannot look at global tables in this model!
     -- the model restricts us to thinking about one record at a time.
     -- so we need to construct single records that have all the info we need.
val contribs = jj.values.flatMap{ case (urls, rank) => urls.map(url => (url, rank / urls.size)) }
  -- for each link, the "from" page's rank divided by number of its links
  -- the key is the "to" url (not the "from" url)
ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
  -- sum up the links that lead to each page
  -- reduceByKey shuffles to bring together all contributions to each url
     and sums
  -- mapValues just modifies each sum

-- second loop iteration
val jj2 = links4.join(ranks)
val contribs2 = jj2.values.flatMap{ case (urls, rank) => urls.map(url => (url, rank / urls.size)) }
ranks = contribs2.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)

-- we're producing a new ranks RDD here, not modifying ranks.

-- the loop &c just creates a lineage graph.
-- it does not do any real work.

val output = ranks.collect()
  -- collect() is an action.
  -- it causes the whole computation to execute!
output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
```

for multi-step computation, more convenient than MapReduce
- we get to say what the stages are!
- when to map, when to shuffle, when to reduce

until the final collect(), this code just creates a lineage graph
- it does not process the data
- "lazy"

what does the lineage graph look like?
- Figure 3
- it's a graph of transform stages -- a data-flow graph
- it's a complete recipe for the computation
- note that the loop added to the graph -- there is not actually a cycle
  - there's a *new* ranks/contribs for each loop iteration
- note the re-use of the links RDD

why is this neat?

- easy program control over construction of lineage graph
- a simple program can drive a huge computation
- programmer doesn't have to worry about details of distribution
- programmer is guided to using constructs that are efficient
- nice integration between language and cluster computation
    - the maps inject a closure and program data into the cluster computation
    - RDDs use language data types, like arrays
    - easy to get results back from the cluster computation

what does execution look like?
[diagram: driver, partitioned input file, workers] -- Figure 2

- Scala code runs in the "driver" machine
- driver uses lineage graph to tell workers what to do
- input in HDFS (like GFS)
- input data files are already "partitioned" over many storage servers
  first 1,000,000 lines in one partition, next lines in another, &c.
- more partitions than machines, for load balance
- each worker machine takes a partition, applies lineage graph in order

narrow dependencies
computations that are independent on different partitions
like map()
a worker can "pipeline" a series of narrow transformations
excellent parallel speedup! like MapReduce's Map phase.

what about distinct()? groupByKey()? join()? reduceByKey()?
these need to look at data from *all* partitions, not just one
because all records with a given key must be considered together
these are the paper's "wide" dependencies

how are wide dependencies implemented?
[diagram]
the driver knows where the wide dependencies are
e.g. between the map() and the distinct() in page-rank
upstream transformation, downstream transformation
the data must be "shuffled" into new partitions
e.g. bring all of a given key together
after the upstream transformation:
split output up by shuffle criterion (typically some key)
arrange into buckets in memory, one per downstream partition
before the downstream transformation:
(wait until upstream transformation completes -- driver manages this)
each worker fetches its bucket from each upstream worker
now the data is partitioned in a different way
just like Map->Reduce shuffle
wide is expensive!
all data is moved across the network
it's a barrier -- all workers must wait until all are done

what if shuffle output is re-used?
  e.g. links4 in our page-rank
  by default, must be re-computed, all the way from original input
  persist() and cache() cause links to be saved in memory for re-use

re-using persisted data is a big advantage over MapReduce

Spark can optimize using its view of the whole lineage graph
  stream records, one at a time, though sequence of narrow transformations
    increases locality, good for CPU data caches
    avoids having to store entire partition of records in memory
  eliminate shuffles when inputs already partitioned in the same way
  can do this b/c it lazily produced lineage graph before starting computation

what about fault tolerance?
  1000s of workers -> frequent worker failures!
    failure recovery strategy important for ordinary performance
    drives the RDD immutable / deterministic design choice
  main trick: recompute rather than replicate

what goes wrong if one worker crashes?
  we lose worker's partition(s) of the work done (RDDs) so far
  [diagram: narrow dependencies]
  driver re-runs transformations on crashed machine's partitions on other machines
    usually each machine is responsible for many partitions
    so work can be spread
    thus re-computation is pretty fast
  for narrow dependencies, only lost partitions have to be re-executed

what about failures when there are wide dependencies?
  [diagram: wide dependency]
  re-computing one failed partition requires information from *all* partitions
  so *all* partitions may need to re-execute from the start!
    even though they didn't fail
  Spark supports checkpoints to HDFS to cope with this
    driver only has to recompute along lineage from latest checkpoint
  for page-rank, perhaps checkpoint ranks every 10th iteration

what about the evaluation section?
  what do we expect? what does the paper promise?
  operate out of memory, rather than from HDFS/GFS
  optimize complex graphs
  but note it's hard to evaluate programmer convenience or flexibility

figure 7 / Section 6.1
  what are they comparing
    Hadoop
    HadoopBinMem
    Spark
  why do they separate first iteration from later ones?
    it includes initial read from HDFS -- slow, and same for all
  why is spark 20x faster for later iterations?
    vs Hadoop?

Hadoops reads/write HDFS on every iteration
vs HadoopBinMem?
lots of overhead for Hadoop to get at even HDFS in memory
Spark keeps data as native Java objects in memory
why is Spark's advantage less for K-Means than for Logistic Regression?
Spark wins mostly by avoiding Hadoop's HDFS reads/writes
if computation time is large compared to HDFS ops, less room to win
take-away: it's faster to keep data in memory than on disk

figure 10 / Section 6.2
this is PageRank
2x to 3x faster than Hadoop, presumably due to mem vs disk
why not 20x? we don't know.
maybe time is dominated by shuffles which are the same for both?
Spark + Controlled Partitioning
eliminates one shuffle by specifying partition
2x improvement
probably 2x b/c each iteration has two shuffles: join, and reduceByKey
and maybe the shuffle dominates performance

limitations?
geared up for batch processing of huge data
all records treated the same way -- data-parallel
not useful for e.g. web site interactions, shopping cart, &c
no notion of reading or updating individual data items
not useful for steady streams of data (but spark has subsequently evolved)

summary
Spark improves expressivity and performance vs MapReduce
what were the keys to success?
application gets to describe the data-flow graph
spark can optimize using prior knowledge of data-flow graph
leave data in memory between transformations, vs write to GFS then read
Spark successful, widely used

LEC 16 Cache Consistency：Memcached at Facebook

中文翻译

6.824 2022 第 16 讲：扩展 Facebook 的 Memcache

Facebook 的 Memcache 扩展，作者：Nishtala 等人，NSDI 2013

我们为什么要读这篇论文?
这是一篇经验论文，而不是新想法/技术
三种阅读方式：
关于从一开始就没有认真对待一致性的警示故事
大多数现成软件的超高容量令人印象深刻
性能和一致性之间的根本斗争
我们可以争论他们的设计，但不能争论他们的成功

当网站获得更多用户时如何应对?

随着时间的推移演变的典型故事:

1.单机w/web服务器+应用程序+数据库

DB提供持久存储、崩溃恢复、事务、SQL

应用程序查询数据库、格式化 HTML 等

但是:随着负载的增长,应用程序会占用过多的 CPU 时间

2. 多个Web FE,一个共享DB

一个简单的改变,因为网络服务器+应用程序已经与数据库分离

FE 是无状态的,所有共享(和并发控制)都通过 DB

无状态 -> 任何 FE 都可以服务任何请求,FE 崩溃不会造成任何损害

但是:随着负载的增长,需要更多的 FE,很快单个数据库服务器就会成为瓶颈

3. 许多 Web FE,数据在 DB 集群上分片

在 DB 上按键对数据进行分区

应用程序查看密钥(例如用户),选择正确的数据库

如果没有超级流行的数据,则数据库并行性良好

痛苦——可能不支持跨分片交易和查询

但是:即使是读取,数据库也很慢,为什么不缓存读取查询结果?

4. 许多Web FE,许多用于读取的缓存,许多用于写入的DB

经济高效的 b/c 读取量大,memcached 比 DB 快 10 倍

memcached只是一个内存中的哈希表,非常简单

复杂的 b/c DB 和 memcached 可能会不同步

脆弱的 b/c 缓存未命中很容易使数据库过载

Facebook 基础设施大图

很多用户、朋友列表、状态、帖子、喜欢、照片

新鲜/一致的数据并不重要

人类是宽容的

高负载:每秒数十亿次操作

这是一台数据库服务器吞吐量的 10,000 倍

mysql 每秒约 100,000 个简单查询

~1,000 笔交易/秒

memcached 约 1,000,000 次 get/puts/s

多个数据中心(至少西海岸和东海岸)

每个数据中心——"区域":

通过 MySQL 数据库分片的"真实"数据

内存缓存层 (mc)

Web 服务器(memcached 的客户端)

每个数据中心的数据库都包含完整副本

西海岸是主要的,其他是通过 MySQL 异步日志复制的副本

FB应用程序如何使用mc? 图1。

FB 使用 mc 作为"后备"缓存

真实数据在数据库中

缓存值(如果有)应与数据库相同

读:

v = get(k)(计算 hash(k) 以选择 mc 服务器)

如果 v 为零{

v = 从数据库获取

集合(k,v)

}

写：
v = 新值
将 k,v 发送到 DB
删除(k)
应用程序确定 mc 与 DB 的关系
mc对DB一无所知

mc 有一个非常简单的 API，但可能很难使用！

FB在mc中存储什么？
纸上没说
可能是用户 ID -> 名称；用户ID -> 好友列表；帖子 ID -> 文本；网址 -> 喜欢
来自数据库查询的数据

纸质课程：
后备缓存比看起来更棘手——一致性
paper 试图整合相互忽略的存储层
缓存很关键：
并不是真正要减少用户可见的延迟
主要是为了保护数据库免受巨大的过载影响！
人类用户可以容忍适度的读取陈旧性
陈旧的读物仍然可能是一个令人头疼的问题
想要避免无限的陈旧性（例如完全缺少删除（））
想要读你自己写的
更多缓存 -> 更多陈旧来源
巨大的“扇出” => 并行获取，in-cast 拥塞

我们先来说说性能
大部分论文都是关于避免陈旧数据的
但陈旧仅源于性能设计

性能来自于许多服务器的并行性
许多活跃用户，许多网络服务器（客户端）
存储的两种基本并行策略：分区和复制

分区或复制会产生最大的 mc 吞吐量吗？
分区：在 mc 服务器上划分密钥
复制：将客户端划分到 mc 服务器上
分割：
+ 内存效率更高（每个 k/v 一份副本）
+ 如果没有钥匙很受欢迎，效果很好
- 每个 Web 服务器必须与许多 mc 服务器通信（开销）
复制：
+ 如果有几个键很受欢迎就很好
+ 可以将副本放在客户附近
- 可以缓存的总数据较少
- 写入成本更高

业绩和地区（第 5 节）

[图：west、db 主分片、mc 服务器、客户端 |
east，数据库辅助分片，...从数据库主分片传送到辅助分片]

问：区域的意义是什么——多个完整的副本？
降低用户的 RTT（东海岸、西海岸）
快速本地读取，从本地 mc 和 DB
（尽管写入成本很高：必须发送到主要区域）
也许是主站点故障的热副本？

问：为什么不对用户进行区域划分？
即为什么不是东海岸地区的东海岸用户数据，&c
那么无需复制：可能会将硬件成本减少一半！
但是：社交网络 -> 地方性不大
可能适用于电子邮件等

问：为什么尽管所有写入都被迫转到主要区域，但性能仍然良好？
写入比读取少得多
向主节点发送写入可能需要 100 毫秒，对于人类用户来说还不错
用户不必等待写入的所有效果完成
即删除所有过时的缓存值

区域内的绩效（第 4 节）

[图：数据库分片，多个集群，每个集群都有 mc 和客户端]

每个区域*内*有多个 mc 集群
cluster == 全套 mc 缓存服务器
即至少是缓存数据的副本

为什么每个区域有多个集群？
为什么不将更多的 mc 服务器划分到单个集群中？

1. 将 mc 服务器添加到集群对单个流行密钥没有帮助
   复制（每个集群一个副本）确实有帮助
2. 集群中有更多 mc -> 每个客户端请求与更多服务器通信
   以及请求 Web 服务器时出现更多的 in-cast 拥塞
   客户端请求获取 20 到 500 个密钥！在许多MC服务器上
   必须并行请求它们（否则总延迟太大）
   所以所有回复都会同时返回
   网络交换机、NIC 缓冲区耗尽
   3、单个大集群网络建设困难
   统一的客户端/服务器访问
   所以横截面黑白必须很大——昂贵
   两个簇 -> 1/2 横截面黑白

但是——对于不太受欢迎的项目来说，复制会浪费 RAM
所有集群共享的“区域池”
不受欢迎的对象（不需要很多副本）
应用程序软件决定将哪些内容放入区域池中
释放 RAM 来复制更流行的对象

启动新的 mc 集群是一个性能问题
新集群命中率为 0%
如果客户端使用它，将会在数据库负载中产生大峰值
如果平时有1%的错过率，
添加“冷”第二个集群将导致 50% 的操作失败。
即数据库负载激增 50 倍！

因此，新集群的客户端首先从现有集群中 get() (4.3)
并 set() 到新集群中
基本上将现有集群惰性复制到新集群
现有集群上的 2 倍负载比数据库上的 30 倍负载要好

另一个过载问题：雷霆群
一个客户端更新数据库并删除()sa密钥
很多客户 get() 但错过了
他们都从数据库获取
他们都设置了()
不好：不必要的数据库负载
mc 只给第一个失踪的客户"租约"
租约=从数据库刷新的权限
mc 告诉其他人"在几毫秒内再次尝试 get()"
效果：只有一个客户端读取数据库并执行 set()
其他人稍后重试 get() 并希望命中

如果 mc 服务器出现故障怎么办？
无法让数据库服务器处理未命中的情况——负载太大
无法将负载转移到另一台 mc 服务器 - 太多
无法重新分区所有数据——耗时
Gutter——空闲 mc 服务器池，客户端仅在 mc 服务器故障后使用
一段时间后，故障的MC服务器将被替换

问题：
为什么客户端不向 Gutter 服务器发送无效信息？
我的猜测：delete() 流量会增加一倍
并向小排水沟发送太多的delete()
因为任何钥匙都可能在阴沟里

重要的实际网络问题：
n^2 TCP 连接状态过多
因此客户端 get() 的 UDP
UDP 不可靠或不有序
因此 TCP 客户端 set()s
和 mcrouter 减少 n^2 中的 n
每个数据包单个请求效率不高（对于 TCP 或 UDP）
每个数据包的开销（中断和c）太高
因此 mcrouter 将许多请求批处理到每个数据包中

现在我们来谈谈一致性

他们的一致性目标是什么？
写入直接写入主数据库，并带有事务，因此数据库保持一致
那么阅读呢？
读取不能保证看到最新的写入
更像是"不超过几秒钟的陈旧"
即最终的
*和*作者看到自己的写入（由于delete()）
阅读自己写的文章是一个很大的推动力

首先，数据库副本如何保持跨区域的一致性？
一个区域是主要的
主数据库将更新日志分发到次要区域的数据库
辅助 DB 适用
辅助数据库是完整的副本（不是缓存）
数据库复制延迟可能相当大（很多秒）

他们如何保持MC内容与DB内容一致？

1. DB 向所有可能缓存的 mc 服务器发送无效（delete()）
   这是图 6 中的 McSqueal
   2.写client也会使本地集群中的mc失效
   用于阅读你自己写的内容

他们遇到了许多 DB-vs-mc 一致性问题
由于多个客户端从 DB 读取数据并将() 放入 mc 时的竞争
或者：不存在一条更新按顺序流动的路径

比赛和修复是什么？

第 1 场比赛：
k 不在缓存中
C1 获得(k)，未命中
C1 v1 = 从 DB 读取 k
C2 将 k = v2 写入 DB
C2 删除(k)
C1集合(k, v1)
现在 mc 具有陈旧数据，delete(k) 已经发生
将无限期地保持陈旧状态，直到下一次写入 k
通过租约解决——C1从mc获得租约，C2的delete()使租约无效，
所以 mc 忽略 C1 的集合
密钥仍然丢失，所以下一个读者将从数据库刷新它

比赛2：
在冷集群预热期间
请记住：如果失败，客户端会在热集群中尝试 get()，然后复制到冷集群
k 从值 v1 开始
C1将DB中的k更新为v2
C1删除(k)——冷集群中
C2 get(k), miss -- 在冷簇中
C2 v1 = 从热集群获取(k)，命中
C2 将（k，v1）设置为冷集群
现在 mc 已经过时了 v1，但是 delete() 已经发生了
将无限期地保持陈旧状态，直到下次写入密钥为止
通过两秒延迟解决，仅用于冷集群
在 C1 delete() 之后，cold mc 忽略 set()s 两秒
到那时，delete() 将（可能）通过数据库传播到热集群

第3场比赛：
k 从值 v1 开始
C1 处于次要区域
C1 更新主数据库中的 k=v2
C1 delete(k) -- 本地区域

C1 得到(k)，未命中
C1 读取本地数据库——看到 v1，而不是 v2！
之后，v2 从主数据库到达
通过“远程标记”解决
C1 delete() 将键标记为“远程”
get() 未命中产生“远程”
告诉 C1 从 *primary* 区域读取
当新数据从主要区域到达时清除“远程”

Q：这些问题不都是客户端将DB数据复制到mc造成的吗?
为什么不让 DB 向 mc 发送新值，这样客户端只读取 mc 呢?
那么就不会有赛车客户端更新等，只是命令写入
A：

1. DB通常不知道如何计算mc的值
   通常客户端应用程序代码根据数据库结果计算它们，
   即 mc 内容通常不仅仅是文字数据库记录
   2.会增加read-your-own写入延迟
   3.数据库不知道缓存了什么，最终会发送大量数据
   未缓存的键的值

存储系统设计人员的 FB/mc 课程?
缓存对于吞吐量生存至关重要，而不仅仅是减少延迟
需要灵活的工具来控制分区与复制
需要更好的想法来集成存储层并保持一致性

- ○ 参考

http://cs.cmu.edu/~beckmann/publications/papers/2020.osdi.cachelib.pdf

英文原文

6.824 2022 Lecture 16: Scaling Memcache at Facebook

Scaling Memcache at Facebook, by Nishtala et al, NSDI 2013

why are we reading this paper?
it's an experience paper, not about new ideas/techniques
three ways to read it:
cautionary tale about not taking consistency seriously from the start
impressive story of super high capacity from mostly-off-the-shelf s/w
fundamental struggle between performance and consistency
we can argue with their design, but not their success

how do web sites cope as they get more users?
a typical story of evolution over time:

1. single machine w/ web server + application + DB
   DB provides persistent storage, crash recovery, transactions, SQL
   application queries DB, formats HTML, &c
   but: as load grows, application takes too much CPU time
2. many web FEs, one shared DB
   an easy change, since web server + app already separate from DB
   FEs are stateless, all sharing (and concurrency control) via DB

stateless -> any FE can serve any request, no harm from FE crash
but: as load grows, need more FEs, soon single DB server is bottleneck
3. many web FEs, data sharded over cluster of DBs
partition data by key over the DBs
app looks at key (e.g. user), chooses the right DB
good DB parallelism if no data is super-popular
painful -- cross-shard transactions and queries probably not supported
but: DBs are slow, even for reads, why not cache read query results?
4. many web FEs, many caches for reads, many DBs for writes
cost-effective b/c read-heavy and memcached 10x faster than a DB
memcached just an in-memory hash table, very simple
complex b/c DB and memcacheds can get out of sync
fragile b/c cache misses can easily overload the DB

the big facebook infrastructure picture
lots of users, friend lists, status, posts, likes, photos
fresh/consistent data not critical
humans are tolerant
high load: billions of operations per second
that's 10,000x the throughput of one DB server
~100,000 simple queries/s for mysql
~1,000s transaction/s
~1,000,000 get/puts/s for memcached
multiple data centers (at least west and east coast)
each data center -- "region":
"real" data sharded over MySQL DBs
memcached layer (mc)
web servers (clients of memcached)
each data center's DBs contain full replica
west coast is primary, others are replicas via MySQL async log replication

how do FB apps use mc? Figure 1.
FB uses mc as a "look-aside" cache
real data is in the DB
cached value (if any) should be same as DB
read:
v = get(k) (computes hash(k) to choose mc server)
if v is nil {
v = fetch from DB
set(k, v)
}
write:
v = new value
send k,v to DB
delete(k)
application determines relationship of mc to DB
mc doesn't know anything about DB

mc has a very simple API but can be hard to use!

what does FB store in mc?
  paper does not say
  maybe userID -> name; userID -> friend list; postID -> text; URL -> likes
  data derived from DB queries

paper lessons:
  look-aside caching is trickier than it looks -- consistency
    paper is trying to integrate mutually-oblivious storage layers
  cache is critical:
    not really about reducing user-visible delay
    mostly about shielding DB from huge overload!
  human users can tolerate modest read staleness
  stale reads nevertheless potentially a big headache
    want to avoid unbounded staleness (e.g. missing a delete() entirely)
    want read-your-own-writes
    more caches -> more sources of staleness
  huge "fan-out" => parallel fetch, in-cast congestion

let's talk about performance first
  majority of paper is about avoiding stale data
  but staleness only arose from performance design

performance comes from parallelism due to many servers
  many active users, many web servers (clients)
  two basic parallel strategies for storage: partition and replication

will partition or replication yield most mc throughput?
  partition: divide keys over mc servers
  replicate: divide clients over mc servers
  partition:
    + more memory-efficient (one copy of each k/v)
    + works well if no key is very popular
    - each web server must talk to many mc servers (overhead)
  replication:
    + good if a few keys are very popular
    + can put replicas near clients
    - less total data can be cached
    - writes are more expensive

performance and regions (Section 5)

[diagram: west, db primary shards, mc servers, clients |
 east, db secondary shards, ... feed from db primary to secondary]

Q: what is the point of regions -- multiple complete replicas?
    lower RTT to users (east coast, west coast)
    quick local reads, from local mc and DB
    (though writes are expensive: must be sent to primary region)
    maybe hot replica for main site failure?

Q: why not partition users over regions?
    i.e. why not east-coast users' data in east-coast region, &c
    then no need to replicate: might cut hardware costs in half!
    but: social net -> not much locality

might work well for e.g. e-mail

Q: why OK performance despite all writes forced to go to the primary region?
writes are much rarer than reads
perhaps 100ms to send write to primary, not so bad for human users
users do not wait for all effects of writes to finish
i.e. for all stale cached values to be deleted

performance within a region (Section 4)

[diagram: db shards, multiple clusters, each w/ mc's and clients]

multiple mc clusters *within* each region
cluster == complete set of mc cache servers
i.e. a replica, at least of cached data

why multiple clusters per region?
why not partition over more mc servers to a single cluster?

1. adding mc servers to cluster doesn't help single popular keys
   replicating (one copy per cluster) does help
2. more mcs in cluster -> each client req talks to more servers
   and more in-cast congestion at requesting web servers
   client requests fetch 20 to 500 keys! over many mc servers
   MUST request them in parallel (otherwise total latency too large)
   so all replies come back at the same time
   network switches, NIC run out of buffers
3. hard to build network for single big cluster
   uniform client/server access
   so cross-section b/w must be large -- expensive
   two clusters -> 1/2 the cross-section b/w

but -- replicating is a waste of RAM for less-popular items
"regional pool" shared by all clusters
unpopular objects (no need for many copies)
the application s/w decides what to put in regional pool
frees RAM to replicate more popular objects

bringing up new mc cluster is a performance problem
new cluster has 0% hit rate
if clients use it, will generate big spike in DB load
if ordinarily 1% miss rate,
adding "cold" second cluster will causes misses for 50% of ops.
i.e. 50x spike in DB load!
thus the clients of new cluster first get() from existing cluster (4.3)
and set() into new cluster
basically lazy copy of existing cluster to new cluster
better 2x load on existing cluster than 30x load on DB

another overload problem: thundering herd
one client updates DB and delete()s a key
lots of clients get() but miss
they all fetch from DB
they all set()

not good: needless DB load
mc gives just the first missing client a "lease"
lease = permission to refresh from DB
mc tells others "try get() again in a few milliseconds"
effect: only one client reads the DB and does set()
others re-try get() later and hopefully hit

what if an mc server fails?
can't have DB servers handle the misses -- too much load
can't shift load to one other mc server -- too much
can't re-partition all data -- time consuming
Gutter -- pool of idle mc servers, clients only use after mc server fails
after a while, failed mc server will be replaced

The Question:
why don't clients send invalidates to Gutter servers?
my guess: would double delete() traffic
and send too many delete()s to small gutter pool
since any key might be in the gutter pool

important practical networking problems:
n^2 TCP connections is too much state
thus UDP for client get()s
UDP is not reliable or ordered
thus TCP for client set()s
and mcrouter to reduce n in n^2
single request per packet is not efficient (for TCP or UDP)
per-packet overhead (interrupt &c) is too high
thus mcrouter batches many requests into each packet

let's talk about consistency now

what is their consistency goal?
writes go direct to primary DB, with transactions, so DB stays consistent
what about reads?
reads not guaranteed to see the latest write
more like "not more than a few seconds stale"
i.e. eventual
and writers see their own writes (due to delete())
read-your-own-writes is a big driving force

first, how are DB replicas kept consistent across regions?
one region is primary
primary DBs distribute log of updates to DBs in secondary regions
secondary DBs apply
secondary DBs are complete replicas (not caches)
DB replication delay can be considerable (many seconds)

how do they keep mc content consistent w/ DB content?

1. DBs send invalidates (delete()s) to all mc servers that might cache
   this is McSqueal in Figure 6

2. writing client also invalidates mc in local cluster
for read-your-own-writes

they ran into a number of DB-vs-mc consistency problems
due to races when multiple clients read from DB and put() into mc
or: there is not a single path along which updates flow in order

what were the races and fixes?

Race 1:
k not in cache
C1 get(k), misses
C1 v1 = read k from DB
C2 writes k = v2 in DB
C2 delete(k)
C1 set(k, v1)
now mc has stale data, delete(k) has already happened
will stay stale indefinitely, until k is next written
solved with leases -- C1 gets a lease from mc, C2's delete() invalidates lease,
so mc ignores C1's set
key still missing, so next reader will refresh it from DB

Race 2:
during cold cluster warm-up
remember: on miss, clients try get() in warm cluster, copy to cold cluster
k starts with value v1
C1 updates k to v2 in DB
C1 delete(k) -- in cold cluster
C2 get(k), miss -- in cold cluster
C2 v1 = get(k) from warm cluster, hits
C2 set(k, v1) into cold cluster
now mc has stale v1, but delete() has already happened
will stay stale indefinitely, until key is next written
solved with two-second hold-off, just used on cold clusters
after C1 delete(), cold mc ignores set()s for two seconds
by then, delete() will (probably) propagate via DB to warm cluster

Race 3:
k starts with value v1
C1 is in a secondary region
C1 updates k=v2 in primary DB
C1 delete(k) -- local region
C1 get(k), miss
C1 read local DB -- sees v1, not v2!
later, v2 arrives from primary DB
solved by "remote mark"
C1 delete() marks key "remote"
get() miss yields "remote"
tells C1 to read from *primary* region
"remote" cleared when new data arrives from primary region

Q: aren't all these problems caused by clients copying DB data to mc?
  why not instead have DB send new values to mc, so clients only read mc?
    then there would be no racing client updates &c, just ordered writes
A:

1. DB doesn't generally know how to compute values for mc
   generally client app code computes them from DB results,
     i.e. mc content is often not simply a literal DB record
2. would increase read-your-own writes delay
3. DB doesn't know what's cached, would end up sending lots
   of values for keys that aren't cached

FB/mc lessons for storage system designers?
  cache is vital for throughput survival, not just to reduce latency
  need flexible tools for controlling partition vs replication
  need better ideas for integrating storage layers with consistency

--- references

http://cs.cmu.edu/~beckmann/publications/papers/2020.osdi.cachelib.pdf

LEC 17 Causal Consistency，COPS

中文翻译

6.824 2022年第17讲：因果一致性，COPS

Lloyd 等人，不要满足于最终结果：可扩展的因果一致性
用于 COPS 的广域存储，SOSP 2011。

设置：大型网站的地理复制
 [3 个数据中心、用户、Web 服务器、分片存储服务器]
 多个数据中心，贴近用户
 每个数据中心都有所有数据的完整副本
 读取速度很快——从本地副本
 写入也可以快吗?

我们已经看到了两种异地复制解决方案
 扳手
  读取是本地的并且速度快
  写入提交等待其他数据中心的回复
  读写事务
 Facebook / 内存缓存
  读取是本地的并且速度快
  所有写入都通过主数据中心
  交易仅用于更新
 问题：
  写入必须等待广域通信
  缓慢的数据中心间网络使写入速度更慢
  发生故障的数据中心可以阻止其他数据中心的写入

我们是否可以拥有永远不必等待其他数据中心的写入操作?
与慢速 WAN 相比，将有助于容错、性能、稳健性
那么我们就可以进行快速的本地读取*和*写入
难题：对于一致性模型我们能做的最好的事情是什么?

理念：最终一致性
客户端读写只需联系本地分片
每个分片将写入异步推送到其他数据中心，分片到分片
也就是说， put() 向每个数据中心发送一条消息，
但不等待，可能会出现长时间的延迟和失序。

最终一致性属性

1. 数据中心/客户端可能会以不同的顺序看到更新
2.如果足够长的时间没有写入，所有客户端都会看到相同的数据
规范相当宽松，实现方法很多，很容易获得良好的性能
用于已部署的系统，例如 Dynamo 和 Cassandra
但对于应用程序程序员来说可能很棘手

示例应用程序代码——照片管理器：
C1和C2位于不同的数据中心
C1上传照片，添加参考列表：
C1: put(照片) put(列表)
C2 内容如下：
C2：获取（列表）获取（照片）

C2能看到什么?

应用程序代码可以看到非直观行为——“异常”
没有错，因为没有更好的承诺
并且可以对这样的系统进行编程
也许等待预期数据（照片）出现
稍后我们将转向 COPS，它提供更好的行为

如何确保所有数据中心最终具有相同的值?
如果有来自多个数据中心的写入
每个人都必须选择相同的最终值，以实现最终的一致性

为什么不在每次放置时附加当前挂钟时间作为版本号?
所有数据中心都喜欢更高的版本号吗?
本地分片服务器收到客户端 put() 时分配 v#=time
远程数据中心接收 put(k, val, v#)
如果 v# 大于当前存储的 k 值的版本
替换为新值 / v#
否则
忽略新值
“时间戳”在这里可能是比“版本”更好的词
如果两个 put(k) 在不同的数据中心同时发生怎么办?
使用 v# 低位中的唯一 ID 打破平局
好的：现在不同的数据中心将就最终值达成一致
所以最终他们会变得一致

如果一个数据中心（或服务器）的时钟快一个小时怎么办
 将导致该数据中心的价值获胜，即使是一个小时前！
 更糟糕的是：一个小时内阻止任何其他更新！

COPS 使用“Lamport 时钟”来分配 v#
 每个服务器都实现一个“Lamport时钟”或“逻辑时钟”
  Tmax = 看到的最高 v#（来自自己和他人）
  T = max(Tmax + 1, 挂钟时间)
 新 put() 的 v# 是当前 T
 所以：如果某个服务器的时钟很快，每个看到版本的人都会看到
  从该服务器将提前他们的兰波特时钟

如果并发写入，是否可以简单地丢弃除一个之外的所有写入？
 该报的“最后作者获胜”
 有时没关系：
  例如，只有一个可能的作者，因此不会出现问题
  可能我是唯一可以写我的照片列表或个人资料的人
 有时，latest-write-wins 没有用：
  如果 put() 试图增加计数器怎么办？
  或者更新购物车以添加新商品？
 问题是“写入冲突”
 我们经常希望有一个更聪明的计划来检测和合并
  迷你事务——原子增量操作，而不仅仅是 get()/put()
  购物车的自定义冲突解决方案（设置并集？）
 冲突写入的解决是最终/因果一致性的问题
  没有单一的主要命令原子操作或事务
 该论文主要忽略了写入冲突解决
  但这对于实际系统来说是一个严重的问题

我们能否拥有比最终更好的一致性？
 并且仍然保留来自任何数据中心的快速本地读取和写入？

记住照片列表的最终一致性异常：
 C1: put(照片) put(列表)
 C2：获取（列表）获取（照片）
C1和C2位于不同的数据中心
问题： put() 到达其他数据中心时可能会乱序。
 所以 get(list) 可能会看到新的照片名称，但 get(photo) 失败
我们需要强制写入顺序
 即使“照片”和“列表”位于不同的分片中

一种可能性：每个数据中心单个写入日志
 每个数据中心都有一个“日志服务器”
 put() 追加到本地服务器的日志，但不等待
 日志服务器在后台按顺序将日志发送到其他数据中心
  远程站点按顺序应用登录
  所以 put(photo), put(list) 会按这个顺序出现在任何地方
 两个问题：
  日志服务器将成为瓶颈
  仅命令从此数据中心写入

来自多个数据中心的写入示例：
最初 x=0 y=0
C1：放置（x，2）
C2: t=get(x) put(y,t)
C3：获取（y）获取（x）
不同数据中心的C1/C2/C3
假设 C2 看到 x=2 （即它观察到 C1 的写入）

在此示例中，每个数据中心的日志都会产生异常
C1 的 put 和 C2 的 put 将位于不同的日志中，未排序
所以 C3 可能会看到 C2 的 put，但看不到 C1 的（之前的）put
即 C3 可能会看到 y=2 但 x=0
我们需要一个传递的顺序概念

所以：
我们想要异步转发 put
我们希望每个分片独立转发 put（没有中央日志服务器）
我们需要足够的排序以使我们的示例具有直观的结果

以下是 COPS 如何实现这些目标

每个 COPS 客户端维护一个“上下文”来反映客户端操作的顺序
客户端在每次 get() 和 put() 之后向上下文添加一个项目
客户端告诉 COPS 在其上下文中的所有内容之后对每个 put() 进行排序
C2 的 get(x) 产生版本 v2
C2 现在的上下文：x/v2
C2 的 put(y,2)
C2发送put(y,2,x/v2)到本地分片服务器
y 的本地分片服务器：
为 y 选择一个新的 v# = 3，
存储 y、2、v3
将 y,2,v3,x/v2 发送到每个数据中心中相应的分片服务器
但不等待回复
远程分片服务器接收 y,2,v3,x/v2
与 x 的本地分片服务器对话
等待 x 达到版本 v2（来自 C1 的数据中心）
然后更新数据库以保存 y,2,v3

COPS 将“y/v3 在 x/v2 之后”这样的关系称为“依赖关系”
x/v2 -> y/v3

依赖性对于 COPS 意味着什么？
如果 C3 看到 y/v3，然后要求 x，它应该至少看到 x/v2
这种依赖的概念是为了符合程序员的直觉
关于 get(x) AND THEN put(y) 的含义
或 put(x) 然后 put(y)

这些一致性语义称为“因果一致性”
论文在第 3 节/图 2 中定义
客户端通过两种方式建立版本之间的依赖关系：
1. 它自己的 put 和 gets 序列（第 3 节中的“执行线程”）
2.读取另一个客户端写入的数据
依赖关系是可传递的（如果 A -> B，且 B -> C，则 A -> C）

系统保证如果 A -> B，并且客户端读取 B，
那么客户端随后必须看到 A（或更高版本）

因果一致性使照片列表示例有效
put(list) 将以照片作为依赖项到达
远程服务器将等待照片出现后再安装更新列表

很好：当更新有因果关系时，读者会看到版本
至少和作者看到的一样新。

很好：当更新没有因果关系时，COPS 没有订单义务
例子：
C1: put(x,1)
C2：放置（y，2）
C3: put(x,3)
这些没有因果关系
因此它们可以以不同的顺序出现在不同的数据中心
无需任何机制来强制执行出场顺序协议
例如，x 没有单一原色
例如，无需像线性化一样等待写入完成
我们可以期望这种自由有助于提高绩效

但是：读者可能会看到比因果依赖关系所需的更新的数据
所以我们没有得到交易或快照

但是：COPS 只看到某些因果关系
COPS 可以从客户端 get() 和 put() 观察到的内容
如果有其他沟通渠道，COPS 只有最终一致
例如我放(k)，通过电话告诉你，你确实得到(k)，也许你看不到我的数据

COPS 优化避免不断增长的客户端环境

- 客户端在 put(x) 之后清除上下文，替换为 x/v#
  所以接下来的 put()，例如 put(y)，仅取决于 x/v#
  因此远程站点将在写入 y 之前等待 x/v# 的到达
  x/v# 本身正在等待（清除）上下文
  所以 y 实际上也在等待（清除的）上下文
- 当所有数据中心都有某个版本时垃圾收集会看到
  该版本永远不需要在上下文中记住
  因为它已经对所有人可见

是否存在有序 put/gets 不够的情况?
是：当您需要事务或原子更新时
论文示例：带有 ACL（访问控制列表）的照片列表
获取（ACL），然后获取（列表）？
如果有人从 ACL 中删除您，然后添加敏感照片怎么办?
在两个 get() 之间
获取（列表），然后获取（ACL）？
如果有人删除敏感照片，然后将您添加到 ACL 该怎么办?
需要一个返回相互一致版本的多键 get

COPS-GT get_trans() 提供只读排序事务
  服务器存储每个值的完整依赖关系集
  服务器存储每个值的一些旧版本
  get_trans(k1,k2,...)
    客户端库执行独立的 get()
    get() 返回依赖项以及 value/v#
  客户端检查依赖关系
    对于每个 get() 结果 R，
      彼此得到 R 依赖项中提到的结果 S，
        R 的依赖项中是否提到了 S/v# >= 版本?
  如果全部为“是”，则可以使用结果
  如果没有，则需要对太旧的值进行第二轮 get()
    每个都获取依赖项中提到的版本
    可能是旧的：避免级联依赖

对于 ACL/列表示例：
  C1: get_trans(ACL, 列表)
  C1：get(ACL) -> v1，无依赖项
      C2：放置（ACL，v2）
      C2: put(列表, v2, deps=ACL/v2)
  C1: get(list) -> v2, 依赖项: ACL/v2
    （C1 检查值版本的依赖关系）
  C1：获取（ACL，v2）-> v2
    （现在 C1 具有一对因果一致的 get() 结果）

为什么 COPS-GET get 交易只需要两个阶段?
  在安装所有依赖项之前，不会安装新值
  因此，如果 get() 返回依赖项，则它必须已在本地安装

表现?
  第 6.2 节：
    每个数据中心有一个客户端、一台服务器
    每秒 52,000 次获取
    每秒 30,000 次放入（异步转发到远程数据中心）
  性能还不错，就像传统的数据库一样
  但没有与其他系统进行比较
    不是为了表现
    不是为了方便编程
  太糟糕了，因为中心论点是 COPS 在
    易于编程和性能

COPS 和因果一致性的局限性/缺点
  冲突的写入是一个严重的困难
  客户很难追踪因果关系
    例如用户和浏览器、多个页面视图、多个服务器
  COPS 没有看到外部因果关系
    软件和人们确实在 COPS 世界之外进行交流
  “交易”的有限概念
    仅用于读取（尽管后来的工作有点概括）
    定义比可序列化事务更微妙
  显着的开销
    跟踪、沟通、强化因果依赖性

垃圾收集
更新延迟可能会级联

影响?
因果一致性是一个流行的研究思想
有充分的理由：保证性能和有用的一致性
COPS 之前以及之后（艾格峰、SNOW、神秘学）做了很多工作
因果一致性很少在已部署的存储系统中使用
实际使用的是什么?
根本没有地理复制，只是本地复制
主站点（PNUTS、Facebook/Memcache）
最终一致性（Dynamo、Cassandra）
强一致（扳手）

英文原文

6.824 2022 Lecture 17: Causal Consistency, COPS

Lloyd et al, Don't Settle for Eventual: Scalable Causal Consistency
for Wide-Area Storage with COPS, SOSP 2011.

the setting: geo-replication for big web sites
[3 datacenters, users, web servers, sharded storage servers]
multiple datacenters, to be close to users
each datacenter has a complete copy of all data
reads are fast -- from local replica
can writes be fast also?

we've seen two solutions for geo-replication
Spanner
reads are local and fast
write commit waits for replies from other datacenters
transactions for read and write
Facebook / Memcache
reads are local and fast
all writes go through primary datacenter
transactions only for updates
problems:
writes must wait for wide-area communication
slow inter-datacenter network makes writes even slower
failed datacenter can block writes at other datacenters

can we have writes that never have to wait for other datacenters?
would help fault tolerance, performance, robustness vs slow WAN
then we'd have fast local reads *and* writes
puzzle: what's the best we can then do for a consistency model?

idea: Eventual Consistency
client reads and writes just contact local shard
each shard pushes writes to other datacenters, shard-to-shard, asynchronously
that is, a put() fires off a message to each other datacenter,
but does not wait, there could be long delays and out-of-order.

Eventual Consistency properties

1. datacenters/clients may see updates in different orders
2. if no writes for long enough, all clients see same data
   a pretty loose spec, many ways to implement, easy to get good performance
   used in deployed systems, e.g. Dynamo and Cassandra
   but can be tricky for app programmers

example app code -- a photo manager:
C1 and C2 in different datacenters
C1 uploads photo, adds reference to list:
C1: put(photo) put(list)
C2 reads:
C2: get(list) get(photo)

what can C2 see?

app code can see non-intuitive behavior -- "anomalies"
not incorrect, since there was no promise of better
and it is possible to program such a system
perhaps wait for expected data (photo) to appear
in a bit we'll turn to COPS, which provides better behavior

how to ensure that all datacenters end up with the same value?
if there are writes from multiple datacenters
everyone has to choose the same final value, for eventual consistency

why not attach the current wall-clock time as version number on each put?
and have all datacenters prefer higher version numbers?
local shard server assigns v#=time when it receives client put()
remote datacenter receives put(k, val, v#)
if v# is larger than version of currently stored value for k
replace with new value / v#
otherwise
ignore new value
"timestamp" might be a better word here than "version"
what if two put(k) happen at exactly the same time at different datacenters?
break tie with a unique ID in the low bits of v#
ok: now different datacenters will agree on final values
so, eventually, they will become consistent

what if one datacenter's (or server's) clock is fast by an hour
will cause that datacenter's values to win, even if an hour old!
worse: prevents any other update for an hour!

COPS uses "Lamport clocks" to assign v#
each server implements a "Lamport clock" or "logical clock"
Tmax = highest v# seen (from self and others)
T = max(Tmax + 1, wall-clock time)
v# for a new put() is current T
so: if some server has a fast clock, everyone who sees a version
from that server will advance their Lamport clock

if concurrent writes, is it OK to simply discard all but one?
  the paper's "last-writer-wins"
  sometimes that's OK:
    e.g. there's only a single possible writer, so the problem can't arise
    probably I'm the only person who can write my photo list or profile
  sometimes latest-write-wins is not useful:
    what if put()s are trying to increment a counter?
    or update a shopping cart to have a new item?
  the problem is "conflicting writes"
  we'd often like to have a more clever plan to detect and merge
    mini-transactions -- atomic increment operation, not just get()/put()
    custom conflict resolution for shopping cart (set union?)
  resolution of conflicting writes is a problem for eventual/causal consistency
    no single primary to order atomic operations or transactions
  the paper mostly ignores write conflict resolution
    but it's a serious problem for real systems

can we have better consistency than eventual?
  and still retain fast local reads and writes, from any datacenter?

remember the eventual consistency anomaly with the photo list:
  C1: put(photo) put(list)
  C2:   get(list) get(photo)
C1 and C2 in different datacenters
the problem: the put()s may arrive out of order at other datacenters.
  so get(list) may see the new photo name, but get(photo) fails
we need to enforce order on writes
  even when "photo" and "list" are in different shards

a possibility: single write log per datacenter
  each datacenter has a single "log server"
  put() appends to the local server's log, but doesn't wait
  log server sends log, in order, to other datacenters, in the background
    remote sites apply log in order
    so put(photo), put(list) will appear in that order everywhere
  two problems:
    log server will be a bottleneck
    only orders writes from this datacenter

example with writes from multiple datacenters:
  initially x=0 y=0
  C1: put(x,2)
  C2:   t=get(x) put(y,t)
  C3:   get(y) get(x)
  C1/C2/C3 in different data centers
  suppose C2 sees x=2 (i.e. it observes C1's write)

a log per datacenter produces an anomaly for this example
  C1's put and C2's put would be in different logs, not ordered
  so C3 might see C2's put but not C1's (prior) put
    i.e. C3 might see y=2 but x=0
  we need a transitive notion of order

so:
  we want to forward puts asynchronously
  we want each shard to forward puts independently (no central log server)
  we want enough ordering to make our examples have intuitive results

here's how COPS achieves these goals

each COPS client maintains a "context" to reflect order of client ops
  client adds an item to context after each get() and put()
  client tells COPS to order each put() after everything in its context
  C2's get(x) yields version v2
    C2's context now: x/v2
  C2's put(y,2)
    C2 sends put(y,2,x/v2) to local shard server
  local shard server for y:
    picks a new v# = 3 for y,
    stores y, 2, v3
    sends y,2,v3,x/v2 to corresponding shard server in each datacenter
      but does not wait for reply
  remote shard server receives y,2,v3,x/v2
    talks to local shard server for x
      waits for x to reach version v2 (from C1's datacenter)
    then updates DB to hold y,2,v3

COPS calls a relationship like "y/v3 comes after x/v2" a "dependency"
  x/v2 -> y/v3

what does a dependency imply for COPS?
  if C3 sees y/v3, then asks for x, it should see at least x/v2
  this notion of dependency is meant to match programmer intuition
    about what it means to get(x) AND THEN put(y)
    or put(x) AND THEN put(y)

these consistency semantics are called "causal consistency"
  paper defines in Section 3 / Figure 2
  a client establishes dependencies between versions in two ways:
    1. its own sequence of puts and gets ("Execution Thread" in Section 3)
    2. reading data written by another client
  dependencies are transitive (if A -> B, and B -> C, then A -> C)
  the system guarantees that if A -> B, and a client reads B,
    then the client must subsequently see A (or a later version)

causal consistency makes the photo list example work
  put(list) will arrive with photo as a dependency
  remote servers will wait for photo to appear before installing updated list

nice: when updates are causally related, readers see versions
  at least as new as the writer saw.

nice: when updates are NOT causally related, COPS has no order obligations
  example:
    C1: put(x,1)
    C2:   put(y,2)

C3: put(x,3)
these are not causally related
so they can appear in different orders at different datacenters
no need for any mechanisms to enforce agreement on order of appearance
e.g. no single primary for x
e.g. no wait for write to complete as in linearizability
we can expect this freedom to help performance

but: readers may see newer data than required by causal dependencies
so we're not getting transactions or snapshots

but: COPS sees only certain causal relationships
ones that COPS can observe from client get()s and put()s
if there are other communication channels, COPS is only eventually consistent
e.g. I put(k), tell you by phone, you do get(k), maybe you won't see my data

COPS optimizations avoid ever-growing client contexts

- client clears context after put(x), replaces with x/v#
  so next put(), e.g. put(y), depends only on x/v#
  so remote sites will wait for arrival of x/v# before writing y
  x/v# itself was waiting for (cleared) context
  so y effectively also waits for (cleared) context
- garbage collection sees when all datacenters have a certain version
  that version never needs to be remembered in context
  since it's already visible to everyone

are there situations where ordered puts/gets aren't sufficient?
yes: when you need transactions or atomic update
paper's example: a photo list with an ACL (Access Control List)
get(ACL), then get(list)?
what if someone deletes you from ACL, then adds a sensitive photo?
between the two get()s
get(list), then get(ACL)?
what if someone deletes sensitive photo, then adds you to ACL?
need a multi-key get that returns mutually consistent versions

COPS-GT get_trans() provides read-only sort-of transactions
servers store full set of dependencies for each value
servers store a few old versions of each value
get_trans(k1,k2,...)
client library does independent get()s
get()s return dependencies as well as value/v#
client checks dependencies
for each get() result R,
for each other get result S mentioned in R's dependencies,
is S/v# >= version mentioned in R's dependency?
if yes for all, can use results
if no for any, need a second round of get()s for values that were too old
each fetches the version mentioned in dependencies
may be old: to avoid cascading dependencies

for ACL / list example:
  C1: get_trans(ACL, list)
  C1: get(ACL) -> v1, no deps
    C2: put(ACL, v2)
    C2: put(list, v2, deps=ACL/v2)
  C1: get(list) -> v2, deps: ACL/v2
  (C1 checks dependencies against value versions)
  C1: get(ACL,v2) -> v2
  (now C1 has a causally consistent pair of get() results)

why are only two phases needed for COPS-GET get transactions?
  a new value won't be installed until all its dependencies are installed
  so if a get() returns a dependency, it must already be locally installed

performance?
  Section 6.2:
    one client, a single server in each datacenter
    52,000 gets per second
    30,000 puts per second (async forwarding to remote datacenter)
  that is OK performance, like a conventional DB
  but no comparisons with other systems
    not for performance
    not for ease of programming
  too bad, since central thesis is that COPS has a better tradeoff between
    ease of programming and performance

limitations / drawbacks, for both COPS and causal consistency
  conflicting writes are a serious difficulty
  awkward for clients to track causality
    e.g. user and browser, multiple page views, multiple servers
  COPS doesn't see external causal dependencies
    s/w and people really do communicate outside of the COPS world
  limited notion of "transaction"
    only for reads (though later work generalized a bit)
    definition is more subtle than serializable transactions
  significant overhead
    track, communicate, enforce causal dependencies
    garbage collection
  update delays can cascade

impact?
  causal consistency is a popular research idea
    with good reason: promises both performance and useful consistency
    much work before COPS -- and after (Eiger, SNOW, Occult)
  causal consistency is rarely used in deployed storage systems
  what is actually used?
    no geographic replication at all, just local
    primary-site (PNUTS, Facebook/Memcache)
    eventual consistency (Dynamo, Cassandra)
    strongly consistent (Spanner)

LEC 18 Fork Consistency，SUNDR

中文翻译

6.824 2022 第 18 讲：安全不受信任的数据存储库 (SUNDR) (2004)

我们为什么要阅读这篇论文?
我们通常信任存储服务，例如 Google mail、AFS、Dropbox。
即使组织总体上是诚实的，坏事也会发生吗?
也许服务器软件或硬件有缺陷，甚至可以被利用。
也许攻击者猜测了服务器管理员密码，并修改了软件。
也许云提供商的员工腐败或草率。
我们能否从不可信的服务器获得可信的存储?
这是一个难题!
SUNDR 包含一些好主意。
类似的想法也出现在 git 和区块链中。
Keybase（通过zoom获取）直接受SUNDR影响
你们中有些人看过 6.858 中的 SUNDR
作为一个分布式系统（一致性、日志记录）都很有趣
以及为了安全。

环境：
一群程序员在一个项目上合作。
他们希望将源文件存储在共享的地方。
他们需要看到彼此的修改。
他们可以将源代码存储在 github、Google Drive 等中
但他们不想相信任何存储服务!

我们事先并不知道到底会出现什么问题。
所以我们假设最坏的情况：
服务器完全被恶意对手控制。
它是故意试图欺骗用户。
如果它不愿意，它就不会遵守任何规则或协议。
“拜占庭式的错误”。
但我们确实假设对手无法破解密码学，
并且只能控制服务器，不能控制我的客户端计算机。

潜在目标：
保密。
对手无法读取文件内容。
正直。
攻击者无法欺骗客户端获取错误的文件内容。
可用性。
对手无法阻止客户端访问其文件。

本文的重点是：诚信。

完整性是源代码存储服务真正关心的问题!
如果有人修改像 Linux 这样的大型项目的源代码怎么办?
数百万行代码!
可能过了很长一段时间才有人注意到。
论文提到 Debian 服务器在 2003 年遭到入侵。
SourceForge 于 2011 年遭到入侵。
[http://sourceforge.net/blog/sourceforge-attack-full-report]
Canonical (Ubuntu) 于 2019 年遭到入侵。

[https://hub.packtpub.com/canonical-the-company-behind-the-ubuntu-linux-distribution-was-hacked-ubuntu-source-code-unaffected/]

完整性不仅仅是防止非法修改，
还要确保合法的修改全部出现并出现在
正确的顺序。

完整性示例：
用户A、B、C正在开发一个简单的O/S
他们将源代码存储在“云端”
我们不希望云存储服务能够修改源。
我们也不希望服务隐藏或重新排序更新！
事实证明，这才是真正的问题。
用户A向login.c添加密码保护。
用户B向net.c添加远程登录。
用户 C 创建一个发布 tar 文件。
我们希望 C 版本同时具有 A 和 B 的更新。
而不仅仅是*B 的net.c。
因为这将允许无密码远程访问。
所以我们要防止服务泄露 A 的更新
但隐藏了B的更新。

这个例子的要点是：仅仅阻止服务器是不够的
改变数据，我们需要防止它隐藏合法的
也有更新！

朴素的设计：用户签署文件内容。
使用现有的网络文件系统（例如 Dropbox）作为起点。
不期望从中得到任何安全保证。
每个文件包含数据，Sig(SK_writer, data)。
当用户写入文件时，客户端使用用户的密钥进行签名。
当用户读取文件时，客户端检查签名。
如果数据与签名时不完全相同，签名检查将失败。
我们假设客户知道所有用户的公钥。

在这种幼稚的设计中，恶意服务器会做什么？
无法向用户 C 发送任意源代码。
可以发送一个文件的数据而不是另一文件的数据。
可能的修复：在签名中包含文件名。
可以发送文件的旧内容。
可以为某些文件发送新内容，为其他文件发送旧内容。
可以声明文件不存在。
不适合我们的操作系统开发场景：
攻击者可能会导致 C 部署 net.c 而无需更改 login.c。

我们需要更复杂的安全设计。
文件版本一致：net.c 更改需要 login.c 更改。
文件的最新版本：不应丢失任何更改。
目录内容的完整性：不应丢失任何文件。
权限：用户可能没有修改每个文件的权限。

SUNDR 的伟大创意：
对任何文件的每次更新都包含当前签名
状态对更新用户可见。
因此，B 对 net.c 的更新包括对 A 更新后的 login.c 的引用
如果C看到更新的net.c，也会知道新的login.c

稻草人设计：论文第 3.1 节。
文件系统状态由用户的操作日志决定。
服务器负责存储此日志。
客户端解释日志。

稻草人细节
日志条目：获取或修改、用户、sig。
签名涵盖了到此为止的整个日志。
客户端步骤：
下载日志（其他客户端正在等待）。
检查日志：
每个条目中的正确签名，覆盖日志前缀。
（实际上只需要检查每个用户的最后一个条目）
存在该客户端的最后一个日志条目。
根据记录的操作构建 FS 状态。
追加客户端的操作并签署新的日志。
上传日志（其他客户端现在可以继续）。
效率低下但推理简单。
日志示例：
X: mod(???), sig
A: mod(login.c), sig
B: mod(net.c), sig
C: fetch(login.c), sig
C: fetch(net.c), sig

至关重要的是，该签名涵盖了日志中所有先前的操作。
防止对手泄露 net.c 更改但隐藏 login.c 更改。
如果对手从日志中忽略了login.c更改，则B的签名无法验证。

对手可以签署虚假的日志条目吗？
它需要知道授权用户的私钥。
客户端如何知道授权用户的公钥？

1. 必须告知所有客户端根目录所有者的公钥。
2. 文件系统存储每个文件/目录所有者的公钥。
聪明：文件系统完整性确保公钥的完整性。

如果 C 不记录其获取会发生什么？
即，即使只是读取文件，也会附加“获取”日志条目。
恶意服务器可能会给 C 提供陈旧的日志视图。
当C获取login.c时，服务器在mod(login.c)之前给出日志。
当C获取net.c时，服务器给出带有mod(login.c)和mod(net.c)的日志。
实际上，服务器假装 C 与 A + B 进行了一场比赛。
结果：C 使用第一次获取中的 login.c 并使用第二次获取中的 net.c
哎呀；这正是我们试图阻止的。

当客户端获取日志时，它会检查其是否是最新的
fetch 或 mod 在日志中，如果没有则拒绝该日志。

记录获取有什么帮助？
服务器仍然可以向 C 显示 mod(login.c) 之前的日志。
但随后 C 根据该日志前缀记录其获取，
并记住该 fetch 作为其最后一个日志条目。
现在服务器无法向 C 显示 mod(login.c) 或 mod(net.c)，
因为他们在日志上有签名，而日志却没有
包含 C 的 fetch，并且 C 期望看到它的 fetch。
“分叉”攻击。

所以：服务器可以通过隐藏对login.c和net.c的修改来分叉C，
但服务器不能再向 C 显示任何后续操作
由 A 或 B，因为他们的签名不包括 C 的获取。
然而，服务器可以继续服务 C 的请求。
C 无法仅通过与服务器通信来意识到它已被分叉。
但服务器必须对 C 隐藏所有未来的 A/B 操作，
因为他们的日志条目签名不会包含 C 的获取。
同样，服务器必须对 A/B 隐藏 C 的操作。
“分叉一致性”：服务器可以分叉客户端，但一旦它
分叉，永远无法向客户端显示来自一个分叉的操作
在另一个叉子上。

对于文件系统来说，分叉一致性非常好。
留下强烈的攻击痕迹：不可能掩盖叉子。
用户可以检测他们是否可以在 SUNDR 之外进行通信。
例如，电子邮件询问“您对我上次提交有何看法？”

使用单独的可信“时间戳盒”进行自动分叉检测。
指定用户负责每 5 秒更新一些文件。
如果客户端看到这些更新，它与时间戳框位于同一个“分叉”中。
如果有一个分叉，时间戳框的更新只能显示在一个分叉中。
另一个分叉中的客户端将看不到时间戳框的更新！

稻草人不实用：
日志不断增长。
解释日志变得很慢。

想法：反映当前状态的块树，而不是日志。
图 2.
服务器存储较小的块，以便快速检索/更新，
通常情况下，您只对一个文件进行操作。
i-handle 指向 i-number->inode 映射表。
索引节点包含类型（文件与目录）和块引用列表。
目录包含名称->i-号码映射的列表。

块是不可变的！
引用（密钥）是内容的加密哈希值。
当客户端向服务器请求块时，它会发送密钥，
客户端可以检查返回的数据哈希值到该密钥。
因此，假设客户端知道正确的密钥，服务器无法修改数据。
但客户端也不能修改任何块！

如何更新?
当客户端C写入文件时，
它构造了一棵反映其修改的新树。
但新树可以与旧树共享几乎所有块，
只需要修改内容路径上的新块
直到 i-handle。
每次修改都有一个新的 i 手柄。

如何保持前叉与 i 型手柄的一致性?
恶意服务器可能会泄露旧的 i-handle，隐藏最近的更新。
我们无法阻止服务器分叉用户，但是（与
稻草人）我们想防止它合并叉子并且
从而掩盖其违法行为。
我们希望每个新的 i 句柄能够以某种方式对其之前的内容进行编码，
我们希望能够记录获取和修改。

想法：为每个用户提供一个签名的“版本结构”（VS）。
图 3.
保存在服务器中。
包含：
i-用户最后一次操作后的句柄。
版本向量（VV）：
对于每个用户，该用户执行了多少次操作。
用户的公钥签名。
要点：VV操作计数可以让客户端检测到
省略旧的操作，并检测服务器的尝试
合并叉子。

客户端U1如何执行操作（读和写）：
从服务器获取所有用户的VS。
证实。
从块服务器获取所需的 i-tables 和 c。
存储新的 i-table、i-handle 和 mods（如果有）。
新VS：
新的 i 型手柄。
增加 U1 的版本号。
将VS放回服务器

版本向量如何在正确的操作中演化?
U1: 1,0 2,2
U2：1,1 1,2 2,3

U2 应如何验证一组获取的 VS?
检查 U2 的 VS 是否是最新的（因此 U2 必须记住它自己的最后一个 VS）。
检查 diff 用户的版本向量是否可以全序。
例如 2,2 <= 2,3

如果服务器隐藏了 U2 的更新，版本向量会是什么样子?
U1: 1,0 [2,1]
U2：1,1 1,2

版本向量能给我们分叉一致性吗?
服务器能否向 U2 显示未来的 U1 VS?
例如3,1
否： 3,1 和 1,2 不能订购!
服务器可以向 U1 显示未来的 U2 VS 吗?
例如1,3
否：无法订购 1,3 和 2,1

对于我们的 A/B/C login.c net.c 示例，正确的执行：
A: 1,1,1 2,1,1 别的，然后修改login.c
B：2,2,1修改net.c
C：2,2,2读取login.c和net.c

如果服务器隐藏A对login.c的更新，但显示B的net.c，
它将向 C 发送这些 vv：
答：1,1,1
乙：2,2,1
C 将计算新的 VV z：1,2,2
但A/B/z不能全序
不是这样：
1,1,1
B 2,2,1
z 1,2,2
不是这样：
1,1,1
z 1,2,2
B 2,2,1
这样C就会知道出了什么问题。

很好：版本结构允许我们使用高效的树
数据结构，消除对不断增长的日志的需要，
但仍然可以强制分叉一致性。

概括。
难题：尽管服务器受到损害，但仍保持完整性。
客户签名可防止彻底伪造。
隐藏和分叉是仍然可能发生的主要攻击。
分叉一致性可防止服务器隐藏分叉创建后的情况。
分叉仍然是可能的，但最终对客户来说会变得显而易见。
检测到分叉后没有恢复计划。

英文原文

6.824 2022 Lecture 18: Secure Untrusted Data Repository (SUNDR) (2004)

Why are we reading this paper?
We routinely trust storage services, e.g Google mail, AFS, Dropbox.
Even if organization is honest overall, can bad things happen?
Maybe the server s/w or h/w is buggy, even exploitable.
Maybe an attacker guessed server admin password, modified s/w.
Maybe an employee of the cloud provider is corrupt or sloppy.
Can we obtain trustworthy storage from non-trustworthy servers?
This is a hard problem!

SUNDR contains some good ideas.
Similar ideas show up in git and blockchains.
Keybase (acquired by zoom) is directly influenced by SUNDR
Some of you have seen SUNDR in 6.858
It's interesting both as a distributed system (consistency, logging)
and for security.

Setting:
A bunch of programmers collaborating on a project.
They want to store their source files somewhere shared.
They need to see each other's modifications.
They could store their source in github, Google Drive, &c
But they do not want to have to trust any storage service!

We don't know in advance what exactly could go wrong.
So we'll assume the worst:
that the server is fully controlled by a malicious adversary.
It is intentionally trying to trick the users.
It will not, if it doesn't want to, follow any rules or protocol.
"Byzantine faults".
But we do assume the adversary cannot break cryptography,
and only has control over the server, not my client computer.

Potential goals:
Confidentiality.
Adversary cannot read file contents.
Integrity.
Adversary cannot trick clients into getting wrong file contents.
Availability.
Adversary can't prevent clients from accessing their files.

This paper's focus: integrity.

Integrity is real concern for source repository services!
What if someone modifies the source in a huge project like Linux?
Millions of lines of code!
It might be a long time before anyone noticed.
Paper mentions Debian server compromised in 2003.
SourceForge compromised in 2011.
[http://sourceforge.net/blog/sourceforge-attack-full-report]
Canonical (Ubuntu) compromised in 2019.
[https://hub.packtpub.com/canonical-the-company-behind-the-ubuntu-linux-distribution-was-hacked-ubuntu-source-code-unaffected/]

Integrity is not just about preventing illegitimate modifications,
but also about ensuring legitimate modifications all appear and in
the right order.

An integrity example:
Users A, B, and C are developing a simple O/S
They store their source "in the cloud"
We don't want the cloud storage service to be able to modify the source.
We don't want the service to hide or re-order updates either!

This turns out to be the real problem.
User A adds password protection to login.c.
User B adds remote login to net.c.
User C creates a release tar file.
We want C's release to have *both* A's and B's updates.
and not *just* B's net.c.
Since that would allow password-less remote access.
So we want to prevent the service from revealing A's update
but concealing B's update.

The point of this example: it's not enough to prevent the server from
changing the data, we need to prevent it from hiding legitimate
updates as well!

Naive design: users sign file contents.
Use existing network file system (say, Dropbox) as a starting point.
Not expecting any security guarantees from it.
Each file contains data, Sig(SK_writer, data).
When user writes to file, client signs with user's key.
When user reads file, client checks signature.
Signature check will fail if the data isn't exactly the same as when signed.
We assume clients know public keys of all users.

What could a malicious server do in this naive design?
Cannot send arbitrary source code to user C.
Can send one file's data instead of another file's data.
Possible fix: include filename in the signature.
Can send old contents of files.
Can send new content for some files, and old for others.
Can claim that a file doesn't exist.
Not good for our O/S development scenario:
Adversary could cause C to deploy net.c without login.c change.

We need a more sophisticated security design.
Consistent versions of files: net.c change requires login.c change.
Latest version of files: should not be missing any changes.
Integrity of directory contents: should not be missing any files.
Permissions: users might not have permissions to modify every file.

Big idea in SUNDR:
Every update to any file includes signature over current
state visible to updating user.
Thus B's update to net.c includes reference to A's updated login.c
If C sees updated net.c, will also know of new login.c

Strawman design: section 3.1 from the paper.
File system state is determined by a log of operations by users.
Server is responsible for storing this log.
Clients interpret the log.

Strawman details
  Log entries: fetch or modify, user, sig.
    Signature covers the entire log up to that point.
  Client step:
    Download log (other clients now wait).
    Check the log:
      Correct signatures in each entry, covering log prefix.
        (really only need to check each user's last entry)
      This client's last log entry is present.
    Construct FS state based on logged operations.
    Append client's operation and sign new log.
    Upload log (other clients can now proceed).
  Inefficient but simple to reason about.
  Example log:
    X: mod(???), sig
    A: mod(login.c), sig
    B: mod(net.c), sig
    C: fetch(login.c), sig
    C: fetch(net.c), sig

Crucial that signature covers all previous operations in the log.
  Prevents adversary from revealing the net.c change but hiding login.c change.
  If adversary omits login.c change from log, B's signature does not verify.

Could an adversary sign a fake log entry?
  It would need to know the private key of an authorized user.
  How do clients know the public keys of authorized users?

    1. All clients must be told public key of root directory owner.
    2. File system stores the public key of each file/directory owner.
       Clever: file system integrity ensures integrity of public keys.

What would happen if C did not log its fetch?
  I.e. append a "fetch" log entry even when just reading a file.
  Malicious server could give C a stale view of the log.
  When C fetches login.c, server gives log before mod(login.c).
  When C fetches net.c, server gives log with mod(login.c) and mod(net.c).
  Effectively, server pretends C had a race with A + B.
  Outcome: C uses login.c from first fetch and uses net.c from second fetch
    Oops; that is what we were trying to prevent.

When a client fetches the log, it checks that its most recent
  fetch or mod is in the log, and rejects the log if not.

How does logging the fetch help?
  Server can still show C the log before mod(login.c).
  But then C logs its fetch based on that log prefix,
    and remembers that fetch as its last log entry.
  Now the server cannot show mod(login.c) or mod(net.c) to C,
    since they have signatures over a log that does not
    contain C's fetch, and C is expecting to see its fetch.
  "Forking" attack.

So: the server can fork C by hiding modifications to login.c and net.c,
  but the server can then no longer show C any subsequent operations
    by A or B, since their signatures won't include C's fetch.
  The server can, however, continue to serve C's requests.
    C can't realize it's been forked just by talking to server.
  But the server has to hide all future A/B operations from C,
    since their log entry signatures won't include C's fetch.
    Similarly, server must hide C's operations from A/B.
  "fork consistency": server can fork clients, but once it
    forks, can never show operations from one fork to clients
    in the other fork.

Fork consistency is pretty good for a file system.
  Leaves strong trace of attack: impossible to cover up a fork.
  Users can detect if they can communicate outside of SUNDR.
    e.g. e-mail asking "what do you think of my last commit?"

Automated fork detection with separate trusted "timestamp box".
  Designated user is responsible for updating some file every 5 seconds.
  If client sees these updates, it's in the same "fork" as the timestamp box.
  If there is a fork, timestamp box's updates can show up in only one fork.
    Clients in the other fork won't see timestamp box's updates!

Strawman is not practical:
  Log keeps growing.
  Interpreting log gets slow.

Idea: tree of blocks reflecting current state, instead of log.
  Figure 2.
  Server stores smallish blocks, for fast retrieval/update,
    in the usual case that you're operating on just one file.
  i-handle points to table of i-number->inode mappings.
  inode contains type (file vs directory) and list of block references.
  Directory contains list of name->i-number mappings.

The blocks are immutable!
  References (keys) are cryptographic hashes of content.
  When a client asks server for a block, it sends the key,
    and client can check that returned data hashes to that key.
  So the server cannot modify the data, assuming client knows the right key.
  But neither can clients modify any blocks!

How to update?
  When client C writes a file,
    it constructs a new tree reflecting its modification.
  But new tree can share almost all blocks with old tree,
    only needs new blocks on path from modified content
    up to i-handle.
  A new i-handle for each modification.

How to maintain fork consistency with i-handles?
A malicious server could give out old i-handle, concealing recent updates.
We can't prevent the server from forking users, but (as with the straw man) we want to prevent it from merging forks and thus concealing its misdeeds.
We want each new i-handle to somehow encode what came before it, and we want to be able to record fetches as well as modifications.

Idea: a signed "version structure" (VS) for each user.
Figure 3.
Stored in server.
Contains:
i-handle after user's last operation.
Version vector (VV):
For each user, how many operations that user has performed.
Public-key signature by user.
The point: the VV operation counts allow clients to detect omitted old operations, and detect attempts by server to merge forks.

How client U1 executes an operation (both reads and writes):
Get all users' VSs from server.
Validate.
Get needed i-tables &c from block server.
Store new i-table, i-handle with mods (if any).
New VS:
New i-handle.
Increment U1's version #.
Put VS back to server

How do version vectors evolve in correct operation?
U1: 1,0 2,2
U2: 1,1 1,2 2,3

How should U2 validate a set of fetched VSs?
Check that U2's VS is up to date (so U2 must remember its own last VS).
Check that version vectors of diff users can be totally ordered.
e.g. 2,2 <= 2,3

What would version vectors look like if server hid an update from U2?
U1: 1,0 [2,1]
U2: 1,1 1,2

Do the version vectors give us fork consistency?
can the server show future U1 VSs to U2?
e.g. 3,1
no: 3,1 and 1,2 cannot be ordered!
can the server show future U2 VSs to U1?
e.g. 1,3
no: 1,3 and 2,1 cannot be ordered

For our A/B/C login.c net.c example, a correct execution:
A: 1,1,1 2,1,1 something else, then modify login.c
B: 2,2,1 modify net.c
C: 2,2,2 read login.c and net.c

If the server hides A's update to login.c, but reveals B's net.c,
it will send C these vv's:
A: 1,1,1
B: 2,2,1
C will compute its new VV z: 1,2,2
But A/B/z cannot be totally ordered
not this way:
A 1,1,1
B 2,2,1
z 1,2,2
not this way:
A 1,1,1
z 1,2,2
B 2,2,1
so C will know something is wrong.

Nice: version structures allow us to use an efficient tree
data structure, eliminate the need for an ever-growing log,
but can still enforce fork consistency.

Summary.
Hard problem: integrity despite compromised servers.
Client signatures prevent outright forgery.
Hiding and forking are the main attacks still possible.
Fork consistency prevents server from hiding a fork once created.
Forking still possible, but will eventually become obvious to clients.
No plan for recovery after a detected fork.

LEC 19 Peer-to-peer：BitCoin

中文翻译

6.824 2022第19讲：比特币

比特币：点对点电子现金系统，中本聪，2008 年

为什么要写这篇论文?
尽管参与者错综复杂，但仍达成协议
像 SUNDR 稻草人一样：
签名操作日志
日志内容协议->状态协议
叉子是一个主要危险
与 SUNDR 不同
面对分叉有继续的计划
比大多数系统更加分布式——去中心化、点对点
参与者的身份不为人知，甚至数量也不为人所知
协议方案新颖有趣
比特币的成功令人惊讶

技术挑战是什么?
彻底伪造（很容易解决）
双重支出（很难，比特币做得很好）
盗窃（硬私钥是一个弱点）

区块链是什么样的?
交易序列（“分类账”）

符号：
pub(user1)：新所有者的公钥
hash(prev): 该币种之前交易记录的哈希值
sig(user2)：由前所有者的私钥对交易进行签名

交易示例（与真实比特币相比简化得多）：
Y 拥有一枚硬币，之前由 X 给予它：
T6：酒吧（X），...
T7：pub（Y），散列（T6），sig（X）
Y 从 Z 那里买了一个汉堡并用这个硬币支付
Z 将公钥发送给 Y
Y 创建一个新交易并对其进行签名
T8：pub（Z），散列（T7），sig（Y）
Y向Z发送交易记录
Z 验证：
真的酒吧(Z)
T7存在
T8的sig(Y)对应T7的pub(Y)
Z给Y汉堡

只有交易存在，而不是货币本身
Z 的“余额”是 Z 知道私钥的未花费交易的集合
硬币的“身份”是其最近的 xaction（的哈希值）

除了主人以外，任何人都可以花硬币吗?
签署下一笔交易需要当前所有者的私钥
危险：攻击者也许可以窃取 Z 的私钥
例如，通过 PC 或智能手机或在线交换
这是实践中的一个严重问题，而且很难很好解决

硬币的所有者可以在这个计划中花费两次吗?
Y为同一个币创建两个交易：Y->Z，Y->Q
两者都带有哈希值(T7)
Y 向 Z 和 Q 显示不同的交易
两笔交易看起来都不错，包括签名和哈希值
现在Z和Q都会给Y汉堡
双重支出是比特币解决的最根本问题

为什么会出现双重支出?
b/c Z 和 Q 不知道完整的交易集

我们需要什么?
发布所有交易的日志
确保每个人都以相同的顺序看到相同的日志
确保没有人可以取消发布或修改日志条目
结果：

Z 会看到 Y->Z 在 Y->Q 之前，并且会接受 Y->Z
Q 将看到 Y->Z 先于 Y->Q，并将拒绝 Y->Q
“公共分类账”
如何创建这样的账本？

比特币区块链
区块链包含所有代币的所有交易
它是一个日志，事务是状态机操作
目标：就区块链达成一致以防止双重支出
许多同行参与协议
这样我们就可以对信任进行统计
例如通过对下一个区块进行投票（尽管比特币没有）
避免 SUNDR 对单个可疑服务器的严格依赖
同行组织
每个都有整个链条的完整副本
每个节点都有与其他几个对等点的 TCP 连接——“网状覆盖”
通过 TCP 转发，新的链区块被淹没到所有对等点
拟议的交易也涌向所有同行
洪泛有助于确保每个人都知道所有交易
每个块：
哈希（前一个块）
交易集
“nonce”（可以是任何东西，我们将看到）
当前时间（挂钟时间戳）
自上一个区块以来，每 10 分钟包含 xactions 新区块
收款人在交易进入区块链之前不接受交易

谁创建每个新块？
这是通过“工作量证明”进行“挖矿”
要求：hash(block)有N个前导零
每个对等点都会尝试随机的随机数值，直到解决为止
尝试一个随机数很快，但大多数随机数不起作用
这就像抛一枚无数面的硬币，直到出现正面
每次翻转都有独立的小成功机会
挖掘一个区块“不是”特定的固定工作量
创建一个块可能需要 1 个 CPU 的时间
但成千上万的同行正在为此努力
这样第一次找到的预期时间约为 10 分钟
尽管方差很大
获胜者将新区块淹没给所有同行

Y->Z 交易如何通过区块链进行？
开始：所有同行都知道...<-B5
正在挖掘区块 B6（尝试不同的随机数）
Y 将 Y->Z 交易发送给对等点，这些交易会淹没它
对等方缓冲事务直到 B6 计算出来
听到 Y->Z 的同行将其包含在下一个块中
所以最终...<-B5<-B6<-B7，其中B7包括Y->Z

问：B6 是否可能有“两个”不同的后继产品？
答： 是的：

1. 两个对等点几乎同时找到随机数，或者
   2）网络速度慢，在第一个块被淹没之前找到的第二个块被淹没
   两个同时发生的块将会不同

```
1 矿工了解略有不同的新交易集等等。
```

如果有两个继任者，区块链会暂时分叉
同行们挖掘他们首先听到的区块的后继者
但如果他们意识到有一条链，就切换到更长的链

分叉是如何解决的?
每个对等点最初都相信它看到的第一个新（且有效）块
试图挖掘继任者
如果看到 Bx 的人多于 By，就会有更多人开采 Bx，
所以 Bx 后继者可能首先被创建
即使正好对半，一个叉子也可能首先伸出
由于采矿时间存在显着差异
同伴看到最长的叉子后就会切换到最长的叉子
这样分叉就可以获得更多的挖矿能力，从而扩展它
因此随着时间的推移，关于区块的协议往往会变得更加稳定
废弃分叉中的交易怎么样?
大部分都会在两个分叉中
但有些可能就在废弃的岔路口——出现，然后消失!

如果Y同时发出Y->Z和Y->Q怎么办?
即 Y 尝试双花
正确的同伴会首先接受他们看到的，然后忽略第二个
因此下一个块将有一个但不是两个

如果 Y 向一些同伴讲述 Y->Z，而向其他同伴讲述 Y->Q，会发生什么?
也许使用网络 DoS 来防止任一网络的完全泛滥
也许会有一个分叉：B6<-BZ 和 B6<-BQ

因此：
由于分叉，暂时的双重支出是可能的
但叉子的一侧或另一侧很可能很快就会消失
因此如果 Z 看到 Y->Z 后面有几个块，
它被超越的可能性很小
包含 Y->Q 的不同叉
如果 Z 正在销售高价值商品，Z 应该等待一些
发货前阻止
如果 Z 卖的东西很便宜，也许可以等待一些同行
查看 Y->Z 并验证它（但不在块中）

攻击者可以只修改区块链中间的现有区块吗?
并告诉新开始的同伴有关修改后的块?
例如删除攻击者硬币的第一次花费?
否：那么下一个块中的“prev”哈希将是错误的，同行将检测到

攻击者能否从旧区块开始分叉，使用 Y->Q 而不是 Y->Z?
是的——但是 fork 必须更长才能让同行接受它
由于攻击者的分叉在主分叉后面开始，
攻击者必须比其他同行的总和“更快”地挖掘区块

仅使用一个 CPU，即使创建几个块也需要数月时间
到时候主链会更长
没有对等点会切换到攻击者的较短链
如果攻击者比所有诚实的人拥有更多的 CPU 能力
比特币对等体——然后攻击者可以创建最长的分叉，
每个人都会转向它，从而允许攻击者双花

为什么比特币的工作量证明挖矿有效?
对所有参与者进行随机选择，谁可以选择延长哪个分叉
按 CPU 功率加权
如果大多数参与者都是诚实的
他们将加强最长分叉的协议
随机选择意味着（小）攻击者不会得到很多
有机会尝试将协议切换到不同的分叉
令人惊讶的是，在不知道的情况下随机选择是可能的
参与者的身份甚至有多少!

但
如果攻击者控制了大部分 CPU 能力，它可以强制诚实
节点从真实链切换到攻击者创建的链
如果参与者很多，效果最好，所以价格昂贵
攻击者拥有 51% 的 CPU 能力
所以很难开始一个新的比特币式计划

矿工的动力是什么?
每个新区块都会向矿工支付一些新创建的比特币
区块包含获取新比特币的公钥
这是人们经营比特币同行的激励
结果：
军备竞赛，因为更多的硬件 -> 更多的奖励
挖矿获胜的特殊硬件
协作的矿工池
能源浪费

验证检查：
同行，新的 xaction：
先前的交易已存在
没有其他交易花费相同的先前交易
签名是通过之前交易中的公钥的私钥
然后将交易添加到 txn 列表以进行下一个区块的挖掘
对等体，新块：
哈希值有足够的前导零（即随机数是正确的，证明有效）
前一个块哈希存在
区块中的所有交易均有效
如果比当前最长的链长，则对等点切换到新链
Z:
（有些客户依赖同行进行上述检查，有些则不需要）
Y->Z 在一个块中
Z的公钥/地址在交易中
链中还有几个区块
(其他的东西也必须检查，很多细节)

问：10分钟很烦人；它可以变得更短吗？

问：如果有很多矿工加入，出块率会更高吗？

问：比特币为什么要延长最长的分叉？为什么不制定其他规则呢？

问：交易是匿名的吗？

问：如果我偷了比特币，花掉它们安全吗？

问：比特币可以被伪造吗？即创造出完全假的硬币吗？

问：对手可以利用全球大部分 CPU 资源做什么？
可以通过分叉进行双花和取消花
无法窃取他人的比特币
可以防止xaction进入链

问：如果需要更改块格式怎么办？
特别是如果新格式不能被以前的软件版本接受？
“硬分叉”

问：同龄人如何找到彼此？

问：如果同行被欺骗只与腐败的同行交谈怎么办？
如果它与一个好同伴和许多共谋的坏同伴对话会怎么样？

问：一个全新的同行是否会被欺骗而完全使用错误的链？
如果某个同伴在断开几年后重新加入怎么办？
断线几天？

问：一台机器挖矿你能发多少钱？

问：为什么挖矿奖励会随着时间的推移而减少？

Q：币数固定有问题吗？
如果实体经济增长（或收缩）怎么办？

问：为什么比特币有价值？
例如，人们似乎愿意为每个比特币支付 42,000 美元（2022 年 4 月）。

问：比特币的扩容性会很好吗？
就CPU时间而言？
显然 CPU 限制为 4,000 tps（签名检查）
比 Visa 多但比现金少
在存储方面？
您是否需要查看非常旧的块？
您是否需要转移整个区块链？
Merkle 树：区块头与 txn 数据。
就网络流量而言？
每十分钟几兆字节（一个块）
遗憾的是，最大块大小仅限于几兆字节

问：比特币可能只是一个没有新货币的分类账吗？
例如，美元是货币吗？
因为货币部分相当尴尬。
（结算...挖矿激励...）

设计中的弱点?
太糟糕了，它是一种新货币和支付系统
交易确认至少需要 10 分钟，或者高可信度需要 60 分钟
洪水限制性能，可能是攻击点
最大块大小加上 10 分钟限制每秒最大事务数
容易受到多数攻击
工作量证明浪费 CPU 时间和电力
不是很匿名
足够匿名以吸引非法活动
用户在保护私钥方面遇到困难

核心思想：区块链
公共商定的分类账是个好主意
权力下放可能是件好事
挖矿是解决分叉/确保协议的巧妙方法

- ○ 参考 - -

比特币 SoK：https://www.ieee-security.org/TC/SP2015/papers-archived/6949a104.pdf
脚本语言：https://en.bitcoin.it/wiki/Script
https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html

英文原文

6.824 2022 Lecture 19: Bitcoin

Bitcoin: A Peer-to-Peer Electronic Cash System, by Satoshi Nakamoto, 2008

why this paper?
agreement despite byzantine participants
like SUNDR straw man:
log of signed operations
agreement on log content -> agreement on state
forks are a key danger
unlike SUNDR
has a plan to continue in the face of forks
more distributed than most systems -- decentralized, peer-to-peer
identities of participants are not known, not even the number
the agreement scheme is new and interesting
Bitcoin's success was a surprise

what are the technical challenges?
outright forgery (easy to solve)
double spending (hard, bitcoin does pretty well)
theft (hard, private keys are a weakness)

what does the block chain look like?
a sequence ("ledger") of transactions

notation:
pub(user1): public key of new owner
hash(prev): hash of this coin's previous transaction record
sig(user2): signature over transaction by previous owner's private key

transaction example (much simplified compared to real bitcoin):
  Y owns a coin, previously given to it by X:
    T6: pub(X), ...
    T7: pub(Y), hash(T6), sig(X)
  Y buys a hamburger from Z and pays with this coin
    Z sends public key to Y
    Y creates a new transaction and signs it
    T8: pub(Z), hash(T7), sig(Y)
  Y sends transaction record to Z
  Z verifies:
    really pub(Z)
    T7 exists
    T8's sig(Y) corresponds to T7's pub(Y)
  Z gives hamburger to Y

only the transactions exist, not the coins themselves
  Z's "balance" is set of unspent transactions for which Z knows private key
  the "identity" of a coin is the (hash of) its most recent xaction

can anyone other than the owner spend a coin?
  current owner's private key needed to sign next transaction
  danger: perhaps attacker can steal Z's private key
    e.g. from PC or smartphone or online exchange
    this is a serious problem in practice, and hard to solve well

can a coin's owner spend it twice in this scheme?
  Y creates two transactions for same coin: Y->Z, Y->Q
    both with hash(T7)
  Y shows different transactions to Z and Q
  both transactions look good, including signatures and hash
  now both Z and Q will give hamburgers to Y
  double-spending is the most fundamental problem that bitcoin solves

why was double-spending possible?
  b/c Z and Q didn't know complete set of transactions

what do we need?
  publish a log of all transactions
  ensure everyone sees the same log, in the same order
  ensure no-one can un-publish or modify a log entry
  result:
    Z will see Y->Z came before Y->Q, and will accept Y->Z
    Q will see Y->Z came before Y->Q, and will reject Y->Q
    a "public ledger"
  how to create such a ledger?

the BitCoin block chain
  the block chain contains all transactions on all coins
    it's a log, and the transactions are state machine operations
  the goal: agreement on the block chain to prevent double-spending
  lots of peers participate in agreement
    so we can be statistical about trust
    e.g. by voting on next block (though Bitcoin doesn't)

  avoids SUNDR's hard-edged dependence on a single suspect server
 peer organization
  each has a complete copy of the whole chain
  each has TCP connections to a few other peers -- a "mesh overlay"
  new chain blocks flooded to all peers, by forwarding over TCP
  proposed transactions also flooded to all peers
  flooding helps ensure everyone is aware of all transactions
 each block:
  hash(prevblock)
  set of transactions
  "nonce" (can be anything, as we'll see)
  current time (wall clock timestamp)
 new block every 10 minutes containing xactions since prev block
 payee doesn't accept transaction until it's in the block chain

who creates each new block?
 this is "mining" via "proof-of-work"
 requirement: hash(block) has N leading zeros
  each peer tries random nonce values until this works out
 trying one nonce is fast, but most nonces won't work
  it's like flipping a zillion-sided coin until it comes up heads
  each flip has an independent small chance of success
  mining a block is *not* a specific fixed amount of work
 it would likely take one CPU months to create one block
 but thousands of peers are working on it
 such that expected time to first to find is about 10 minutes
  though the variance is high
 the winner floods the new block to all peers

how does a Y->Z transaction work w/ block chain?
 start: all peers know ...<-B5
  and are mining block B6 (trying different nonces)
 Y sends Y->Z transaction to peers, which flood it
 peers buffer the transaction until B6 computed
 peers that heard Y->Z include it in next block
 so eventually ...<-B5<-B6<-B7, where B7 includes Y->Z

Q: could there be *two* different successors to B6?
A: yes:

1. two peers find nonces at about the same time, or
2. slow network, 2nd block found before 1st is flooded to all
   two simultaneous blocks will be different

```
1 miners know about slightly different sets of new transactions, &c.
```

 if two successors, the blockchain temporarily forks
  peers mine a successor to whichever block they heard first
  but switch to longer chain if they become aware of one

how is a fork resolved?
  each peer initially believes the first new (and valid) block it sees
  tries to mine a successor
  if more saw Bx than By, more will mine for Bx,
    so Bx successor likely to be created first
  even if exactly half-and-half, one fork likely to be extended first
    since significant variance in mining time
  peers switch to the longest fork once they see it
    so that fork gets more mining power extending it
    so agreement on a block tends to get more stable over time
  what about transactions in the abandoned fork?
    most will be in both forks
    but some may be in just the abandoned fork -- appear, then disappear!

what if Y sends out Y->Z and Y->Q at the same time?
  i.e. Y attempts to double-spend
  correct peers will accept first they see, ignore second
  thus next block will have one but not both

what happens if Y tells some peers about Y->Z, others about Y->Q?
  perhaps use network DoS to prevent full flooding of either
  perhaps there will be a fork: B6<-BZ and B6<-BQ

thus:
  temporary double spending is possible, due to forks
  but one side or the other of the fork highly likely to disappear soon
  thus if Z sees Y->Z with a few blocks after it,
    it's very unlikely that it could be overtaken by a
    different fork containing Y->Q
  if Z is selling a high-value item, Z should wait for a few
    blocks before shipping it
  if Z is selling something cheap, maybe OK to wait just for some peers
    to see Y->Z and validate it (but not in block)

can an attacker modify just an existing block in the middle of the block chain?
  and tell newly starting peers about the modified block?
  e.g. to delete the first spend of the attacker's coin?
  no: then "prev" hash in next block will be wrong, peers will detect

could attacker start a fork from an old block, with Y->Q instead of Y->Z?
  yes -- but fork must be longer in order for peers to accept it
  since attacker's fork starts behind main fork,
    attacker must mine blocks *faster* than total of other peers
  with just one CPU, will take months to create even a few blocks
    by that time the main chain will be much longer
    no peer will switch to the attacker's shorter chain
  if the attacker has more CPU power than all the honest
    bitcoin peers -- then the attacker can create the longest fork,
    everyone will switch to it, allowing the attacker to double-spend

why does bitcoin's proof-of-work mining work?
  random choice over all participants for who gets to choose which fork to extend
    weighted by CPU power
  if most participants are honest,
    they will re-inforce agreement on longest fork
  random choice means a (small) attacker won't get many
    chances to try to switch agreement to a different fork
  surprising that random choice is possible without knowing
    participant identities or even how many!

but
  if attacker controls majority of CPU power, it can force honest
    peers to switch from real chain to one created by the attacker
  works best if lots of participants, so it's expensive for
    attacker have 51% of CPU power
  so it's hard to start a new bitcoin-style scheme

what motivates miners?
  each new block pays miner a few newly created bitcoins
    block contains public key that gets the new bitcoins
    this is incentive for people to operate bitcoin peers
  consequences:
    arms race, since more hardware -> more reward
    special hardware for mining to win
    pools of miners that collaborate
    energy waste

validation checks:
  peer, new xaction:
    previous transaction exists
    no other transaction spends the same previous transaction
    signature is by private key of pub key in previous transaction
    then will add transaction to txn list for next block to mine
  peer, new block:
    hash value has enough leading zeroes (i.e. nonce is right, proves work)
    previous block hash exists
    all transactions in block are valid
    peer switches to new chain if longer than current longest
  Z:
    (some clients rely on peers to do above checks, some don't)
    Y->Z is in a block
    Z's public key / address is in the transaction
    there's several more blocks in the chain
  (other stuff has to be checked as well, lots of details)

Q: 10 minutes is annoying; could it be made much shorter?

Q: if lots of miners join, will blocks be created at a higher rate?

Q: why does Bitcoin extend the longest fork? why not some other rule?

Q: are transactions anonymous?

Q: if I steal bitcoins, is it safe to spend them?

Q: can bitcoins be forged, i.e. a totally fake coin created?

Q: what can adversary do with a majority of CPU power in the world?
  can double-spend and un-spend, by forking
  cannot steal others' bitcoins
  can prevent xaction from entering chain

Q: what if the block format needs to be changed?
  esp if new format wouldn't be acceptable to previous s/w version?
  "hard fork"

Q: how do peers find each other?

Q: what if a peer has been tricked into only talking to corrupt peers?
  how about if it talks to one good peer and many colluding bad peers?

Q: could a brand-new peer be tricked into using the wrong chain entirely?
  what if a peer rejoins after a few years disconnection?
  a few days of disconnection?

Q: how rich are you likely to get with one machine mining?

Q: why does it make sense for the mining reward to decrease with time?

Q: is it a problem that there will be a fixed number of coins?
  what if the real economy grows (or shrinks)?

Q: why do bitcoins have value?
  e.g. people seem willing to pay $42,000 per bitcoin (April 2022).

Q: will bitcoin scale well?
  in terms of CPU time?
    apparently CPU limits to 4,000 tps (signature checks)
    more than Visa but less than cash
  in terms of storage?
    do you ever need to look at very old blocks?
    do you ever need to xfer the whole block chain?
    merkle tree: block headers vs txn data.
  in terms of network traffic?
     a few megabytes (one block) every ten minutes
  sadly, the maximum block size is limited to a few megabytes

Q: could Bitcoin have been just a ledger w/o a new currency?
  e.g. have dollars be the currency?
  since the currency part is pretty awkward.
  (settlement... mining incentive...)

weak points in the design?
  too bad it's a new currency as well as a payment system
  transaction confirmation takes at least 10 minutes, or 60 for high confidence
  flooding limits performance, may be a point of attack
  maximum block size plus 10 minutes limits max transactions per second
  vulnerable to majority attack
  proof-of-work wastes CPU time, power
  not very anonmyous

anonymous enough to attract illegal activity
users have trouble securing private keys

key idea: block chain
public agreed-on ledger is a great idea
decentralization might be good
mining is a clever way to resolve forks / ensure agreement

---- References ----

Bitcoin SoK: https://www.ieee-security.org/TC/SP2015/papers-archived/6949a104.pdf
Scripting language: https://en.bitcoin.it/wiki/Script
https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html

LEC 20 Blockstack

中文翻译

6.824 2022年第20讲：Blockstack（2016）

今天的主题：去中心化应用
以将数据控制权转移到用户手中的方式构建的应用程序
以及集中控制的网站之外
许多人正在探索这个总体愿景
早期的P2P应用程序、密钥库、实体、智能合约等
推测，不清楚会发生什么
本讲座借鉴了 Blockstack 的后续发展
现在是一家公司，拥有更全面的去中心化应用程序设计
它有为其编写的用户和应用程序

旧：典型的（集中式）网站
[用户浏览器、网络、带有应用程序代码的网站网络服务器、网站数据库]
隐藏在专有应用程序代码后面的用户数据
例如博客文章、gmail、广场、reddit 评论、照片共享、
日历、医疗记录等
这个安排非常成功
编程很容易
例如 eBay 软件可以看到所有用户的出价，即使用户看不到
为什么这不理想？
如果用户想要使用他们的数据，则必须使用此网站的 UI
网站设置（并更改！）谁可以访问的规则
网站可能会窥探、向广告商出售信息
网站员工可能会出于个人原因进行窥探
令人失望的是，因为它通常是用户自己的数据！
问题的设计视图：
大的界面划分是在用户和应用程序+数据之间
应用程序+数据集成方便网站所有者
但 HTML 作为界面是面向 UI 的。
通常不擅长让用户控制和访问数据

新：去中心化应用程序
[用户应用程序、通用云存储服务]
在用户的计算机、ipad、浏览器 JavaScript 等上运行应用程序
云存储提供商中的数据
用户向提供商付费——用户拥有自己的数据
用户控制访问
用户可以互相共享数据
用户可以使用自己的数据使用任何他们想要的应用程序
该架构将应用程序代码与用户数据分开
关键接口是用户+应用和数据之间
因此，用户数据有更清晰的概念，由用户拥有/控制
就像您拥有笔记本电脑或 Athena 帐户中的数据一样

对存储系统的要求
我们非常关心它的设计，因为存储现在是主要的 API
在云端，因此可以从任何设备访问
存储设计不能特定于应用程序
通用，如文件系统或 k/v 存储或数据库
由拥有数据的用户付费并控制
我们希望在多用户应用程序的用户之间共享
权限
用户对用户
应用程序与应用程序：我不想让游戏查看我的电子邮件！
需要一种方法来查找其他用户的数据，命名用户以访问 ctl
并非不可能：Amazon S3 有一些这种味道

去中心化应用程序如何运作？
app：两个用户共享的待办事项列表
[UI x2，复选框列表，“添加”按钮]
用户 U1 和 U2 在其计算机上运行应用程序
可能作为浏览器中的 JavaScript，或者 iPad 应用程序
应用程序读取其他用户的公共数据，写入自己用户的数据
两者都贡献列表项
两者都可以将项目标记为完成
每个用户都有一个包含待办事项的文件
和一个带有“完成”标记的文件
每个都由其存储提供商存储
每个用户的 UI 代码定期扫描其他用户的待办事项文件
该应用程序没有任何关联的服务器，它只使用存储系统

重点是什么？
方便用户切换应用
数据未隐藏在网站内
数据不与应用程序绑定
更容易拥有查看多种数据的应用程序
日历/电子邮件/待办事项，或备份，或文件浏览器
隐私与窥探（假设端到端加密）

为什么这个愿景可能难以实现？
共享需要构建存储数据的标准
两种文件格式以及更大的排列
不再是网站的私人业务

历史双方都在说话（JPEG、Microsoft Word）
可能对性能不利
每用户的类 FS 存储的灵活性远不如 SQL DB
例如服务器端执行复杂的SQL
有时您可能需要一个可信/中央服务器
例如生成reddit首页、索引、投票数
例如查看所有用户的拍卖出价而不透露
加密隐私/身份验证使其他一切变得更加困难
组、撤销
尚不清楚是否有推动采用的杀手级优势

现在为 Blockstack
他们一开始只是去中心化命名
后来的论文和公司追求去中心化应用程序

为什么命名是一个问题?
Blockstack 希望允许所有人之间共享
需要一种方法让人们明确指定与谁共享 &c
现有系统无法满足他们的需求
name -> 用户数据的位置，以便多个用户可以交互
名称 -> 公钥，用于端到端数据安全
这样我就可以检查我是否确实检索到了您的真实数据
这样我就可以加密私人数据，这样只有我的同事才能解密它
由于存储系统不受信任
这是一个公钥基础设施 (PKI)
PKI 对于许多全球安全理念至关重要，例如安全电子邮件
存在于网站——https 证书
但没有适合人类用户的良好的全球 PKI
所以 Blockstack 从 PKI 开始——一个用户的命名方案

Blockstack 声称命名很困难，用“Zooko 三角”来概括：

1. 独特/全球——每个名字对每个人来说都有相同的含义
2. 人类可读
3.去中心化——我们不需要任何人的许可来创建/使用名称
主张：这三者都很有价值
主张：任何两个都是容易的；这三个都很难

```
没有中心实体，如何决定谁获得“rtm”？
```

每对属性的示例?
唯一+人类可读：电子邮件地址
独特+去中心化：随机选择公钥
人类可读+去中心化：我的联系人列表

Blockstack 如何获得 Zooko 的所有三个属性的摘要?
比特币产生有序的区块链
每个人都同意这个命令，即使面对恶意
Blockstack 将名称声明记录嵌入到
有效比特币交易的元数据
[图：块，嵌入的命名操作]
如果我声称“rtm”的记录在比特币链中是第一，那么我拥有它

Blockstack 同行关注比特币链
他们都执行规则
例如，如果我声明“rtm”，然后其他人声明“rtm”怎么办？
先到先得政策
唯一（==全局相同）？
人类可读的？
去中心化？

现在我控制了“rtm”，我们将看到 Blockstack 将
让我指出我的博客文章、开源贡献，
照片集、待办事项列表、公钥等。
然后其他人一旦知道我是“rtm”就可以找到我的东西。

这种名称空间对于去中心化应用程序有好处吗？
好：我可以告诉你某人的名字，你可以使用它（全球的，唯一的）。
好：我们可能可以记住这些名字（人类可读）。
好：没有人可以通过审查名称系统来阻止访问（去中心化）。
坏：如果全局唯一，人类可读的名称就不会很有意义
我可能不能“rtm”——其他人可能已经拿走了它。
你很难猜到我是“rtm229”。
你可能认为“rtm@mit.edu”肯定是我。
但blockstack是FCFS，它不检查！
bad：我怎样才能找到你的 Blockstack 名称？
bad：我如何验证 Blockstack 名称是否真的是您？

替代命名方法：
Keybase 通过链接到其他命名系统来增加强化
（例如电子邮件或github名称），与现有朋友的协议
可以直接使用公钥，没有人类可读的名称
人们在带外交换密钥，例如在纸上
每个人都保留单独的“联系人列表”，其中包含他们理解的名字
自然去中心化
不需要唯一性机制，所以不需要比特币
可以拥有可靠地验证人类身份的中央实体
可能与护照、驾驶执照、社会保障号码等相关

Blockstack 去中心化应用程序设计有哪些部分？
（其中大部分来自他们后来的公司白皮书）
[比特币链、BNS 服务器、Atlas、Gaia、S3、客户端应用程序]
比特币的区块链
具有嵌入的 Blockstack 名称操作
块堆栈服务器 (BNS)
阅读比特币链
解释 Blockstack 命名记录以更新 DB
服务来自客户端的命名 RPC
名称 -> 区域哈希、公钥
Atlas 服务器——存储“区域文件”
比特币中的名称记录映射到 Atlas 中的区域文件
区域文件指示我的 Gaia 数据的存储位置
由内容哈希键控，因此项目是不可变的
Atlas 在每台服务器中保留完整的数据库
盖亚服务器

每个用户都有独立的存储区域
键->值
Amazon S3、Dropbox 等的网关
盖亚让它们看起来都一样，充当网关/翻译器
用户的个人资料包含用户的公钥、每个应用程序的公钥
用户可以拥有许多其他文件，其中包含应用程序数据
应用程序在 Gaia 中签署数据
所有者也可以更新，只要正确签名即可
对于私人数据，应用程序可以加密

BNS服务器详细信息
BNS 服务器必须忽略无效的 Blockstack 记录
BNS 服务器强制执行比特币支付以销毁地址
为什么需要付费才能注册名字?
防止人们注册每一个有用的名字
免费的东西往往会被滥用
您需要使用您信任的BNS服务器
也许你自己运行
如果需要的话，可以通过查看区块链来检查

用户U1的应用如何获取用户U2的数据?
要求BNS服务器查找“U2”
生成 U2 的区域文件的内容哈希和 U2 的 pub 密钥
（U1必须信任其BNS服务器）
要求 Atlas 查找哈希值，获取区域文件
不必相信 Atlas，因为 U1 拥有哈希值
区域文件有一个到 Gaia 存储柜的链接
向 Gaia 询问 U2 的个人资料文件
用U2的公钥验证
从配置文件中提取 U2 的每个应用程序公钥
向盖亚询问感兴趣的文件
使用 U2 个人资料中的公钥进行验证
U1必须相信Gaia返回了最新版本

私钥处理
我的应用程序在我自己的计算机上运行
我的应用程序需要私钥才能写入 GAIA
但我不能信任带有我真实私钥的随机应用程序!
例如，由不受信任的供应商编写的游戏
所以每个应用程序的私钥，每个应用程序的数据
Blockstack 浏览器管理；值得信赖的
私钥永远不会离开我的设备!
静态时受密码保护

保证私钥的安全很难!
用户通常对私钥不太小心
如果我丢失手机怎么办?
如果我忘记了关键密码怎么办?
如果我怀疑丢失并想更换钥匙怎么办?
也许将我的私钥打印在安全的地方的一张纸上?
也许“母亲的婚前姓名”用于密钥恢复?
但这可能是最薄弱的安全环节

这意味着恢复代理需要知道我的私钥
（blockstack 不这样做）
一个令人烦恼的普遍问题！

程序员会渴望转向这种架构吗？
我已经对它进行了一些编程，似乎比 apache/mysql 更难
与每个用户相比，很难获得特定于应用程序的数据
Reddit 或 Hacker News 的指数、投票计数、头版排名
很难既查看其他用户的秘密又保守秘密
例如易趣
访问控制比 apache/mysql 更痛苦
因为你需要加密
例如 6.824 组，由注册商控制，可能会更改、撤销 &c
所以目前还不清楚程序员是否会急于转换。

用户会对信任/隐私/去中心化的争论感到兴奋吗？
也许这会改善用户数据的隐私
也许比相信 facebook、google 等更好
对员工、广告商、朋友、黑客保密数据
但你仍然必须信任云存储提供商，
可能是谷歌或亚马逊 aws
客户端应用程序是由 facebook 或其他任何人编写的！
所以，除非你审核代码，否则它会做 Facebook 想要的任何事情
所以信任优势也许并不大
必须信任存储提供商不会丢失数据
并产生最新版本
鉴于此，也许也愿意相信谷歌来运行我的电子邮件软件

用户是否会对对其数据的更多控制感兴趣？
能够切换照片管理器等？
用户是否想在多个应用程序中使用相同的数据？
应用程序是否会使用标准存储方案来实现切换？

用户愿意为自己的存储付费吗？
无可匹敌的免费广告支持服务

名称创建

如何注册 Blockstack 名称？
（https://docs.blockstack.org/core/wire-format.html）
用户执行此操作（通过运行 Blockstack 软件）
用户必须拥有一些比特币
两种比特币交易：预购、注册
预购交易
注册费到“燃烧”地址
哈希（名称）
登记交易
名称（未散列）
所有者公钥
哈希（区域文件）

为什么有*两笔*交易？
抢先交易

为什么要收注册费？毕竟没有真正的成本。

如果客户尝试注册已被占用的名称怎么办？

如果两个客户尝试同时注册相同的名称怎么办？

攻击者是否有可能更改名称->密钥绑定？
 毕竟，任何人都可以提交他们喜欢的任何比特币交易

Blockstack 是否可以更改名称->键绑定？

结论

我能从 Blockstack 中得到什么？
 将云数据与应用程序分开原则上听起来不错
  但开发人员会讨厌它（例如没有 SQL）。
  不清楚的用户会关心。
  尚不清楚用户是否愿意为存储付费。
 令人惊讶的是我们可以进行去中心化的人类可读的名称分配
 整个事情都取决于 PKI——这里的任何进展都会很棒
  为了安全地命名人类，映射到公钥
 端到端隐私加密听起来是个好主意
  私钥管理是痛苦且脆弱的
  加密使共享和访问控制变得尴尬
 您仍然必须信任供应商软件；不清楚这是一个
  巨大的胜利是它在你的笔记本电脑上运行，而不是
  供应商的服务器。

话虽如此，如果 Blockstack 之类的东西那就太棒了
 就像要成功一样。

- 参考 - -

https://dataspace.princeton.edu/handle/88435/dsp019306t191k
https://econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf

英文原文

6.824 2022 Lecture 20: Blockstack (2016)

topic today: decentralized apps
 apps built in a way that moves control over data into users's hands
  and out of centrally-controlled web sites
 many people are exploring this general vision
  early P2P apps, keybase, solid, smart contracts, etc
 speculative, not clear what will happen
 this lecture draws on later developments by Blockstack
  now a company, with a fuller design for decentralized apps
  it has users and apps written for it

old: a typical (centralized) web site
 [user browsers, net, site's web servers w/ app code, site's DB]
 users' data hidden behind proprietary app code
 e.g. blog posts, gmail, piazza, reddit comments, photo sharing,
  calendar, medical records, &c

this arrangement has been very successful
it's easy to program
e.g. ebay s/w can see all users' bids, even though users can't
why is this not ideal?
users have to use this web site's UI if they want to use their data
web site sets (and changes!) the rules for who gets access
web site may snoop, sell information to advertisers
web site's employees may snoop for personal reasons
disappointing since it's often the user's own data!
a design view of the problem:
the big interface division is between users and app+data
app+data integration is convenient for web site owner
but HTML as an interface is UI-oriented.
and is usually not good about giving users control and access to data

new: decentralized apps
[user apps, general-purpose cloud storage service]
run apps on users' computers, ipads, browser javascript, &c
data in cloud storage providers
users pay providers -- users own their data
users control access
users can share data with each other
users can use whatever apps they want with their own data
this architecture separates app code from user data
the key interface is between user+app and data
so there's a clearer notion of a user's data, owned/controlled by user
much as you own the data on your laptop, or in your Athena account

requirements for the storage system
we care about its design a lot, since storage is now a primary API
in the cloud, so can be accessed from any device
storage design can't be app-specific
general-purpose, like a file system or k/v store or DB
paid for and controlled by user who owns the data
we want sharing between users for multi-user apps
permissions
user vs user
app vs app: i don't want a game to look at my e-mail!
need a way to find other user's data, name users for access ctl
not impossible: Amazon S3 has some of this flavor

how might a decentralized application work?
app: a to-do list shared by two users
[UI x2, check-box list, "add" button]
users U1 and U2 run apps on their computers
maybe as JavaScript in browsers, or iPad app
the apps read other user's public data, write own user's data
both contribute list items
both can mark an item as finished
each user has a file with to-do items
and a file with "done" marks

each stored by their storage provider
each user's UI code periodically scans the other user's to-do files
the app doesn't have any associated server, it just uses the storage system

what's the point?
easier for users to switch apps
data not hidden inside web sites
data not tied to apps
easier to have apps that look at multiple kinds of data
calendar/email/todo, or backup, or file browser
privacy vs snooping (assuming end-to-end encryption)

why might this vision be difficult to realize?
sharing requires standards for structuring stored data
both file formats, and larger arrangements
no longer private business of web sites
history speaks on both sides (JPEG, Microsoft Word)
probably bad for performance
per-user FS-like storage much less flexible than SQL DB
e.g. server-side execution of complex SQL
sometimes you may need a trusted/central server
e.g. to generate reddit front page, indexes, vote counts
e.g. to look at all users' auction bids w/o revealing
cryptographic privacy/authentication makes everything else harder
groups, revocation
not clear there are killer advantages to drive adoption

now for Blockstack
they started out just with decentralized naming
later papers -- and company -- pursuing decentralized apps

why is naming a concern?
Blockstack wants to allow sharing among potentially all humans
need a way for humans to unambiguously specify who to share with &c
no existing system does what they need
name -> location of user's data, so multiple users can interact
name -> public key, for end-to-end data security
so I can check I've really retrieved your authentic data
so I can encrypt private data so only my co-workers can decrypt it
since storage system is not trusted
this is a public key infrastructure (PKI)
PKI critical to many global security ideas e.g. secure e-mail
exists for web sites -- https certificates
but no good global PKI for human users
so Blockstack started with a PKI -- a naming scheme for users

Blockstack claims naming is hard, summarized by "Zooko's triangle":

1. unique / global -- each name has the same meaning to everyone
2. human-readable
3. decentralized -- we don't need anyone's permission to create/use names

claim: all three would be valuable
claim: any two is easy; all three is hard

```
without central entity, how to decide who gets "rtm"?
```

example for each pair of properties?
  unique + human-readable : e-mail addresses
  unique + decentralized : randomly chosen public keys
  human-readable + decentralized : my contact list

summary of how Blockstack gets all three of Zooko's properties?
  Bitcoin produces an ordered chain of blocks
    and everyone agrees on the order, even in the face of malice
  Blockstack embeds name-claiming records in
    meta-data of valid bitcoin transactions
  [diagram: blocks, naming operations embedded]
  if my record claiming "rtm" is first in Bitcoin chain, I own it
    blockstack peers watch the bitcoin chain
    they all enforce rules
    e.g. what if i claim "rtm", then someone else claims "rtm"
    first-come-first-served policy
  unique (== globally the same)?
  human-readable?
  decentralized?

now that I control "rtm", we'll see that Blockstack will
  let me point it at my blog posts, open source contributions,
  photo collection, to-do list, public keys, &c.
and then others can find my stuff once they know I'm "rtm".

is this kind of name space good for decentralized apps?
  good: I can tell you someone's name, and you can use it (global, unique).
  good: we can probably remember the names (human readable).
  good: no-one can block access by censoring the name system (decentralized).
  bad: human-readable names won't be very meaningful if globally unique
      I probably can't be "rtm" -- someone else likely to have taken it.
      Hard for you to guess that I'm "rtm229".
      You might think "rtm@mit.edu" is sure to be me.
      but blockstack is FCFS, it doesn't check!
  bad: how can I find your Blockstack name?
  bad: how can I verify that a Blockstack name is really you?

alternate naming approaches:
  Keybase adds re-inforcement via linking to other naming systems
    (e.g. e-mail or github name), agreement with existing friends
  could use public keys directly, no human-readable names
    people exchange their keys out-of-band, e.g. on paper
    each person keeps separate "contact list" with names they understand
    naturally decentralized
    no need for uniqueness mechanism, so no need for Bitcoin
  could have central entity that reliably verifies human identity
    maybe tied to passports, drivers licences, social security #s, &c

what are the pieces in Blockstack decentralized app design?
 (much of this is from their later company white-papers)
 [bitcoin chain, BNS servers, Atlas, Gaia, S3, client apps]
 Bitcoin's block-chain
  with embedded Blockstack name operations
 Blockstack servers (BNS)
  read Bitcoin chain
  interpret Blockstack naming records to update DB
  serve naming RPCs from clients
  name -> zone hash, public key
 Atlas servers -- store "zone files"
  a name record in bitcoin maps to a zone file in Atlas
  zone file indicates where my Gaia data is stored
  keyed by content-hash, so items are immutable
  Atlas keeps the full DB in every server
 Gaia servers
  separate storage area for each user
  key -> value
  gateway to Amazon S3, Dropbox, &c
   Gaia makes them all look the same, acts as gateway/translator
  user's profile contains user's public key, per-app public keys
  user can have lots of other files, containing app data
  apps sign data in Gaia
   owner can update as well, as long as correctly signed
  for private data, apps can encrypt

BNS server details
 BNS servers have to ignore invalid Blockstack records
 BNS servers enforce bitcoin payments to burn address
  why require payment to register a name?
  prevent people from registering every single useful name
  free stuff tends to be abused
 you need to use a BNS server you trust
  maybe run it yourself
  can check, if you need to, by looking at the block-chain

how does user U1's app fetch user U2's data?
 ask BNS server to look up "U2"
  yields content-hash of U2's zone file, and U2's pub key
  (U1 has to trust its BNS server)
 ask Atlas to look up hash, get zone file
  don't have to trust Atlas, since U1 has the hash
 zone file has a link to a Gaia storage locker
 ask Gaia for U2's profile file
  verify with U2's public key
  extract U2's per-app public keys from profile
 ask Gaia for the file of interest
  verify with public key from U2's profile
 U1 has to trust that Gaia returned the latest version

Private key handling
  my apps run on my own computers
  my apps need a private key to write to gaia
  but I can't trust random apps with my real private key!
    e.g. a game written by an untrusted vendor
  so per-app private keys, per-app data
  Blockstack Browser manages; trusted
  private key never leaves my device(s)!
    protected at rest by a pass-phrase

It's hard to keep private keys secure!
  users often not very careful with private keys
  what if I lose my phone?
  what if I forget my key pass-phrase?
  what if I suspect loss and want to change my key?
  maybe print my private key on a sheet of paper somewhere secure?
  maybe "mother's maiden name" for key recovery?
    but then that's likely to be the weakest security link
    and it means the recovery agent needs to know my private key
    (blockstack doesn't do this)
  a vexing general problem!

Will programmers be eager to switch to this kind of architecture?
  I have programmed it a bit, seems signif harder than apache/mysql
  hard to have data that's specific to the app, vs each user
    indices, vote counts, front-page rankings for Reddit or Hacker News
  hard to both look at other users' secrets, and keep the secrets
    e.g. for eBay
  access control is more painful than with apache/mysql
    since you need to encrypt
    e.g. 6.824 group, controlled by registrar, may change, revoke &c
  so it's not clear programmers will be eager to switch.

Will users be excited by trust/privacy/decentralization arguments?
  maybe it will improve privacy of user data
    perhaps better than trusting facebook, google, &c
    to keep data private from employees, advertisers, friends, hackers
  but you still have to trust a cloud storage provider,
    and it will probably be google or amazon aws
  client app is written by facebook, or whoever!
    so, unless you audit the code, it does whatever facebook wants anyway
    so the trust advantage is perhaps not huge
  must trust storage provider to not lose data
    and to yield most recent version
    given that, maybe willing to trust Google to run my email s/w too

Will users be interested in more control over their data?
  to be able to switch photo organizers &c?
  do users want to use same data in multiple applications?
  will apps use standard storage schemes to enable switching?

Will users be willing to pay for their own storage?
  hard to beat free ad-supported services

NAME CREATION

how does one register a Blockstack name?
  (https://docs.blockstack.org/core/wire-format.html)
  the user does it (by running Blockstack software)
  user must own some bitcoin
  two bitcoin transactions: preorder, registration
  preorder transaction
    registration fee to "burn" address
    hash(name)
  registration transaction
    name (not hashed)
    owner public key
    hash(zonefile)

why *two* transactions?
  front-running

why the registration fee? after all there's no real cost.

what if a client tries to register a name that's already taken?

what if two clients try to register same name at same time?

is it possible for an attacker to change a name->key binding?
  after all, anyone can submit any bitcoin transaction they like

is it possible for Blockstack to change a name->key binding?

CONCLUSION

what do I take away from Blockstack?
  separating cloud data from applications sounds good in principle
    but developers will hate it (e.g. no SQL).
    not clear users will care.
    not clear whether users will want to pay for storage.
  surprising that we can have decentralized human-readable name allocation
  the whole thing rests on a PKI -- any progress here would be great
    to securely name human beings, map to public keys
  end-to-end encryption for privacy sound like a good idea
    private key management is a pain, and fragile
    encryption makes sharing and access control awkward
  you still have to trust vendor software; not clear it's a
    huge win that it's running on your laptop rather than
    vendor's server.

all that said, it would be fantastic if Blockstack or something
  like it were to be successful.

--- references ---

https://dataspace.princeton.edu/handle/88435/dsp019306t191k
https://econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf

LEC 21 Smart Contracts

中文翻译

6.824 2022年第21讲：以太坊（2014）

主题：用于去中心化应用的可编程区块链

旧：集中式应用程序
 示例应用程序：金融交易、DNS/PKI、赌场
 前端（客户端）<->后端（服务），由应用程序操作员编写
 缺点：必须信任应用程序作者/服务运营商
  可能会出现什么问题：恶意操作员、受损操作员
   金融交易：例如抢先交易
   DNS：例如受损域名
   赌场：例如增加赌场优势
 无法判断后端是否正常运行
  可以运行不同的代码，直接操作数据库内容，...
 演示：硬币翻转赌场心理游戏

新：去中心化（无信任）应用程序
 目标：减少对网站运营商的信任或不信任
 用一些无需信任（去中心化？）的服务替换后端
 可以想象使用定制解决方案
  例如赌场：下注前签署承诺书
 雄心勃勃的目标：以通用且易于编程的方式支持大量应用程序
  云 -> 用于去中心化应用程序的新计算层
 这如何运作？
  《世界计算机》
  行为就像单个共享机器
   安全/单一事实来源
  开放参与
  提供计算+存储
  任何人都可以将程序加载到上面
  所有状态都是公共的
  内置货币系统：用于支付计算费用等。

演示：世界计算机的接口是什么？
 智能合约~=程序
  既不是智能，也不是合同（法律意义上的）
 广告牌示例 (1_Billboard.sol)
  混音IDE
  用高级语言编写程序（可靠性）
  智能合约是程序的实例，具有状态和方法
   状态机
  合约的状态包括其余额
  在某个地址编译并部署程序
  外部拥有的账户可以调用程序的方法，发送值
  尚未在真实网络上：内存虚拟机
 赌场示例 (2_Casino.sol)
  部署在真实（测试网）以太坊网络上
  任何人都可以与其方法交互

remix ide（网络应用程序）与metamask（钱包应用程序/浏览器扩展）交互
以太扫描

以太坊
在 POW 区块链之上实现，将 txns 批量放入块中
6.824 图片：共享日志之上的复制状态机
世界计算机作为复制状态机实现
元状态机，您可以在其中加载和运行程序
世界计算机状况
地址->
| 外部账户 { 余额 }
| 合约账户 { 余额、代码、状态 }
转换函数
发送:=
| 部署合约（代码，值）
| 消息（地址、方法、数据、值）
语义
部署合约
需要合约代码
选择一个新地址并使合约代码在该地址可用
将状态初始化为初始值
将余额设置为初始值
运行构造函数
返回地址
信息
接受地址、方法、数据、值
查找地址：如果是外部账户，则仅转移价值
查找代码，查找方法
增加合约余额的价值
从初始状态开始，使用作为参数给出的数据运行方法
产生一个新的状态
EVM：支持智能合约的低级语言
许多编译为 EVM 的 HLL
图灵完备->气体
气体耗尽 -> 交易中止

智能合约的应用
以太坊名称服务 (ENS)
像块堆栈
ERC20代币
基本上，银行（∽比特币）状态机作为智能合约
发行代币
跟踪每个用户的余额
交易代币
例子
股票：通过拍卖首次出售，向代币持有者支付股息，代币持有者可以投票
资产支持代币（例如 USDC），简化国际金融
奖励计划（例如航空里程）
DeFi、去中心化交易所
交易 ERC20 代币

去中心化治理（DAO）
去中心化投资基金，集体投票决定投资什么

dapps 实践：桥接以太坊和网络应用程序
通常，应用程序的一部分是使用智能合约实现的
仍然使用在标准云上运行的后端计算

高级智能合约：编程模型的复杂性
（不是完整列表）
有些东西我们在演示中没有看到
合约可以调用其他合约的方法（并发送以太币）
可选的气体限制
创建其他合同
试着抓
涌现的复杂性
如果你在执行合约的过程中耗尽了gas怎么办?
交易中止：写入恢复并退还剩余气体
如果你调用另一个合约，而该合约的 Gas 耗尽了怎么办?
整个事务中止：恢复所有状态更改
如果您的合约遇到错误（例如除以零）会发生什么?
事务中止，状态恢复
如果被调用者合约遇到错误会发生什么?
同样的事情：事务中止，错误冒泡
试着抓
发生错误时，仅恢复被调用者的更改
解决不变性问题
合约可以调用其他合约
合约可以“间接分支”，例如 call(address=storage[0])
改变存储可以改变合约运行的代码
神谕

局限性 + 以太坊是否实现了其愿景
计算能力有限
存储容量有限
没有秘密数据
确定性的；矿工控制下的“随机性”
交易在执行之前是公开的：抢先交易
预言机需要与系统外部的数据进行交互
“代码就是法律”、不可变代码、不可逆交易：并不总是好事
DAO 事件、硬分叉、ETH 与 ETC
不太容易编程
虫子
https://rekt.news/
新的编程模型和环境，有时语义不直观
例如，资金转移是一条消息，合约可以“实现”该消息的处理程序
例如重入漏洞，在许多现实世界的攻击中被利用

提取（）：
sender.transfer(余额[发件人])
余额[发件人] = 0

充满敌意且难以理解的编程环境，例如闪电贷款
 无需信任的贷款（在单笔交易中）
 使用回调和使用 try/catch 进行仔细编程来实现

问：合约在哪里执行?
答：在所有全节点上，任何生成新块或验证块的矿工都会执行交易

问：如何选择 Gas 价格?
答：基于网络拥塞情况，可以使用 https://ethgasstation.info/ 等工具来提供帮助

问：如何选择 Gas Limit?
答：钱包可以帮助估算 Gas 成本
 （例如，在给定系统当前状态的情况下，运行交易需要多少gas）

英文原文

6.824 2022 Lecture 21: Ethereum (2014)

topic: programmable blockchains for decentralized applications

old: centralized app
 example apps: financial exchange, DNS/PKI, casino
 frontend (client) <-> backend (service), written by app operator
 disadvantage: have to trust app author / service operator
  what can go wrong: malicious operator, compromised operator
   financial exchange: e.g. front-running
   DNS: e.g. compromised domain
   casino: e.g. increasing house edge
 can't tell if backend is running honestly
  can run different code altogether, manipulate database contents directly, ...
 demo: coin flip casino mental game

new: decentralized (trust-less) app
 goal: reduced trust or no trust in site operator
 replace backend with some trust-less (decentralized?) service
 could imagine using bespoke solutions
  e.g. casino: signed commitments before bets are placed
 ambitious goal: support large class of apps in a general-purpose and easy-to-program way
  cloud -> new compute layer for decentralized apps
 how might this work?
  "world computer"
  behaves like single shared machine
   security / single source of truth
  open participation
  offers compute + storage
  anyone can load programs onto it
  all state is public
  built-in currency system: to pay for compute, etc.

demo: what is the interface to the world computer?
 smart contracts ~= programs
  neither smart, nor contracts (in the legal sense)
 billboard example (1_Billboard.sol)
  remix ide

- write programs in high-level language (solidity)
- smart contracts are instances of programs, with state and methods
  - state machines
- contract's state includes its balance
- compile and deploy programs at an address
- externally-owned accounts can call programs' methods, send value
- not on real network yet: in-memory vm
- casino example (2_Casino.sol)
- deploy on real (testnet) ethereum network
- anyone can interact with its methods
- remix ide (web app) interacts with metamask (wallet app / browser extension)
- etherscan

ethereum

- implemented on top of POW blockchain, batching txns into blocks
- 6.824 picture: replicated state machine on top of a shared log
- world computer implemented as replicated state machine
- meta state machine where you can load and run programs
- state of world computer
  - address ->
    - | externally owned account { balance }
    - | contract account { balance, code, state }
- transition function
  - tx :=
  - | deploy contract (code, value)
  - | message (address, method, data, value)
- semantics
  - deploy contract
    - takes contract code
    - chooses a fresh address and makes contract code available at that address
    - initializes state to initial values
    - sets balance to initial value
    - runs constructor
    - returns address
  - message
    - takes address, method, data, value
    - looks up address: if externally-owned account, just transfers value
    - looks up code, looks up method
    - adds value to contract's balance
    - runs method with data given as an argument, starting from the initial state
    - produces a new state
- EVM: low-level language to support smart contracts
  - many HLLs that compile to EVM
  - Turing-complete -> gas
  - run out of gas -> transaction abort

applications of smart contracts

- ethereum name service (ENS)
  - like blockstack
- ERC20 tokens
  - basically, bank (~ bitcoin) state machine as a smart contract

- issue tokens
- track balances for every user
- trade tokens
- examples
  - stock: initial sale via auction, pays dividends to token holders, token holders can vote
  - asset-backed tokens (e.g. USDC), eases international finance
  - rewards programs (e.g. airline miles)
- DeFi, decentralized exchanges
  - trade ERC20 tokens
- decentralized governance (DAOs)
  - decentralized investment fund, group votes on what to invest in

dapps in practice: bridging ethereum and web apps
- usually, a portion of an app is implemented using smart contracts
- still use backend compute running on standard cloud

advanced smart contracts: complexities in the programming model
- (not a complete list)
- some things we haven't seen in the demos
  - contracts can call other contracts' methods (and send ether)
    - optionally with a gas limit
  - create other contracts
  - try/catch
- emergent complexity
  - what if you run out of gas in the middle of executing a contract?
    - transaction abort: writes reverted and leftover gas refunded
  - what if you call another contract, and that runs out of gas?
    - entire transaction aborts: all state changes reverted
  - what happens if your contract encounters an error (e.g. div by zero)?
    - transaction abort, state reverted
  - what happens if a callee contract encounters an error?
    - same thing: transaction abort, error bubbles up
  - try/catch
    - on error, only callee's changes are reverted
  - working around immutability
    - contracts can call other contracts
    - contracts can "indirect branch", e.g. call(address=storage[0])
    - mutating storage can change code that contract runs
  - oracles

limitations + has ethereum lived up to its vision
- limited compute capacity
- limited storage capacity
- no secret data
- deterministic; "randomness" under control of miners
- transactions are public before they are executed: front-running transactions
- need for oracles to interact with data outside system
- "code is law", immutable code, irreversible transactions: not always a good thing
  - DAO incident, hard fork, ETH vs ETC
- not so easy to program
  - bugs

https://rekt.news/

new programming model and environment, sometimes unintuitive semantics

e.g. funds transfer is a message, contracts can "implement" a handler for this message

e.g. reentrancy vulnerability, exploited in many real-world attacks

```
      Withdraw():
        sender.transfer(balance[sender])
        balance[sender] = 0

hostile and poorly-understood programming environment, e.g. flash loans
  trustless loans (within a single transaction)
  implemented using callbacks and careful programming with try/catch
```

Q: where do contracts execute?
A: on all full nodes, any miner producing a new block or validating a block executes the transactions

Q: how do you choose a gas price?
A: based on network congestion, tools to help like https://ethgasstation.info/

Q: how do you choose a gas limit?
A: wallets can help estimate gas cost
(e.g. how much gas do you need to run the transaction given the current state of the system)