CAP 理论与分布式系统设计

一 引言

在现代分布式系统中，节点数目是巨大的。在 CAP 理论的范围内，Michael Stonebraker 断言分区必然会发生，并且系统内发生节点失败的机会随着节点数的增加而呈指数级增加：

$$P(any\ failure) = 1 - P(individual\ node\ not\ failing)^{number\ of\ nodes}$$

基于这样的事实，很多人会问：

如果发生分区（故障），系统会牺牲什么？一致性还是可用性？

对于这个问题，我有几个反问：

发生分区这个假设在多大概率上会成立？哪些因素会造成分区？

如果发生分区，一致性如何定义？应用需要什么样的一致性？

如果发生分区，可用性如何定义？应用需要什么样的可用性？

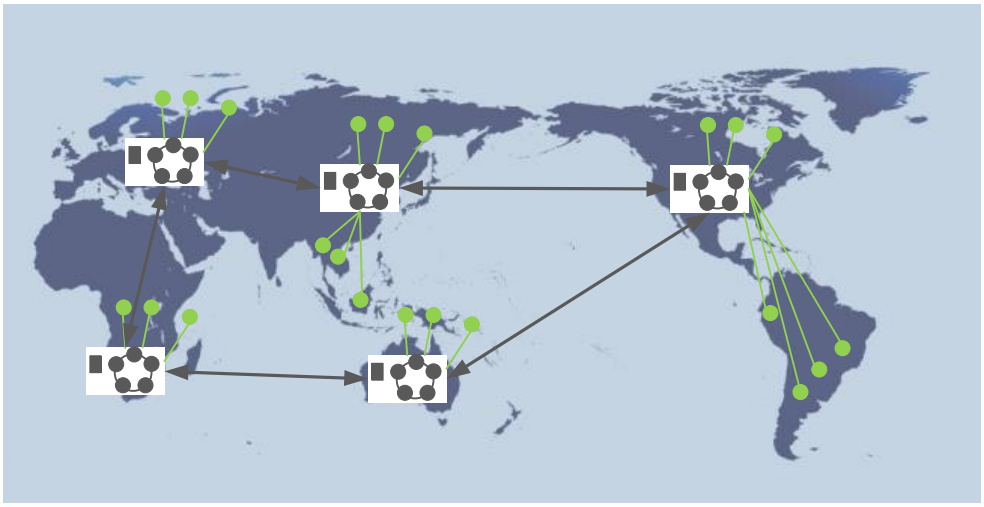
如果不发生分区，一致性和可用性又该如何取舍？

二 全球规模分布式系统

分布式系统是指联网的计算机通过消息传递来协调行为的系统。在这样的系统中机器之间并发执行，独立故障，并且没有全局状态和全局锁。

Distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages.

随着互联网公司的全球化，为了保证服务质量和响应速度，各大互联网公司（Google,Amzon,Facebook,Alibaba 等）纷纷在全球建立数据中心，部署服务和放置数据。为了提高可用性和响应速度，以及满足容灾等需求，这些系统都采用了复制技术。这就带来了服务和数据状态的全球范围内的数据复制和一致性问题。



类似这样的系统有：Dynamo，PNUTS，Cassandra，Megastore，Mesa，Walter，COPS，Spanner，Gemini 等。

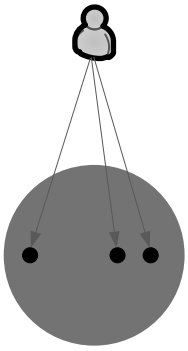
三 分布式系统设计的两大原则

分布式系统设计的原则有很多，这里介绍两个基础性的原则。

通过复制来提高可用性

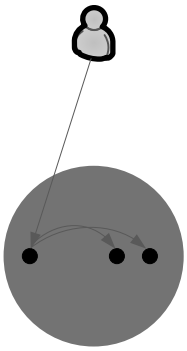
通过复制来提高可用性，这是分布式系统设计的首要原则。从复制类型来看，复制可以分为主动复制和被动复制。

Active replication



在**主动复制**中，每个客户端请求都由所有服务器处理。主动复制首先由 Leslie Lamport 以“ 状态机复制 ”命名。这要求由服务器托管的进程是确定性的。确定性意味着，给定相同的初始状态和请求序列，所有过程将产生相同的响应序列，并且最终处于相同的最终状态。为了使所有的服务器接收到相同的操作序列，一般都使用原子广播协议。原子广播协议保证所有服务器都接收到消息或没有消息，并且他们都以相同的顺序接收消息。主动复制的主要缺点是实际上大多数真实世界的服务器都是非确定性的。

Passive replication

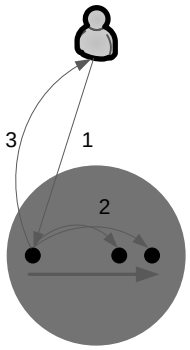


在**被动复制中**，只有一个服务器（称为主服务器）处理客户机请求。处理请求后，主服务器更新其他（备份）服务器上的状态，并将响应发送回客户端。如果主服务器发生故障，则其中一台备份服务器就会接管它。被动复制可以用于非确定性过程。被动复制与主动复制相比的主要缺点是在失败的情况下，客户端的响应会被延迟。

从进程与系统交互角度来看，复制分为同步复制和异步复制。

同步复制 - 通过原子写入操作来保证“零数据丢失”，即完全写入。在本地和远程副本存储的确认之前，写入不被认为是完整的。

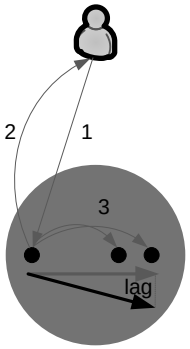
Synchronous replication



Time difference is 0 ms

异步复制 - 本地存储确认后，写入即被认为是完整的。远程存储已更新，但可能滞后很小。系统性能会因异步复制而大大提高。但是在丢失本地存储的情况下，远程存储不能保证具有当前的数据副本，并且最近的数据可能会丢失。

Asynchronous replication



Time difference > 0 ms

关于复制技术，目前有三大模型：事务复制，paxos 复制和虚拟同步。事务复制，paxos 复制讨论的已经比较多，虚拟同步则较少看到有产品采用。虚拟同步定义了一个动态的自组织进程组，这个进程组本身可以看作是一个复制变量，那么这个变量需要特定应用中保持一致。虚拟同步常用的场景是订阅发布和 DHT 中相邻节点的成员关系。虚拟同步和 paxos 协议的区别在于不同的层次：paxos 协议可以用来保证虚拟同步的进程组视图一致。事务复制和 paxos 复制的区别在于：事务复制满足 ACID 语义，有明确的 begin/commit/abort 接口；paxos 复制内部可能使用弱一致性的传言协议，并可以呈现外部的一致性。paxos 复制没有保证提供 ACID 语义和 begin/commit/abort 接口。

https://en.wikipedia.org/wiki/Virtual_synchrony

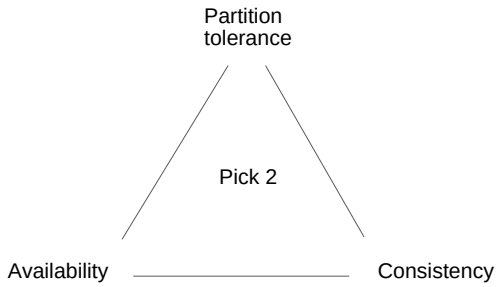
https://en.wikipedia.org/wiki/Replication_(computing)

我个人认为复制的分类方式可以根据节点组织关系分为以下三种：master/slave 复制，paxos 复制和链式复制。这里不赘述。

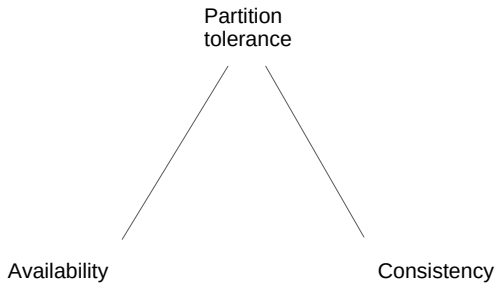
关于复制副本的数量，通常我们讨论的都是 3 个副本，已经满足容灾和高可用的需要。但是在 Chubby ， F1 和 Aurora 中为了更高的可用性，都采用了 5 或 6 个副本。结合同步复制和异步复制，以及链式复制，可以实现混合复制类型的系统，即 5 个副本中部分是实时同步，其他副本可以采用链式复制的方式，或者 paxos 多数原则的方式，实现异步复制。异步复制的副本可以作为快照读取的副本和 OLAP 的副本。

使用 CAP 理论指导分布式系统设计

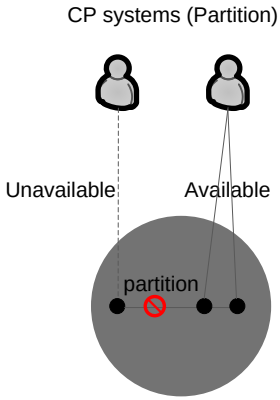
复制技术是产生一致性问题的根源。由此带来了分布式系统设计的第二个原则。



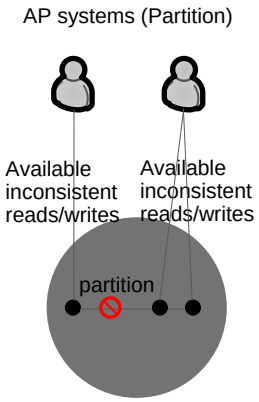
对于 Internet 这样的全球规模的分布式系统，一直以来讨论最多的是 AP 和 CP 系统。



CP 系统是牺牲可用性的系统。复制同步的协议一般使用严格的法定数协议 (paxos, raft, zab)或者 2PC 协议。CP 类型的系统有 MongoDB， HBase， Zookeeper 等。



AP 系统是牺牲一致性的系统。复制同步的协议一般使用非严格的法定数协议。AP 类型的系统有 Couch DB，Cassandra，Amazon Dynamo 等。



那么有没有 CA 系统？如何实现 CA 系统？本文将尝试解答这个问题。

四 重新理解 CAP

CAP 三者并不对等，三选二是误导

在全球广域地理分布环境下（全球规模的分布式系统），网络分区是一个自然的事实，甚至有人认为是必然的。

在这样的情况下，有两种声音：

- 因为分区是必然的，系统设计时，只能实现 AP 和 CP 系统，CA 系统是不可能的。

- 从技术上来说，分区确实会出现，但从效果来说，或者从概率来说，分区很少出现，可以认为系统不会发生分区。由于分区很少发生，那么在系统不存在分区的情况下没什么理由牺牲 C 或 A。

从更广阔的分布式计算理论背景下审视 CAP 理论，可以发现 C，A，P 三者并不对等。

CAP 之父在《Spanner，真时，CAP 理论》一文中写道：如果说 Spanner 真有什么特别之处，那就是谷歌的广域网。Google 通过建立私有网络以及强大的网络工程能力来保证 P，在多年运营改进的基础上，在生产环境中可以最大程度的减少分区发生，从而实现高可用性。

从 Google 的经验中可以得到的结论是，一直以来我们可能被 CAP 理论蒙蔽了双眼，CAP 三者之间并不对称，C 和 A 不是 P 的原因啊（P 不能和 CA trade-off，CP 和 AP 中不存在 tradeoff，tradeoff 在 CA 之间）。提高一个系统的抗毁能力，或者说提高 P（分区容忍能力）是通过提高基础设施的稳定性来获得的，而不是通过降低 C 和 A 来获得的。也就说牺牲 C 和 A 也不能提高 P。

还有一种说法是，放弃 C 不是为了获得 A，而是为了低延迟（延迟不也是可用性的内涵吗？我这里有疑问）。PNUTS 为了降低 WAN 上的消息事务的延迟（几百毫秒，对于像亚马逊和雅虎这样的企业需要实施的许多 Web 应用程序来说，成本太高），采用放弃一致性的做法。

而 CA 是系统的刚性强需求，但是 CA 两者也不对等。系统无论如何要保证一致性（无论事先还是事后，这是系统设计的最大不变性约束，后文会详述），在这个前提下，可以谈谈可用性的程度。Google 的 Spanner 就是这样的思路。

总结：P 是一个自然的事实，CA 是强需求。三者并不对等。

保证不发生分区，CA 也不容易兼得

在分布式系统中，安全性，活性是本质需求，并且广泛的研究结果是分布式系统中一直存在一个广泛意义的 trade-off：在不可靠的分布式系统中无法同时实现安全性和活性。分布式系统的设计中充满了安全性和活性的 trade-off，FLA 著名的论文《Impossibility of Distributed Consensus with One Faulty process》就是说我们不可能设计一个算法既可以绝对保证一致性（安全性）又无需时间等待的实现一致性（活性）。

CAP 就是这个 trade-off 的的集中体现。分别对应于:

Safety:非正式的说，一个算法没有任何坏的事情发生，那么该算法就是安全的。CAP 中的 C 就是典型的 safety 属性：所有对客户的响应都是正确的。

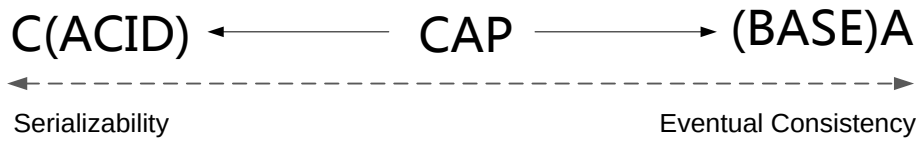
Liveness:相反， 一个算法最终有有一些好的事情发生，那么该算法就是活性的。CAP 中的 A 就是典型的 liveness 属性：所有的客户最终都会收到回应。

FLA 中的故障是指：

Unreliable: 有很多种方式可以让一个系统不可靠，CAP 中的 P，以及其他故障：系统崩溃，消息丢失，恶意攻击，拜占庭故障等。

所以，CAP 理论可以说是 FLA 不可能性的不同表达方式。P 只是 Unreliable 的一种极端形式而已。在 Google 的 Chubby 文章中，指出 paxos 协议维护系统的 safety，引入时钟来保证 liveness，由此克服 FLA 的不可能性。实际上，基本的 Paxos 协议可以保证值一旦被选出后就一定不会改变，但不能保证一定会选出值来。换句话说，这个投票算法不一定收敛。所以在系统设计时，paxos 算法的使用是有条件的。

在数据库领域，CAP 也正是 ACID 和 BASE 长期博弈(tradeoff)的结果。ACID 伴随数据库的诞生定义了系统基本设计思路，所谓先入为主。2000 年左右，随着互联网的发展，高可用的话题被摆上桌面，所以提出了 BASE。从此 C 和 A 的取舍消长此起彼伏，其结晶就是 CAP 理论。



从 ACID 和 BASE 来说，ACID 是为了保证一致性而诞生，因而侧重一致性；BASE 是为了高可用系统的设计而诞生，因而侧重可用性。在分解 C 和 A 的情况时，肯定要涉及 P，所以 CAP 理论统一了这一切。如果非要说酸碱，或者说酸碱平衡，那就是平衡于 CAP 理论。

CAP 并不与 ACID 中的 A（原子性）冲突，值得讨论的是 ACID 中的 C（一致性）和 I（隔离性）。ACID 的 C 指的是事务不能破坏任何数据库规则，如键的唯一性。与之相比，CAP 的 C 仅指单一副本这个意义上的一致性，因此只是 ACID 一致性约束的一个严格的子集。如果系统要求 ACID 中的 I（隔离性），那么它在分区期间最多可以在分区一侧维持操作。事务的可串行性（serializability）要求全局的通信，因此在分区的情况下不能成立。

C 与 A 之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可能因为具体的操作，乃至因为牵涉到特定的数据或用户而有所不同。我们在分布式系统设计的两大原则中讨论过保持一致性的手段：同步复制和异步复制，结合复制协议的各种模式，请参考下表。例如简单满足了 C，但延迟升高了，吞吐量下来了，还有什么可用性？

	Backups	M/S	MM	2PC	Paxos
Consistency	Weak	Eventual		Strong	
Transactions	No	Full	Local	Full	
Latency	Low			High	
Throughput	High			Low	Medium
Data loss	Lots	Some		None	
Failover	Down	Read only		Read/write	

Backups： Make a copy

M/S: Master/slave replication: Usually asynchronous

MM: Multi-master replication: concurrent writes,Usually asynchronous, eventual consistency

2PC: Two Phase Commit: Centralized consensus protocol

Paxos: Protocol similar to 2PC/3PC : Decentralized, distributed consensus protocol

重新审视本文的时候，恰好看到一个新的理论 PACELC：even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C). 可谓和我的想法不谋而合。

https://en.wikipedia.org/wiki/PACELC_theorem

可用性并不是简单的网络连通，服务可以访问，数据可以读取就是可用性，对于互联网业务，可用性是完整的用户体验，甚至会延伸到用户现实生活中（补偿）。有的系统必须容忍大规模可靠分布式系统中的数据不一致，其中原因就是为了在高并发条件下提高读写性能。

必须容忍大规模可靠分布式系统中的数据不一致，有两个原因：在高并发条件下提高读写性能，并要区分物理上导致的不一致和协议规定的不一致：

- 节点已经宕机，副本无法访问（物理）

- 法定数模型会使部分系统不可用的分区情况，即使节点已启动并运行（paxos 协议）
- 网络断开，节点孤立（物理）

所以，保证不发生分区，CA 也不是免费午餐：尽管保证了网络可靠性，尽量不发生分区，同时获得 CA 也不是一件简单的事情。

CA 系统才是真正的难点。

宣称是 CA 系统的，目前有两家：一家是 Google 的 Spanner，一家是 Alibaba 的 OceanBase。

发生分区时，也不要随意牺牲 CA

虽然架构师仍然需要在分区的前提下对数据一致性和可用性做取舍，但具体如何处理分区和恢复一致性，这里面有不计其数的变通方案和灵活度。

当发生分区时，系统设计人员不应该盲目地牺牲一致性或可用性。当分区存在或可感知其影响的情况下，就要预备一种策略去探知分区并显式处理其影响。这样的策略应分为三个步骤：探知分区发生，进入显式的分区模式以限制某些操作，启动恢复过程以恢复数据一致性并补偿分区期间发生的错误。

这一切都需要在系统设计之初考虑到，并在测试时模拟各种故障保证覆盖到你的测试点。

构建高度稳健的基础设施永远是第一要务，所以我不认为网络分区与 CA 属性是对等的。

五 分解，分解，分解

分解 CAP：“三选二”的公式一直存在着误导性，它会过分简单化各性质之间的相互关系。现在我们有必要辨析其中的细节。

CAP 三种性质都可以在程度上衡量，并不是非黑即白的有或无。可用性显然是在 0%到 100%之间连续变化的，一致性分很多级别，连分区也可以细分为不同含义，如系统内的不同部分对于是否存在分区可以有不一样的认知。

P 分解：延迟？故障？分区？

if you're working with distributed systems, you should always be thinking about failure.

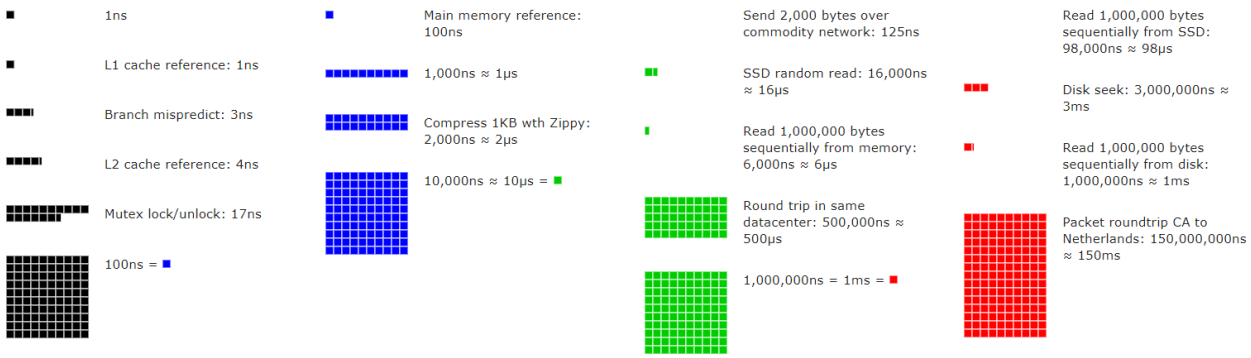
如果你正在使用分布式系统，你应该永远考虑失败。

故障，延迟，分区，是一组非常相关的概念。

在通信网络中，最重要的两个属性是带宽和延迟。延迟也往往取决于链路转发节点的效率。由于延迟的存在，系统很难就全局状态达成一致。当链路发生故障，就会导致网络分区。由于延迟的特性，就算链路没有发生故障，系统也可能判断发生了分区。

对 P 的分解需要从网络开始。网络包含了基础设施，光速限制以及软件配置与升级等。Google 通过建设自己广域网获得高可靠的基础设施支撑，对于 Google Spanner 的 CA 系统，CAP 之父曾总结说网络才是根本。

而光速限制则告诫我们：一致性是一个结果，不是实时的状态。由于光速无法超越，则延迟必然存在（下图显示从加拿大到荷兰的网络延迟在 150 毫秒左右）。延迟的存在让一个节点无法获得对方节点的实时状态。

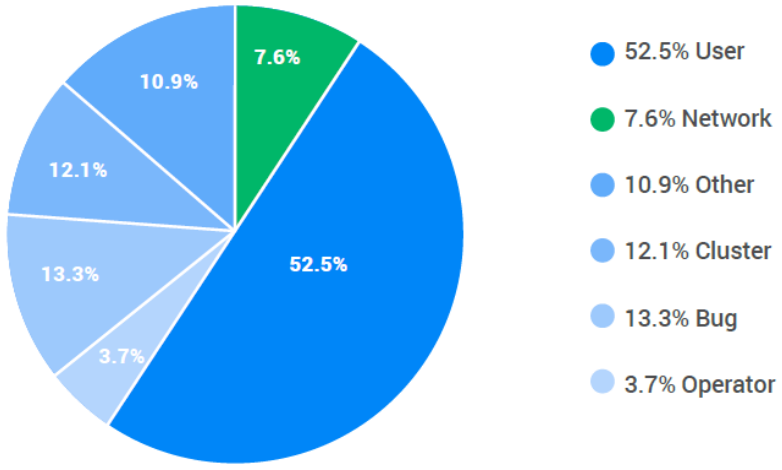


Latency Numbers Every Programmer Should Know

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

由于延迟的存在，有人说，即时性和全球性的一致性是不可能的。宇宙根本不允许它。我以前从事分布式系统研发时，带我的博士总是告诫说，系统设计不要超越时空的限制。

软件配置与升级则体现基础设施搭建工程能力和运维能力。Google 调查了 Spanner 事故的内部原因分类显示，网络类别的事故是由网络分区和网络配置问题造成的。



Michael Stonebraker 在《Errors in Database Systems, Eventual Consistency, and the CAP Theorem》一文中通过对故障进行分解，给出进入 CAP 讨论范围的部分故障列表：

1）应用程序错误。应用程序执行一个或多个不正确的更新。一般来说，这不是几分钟到几个小时之后才发现的。必须将数据库备份到违规事务之前的一个时间点，然后重做后续活动。

2）可重复的 DBMS 错误。DBMS 在处理节点上崩溃。在具有副本的处理节点上执行相同的事务将导致备份崩溃。这些错误被称为 Bohr 错误。

3）不可重复的 DBMS 错误。数据库崩溃了，但是一个副本很可能没问题。这些通常是由处理异步操作的奇怪角落造成的，并且被称为 Heisenbugs。

4）操作系统错误。操作系统崩溃在一个节点，产生“蓝屏死亡”。

5）本地集群中的硬件故障。这些包括内存故障，磁盘故障等。通常，这些会导致操作系统或 DBMS 的“紧急停止”。但是，有些时候这些失败就像 Heisenbugs 一样。

6）本地集群中的网络分区。LAN 失败，节点不能再相互通信。

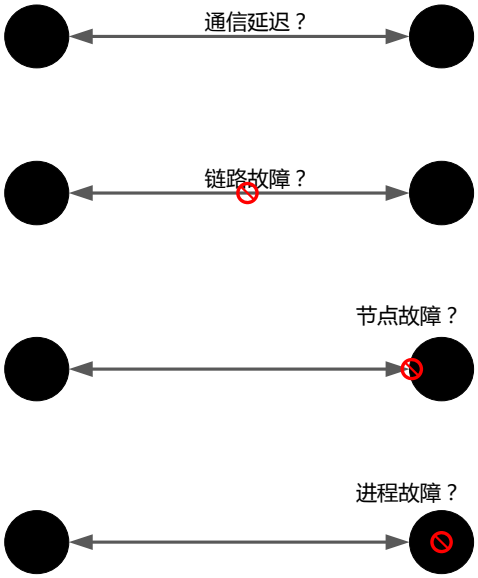
7）灾难。地方数据中心被洪水，地震等等所消灭。群集不再存在。

8）WAN 中的网络故障将集群连接在一起。WAN 失败，群集不能再相互通信。

Michael Stonebraker 总结认为错误 1,2 和 7 是 CAP 定理根本不适用的情况的示例，3,4,5 和 6 是本地故障，错误 8 是 WAN 网络中的一个分区，但是是非常罕见的。所以不要盲目的 follow CAP 理论，轻易的放弃一致性。分解故障，进行针对性的设计。

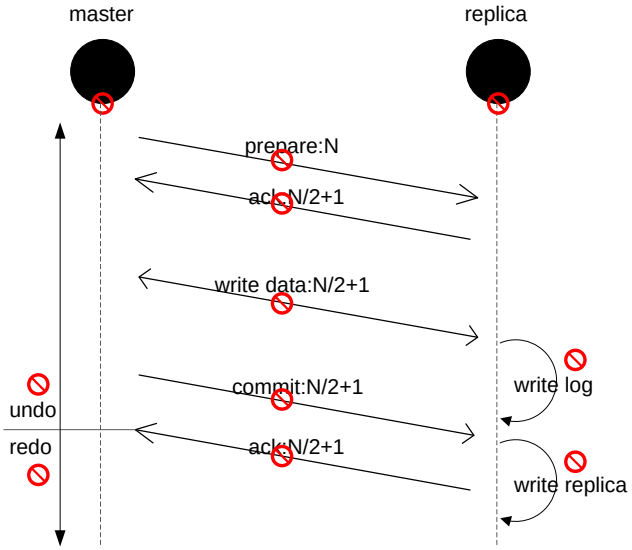
PACELC 理论本质就是对分区进行分解：发生分区时，在 CA 之间取舍；没有发生分区时，在 C 和延迟之间取舍。（我们系统设计的大多数情况就是在没有发生分区的时候）

如何探知分区？这涉及到分布式系统中常见且重要的话题：故障检测。故障检测需要从从分布式系统的定义谈起：节点和连线的模型。在分布式系统中，通过消息通信建立的连接，连接两端的节点在任意时刻，以及节点中运行的进程，随时都可能发生故障。



在分布式系统中，节点故障判断必然依赖超时（时限设置），在一定的超时时间内，消息可能延迟，也可能链路故障造成的节点不可达，在这样的情况下，如何判断节点故障？常见的策略是间隔一段时间重新尝试连接，降低误判的风险，这样就增加了系统的延迟时间，也就是降低了可用性。远端节点无法准确的判断存活还是故障，就无法准确的判断分区是否真的发生。所以 CAP 之父在其著名的文章《CAP Twelve Years Later: How the "Rules" Have Changed》中提出了分布式设计的核心问题：分区两侧是否能够在无通信的情况下继续其操作？

下面举一个复杂的例子。在分布式事务这样的场景中，涉及的消息和操作很多。每一条消息和操作都有可能故障，如下图所示。这就要求我们在程序的设计和实现过程中，针对大量的异常和故障编写代码。这就是故障检测之后的故障处理。



故障处理如何做？有以下模型可以考虑。

Fail-Fast：从字面含义看就是“快速失败”，尽可能的发现系统中的错误，使系统能够按照事先设定好的错误的流程执行，对应的方式是“fault-tolerant（容错）”。只发起一次调用，失败立即报错,通常用于非幂等性的写操作。 如果有机器正在重启，可能会出现调用失败 。

Fail-Over：含义为“失效转移”，是一种备份操作模式，当主要组件异常时，其功能转移到备份组件。其要点在于有主有备，且主故障时备可启用，并设置为主。如 Mysql 的双 Master 模式，当正在使用的 Master 出现故障时，可以拿备 Master 做主使用。阿里同学认为这里可以指失败自动切换。当出现失败，重试其它服务器，通常用于读操作（推荐使用）。 重试会带来更长延迟。

Fail-Safe：含义为“失效安全”，即使在故障的情况下也不会造成伤害或者尽量减少伤害。维基百科上一个形象的例子是红绿灯的“冲突监测模块”当监测到错误或者冲突的信号时会将十字路口的红绿灯变为闪烁错误模式，而不是全部显示为绿灯。有时候来指代“自动功能降级” (Auto-Degrade)。阿里的同学认为失败安全，出现异常时，直接忽略，通常用于写入审计日志等操作。调用信息丢失 可用于生产环境 Monitor。

Fail-Back：Fail-over 之后的自动恢复，在簇网络系统（有两台或多台服务器互联的网络）中，由于要某台服务器进行维修，需要网络资源和服务暂时重定向到备用系统。在此之后将网络资源和服务器恢复为由原始主机提供的过程，称为自动恢复。阿里的同学认为失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作 不可靠，重启丢失。可用于生产环境 Registry。

Forking 并行调用多个服务器，只要一个成功即返回，通常用于实时性要求较高的读操作。 需要浪费更多服务资源 。

Broadcast 广播调用，所有提供逐个调用，任意一台报错则报错。通常用于更新提供方本地状态速度慢，任意一台报错则报错。

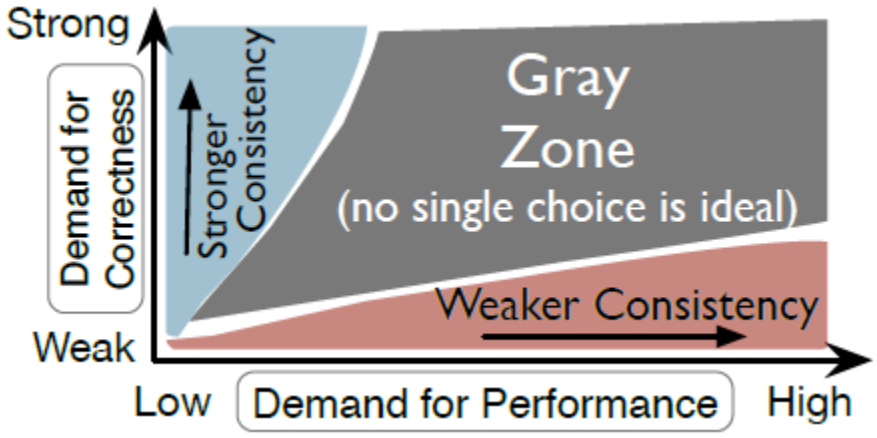
上述故障模型是从系统设计的角度出发的，根据不同的需要设计不同故障处理方案。现在看来，系统的外延已经扩大。系统的容错性，或者分区容错能力，不能仅仅使用事先和事中的方案解决，系统的容错性还包括事后处理。

总之，分区发现的要义是工程问题，即如何构建和加强基础设置的稳定性，如何设计出准确高效的故障检测算法。

C 分解：服务器端与客户端

曾经，透明性也是系统的一个要求和属性（透明性：对于系统的用户来说，只能感受到一个系统而不是多个协作系统）。曾经，系统的一致性模型只有一个：当进行更新时，所有观察者都会看到更新。曾经，我们的系统可以用“邻国相望，鸡犬之声相闻，民至老死不相往来”来描述，不是全球部署，不需要复制技术来保证高可用性，也就不会有一致性问题。然而，随着互联网的发展，可用性被认为是互联网系统中最重要的属性，特别是 CAP 理论的提出，使得我们不得不重新审视当初的一些系统原则。我们必须打破系统透明性，把其中的内部细节暴露出来，同时也思考我们到底需要什么？并且需要付出什么？

关于一致性的讨论主要分为三个方面：这个分法也是性能与一致性之间的平衡。



强一致性

Spanner

PNUTS: Yahoo!’s Hosted Data Serving Platform

弱一致性

Dynamo: Amazon’s Highly Available Key-value Store

CAP 理论：一致性与性能之间的 trade-off，目前关于这方面的研究有很多。比如：

Consistency tradeoffs in modern distributed database system design: CAP is only part of the story

Don’t Settle for Eventual:Scalable Causal Consistency for Wide-Area Storage with COPS

Consistency Tradeoffs in Modern Distributed Database System Design

Consistency rationing in the cloud pay only when it matters

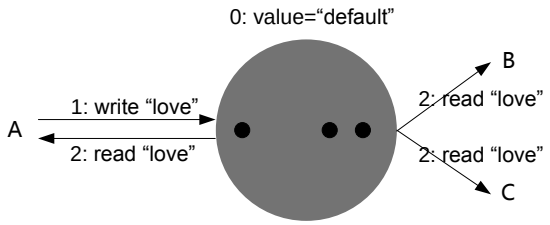
Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary

分解一致性的目的是，在系统设计时，不要盲目的谈 CAP 的三选二，而是通过分解不同业务场景和业务操作，使用合适的一致性模型。如上图所示，有大量操作是属于一致性的灰色区域。系统的设计不要以 CAP 为中心，而是以业务为中心。例如，用户名和密码必须强一致，而用户的属性信息可以是弱一致性的。

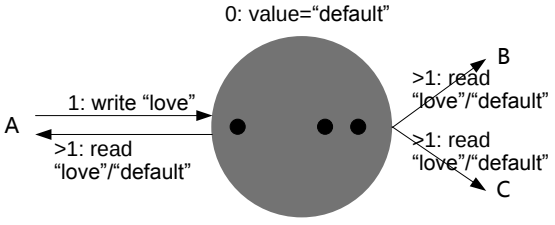
一致性模型本质上是进程和数据存储之间关于一致性所达成的约定（contract）。

首先，我们从客户端的角度，看看有哪些一致性类型。

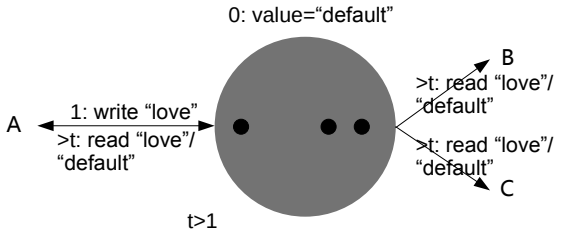
强一致性：更新完成后，所有的后续读操作都会返回更新值。



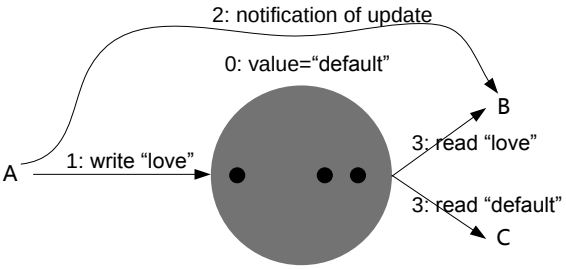
弱一致性：系统并不保证后续读操作获得更新值的时间点。



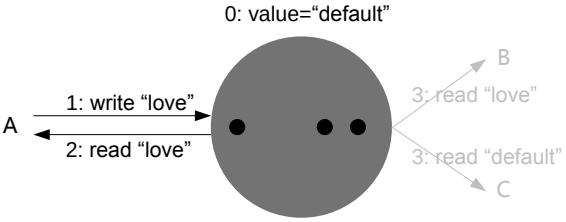
最终一致性：如果没有更新，最终系统会返回最后更新的值。换句话说，如果系统在持续更新，则永远无法达到一致性。



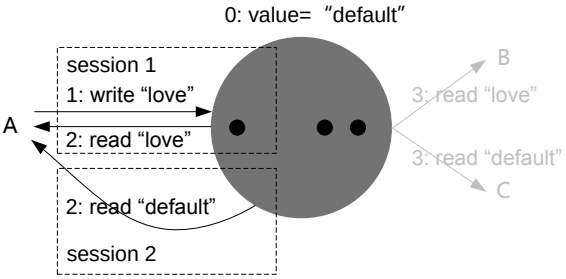
因果一致性：和写进程具有因果关系的进程将会读取到更新的数据，写进程保证取代上次更更新。



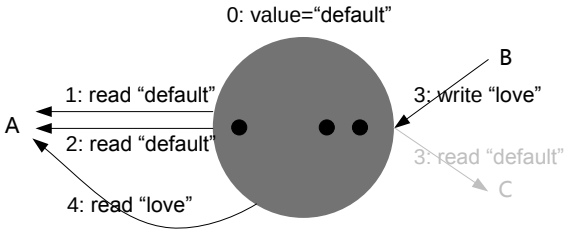
读己所写一致性：进程永远读取自己上次更新写入的最新值，而不可能读取到任何历史数据。这是传统操作系统默认的一致性行为。



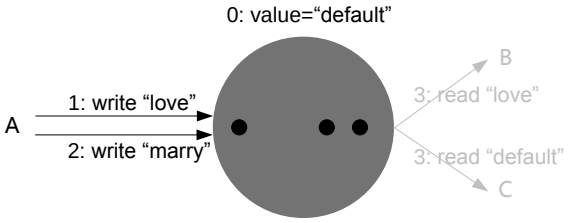
会话一致性：在同一个会话内，系统保证读己所写的一致性。



单调读一致性：进程在读取到系统的一个特定值，则系统永远不会再返回该值以前的任何值

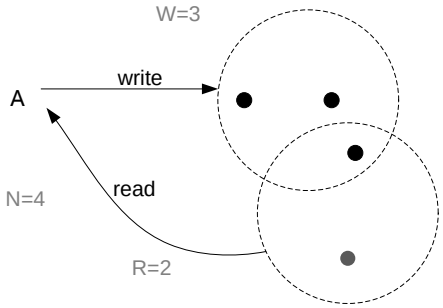


单调写一致性：系统保证同一个进程写入操作的串行化。对于多副本系统来说，保证写顺序的一致性（串行化），是很重要的



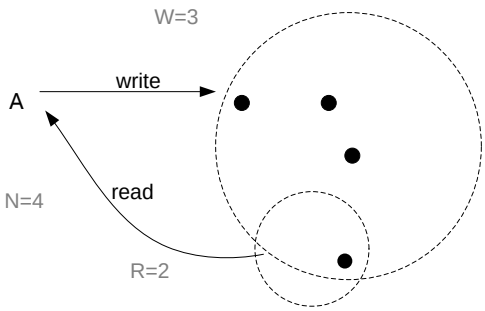
然后我们看下服务器端。

强一致性：



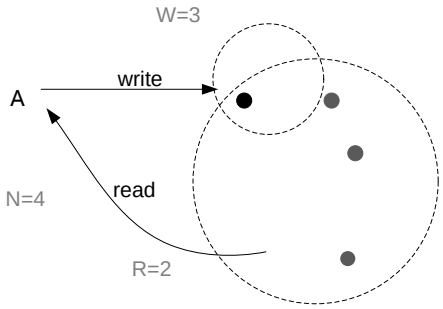
Strong consistency: W + R > N

为读而优化：



Optimize read: R = 1, N = W

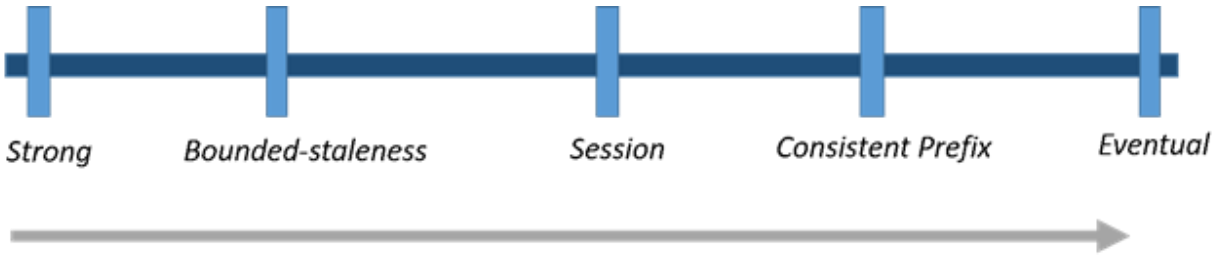
为写而优化：



Optimize write: W = 1, N = R

还有一种一致性模型是 PNUTS 中中提出的，仅保证“时间线一致性”来放松一致性，其中副本可能不一致，但保证在所有副本上以相同的顺序应用更新。

Azure Cosmos DB 中也提出了几种支持的一致性模型，并宣称可选的支持几种不同的一致性模型。



A 分解：出来混，迟早要还的

可用性首先体现在容错。容错不是系统能够在各种故障情况下都能工作，容错是系统发生故障时，系统能够以明确的预定方式运行。下面列出一下可用性指标。

Availability %	How much downtime is allowed per year?
90% ("one nine")	More than a month
99% ("two nines")	Less than 4 days
99.9% ("three nines")	Less than 9 hours
99.99% ("four nines")	Less than an hour
99.999% ("five nines")	~ 5 minutes
99.9999% ("six nines")	~ 31 seconds

在系统设计之初，要考虑系统提供什么程度的可用性，并采用相应的一致性模型。可用性体现在用户体验。在现实生活中，更高的可用性意味着更高的收入。

另外，因为分区引起的可用性问题可以通过事后补偿来获得。

$$C = A + compensation$$

总结起来，C>A>P，这里的大于号，不是包含关系，也不是优先级关系，而是尽力保证 P 是为了 CA，为了保证 C,需要牺牲一点点 A。或者也可以说 A> C>P，为了保证可用性，需要牺牲暂时的不一致性。

这里有人可能有疑问，在某一个场景下，我选择了可用性，放弃了一致性啊？那我说，你一定有补偿措施。也就是说无论事先还是事后，你总要保证一致性。

当代 CAP 实践应将目标定为针对具体的应用，在合理范围内最大化数据一致性和可用性的“合力”。这样的思路延伸为如何规划分区期间的操作和分区之后的恢复，从而启发设计师加深对 CAP 的认识，突破过去由于 CAP 理论的表述而产生的思维局限。

当发生分区时，如何设计可用性？以下几个方面供思考和探讨。

明确系统的不变性约束：通过仔细分析和管理在分区期间系统的不变性约束来优化 CA 属性。我认为系统唯一的不变性约束就是数据的一致性约束。当然你也可以设计 C 和 A 都是系统的不变性约束，然后用 C=A+compensation 来保证一致性，这样就是 CA 系统。就一致性不变性约束来说，唯一的是不变性，有两种：系统内加锁访问的对象和现实生活设置的阈值。例如航空公司，不变性约束必须至少与乘客座位一样多。如果乘客太多，有人会失去座位，理想的客户服务会以某种方式补偿这些乘客。在飞机航班“超售”的情况下，可以把乘客登机看作是对之前售票情况的分区恢复，必须恢复“座位数不少于乘客数”这项不变性约束。那么当乘客太多的时候，有些乘客将失去座位，客服需要补偿他们。航空公司的例子就是系统内需要加锁访问的对象成为不变性约束。而像 ATM 机最后的余额不能低于零的例子就是现实生活设置的阈值，这也是不能违背的不变

性约束。ATM 机的例子也可以引入政府或者操作者的背书机制：比如引入操作者的信用，如果操作者具有超过阈值的信用，就可以继续操作。还有一种补偿方式法律的形式，透支的金额由用户补偿，多扣的手续费退回给客户（和信用绑定：既然一致性已经打破系统的透明性，应用已经参与进来，那就干脆引入社会信用吧）。现在很多系统设计时没有梳理清楚系统的所有不变性约束，并且对其影响也不明确，大多选择了一致性，牺牲一点可用性。

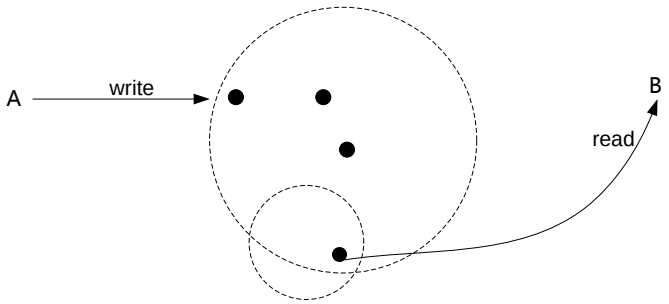
设计补偿机制：管理分区就是管理补偿。根据 C=A+compensation 的思考，所谓不变性约束，前提是无法补偿，如果可以补偿，一切都是可变性约束。一般的补偿分为内部补偿和外部补偿。以重复的订单为例（还有一种情况是删除的数据重新出现），系统可以将合并后的订单中取消一个重复的订单，并不通知用户，这是内部补偿。如果这次错误产生了外在影响，补偿策略可以是自动生成一封电子邮件，向顾客解释系统意外将订单执行了两次，现在错误已经被纠正，附上一张优惠券下次可以用。但 C=A+compensation 并没有打破数据一致性约束，只是让用户在分区的情况下继续保持可用性。

设计数据不变性：补偿依赖完善的日志，我在这里借用《How to beat the CAP theorem》的数据不变性原理，把日志称为数据不变性。**这里的日志是广泛意义的日志，不仅包含数据的所有历史版本，还包括分区历史（时间，地点，人物，加上这些，世界上不存在重复的数据主键）和分区合并历史，以及节点成员关系等。是系统重构？还是节点启动加入系统？还是系统发生了分区？这些都要记录下来。这样当节点加入系统，应该能够找到自己曾经所在的父分区。我认为数据不变性+补偿机制可以打败 CAP 定理。**如果分区之后要保证可用性，用于用户继续操作数据。那么在分区合并之后，继续保持数据的分区状态也是可以的（相当于多了一条分支数据）。可以让数据的元数据中记录分区信息，甚至记录所有分区历史。分区模式操作的跟踪和限制确保知道哪些不变量可能被违反，这又使得设计者能够为每个这样的不变量创建恢复策略。这里和不变性原理是一致的：设计利用的基本原则是“数据本质上是不可变的”。这是因为数据总是与一个时间，人物，地点（分区）相关联，你可以将数据视为当时的事实。所以当你的银行帐户的余额可能会从时间 T 到时间 T+ 1 从 10 变化到 20，以及操作的分区，这些数据，时间，人物，地点永远是真实的。有了这些数据，我们可以任何的补偿操作。上面订单重复的例子，假如没有完善的历史记录，就只好靠顾客亲自去发现错误了。

应用分解：有两个层面。第一是分解业务细节，有些时候这些细节也对应于系统调用或者 API。例如 ATM 的基本操作是存款、取款、查看余额。在分区发生时，存款和查看是可以进行的，取款可以设置取款限额或者不设限额后续补偿。第二是应用程序处理补偿。不一致是否可以接受取决于客户端应用程序。在所有情况下，开发人员需要注意，存储系统提供一致性保证，并在开发应用程序时需要考虑。最终的一致性模型有一些实际的改进，例如会话级一致性和单调读取，为开发人员提供更好的工具。许多时候，应用程序能够处理存储系统的最终一致性保证，没有任何问题。 从可用性的角度来看，阿里 OceanBase 的观点可见一斑：数据库一定是时延很低的，时延高就会导致应用出问题，实际上这个问题要花另外一个篇幅去讲，那就是应用程序必须要去适应这种时延高的数据库系统。当然用了 batching 和 pipeling 技术，本质上都是通用的工程优化，让跨网络多副本同步变得高效，但是时延一定会增加。

六 真正的难题

分布式并发读写事务。如果下图所示，进程 A 和 B 是地理上分布的两个进程，A 进程对系统发起写操作，B 进程同时并发的读取。



首先第一个难题，是否允许任意节点并发可写。在 Google 的 F1，蚂蚁的 OceanBase，亚马逊的 Aurora 中，都是指定一个写节点或者更新节点的（据说 OB 升级 1.0 后，所有节点都是同等地位的）。

第二个难题，是否支持读写并发。这里涉及到读写一致性的问题。比如上图，当用户 A 在写入系统的时候，用户 B 的读取情况是什么样子的？是读取数据的上一个快照，还是读取 A 写入的最新数据？用户 A 和用户 B 在读取的过程中如何加锁？特别跨越广域网的不同的数据中心的时候。这里 tricky 的地方在于是否要对整个数据加读写锁。目前我看 Google 的主要方法是目前 A 进程在写的时候采用多版本数据存储，并保证分布式事务。B 进程可以实现无锁的快照读取。基于中心节点的机制，如果读写冲突或者写写冲突，会被锁机制拒绝，用户操作失败。

第三个难题，元数据如何保存。用户 A 或 B 在读取或者写入系统的时候，如何获得数据的版本和时间戳？这在 OceanBase， spanner/F1，以及 TiDB 中 PD 机制中略有涉及，但都不详细。如果元数据在异地数据中心，获得元数据将会有一个广域网延迟到时间开销。我认为 Google 的 truetime 是用了物理时间代替经典的逻辑时钟，并且作为副本的时间戳，也就是版本号，这样基于 truetime 的精巧 API 设计，让版本号和物理时间线性对齐，无需去查询副本的元数据信息。相反的例子是 Google Chubby 提到的：在一个已经初步成熟的复杂应用程序的每次操作中引入版本号

的开销是非常大的。所以后来退一步求其次，Chubby 采用了仅仅在所有使用锁的操作中引入版本号。

第难个问题，在读写事务期间，节点故障。注意，这里是指任意节点故障，包括一次事务中的 leader 节点，参与节点，以及用户节点。特别是用户所在的节点故障要求系统必须有加锁租约等自恢复机制。

关于锁的设计，在 CAP 的范围内，需要满足三点：

- 锁对象信息的要写入多副本以应对故障
- 不同对象的锁信息需要分布化和负载均衡
- 锁信息写入持久化存储系统

注意，这里锁的概念和 Google Chubby 中锁的概念是不同的。这里是一种粗粒度的锁（leader 选举），其作者也不建议把 Chubby 应用在细粒度锁（事务更新）的场景中。这两种锁在 CAP 的范围内使用时值得非常细心的研究和讨论，特别是在分布式数据库领域内。

第五个难题，嵌套事务或者链式事务。这是电子商务的基础。下面以 Percolator 的实现说明这个问题。银行转账，Bob 向 Joe 转账 7 元。该事务于 start timestamp =7 开始，commit timestamp=8 结束。具体过程如下：

初始状态下，Bob 的帐户下有 10（首先查询 column write 获取最新时间戳数据,获取到 data@5,然后从 column data 里面获取时间戳为 5 的数据，即$10），Joe 的帐户下有 2。

key	*bal:data*	*bal:lock*	*bal:write*
Bob	*6:* *5:* $10	*6:* *5:*	*6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

2、转账开始，使用 stat timestamp=7 作为当前事务的开始时间戳，将 Bob 选为本事务的 primary，通过写入 Column Lock 锁定 Bob 的帐户，同时将数据 7:$3 写入到 Column,data 列。

Bob	*7:* **$3** *6:* *5:* $10	*7:* **I am primary** *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

3、同样的，使用 stat timestamp=7，锁定 Joe 的帐户，并将 Joe 改变后的余额写入到 Column,data,当前锁作为 secondary 并存储一个指向 primary 的引用（当失败时，能够快速定位到 primary 锁，并根据其状态异步清理）

Bob	*7:* $3 *6:* *5:* $10	*7:* I am primary *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	***7:* $9** *6:* *5:* $2	***7:* primary @ Bob.bal** *6:* *5:*	*7:* *6:* data @ 5 *5:*

4、事务带着当前时间戳 commit timestamp=8 进入 commit 阶段：删除 primary 所在的 lock，并在 write 列中写入从提交时间戳指向数据存储的一个指针 commit_ts=>data @7。至此，读请求过来时将看到 Bob 的余额为 3。

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* ***7:*** *6:* *5:*	*8:* **data @ 7** *7:* *6:* data @ 5 *5:*
Joe	*7:* $9 *6:* *5:* $2	*7:* primary @ Bob.bal *6:* *5:*	*7:* *6:*data @ 5 *5:*

5、依次在 secondary 项中写入 write 并清理锁，整个事务提交结束。在本例中，只有一个 secondary:Joe.

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* *7:* *6:* *5:*	*8:* data @ 7 *7:* *6:* data @ 5 *5:*
Joe	*8:* *7:* $9 *6:* *5:*$2	*8:* ***7:*** *6:* *5:*	***8:* data @ 7** *7:* *6:* data @ 5 *5:*

七 总结

本文的主要观点是：

1. CAP 中的三个因素并不对等，系统设计不是三选二的取舍。
2. CAP 真正的 trade-off 在 CA 之间，系统设计需要细心分解 C 和 A，不同的系统有不同的需求。本文在对 CAP 分解的基础上，提供了系统设计的一些思考方法。未来系统的设计必然是要满足多种一致性模型和多种可用性需求（例如微软的 cosmos DB 声称支持多种一致性模型）。

3. 针对分区设计数据不变性，记录所有的分区历史，这让分区合并之后的 compensation 有据可依。借助社会和法律因素，一致性最终都是可以保证的。另外，如果像阿里日照的见解，可用性是一定时间延迟（可能是一天）之后返回响应（在这期间实现服务切换），那么可用性也是可以保证的。
4. 在第 3 点的基础上，未来分布式系统需要从整体上考虑，即需要考虑 IT 基础设施，也要考虑应用的适应和配合，以及人类社会中的法律和补偿。
5. 本文讨论了在 CAP 范围内，分布式系统设计的一些难点。
6. 注意本文的分区和数据库的分片（分区）不是一个概念。

八 附录

附录 1：不变性如何打败 CAP 定理？

以下翻译自《how to beat CAP》。

CAP 定理仍然适用，所以你需要在可用性和一致性上做出选择，这里的漂亮之处在于，一旦你权衡之后做出了选择，你就做完了所有的事情。通常的那些因为 CAP 定理带来的问题，都可以通过不可改变的数据和从原始数据中计算查询来规避。

如果你选择一致性而不是可用性，那么跟以前并没有多大的区别，因为你放弃了可用性，所以一些时候你将无法读取或者写入数据。当然这只是针对对强一致性有要求的系统。

如果你选择可用性而不是一致性，在这种情况下，系统可以达到最终一致性而且规避了所有最终一致性带来的复杂问题。由于系统总是可用的，所以你总可以写入新数据或者进行查询。在出错情况下，查询可能返回的不是最近写入的数据，但根据最终一致性，这个数据最终会一致，而查询函数最终会把这个数据计算进去。

这里的关键在于数据是不可变的。不可变数据意味着这里没有更新操作，所以不可能出现数据复制不同这种不一致的情况，也意味着不需要版本化的数据、矢量时钟或者读取修复。在一个查询场景中，一个数据只有存在或者不存在两种情况。这里只有数据和在数据之上的函数。这里没有需要你为确保最终一致性额外做的事情，最终一致性也不会因此使你的系统变得复杂。

附录 2：放弃 P,选择 C

在《Errors in Database Systems, Eventual Consistency, and the CAP Theorem》一文中，Michael Stonebraker 指出：

A partition in a WAN network. There is enough redundancy engineered into today's WANs that a partition **is quite rare**. **My experience is that local failures and application errors are way more likely.** Moreover, the most likely WAN failure is to separate a small portion of the network from the majority. In this case, the majority can continue with straightforward algorithms, and only the small portion must block. Hence, **it seems unwise to give up consistency all the time in exchange for availability of a small subset of the nodes in a fairly rare scenario.**

In summary, **one should not throw out the C so quickly,** since there are real error scenarios where CAP does not apply and it seems like a bad tradeoff in many of the other situations.

附录 3：用 PACELC 替换 CAP

《Problems with CAP, and Yahoo's little known NoSQL system》

Not only is the asymmetry of the contributions of C, A, and P confusing, but the lack of latency considerations in CAP significantly reduces its utility.

In conclusion, rewriting CAP as PACELC removes some confusing asymmetry in CAP, and, in my opinion, comes closer to explaining the design of NoSQL systems.

参考文献

Immutability Changes Everything

Transactions Across Datacenters

https://snarfed.org/transactions_across_datacenters_io.html

From the Ground Up: Reasoning About Distributed Systems in the Real World

http://bravenewgeek.com/from-the-ground-up-reasoning-about-distributed-systems-in-the-real-world/

How Google Serves Data From Multiple Datacenters

http://highscalability.com/blog/2009/8/24/how-google-serves-data-from-multiple-datacenters.html

Everything You Know About Latency Is Wrong

http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/

You Can't Sacrifice Partition Tolerance

https://codahale.com/you-cant-sacrifice-partition-tolerance/

Distributed systems for fun and profit

http://book.mixu.net/distsys/single-page.html

Distributed Systems and the CAP Theorem

http://ternarysearch.blogspot.fi/2014/03/distributed-systems-and-cap-theorem.html

CAP Twelve Years Later: How the "Rules" Have Changed

http://www.infoq.com/cn/articles/cap-twelve-years-later-how-the-rules-have-changed

《How to beat the CAP theorem》

A Conversation with Bruce Lindsay

http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

http://ternarysearch.blogspot.fi/2014/03/distributed-systems-and-cap-theorem.html

Rethinking Eventual Consistency: Can we do better?

Rethinking Eventual Consistency

Perspectives on the CAP Theorem

Spanner, TrueTime & The CAP Theorem

Eventual Consistent Databases: State of the Art

Incremental Consistency Guarantees for Replicated Objects

http://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

https://github.com/aphyr/partitions-post

https://ying-zhang.github.io/cloud/2017/spanner-truetime-cap/index.html