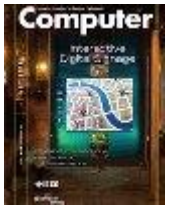
CAP 理论十二年回顾："规则"变了

编者按：由 InfoQ 主办的全球架构师峰会将于 2012 年 8 月 10 日-12 日在深圳举行，为了更好地诠释架构的意义、方法和实践，InfoQ 中文站近期会集中发布一批与架构相关的文章，本篇即为其中之一。InfoQ 也欢迎读者亲身参与到本次全球架构师峰会中，与来自国内外的顶尖架构师进行面对面的交流。报名参会请点击这里。



本文首发于 Computer 杂志，由 InfoQ 和 IEEE 呈现给您。

CAP 理论断言任何基于网络的数据共享系统，最多只能满足数据一致性、可用性、分区容忍性三要素中的两个要素。但是通过显式处理分区情形，系统设计师可以做到优化数据一致性和可用性，进而取得三者之间的平衡。

自打引入 CAP 理论的十几年里，设计师和研究者已经以它为理论基础探索了各式各样新颖的分布式系统，甚至到了滥用的程度。NoSQL 运动也将 CAP 理论当作对抗传统关系型数据库的依据。

CAP 理论主张任何基于网络的数据共享系统，都最多只能拥有以下三条中的两条：

- 数据一致性（C），等同于所有节点访问同一份最新的数据副本；
- 对数据更新具备高可用性（A）；
- 能容忍网络分区（P）。

CAP 理论的表述很好地服务了它的目的，即开阔设计师的思路，在多样化的取舍方案下设计出多样化的系统。在过去的十几年里确实涌现了不计其数的新系统，也随之在数据一致性和可用性的相对关系上产生了相当多的争论。“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。实际上只有“在分区存在的前提下呈现完美的数据一致性和可用性”这种很少见的情况是 CAP 理论不允许出现的。

虽然设计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。

为什么“三选二”公式有误导性

理解 CAP 理论的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了 C 性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了 A 性质。除非两个节点可以互相通信，才能既保证 C 又保证 A，这又会导致丧失 P 性质。一般来说跨区域的系统，设计师无法舍弃 P 性质，那么就只能在数据一致性和可用性上做一个艰难选择。不确切地说，NoSQL 运动的主题其实是创造各种可用性优先、数据一致性其次的方案；而传统数据库坚守 ACID 特性（原子性、一致性、隔离性、持久性），做的是相反的事情。下文“ACID、BASE、CAP”小节详细说明了它们的差异。

事实上，CAP 理论本身就是在类似的讨论中诞生的。早在 1990 年代中期，我和同事构建了一系列的基于集群的跨区域系统（实质上是早期的云计算），包括搜索引擎、缓存代理以及内容分发系统[1]。从收入目标以及合约规定来讲，系统可用性是首要目标，因而我们常规会使用缓存或者事后校核更新日志来优化系统的可用性。尽管这些策略提升了系统的可用性，但这是以牺牲系统数据一致性为代价的。
关于“数据一致性 VS 可用性”的第一回合争论，表现为 ACID 与 BASE 之争[2]。当时 BASE 还不怎么被人们接受，主要是大家看重 ACID 的优点而不愿意放弃。提出 CAP 理论，目的是证明有必要开拓更广阔的设计空间，因此才有了“三选二”公式。CAP 理论最早在 1998 年秋季提出，1999 年正式发表[3]，并在 2000 年登上 Symposium on Principles of Distributed Computing 大会的主题演讲[4]，最终确立了该理论的正确性。
“三选二”的观点在几个方面起了误导作用，详见下文“CAP 之惑”小节的解释。首先，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。其次，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。最后，这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。

要探索这些细微的差别，就要突破传统的分区处理方式，而这是一项根本性的挑战。因为分区很少出现，CAP 在大多数时候允许完美的 C 和 A。但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。

ACID、BASE、CAP

ACID 和 BASE 代表了两种截然相反的设计哲学，分处一致性-可用性分布图谱的两极。ACID 注重一致性，是数据库的传统设计思路。我和同事在 1990 年代晚期提出 BASE，目的是抓住当时正逐渐成型的一些针对高可用性的设计思路，并且把不同性质之间的取舍和消长关系摆上台面。现代大规模跨区域分布的系统，包括云在内，同时运用了这两种思路。

这两个术语都好记有余而精确不足，出现较晚的 BASE 硬凑的感觉更明显，它是“Basically Available, Soft state, Eventually consistent（基本可用、软状态、最终一致性）”的首字母缩写。其中的软状态和最终一致性这两种技巧擅于对付存在分区的场合，并因此提高了可用性。

CAP 与 ACID 的关系更复杂一些，也因此引起更多误解。其中一个原因是 ACID 的 C 和 A 字母所代表的概念不同于 CAP 的 C 和 A。还有一个原因是选择可用性只部分地影响 ACID 约束。ACID 四项特性分别为：

原子性（A）。所有的系统都受惠于原子性操作。当我们考虑可用性的时候，没有理由去改变分区两侧操作的原子性。而且满足 ACID 定义的、高抽象层次的原子操作，实际上会简化分区恢复。
一致性（C）。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。ACID 一致性不可能在分区过程中保持，因此分区恢复时需要重建 ACID 一致性。推而广之，分区期间也许不可能维持某些不变性约束，所以有必要仔细考虑哪些操作应该禁止，分区后又如何恢复这些不变性约束。
隔离性（I）。隔离是 CAP 理论的核心：如果系统要求 ACID 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。只要在分区恢复时进行补偿，在分区前后保持一个较弱的正确性定义是可行的。
持久性（D）。牺牲持久性没有意义，理由和原子性一样，虽然开发者有理由（持久性成本太高）选择 BASE 风格的软状态来避免实现持久性。这里有一个细节，分区恢复可能因为回退持久性操作，而无意中破坏某项不变性约束。但只要恢复时给定分区两侧的持久性操作历史记录，破坏不变性约束的操作还是可以被检测出来并修正的。通常来讲，让分区两侧的事务都满足 ACID 特性会使得后续的分区恢复变得更容易，并且为分区恢复时事务的补偿工作奠定了基本的条件。

CAP 和延迟的联系

CAP 理论的经典解释，是忽略网络延迟的，但在实际中延迟和分区紧密相关。CAP 从理论变为现实的场景发生在操作的间歇，系统需要在这段时间内做出关于分区的一个重要决定：

- 取消操作因而降低系统的可用性，还是
- 继续操作，以冒险损失系统一致性为代价

依靠多次尝试通信的方法来达到一致性，比如 Paxos 算法或者两阶段事务提交，仅仅是推迟了决策的时间。系统终究要做一个决定；无限期地尝试下去，本身就是选择一致性牺牲可用性的表现。

因此以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。这就从延迟的角度抓住了设计的核心问题：分区两侧是否在无通信的情况下继续其操作？

从这个实用的观察角度出发可以导出若干重要的推论。第一，分区并不是全体节点的一致见解，因为有些节点检测到了分区，有些可能没有。第二，检测到分区的节点即进入**分区模式**——这是优化 C 和 A 的核心环节。
最后，这个观察角度还意味着设计师可以根据期望中的响应时间，有意识地设置时限；时限设得越短，系统进入分区模式越频繁，其中有些时候并不一定真的发生了分区的情况，可能只是网络变慢而已。

有时候在跨区域的系统，放弃强一致性来避免保持数据一致所带来的高延迟是非常有意义的。Yahoo 的 PNUTS 系统因为以异步的方式维护远程副本而带来数据一致性的问题[5]。但好处是主副本就放在本地，减小操作的等待时间。这个策略在实际中很实用，因为一般来讲，用户数据大都会根据用户的（日常）地理位置做分区。最理想的状况是每一位用户都在他的数据主副本附近。
Facebook 使用了相反的策略[6]：主副本被固定在一个地方，因此远程用户一般访问到的是离他较近，但可能已经过时的数据副本。不过当用户更新其页面的时候是直接对主副本进行更新，而且该用户的所有读操作也被短暂转向从主副本读取，尽管这样延迟会比较高。20 秒后，该用户的流量被重新切换回离他较近的副本，此时副本应该已经同步好了刚才的更新。

CAP 之惑

CAP 理论经常在不同方面被人误解，对于可用性和一致性的作用范围的误解尤为严重，可能造成不希望看到的结果。如果用户根本获取不到服务，那么其实谈不上 C 和 A 之间做取舍，除非把一部分服务放在客户端上运行，即所谓的无连接操作或称离线模式[7]。离线模式正变得越来越重要。HTML5 的一些特性，特别是客户端持久化存储特性，将会促进离线操作的发展。支持离线模式的系统通常会在 C 和 A 中选择 A，那么就不得不在长时间处于分区状态后进行恢复。
“一致性的作用范围”其实反映了这样一种观念，即在一定的边界内状态是一致的，但超出了边界就无从谈起。比如在一个主分区内可以保证完备的一致性和可用性，而在分区外服务是不可用的。Paxos 算法和原子性多播（atomic multicast）系统一般符合这样的场景[8]。像 Google 的一般做法是将主分区归属在单一个数据中心里面，然后交给 Paxos 算法去解决跨区域的问题，一方面保证全局协商一致（global consensus）如 Chubby[9]，一方面实现高可用的持久性存储如 Megastore[10]。
分区期间，独立且能自我保证一致性的节点子集合可以继续执行操作，只是无法保证全局范围的不变性约束不受破坏。数据分片（sharding）就是这样的例子，设计师预先将数据划分到不同的分区节点，分区期间单个数据分片多半可以继续操作。相反，如果被分区的是内在关系密切的状态，或者有某些全局性的不变性约束非保持不可，那么最好的情况是只有分区一侧可以进行操作，最坏情况是操作完全不能进行。

“三选二”的时候取 CA 而舍 P 是否合理？已经有研究者指出了其中的要害——怎样才算“舍 P”含义并不明确[11,12]。设计师可以选择不要分区吗？哪怕原来选了 CA，当分区出现的时候，你也只能回头重新在 C 和 A 之间再选一次。我们最好从概率的角度去理解：选择 CA 意味着我们假定，分区出现的可能性要比其他的系统性错误（如自然灾难、并发故障）低很多。
这种观点在实际中很有意义，因为某些故障组合可能导致同时丢掉 C 和 A，所以说 CAP 三个性质都是一个度的问题。实践中，大部分团体认为（位于单一地点的）数据中心内部

是没有分区的，因此在单一数据中心之内可以选择 CA；CAP 理论出现之前，系统都默认这样的设计思路，包括传统数据库在内。然而就算可能性不高，单一数据中心完全有可能出现分区的情况，一旦出现就会动摇以 CA 为取向的设计基础。最后，考虑到跨区域时出现的高延迟，在数据一致性上让步来换取更好性能的做法相对比较常见。

CAP 还有一个方面很多人认识不清，那就是放弃一致性其实有隐藏负担，即需要明确了解系统中存在的不变性约束。满足一致性的系统有一种保持其不变性约束的自然倾向，即便设计师不清楚系统中所有的不变性约束，相当一部分合理的不变性约束会自动地维持下去。相反，当设计师选择可用性的时候，因为需要在分区结束后恢复被破坏的不变性约束，显然必须将各种不变性约束一一列举出来，可想而知这件工作很有挑战又很容易犯错。放弃一致性为什么难，其核心还是“并发更新问题”，跟多线程编程比顺序编程难的原因是一样的。

管理分区

怎样缓和分区对一致性和可用性的影响是对设计师的挑战。其关键是以非常明确、公开的方式去管理分区，不仅需要主动察觉分区的发生，还需要为分区期间所有可能受侵害的不变性约束预备专门的恢复过程和计划。管理分区有三个步骤：

（点击看大图）

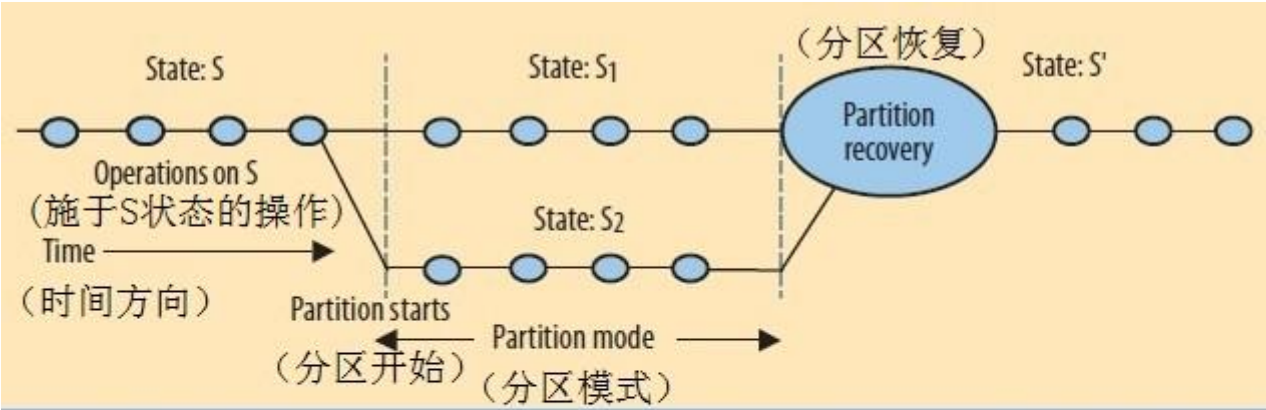


图1. 一开始状态是一致的，并一直保持到分区开始的时刻。为了维持可用性，两侧都进入分区模式并继续执行操作，因此产生不一致的并发状态S_1和S_2。到分区结束时，尘埃落定，分区恢复开始。恢复期间，系统合并S_1和S_2成为一致的状态S'，并且弥补分区期间发生的任何错误。

- 检测到分区开始
- 明确进入分区模式，限制某些操作，并且
- 当通信恢复后启动分区恢复过程

最后一步的目的是恢复一致性，以及补偿在系统分区期间程序产生的错误。

图 1 可见分区的演变过程。普通的操作都是顺序的原子操作，因此分区总是在两笔操作之间开始。一旦系统在操作间歇检测到分区发生，检测方一侧即进入分区模式。如果确实发

生了分区的情况，那么一般分区两侧都会进入到分区模式，不过单方面完成分区也是可能的。单方面分区要求在对方按需要通信的时候，本方要么能正确响应，要么不需要通信；总之操作不得破坏一致性。但不管怎么样，由于检测方可能有不一致的操作，它必须进入分区模式。采取了 quorum 决定机制的系统即为单方面分区的例子。其中一方拥有“法定通过节点数”，因此可以执行操作，而另一方不可以执行操作。支持离线操作的系统明显地含有“分区模式”的概念，一些支持原子多播（atomic multicast）的系统也含有这个概念，如 Java 平台的 JGroups。

当系统进入到分区模式，它有两种可行的策略。其一是限制部分操作，因此会削弱可用性。其二是额外记录一些有利于后面分区恢复的操作信息。系统可通过持续尝试恢复通信来察觉分区何时结束。

哪些操作可以执行？

决定限制哪些操作，主要取决于系统需要维持哪几项不变性约束。在给定了不变性约束条件之后，设计师需要决定在分区模式下，是否坚持不触动某项不变性约束，抑或以事后恢复为前提去冒险触犯它。例如，对于“表中键的惟一性”这项不变性约束，设计师一般都选择在分区期间放宽要求，容许重复的键。重复的键很容易在恢复阶段检查出来，假如重复键可以合并，那么设计师不难恢复这项不变性约束。

对于分区期间必须维持的不变性约束，设计师应当禁止或改动可能触犯该不变性约束的操作。（一般而言，我们没办法知道操作是否真的会破坏不变性约束，因为无法知道分区另一侧的状态。）信用卡扣费等具有外部化特征的事件常以这种方式工作。适合这种情况的策略，是记录下操作意图，然后在分区恢复后再执行操作。这类事务往往从属于一些更大的工作流，在工作流明确含有类似“订单处理中”状态的情况下，将操作推迟到分区结束并无明显的坏处。设计师以用户不易察觉的方式牺牲了可用性。用户只知道自己下了指令，系统稍后会执行。

说得更概括一点，分区模式给用户界面提出了一种根本性的挑战，即如何传达“任务正在进行尚未完成”的信息。研究者已经从离线操作的角度对此问题进行了一些深入的探索，离线操作可以看成时间很长的一次分区。例如 Bayou 的日历程序用颜色来区分显示可能（暂时）不一致的条目 [13]。工作流应用和带离线模式的云服务中也常见类似的提醒，前者的例子如交易中的电子邮件通知，后者的例子如 Google Docs。
在分区模式的讨论中，我们将关注点放在有明确意义的原子操作而非单纯的读写，其中一个原因是操作的抽象级别越高，对不变性约束的影响通常就越容易分析清楚。大体来说，设计师要建立一张所有操作与所有不变性约束的叉乘表格（cross product），观察并确定其中每一处操作可能与不变性约束相冲突的地方。对于这些冲突情况，设计师必须决定是否禁止、推迟或修改相应的操作。在实践中，这类决定还受到分区前状态和/或环境参数的影响。例如有的系统为特定的数据设立了主节点，那么一般允许主节点执行操作，不允许其他节点操作。

对分区两侧跟踪操作历史的最佳方式是使用版本向量，版本向量可以反映操作间的因果依赖关系。向量的元素是（节点, 逻辑时间）数值对，分别对应一个更新了对象的节点和它最后更新的时间。对于同一对象的两个给定的版本 A 和 B，当所有结点的版本向量一致有

A 的时间大于或等于 B 的时间，且至少有一个节点的版本向量有 A 的时间较大，则 A 新于 B。

如果不可能对版本向量排序，那么更新操作是并发的，而且有可能出现不一致的情况。只要知道分区两侧版本向量的沿革。系统不难判断哪些操作的执行顺序是确定的，哪些操作是并发的。最近的研究成果证明 [14]，当设计师选择可用性优先，一般最多只能将一致性收紧到这样的程度。

分区恢复

到了某个时刻，通信恢复，分区结束。由于每一侧在分区期间都是可用的，其状态仍继续向前进展，但是分区会推迟某些操作并侵犯一些不变性约束。分区结束的时刻，系统知道分区两侧的当前状态和历史记录，因为它在分区模式下记录了详尽的日志。当前状态不如历史记录有价值，因为通过历史记录，系统可以判断哪些操作违反了不变性约束，产生了何种外在的后果（如发送了响应给用户）。在分区恢复过程中，设计师必须解决两个问题：

- 分区两侧的状态最终必须保持一致，
- 并且必须补偿分区期间产生的错误。

通常情况，矫正当前状态最简单的解决方法是回退到分区开始时的状态，以特定方式推进分区两侧的一系列操作，并在过程中一直保持一致的状态。Bayou 就是这个实现机制，它会回滚数据库到正确的时刻并按无歧义的、确定性的顺序重新执行所有的操作，最终使所有的节点达到相同的状态 [15]。同样地，并发版本控制系统 CVS 在合并分支的时候，也是从从一个共享的状态一致点开始，逐步将更新合并上去。。

大部分系统都存在不能自动合并的冲突。比如，CVS 时不时有些冲突需要手动介入，带离线模式的 wiki 系统总是把冲突留在产生的文档里给用户处理 [16]。

相反，有些系统用了限制操作的办法来保证冲突总能合并。一个例子就是 Google Docs 将其文本编辑操作 [17] 精简为应用样式、添加文本和删除文本。因此，虽然总的来说冲突问题不可解，但现实中设计师可以选择在分区期间限制使用部分操作，以便系统在恢复的时候能够自动合并状态。如果要实施这种策略，推迟有风险的操作是相对简单的实现方式。

还有一种办法是让操作可以交换顺序，这种办法最接近于形成一种解决自动状态合并问题的通用框架。此类系统将线性合并各日志并重排操作的顺序，然后执行。操作满足交换率，意味着操作有可能重新排列成一种全局一致的最佳顺序。不幸的是，只允许满足交换率的操作这个想法实现起来没那么容易。比如加法操作可以交换顺序，但是加入了越界检查的加法就不行了。

Marc Shapiro 及其 INRIA 同事最近的工作 [18,19] 对于可交换顺序的操作在状态合并方面的应用起了很大的促进作用。该团队提出一种从理论上证明可以保证分区后合并的数据类型，称为可交换多副本数据类型（commutative replicated data types，CRDTs）。他们介绍了如何使用此类数据结构来

- 保证分区期间进行的所有操作都是可交换顺序的，或者
- 用“格（lattice）”的数学概念来表示数据，并保证相对于“格”来说，分区期间的所有操作都是单调递增的。

用后一种方法合并状态会汇总分区两边的最大集合。这种方法是对亚马逊购物车合并算法 [20] 的形式化总结和改良，合并后的数据是两边购物车的并集，而并运算是一种单调的集合运算。这种策略的坏处是删掉的购物车商品有可能再次出现。

其实 CRDTs 完全可以实现同时支持增、删操作的分区耐受集合。此方法的本质是维护两个集合：一个放增加的项目，一个放删除的项目，两集合之差即为真正的集合成员。增集合、删集合分别合并起来都不困难，因而增删集合之差合并起来也不困难。在某个时间点上，系统可以从两个集合中清理掉删除的数据项。假如按照一般的设计，像这种清理操作仅在系统没分区的时候才可行，属于设计师必须在分区期间禁止或推迟的特定操作，但是 CRDTs 的清理操作并不会对可用性产生外在的影响。因此通过 CRDTs 来实现状态，设计师既保证了可用性，又保证了分区后系统自动合并状态。

补偿错误

比计算分区后状态更难解决的问题是如何弥补分区期间造成的错误。跟踪和限制分区模式下的操作，这两种措施足以使设计师确知哪些不变性约束可能被违反，然后分别为它们制定恢复策略。一般系统在分区恢复期间检查违反情况，修复工作也必须在这段时间内完成。

恢复不变性约束的方法有很多，粗陋一点的办法如“最后写入者胜”（因此会忽略部分更新），聪明一点的办法如合并操作和人为跟进事态（human escalation）。人为跟进事态的例子如飞机航班“超售”的情形：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，客服最好能设法补偿他们。

航班的例子揭示了一个外在错误（externalized mistake）：假如航空公司没说过乘客一定有座位，这个问题会好解决得多。因此我们看到推迟有风险的操作的又一个理由——到了分区恢复的时候，我们才知道真实的情况。矫正此类错误的核心概念是“补偿（compensation）”；设计师必须设立补偿操作，除了恢复不变性约束，还要纠正外在错误。

技术上 CRDTs 只允许局部可验证的不变性约束，所以没有补偿的必要，虽然这种限制降低了 CRDTs 方法本身的能力。用了 CRDTs 来处理状态合并的设计方案可以允许暂时违反全局性的不变量约束，分区结束后才合并状态，以及履行必要的补偿。

恢复外在错误通常要求知道一些有关外在输出的历史信息。以“喝醉酒打电话”为例，一位老兄不记得自己昨晚喝高了的时候打过几个电话，虽然他第二天白天恢复了正常状态，但通话日志上的记录都还在，其中有些通话很可能是错误的。拨出的电话就是这位老兄的状态（喝高了）的外在影响。而由于这位老兄不记得打过什么电话，也就很难补偿其中可能造成的麻烦。

又以机器为例，电脑可能在分区期间把一份订单执行了两次。如果系统能区分两份一样的订单是有意的还是重复了，它就能取消掉一份重复的订单。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。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。

曾经有人正式研究过将补偿性事务作为处理长寿命事务（long-lived transactions）的一种手段 [21,22]。长时间运行的事务会面临另一种形态的分区决策：是长时间持有锁来保证一致性比较好呢？还是及早释放锁向其他事务暴露未提交的数据，提高并发能力比较好呢？比

如在单笔事务中更新所有的员工记录就是一个典型例子。按照一般的方式串行化这笔事务，将导致所有的记录都被锁定，阻止并发。而补偿性事务采取另一种方式，它将大事务拆成多个分别提交的子事务。如果要中止大事务，系统必须发起一笔新的、起纠正作用的事务，逐一撤销所有已经提交的子事务，这笔新事务就是所谓的补偿性事务。
总的来说，补偿性事务的目的是避免中止其他用了未正确提交数据的事务（即不允许级联取消）。这种方案不依赖串行化或隔离的手段来保障正确性，其正确性取决于事务序列对状态和输出所产生的净影响。那么，经过补偿，数据库的状态究竟是不是相当于那些子事务根本没执行过一样呢？考虑等价必须连外在行为也包括在内；举个例子，把重复扣取的交易款退还给顾客，很难说成等于一开始就没多收顾客的钱，但从结果上看勉强算扯平了。分区恢复也延续同样的思路。虽然服务不一定总能直接撤销其错误，但起码承认错误并做出新的补偿行为。怎样在分区恢复中运用这种思路效果最好，这个问题没有固定的答案。“自动柜员机上的补偿问题”小节以一个很小的应用领域为例点出了一些思考方向。

当系统中存在分区，系统设计师不应该盲目地牺牲一致性或可用性。运用以上讨论的方法，设计师通过细致地管理分区期间的不变性约束，两方面的性质都可以取得最佳的表现。随着版本向量和 CRDTs 等比较新的技术逐渐被纳入一些简化其用法的框架，这方面的优化手段会得到比较普遍的应用。但引入 CAP 实践毕竟不像引入 ACID 事务那么简单，实施的时候需要对过去的策略进行全面的考虑，最佳的实施方案极大地依赖于具体服务的不变性约束和操作细节。

自动柜员机上的补偿问题

以自动柜员机（ATM）的设计来说，强一致性看似符合逻辑的选择，但现实情况是可用性远比一致性重要。理由很简单：高可用性意味着高收入。不管怎么样，讨论如何补偿分区期间被破坏的不变性约束，ATM 的设计很适合作为例子。

ATM 的基本操作是存款、取款、查看余额。关键的不变性约束是余额应大于或等于零。因为只有取款操作会触犯这项不变性约束，也就只有取款操作将受到特别对待，其他两种操作随时都可以执行。

ATM 系统设计师可以选择在分区期间禁止取款操作，因为在那段时间里没办法知道真实的余额，当然这样会损害可用性。现代 ATM 的做法正相反，在 stand-in 模式下（即分区模式），ATM 限制净取款额不得高于 k，比如 k 为$200。低于限额的时候，取款完全正常；当超过限额的时候，系统拒绝取款操作。这样，ATM 成功将可用性限制在一个合理的水平上，既允许取款操作，又限制了风险。

分区结束的时候，必须有一些措施来恢复一致性和补偿分区期间系统所造成的错误。状态的恢复比较简单，因为操作都是符合交换率的，补偿就要分几种情况去考虑。最后的余额低于零违反了不变性约束。由于 ATM 已经把钱吐出去了，错误成了外部实在。银行的补偿办法是收取透支费并指望顾客偿还。因为风险已经受到限制，问题并不严重。还有一种情况是分区期间的某一刻余额已经小于零（但 ATM 不知道），此时一笔存款重新将余额变为正的。银行可以追溯产生透支费，也可以因为顾客已经缴付而忽略该违反情况。

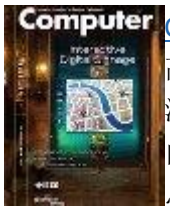
总而言之，因为通信延迟的存在，银行系统不依靠一致性来保证正确性，而更多地依靠审计和补偿。“空头支票诈骗”也是类似的例子，顾客赶在多家分行对账之前分别取出钱来然后逃跑。透支的错误过后才会被发现，对错误的补偿也许体现为法律行动的形式。

致谢

感谢 Mike Dahlin、Hank Korth、Marc Shapiro、Justin Sheehy、Amin Vahdat、Ben Zhao 以及 IEEE Computer Society 的志愿者们，感谢他们对本文的有益反馈。

作者简介

Eric Brewer 是 University of California, Berkeley 的计算机科学教授，在 Google 担任基础设施方面的 VP。他的研究兴趣包括云计算、可伸缩的服务器、传感器网络，还有适合发展中地区应用的技术。他还帮助建立了美国联邦政府的门户网站 USA.gov。Brewer 从 MIT 获得电子工程和计算机科学的博士学位。他是 National Academy of Engineering 的院士。联系方式：brewer@cs.berkeley.edu

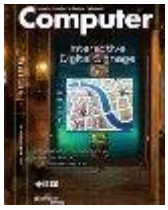


参考文献

1. E. Brewer, "Lessons from Giant-Scale Services," IEEE Internet Computing, July/Aug. 2001, pp. 46-55.
2. A. Fox et al., "Cluster-Based Scalable Network Services," Proc. 16th ACM Symp. Operating Systems Principles (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox and E.A. Brewer, "Harvest, Yield and Scalable Tolerant Systems," Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, "Towards Robust Distributed Systems," Proc. 19th Ann. ACM Symp.Principles of Distributed Computing (PODC 00), ACM, 2000, pp. 7-10; on-line resource.
5. B. Cooper et al., "PNUTS: Yahoo!’s Hosted Data Serving Platform," Proc. VLDB Endowment (VLDB 08), ACM, 2008, pp. 1277-1288.
6. J. Sobel, "Scaling Out," Facebook Engineering Notes, 20 Aug. 2008; on-line resource.
7. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System" ACM Trans. Computer Systems, Feb. 1992, pp. 3-25.
8. K. Birman, Q. Huang, and D. Freedman, "Overcoming the ‘D’ in CAP: Using Isis2 to Build Locally Responsive Cloud Services," Computer, Feb. 2011, pp. 50-58.
9. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," Proc. Symp. Operating Systems Design and Implementation (OSDI 06), Usenix, 2006, pp. 335-350.

10. J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," Proc. 5th Biennial Conf. Innovative Data Systems Research (CIDR 11), ACM, 2011, pp. 223-234.
11. D. Abadi, "Problems with CAP, and Yahoo's Little Known NoSQL System," DBMS Musings, blog, 23 Apr. 2010; on-line resource.
12. C. Hale, "You Can't Sacrifice Partition Tolerance," 7 Oct. 2010; on-line resource.
13. W. K. Edwards et al., "Designing and Implementing Asynchronous Collaborative Applications with Bayou," Proc. 10th Ann. ACM Symp. User Interface Software and Technology (UIST 97), ACM, 1999, pp. 119-128.
14. P. Mahajan, L. Alvisi, and M. Dahlin, Consistency, Availability, and Convergence, tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011.
15. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," Proc. 15th ACM Symp. Operating Systems Principles (SOSP 95), ACM, 1995, pp. 172-182.
16. B. Du and E.A. Brewer, "DTWiki: A Disconnection and Intermittency Tolerant Wiki," Proc. 17th Int'l Conf. World Wide Web (WWW 08), ACM, 2008, pp. 945-952.
17. "What's Different about the New Google Docs: Conflict Resolution" blog.
18. M. Shapiro et al., "Conflict-Free Replicated Data Types," Proc. 13th Int'l Conf. Stabilization, Safety, and Security of Distributed Systems (SSS 11), ACM, 2011, pp. 386-400.
19. M. Shapiro et al., "Convergent and Commutative Replicated Data Types," Bulletin of the EATCS, no. 104, June 2011, pp. 67-88.
20. G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," Proc. 21st ACM SIGOPS Symp. Operating Systems Principles (SOSP 07), ACM, 2007, pp. 205-220.
21. H. Garcia-Molina and K. Salem, "SAGAS," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD 87), ACM, 1987, pp. 249-259.
22. H. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," Proc. VLDB Endowment (VLDB 90), ACM, 1990, pp. 95-106

原文链接：CAP Twelve Years Later: How the "Rules" Have Changed



This article first appeared in Computer magazine and is brought to you by InfoQ & IEEE Computer Society.

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off of all three.

In the decade since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. The NoSQL movement also has applied it as an argument against traditional databases.

How to Build (and Scale) with Microservices

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

- consistency (C) equivalent to having a single up-to-date copy of the data;
- high availability (A) of that data (for updates); and
- tolerance to network partitions (P).

This expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The "2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

Why "2 of 3" is missleading

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P. The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A. In some sense, the NoSQL movement is about creating choices that focus on availability first and consistency second; databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite. The "ACID, BASE, and CAP" sidebar explains this difference in more detail.

In fact, this exact discussion led to the CAP theorem. In the mid-1990s, my colleagues and I were building a variety of cluster-based wide-area systems (essentially early cloud computing), including search engines, proxy caches, and content distribution systems.[1] Because of both revenue goals and contract specifications, system availability was at a premium, so we found ourselves regularly choosing to optimize availability through strategies such as employing caches or logging updates for later reconciliation. Although these strategies did increase availability, the gain came at the cost of decreased consistency.

The first version of this consistency-versus-availability argument appeared as ACID versus BASE,[2] which was not well received at the time, primarily because people love the ACID properties and are hesitant to give them up. The CAP theorem's aim was to justify the need to explore a wider design space-hence the "2 of 3" formulation. The theorem first appeared in fall 1998. It was published in 1999[3] and in the keynote address at the 2000 Symposium on Principles of Distributed Computing,[4] which led to its proof.

As the "CAP Confusion" sidebar explains, the "2 of 3" view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Exploring these nuances requires pushing the traditional way of dealing with partitions, which is the fundamental challenge. Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

Acid, base, and cap

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum. The ACID properties focus on consistency and are the traditional approach of databases. My colleagues and I created BASE in the late 1990s to capture the emerging design approaches for high availability and to make explicit both the choice and the spectrum. Modern large-scale wide-area systems, including the cloud, use a mix of both approaches.

Although both terms are more mnemonic than precise, the BASE acronym (being second) is a bit more awkward: Basically Available, Soft state, Eventually consistent. Soft state and eventual consistency are techniques that work well in the presence of partitions and thus promote availability.

The relationship between CAP and ACID is more complex and often misunderstood, in part because the C and A in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees. The four ACID properties are:

Atomicity (A). All systems benefit from atomic operations. When the focus is availability, both sides of a partition should still use atomic operations. Moreover, higher-level atomic operations (the kind that ACID implies) actually simplify recovery.
Consistency (C). In ACID, the C means that a transaction pre-serves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single•]copy consistency, a strict subset of ACID consistency. ACID consistency also cannot be maintained across partitions.partition recovery will need to restore ACID consistency. More generally, maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery.
Isolation (I). Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition. Serializability requires communication in general and thus fails across partitions. Weaker definitions of correctness are viable across partitions via compensation during partition recovery.
Durability (D). As with atomicity, there is no reason to forfeit durability, although the developer might choose to avoid needing it via soft state (in the style of BASE) due to its expense. A subtle point is that, during partition recovery, it is possible to reverse durable operations that unknowingly violated an invariant during the operation. However, at the time of recovery, given a durable history from both sides, such operations can be detected and corrected. In general, running ACID transactions on each side of a partition makes recovery easier and enables a framework for compensating transactions that can be used for recovery from a partition.

Cap-latency connection

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision-the *partition decision*:

- cancel the operation and thus decrease availability, *or*
- proceed with the operation and thus risk inconsistency.

Retrying communication to achieve consistency, for example, via Paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

Thus, pragmatically, a partition is a time bound on communication. Failing to achieve consistency within the time bound implies a partition and thus a choice between C and A for this operation. These concepts capture the core design issue with regard to latency: are two sides moving forward without communication?

This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is that nodes can detect a partition and enter a *partition mode*-a central part of optimizing C and A.
Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

Sometimes it makes sense to forfeit strong C to avoid the high latency of maintaining consistency over a wide area. Yahoo's PNUTS system incurs inconsistency by maintaining remote copies asynchronously.[5] However, it makes the master copy local, which decreases latency. This strategy works well in practice because single user data is naturally partitioned according to the user's (normal) location. Ideally, each user's data master is nearby.
Facebook uses the opposite strategy:[6] the master copy is always in one location, so a remote user typically has a closer but potentially stale copy. However, when users update their pages, the update goes to the master copy directly as do all the user's reads for a short time, despite higher latency. After 20 seconds, the user's traffic reverts to the closer copy, which by that time should reflect the update.

Cap confusion

Aspects of the CAP theorem are often misunderstood, particularly the scope of availability and consistency, which can lead to undesirable results. If users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as disconnected operation or offline mode,[7] is becoming increasingly important. Some HTML5 features-in particular, on-client persistent storage-make disconnected operation easier going forward. These systems normally choose A over C and thus must recover from long partitions.
Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available. Paxos and atomic multicast systems typically match this scenario.[8] In Google, the primary partition usually resides within one datacenter;

however, Paxos is used on the wide area to ensure global consensus, as in Chubby,[9] and highly available durable storage, as in Megastore.[10]
Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible.

Does choosing consistency and availability (CA) as the "2 of 3" make sense? As some researchers correctly point out, exactly what it means to forfeit P is unclear.[11,12] Can a designer choose not to have partitions? If the choice is CA, and then there is a partition, the choice must revert to C or A. It is best to think about this probabilistically: choosing CA should mean that the probability of a partition is far less than that of other systemic failures, such as disasters or multiple simultaneous faults.
Such a view makes sense because real systems lose both C and A under some sets of faults, so all three properties are a matter of degree. In practice, most groups assume that a datacenter (single site) has no partitions within, and thus design for CA within a single site; such designs, including traditional databases, are the pre-CAP default. However, although partitions are less likely within a datacenter, they are indeed possible, which makes a CA goal problematic. Finally, given the high latency across the wide area, it is relatively common to forfeit perfect consistency across the wide area for better performance.

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system’s invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

Managing partitions

The challenging case for designers is to mitigate a partition’s effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

(Click on the image to enlarge it)

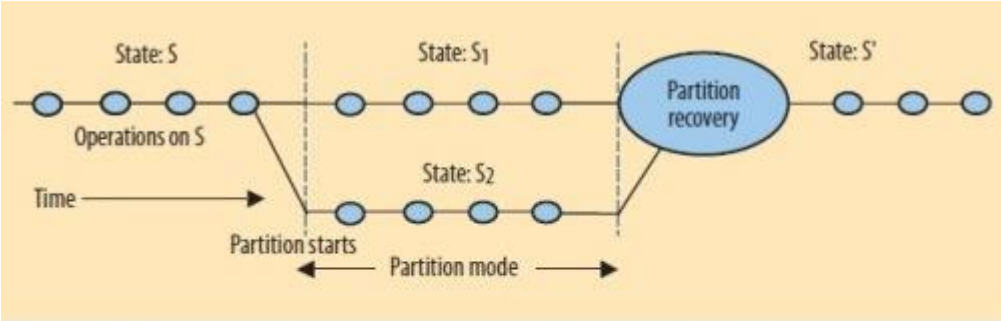


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.

- detect the start of a partition,
- enter an explicit partition mode that may limit some operations, and
- initiate partition recovery when communication is restored.

The last step aims to restore consistency and compensate for mistakes the program made while the system was partitioned.

Figure 1 shows a partition's evolution. Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. In such cases, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could have inconsistent operations, it must enter partition mode. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Systems that support disconnected operation clearly have a notion of partition mode, as do some atomic multicast systems, such as Java's JGroups.

Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

Which operations should proceed?

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recovery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant.

For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. (In general, there is no way to tell if the operation will actually violate the invariant, since the state of the other side is not knowable.) Externalized events, such as charging a credit card, often work this way. In this case, the strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing state, and there is little downside to delaying the operation until the partition ends. The designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition. Bayou's calendar application, for example, shows potentially inconsistent (tentative) entries in a different color.[13] Such notifications are regularly visible both in workflow applications, such as commerce with e-mail notifications, and in cloud services with an offline mode, such as Google Docs.

One reason to focus on explicit atomic operations, rather than just reads and writes, is that it is vastly easier to analyze the impact of higher-level operations on invariants. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation. In practice, these decisions can also depend on the known state, on the arguments, or on both. For example, in systems with a home node for certain data, 5 operations can typically proceed on the home node but not on other nodes.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given two versions of an object, A and B, A is newer than B if, for every node in common in their vectors, A's times are greater than or equal to B's and at least one of A's times is greater.

If it is impossible to order the vectors, then the updates were concurrent and possibly inconsistent. Thus, given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. Recent work[14]proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

Partition recovery

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually

violated invariants and what results were externalized, including the responses sent to the user. The designer must solve two hard problems during recovery:

- the state on both sides must become consistent, and
- there must be compensation for the mistakes made during partition mode.

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. Bayou did this explicitly by rolling back the database to a correct time and replaying the full set of operations in a well-defined, deterministic order so that all nodes reached the same state.[15] Similarly, source-code control systems such as the Concurrent Versioning System (CVS) start from a shared consistent point and roll forward updates to merge branches.

Most systems cannot always merge conflicts. For example, CVS occasionally has conflicts that the user must resolve manually, and wiki systems with offline mode typically leave conflicts in the resulting document that require manual editing.[16]

Conversely, some systems can always merge conflicts by choosing certain operations. A case in point is text editing in Google Docs,[17] which limits operations to applying a style and adding or deleting text. Thus, although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy.

Using commutative operations is the closest approach to a general framework for automatic state convergence. The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is commutative, but addition with a bounds check is not (a zero balance, for example).

Recent work by Marc Shapiro and colleagues at INRIA[18,19] has greatly improved the use of commutative operations for state convergence. The team has developed commutative replicated data types (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to

- ensure that all operations during a partition are commutative, *or*
- represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice.

The latter approach converges state by moving to the maximum of each side's values. It is a formalization and improvement of what Amazon does with its shopping cart:[20] after a partition, the converged value is the union of the two carts, with union being a monotonic set operation. The consequence of this choice is that deleted items may reappear. However, CRDTs can also implement partition-tolerant sets that both add and delete items. The essence of this approach is to maintain two sets: one each for the added and deleted items, with the difference being the set's membership. Each simplified set converges, and thus so does the difference. At some point, the system can clean things up simply by removing the deleted items from both sets. However, such cleanup generally is possible only while the system is not partitioned. In other words, the designer

must prohibit or postpone some operations during a partition, but these are cleanup operations that do not limit perceived availability. Thus, by implementing state through CRDTs, a designer can choose A and still ensure that state converges automatically after a partition.

Compensating for mistakes

In addition to computing the postpartition state, there is the somewhat harder problem of fixing mistakes made during partitioning. The tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated, which in turn enables the designer to create a restoration strategy for each such invariant. Typically, the system discovers the violation during recovery and must implement any fix at that time.

There are various ways to fix the invariants, including trivial ways such as "last writer wins" (which ignores some updates), smarter approaches that merge operations, and human escalation. An example of the latter is airplane overbooking: boarding the plane is in some sense partition recovery with the invariant that there must be at least as many seats as passengers. If there are too many passengers, some will lose their seats, and ideally customer service will compensate those passengers in some way.

The airplane example also exhibits an externalized mistake: if the airline had not said that the passenger had a seat, fixing the problem would be much easier. This is another reason to delay risky operations: at the time of recovery, the truth is known. The idea of compensation is really at the core of fixing such mistakes; designers must create compensating operations that both restore an invariant and more broadly correct an externalized mistake.

Technically, CRDTs allow only locally verifiable invariants-a limitation that makes compensation unnecessary but that somewhat decreases the approach’s power. However, a solution that uses CRDTs for state convergence could allow the temporary violation of a global invariant, converge the state after the partition, and then execute any needed compensations.

Recovering from externalized mistakes typically requires some history about externalized outputs. Consider the drunk "dialing" scenario, in which a person does not remember making various telephone calls while intoxicated the previous night. That person’s state in the light of day might be sound, but the log still shows a list of calls, some of which might have been mistakes. The calls are the external effects of the person’s state (intoxication). Because the person failed to remember the calls, it could be hard to compensate for any trouble they have caused.

In a machine context, a computer could execute orders twice during a partition. If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates. If externalized, one compensation strategy would be to autogenerate an e-mail to the customer explaining that the system accidentally executed the order

twice but that the mistake has been fixed and to attach a coupon for a discount on the next order. With-out the proper history, however, the burden of catching the mistake is on the customer.

Some researchers have formally explored compensating transactions as a way to deal with long-lived transactions.[21,22] Long-running transactions face a variation of the partition decision: is it better to hold locks for a long time to ensure consistency, or release them early and expose uncommitted data to other transactions but allow higher concurrency? A typical example is trying to update all employee records as a single transaction. Serializing this transaction in the normal way locks all records and prevents concurrency. Compensating transactions take a different approach by breaking the large transaction into a saga, which consists of multiple subtransactions, each of which commits along the way. Thus, to abort the larger transaction, the system must undo each already committed subtransaction by issuing a new transaction that corrects for its effects-the compensating transaction.
In general, the goal is to avoid aborting other transactions that used the incorrectly committed data (no cascading aborts). The correctness of this approach depends not on serializability or isolation, but rather on the net effect of the transaction sequence on state and outputs. That is, after compensations, does the database essentially end up in a place equivalent to where it would have been had the subtransactions never executed? The equivalence must include externalized actions; for example, refunding a duplicate purchase is hardly the same as not charging that customer in the first place, but it is arguably equivalent. The same idea holds in partition recovery. A service or product provider cannot always undo mistakes directly, but it aims to admit them and take new, compensating actions. How best to apply these ideas to partition recovery is an open problem. The "Compensation Issues in an Automated Teller Machine" sidebar describes some of the concerns in just one application area.

System designers should not blindly sacrifice consistency or availability when partitions exist. Using the proposed approach, they can optimize both properties through careful management of invariants during partitions. As newer techniques, such as version vectors and CRDTs, move into frameworks that simplify their use, this kind of optimization should become more wide-spread. However, unlike ACID transactions, this approach requires more thoughtful deployment relative to past strategies, and the best solutions will depend heavily on details about the service’s invariants and operations.

Compensation issues in an automated teller machine

In the design of an automated teller machine (ATM), strong consistency would appear to be the logical choice, but in practice, A trumps C. The reason is straightforward enough: higher availability means higher revenue. Regardless, ATM design serves as a good context for reviewing some of the challenges involved in compensating for invariant violations during a partition.

The essential ATM operations are deposit, withdraw, and check balance. The key invariant is that the balance should be zero or higher. Because only withdraw can violate the invariant, it will need special treatment, but the other two operations can always execute.

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using stand-in mode (partition mode), modern ATMs limit the net withdrawal to at most k, where k might be $200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but compensation can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment. Given that the risk is bounded, the problem is not severe. However, suppose that the balance was below zero at some point during the partition (unknown to the ATM), but that a later deposit brought it back up. In this case, the bank might still charge an overdraft fee retroactively, or it might ignore the violation, since the customer has already made the necessary payment.

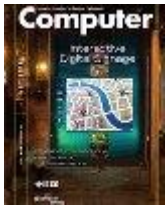
In general, because of communication delays, the banking system depends not on consistency for correctness, but rather on auditing and compensation. Another example of this is "check kiting," in which a customer withdraws money from multiple branches before they can communicate and then flees. The overdraft will be caught later, perhaps leading to compensation in the form of legal action.

Acknowledgments

I thank Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao, and the IEEE Computer Society volunteers for their helpful feedback on this work.

About the Author

Eric Brewer is a professor of computer science at the University of California, Berkeley, and vice president of infrastructure at Google. His research interests include cloud computing, scalable servers, sensor networks, and technology for developing regions. He also helped create USA.gov, the official portal of the federal government. Brewer received a PhD in electrical engineering and computer science from MIT. He is a member of the National Academy of Engineering. Contact him at brewer@cs.berkeley.edu



Computer, the flagship publication of the IEEE Computer Society, publishes highly acclaimed peer-reviewed articles written for and by professionals representing the full spectrum of computing technology from hardware to software and from current research to new applications. Providing more technical substance than trade magazines and more practical ideas than research journals. Computer delivers useful information that is applicable to everyday work environments.

References

1. E. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, July/Aug. 2001, pp. 46-55.
2. A. Fox et al., "Cluster-Based Scalable Network Services," Proc. 16th ACM Symp. *Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox and E.A. Brewer, "Harvest, Yield and Scalable Tolerant Systems," *Proc. 7th Workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, "Towards Robust Distributed Systems," *Proc. 19th Ann. ACM Symp.Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10; on-line resource.
5. B. Cooper et al., "PNUTS: Yahoo!’s Hosted Data Serving Platform," *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
6. J. Sobel, "Scaling Out," *Facebook Engineering Notes*, 20 Aug. 2008; on-line resource.
7. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System" *ACM Trans. Computer Systems*, Feb. 1992, pp. 3-25.
8. K. Birman, Q. Huang, and D. Freedman, "Overcoming the ‘D’ in CAP: Using Isis2 to Build Locally Responsive Cloud Services," *Computer*, Feb. 2011, pp. 50-58.
9. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," *Proc. Symp. Operating Systems Design and Implementation* (OSDI 06), Usenix, 2006, pp. 335-350.
10. J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," *Proc. 5th Biennial Conf. Innovative Data Systems Research* (CIDR 11), ACM, 2011, pp. 223-234.
11. D. Abadi, "Problems with CAP, and Yahoo’s Little Known NoSQL System," *DBMS Musings*, blog, 23 Apr. 2010; on-line resource.
12. C. Hale, "You Can’t Sacrifice Partition Tolerance," 7 Oct. 2010; on-line resource.
13. W. K. Edwards et al., "Designing and Implementing Asynchronous Collaborative Applications with Bayou," *Proc. 10th Ann. ACM Symp. User Interface Software and Technology* (UIST 97), ACM, 1999, pp. 119-128.
14. P. Mahajan, L. Alvisi, and M. Dahlin, *Consistency, Availability, and Convergence*, tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011.
15. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Proc. 15th ACM Symp. Operating Systems*

Principles (SOSP 95), ACM, 1995, pp. 172-182.
16. B. Du and E.A. Brewer, "DTWiki: A Disconnection and Intermittency Tolerant Wiki," Proc. 17th Int'l Conf. World Wide Web (WWW 08), ACM, 2008, pp. 945-952.
17. "What's Different about the New Google Docs: Conflict Resolution" blog.
18. M. Shapiro et al., "Conflict-Free Replicated Data Types," *Proc. 13th Int'l Conf. Stabilization, Safety, and Security of Distributed Systems* (SSS 11), ACM, 2011, pp. 386-400.
19. M. Shapiro et al., "Convergent and Commutative Replicated Data Types," *Bulletin of the EATCS*, no. 104, June 2011, pp. 67-88.
20. G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.
21. H. Garcia-Molina and K. Salem, "SAGAS," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259.
22. H. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pp. 95-106