

# Lecture 17 COPS

---

## 1. 背景知识

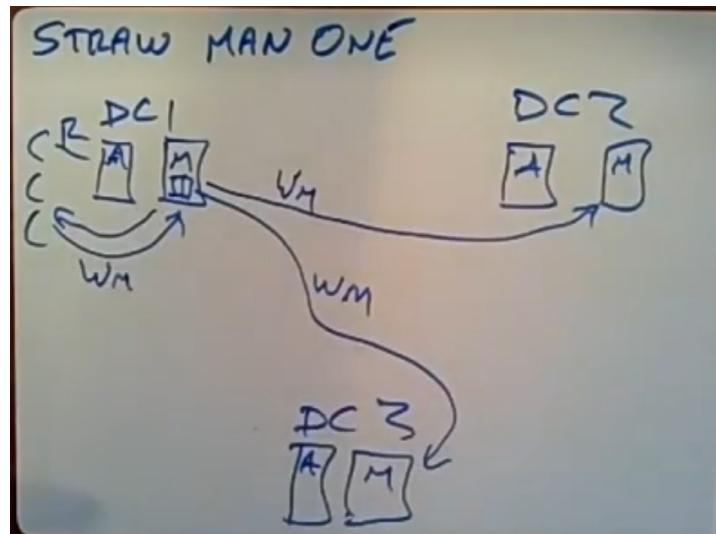
---

- 回顾memcached和spanner的相关架构
  - spanner 有多个DC，多个DC中的不同节点组成一个Paxos Group，形成复制状态机，而后各个Paxos Group作为2PC协议的slave/Coordinator
  - memcached：写操作发送给Master Region，由MySQL提供的同步机制将其分发到各个Region。
  - 这两种机制都需要DC之间的各种交互信息

## 2. 简单的本地读取/写入的实现1 (非COPS)

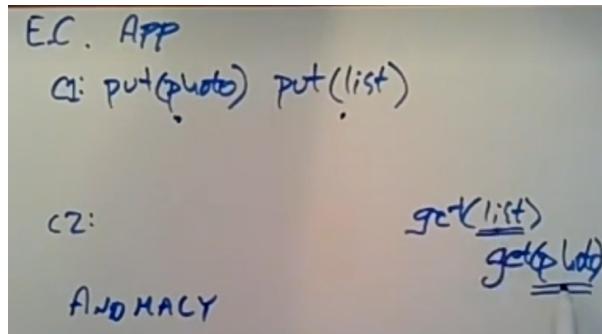
---

- 背景仍然为底层存储作为Server + Web Server作为服务器的形式。
  - 底层存储 (data store) 支持的操作 (类似kv数据库)：
    - get(key)
    - version(V#) := put(key, value)
    - sync(k, V#)， sync操作要求大多数的DC中，关于键k的版本**至少为v#**。sync一方面提供了容错机制，另一方面起到barrier的作用。
  - 数据载荷：读操作占主导
  - **牺牲了强一致性**
- 从client的视角的目的：
  - 本地读取
  - 本地写入
    - 可以提供FT，而不需要考虑其他DC是否正常工作
    - 本地写入更快
- 各个DC内部通过key进行分片 (shard)， DC之间为replication state machine，总体结构如下图



- client向本地DC写入数据，本地执行这些写入，而后本地底层存储维护一个**写入队列**，写入内存中维护的是在本地写入但是未发送到别的DC的数据。队列中的写入操作会被**异步地**发送到别的DC来达成一致性（显然不是强一致性，可以达成最终一致性，换言之，如果经过了足够长的时间，从client的视图来看，各个DC之间的数据应该是一致的）。
- No Order Guarantee:** 这里不是指对不同key的写入操作会以不同的顺序呈现，而是指一个**写入操作下**，不同的DC被更新的顺序可能不同，因而不同的client将会看到不同的数据视图。
- sync 的作用

无sync 操作时，程序猿的编程方式：

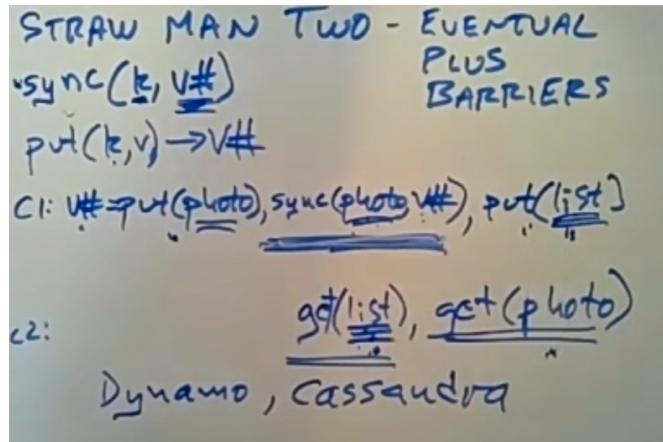


从强一致性的角度来看，上面的代码是正确的，但在最终一致性的系统中可能不正确。例如C1先将自己照片存到kv存储系统中，而后向自己的照片列表中添加关于这条照片的一个引用。但是对于C2而言，其所在的DC可能会先收到put(list)。这就意味着C2根据list去访问照片时，照片尚未传到C2所在的DC之中。

- 程序猿在最终一致性下的编程实现：

使用防御性编程的思路，当获取list之后，检查对应的照片在本地DC中是否可获得，如果不能获得就等待一段时间，而后重试。这种现象被称为**Anomaly (异常)**。显然给程序猿增加了工作量，也使得应用程序的逻辑变得更加复杂。

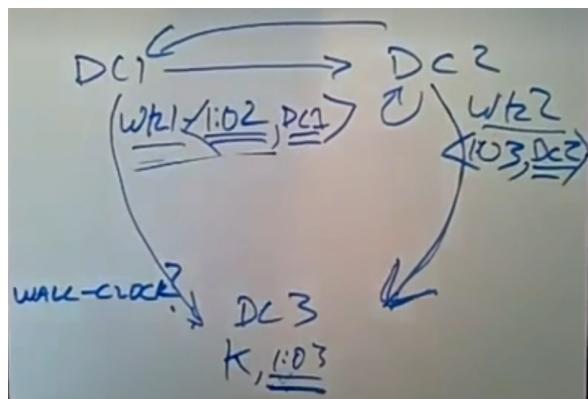
若有sync操作，则编程可以用如下方式实现，可以保证当能获取list时，一定能访问到照片：



## 2.1. 版本号机制

- 对于一个key的并发写入：

考虑下面的情况：



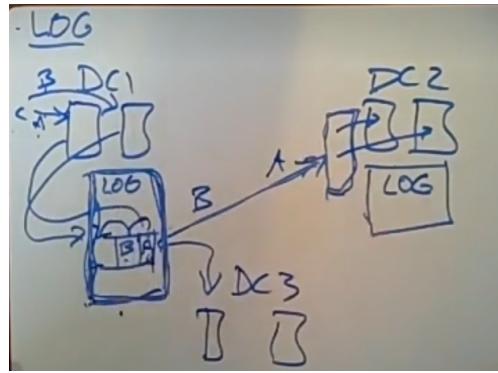
在这里DC1与DC2同时向同一个key写入不同的数值。很有可能导致DC1和DC2最终的数据是不一致的。为了纠正这种情况，为每个写操作定义一个时间戳作为**版本号**，使用最新的写入操作。为了确保时间相同时也能保持最终一致，可以额外在版本号中加入别的数据，例如DC编号等，能保证一定能达成最终一致性。

## 2.2. 时钟同步问题

当各个结点的时钟不同步时，各个写操作的顺序可能会发生错误。可以使用Lamport Clocks，具体如下

- $T_{max}$  = 当前结点所观察到的最大的版本号
- 待写入数据版本号  $T = \max(T_{max} + 1, RealTime)$

## 3. 简单的本地读取/写入的实现2 (非COPS)



- 上面的实现1中，引入了sync操作，因此client需要等待其他的DC的回复消息。在本实现中，在每个DC中引入了一个Log Server/Receive Server。写入操作会被写到本地存储系统中，也会被写入到本地的Log Server中。Log Server会保证，所有的写入操作按照自己队列中的顺序到达每一个DC的Receive Server。
- 这种方案的弊端在于Log Server的负荷过大

## 4. 因果一致性 (Casual Consistency)

- 定义：Values returned from **get** operations at a replica are consistent with the order defined by  $\rightarrow$ 。换言之，对于任意一个写操作，必须发生在任何一个它所**依赖**的操作之前。
- 依赖的定义：
  - Execution Thread：单线程下（可以简单地理解为单一client），先发生的操作  $\rightarrow$  后发生的操作
  - Gets From：若a是put操作，b是get操作，并且b get到的数据是来自a put的数据，则  $a \rightarrow b$
  - Transitivity：传递性
- 缺点：级联依赖问题。依赖的不可满足性是由于对应key在等待其put操作给出的依赖被满足。例子：DC1向DC2和DC3发送put (a, Xv3) 操作，DC2由于dependency中部分键（设为x）不能满足条件而等待，但事实上x已经到达了DC2，但是由于不能满足put(x, yV4)的依赖关系而等待。

## 5. COPS-无GT 原理

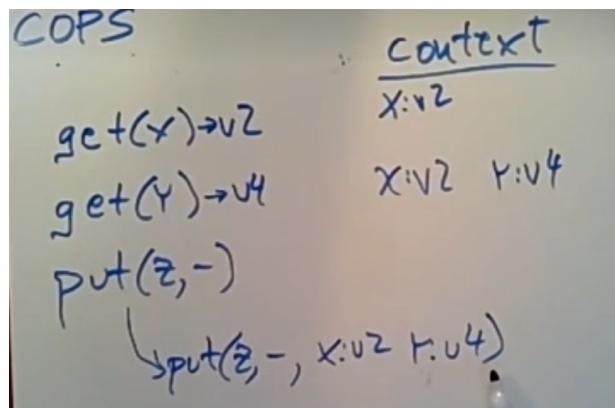
整体结构类似于前文讨论的实现1，没有sync操作

- 库提供给client的接口
  - get(key)

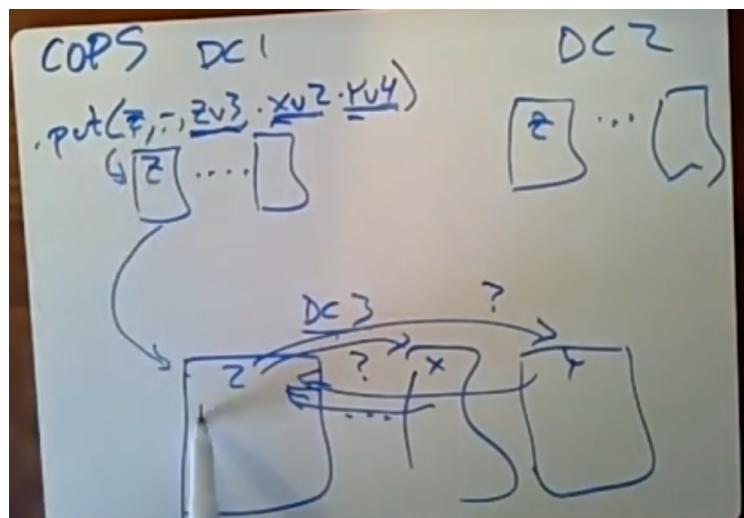
- put(key, value)

## 5.1. Dependency

- 在每个结点（存储了整个键空间的一个分片）收到来自client的请求后，会自动维护一个请求之间的依赖（论文中称为dependency），如图：

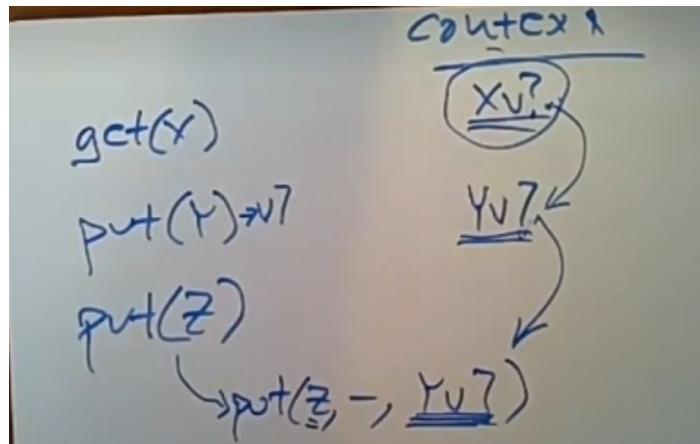


并且由库自动进行上下文的维护，依赖关系对client不可见，也不需要client显式给出。当client向本地DC写入数据后，写入数据的结点（存储对应key的结点）需要向别的DC发送对应的写操作及其依赖，如图：



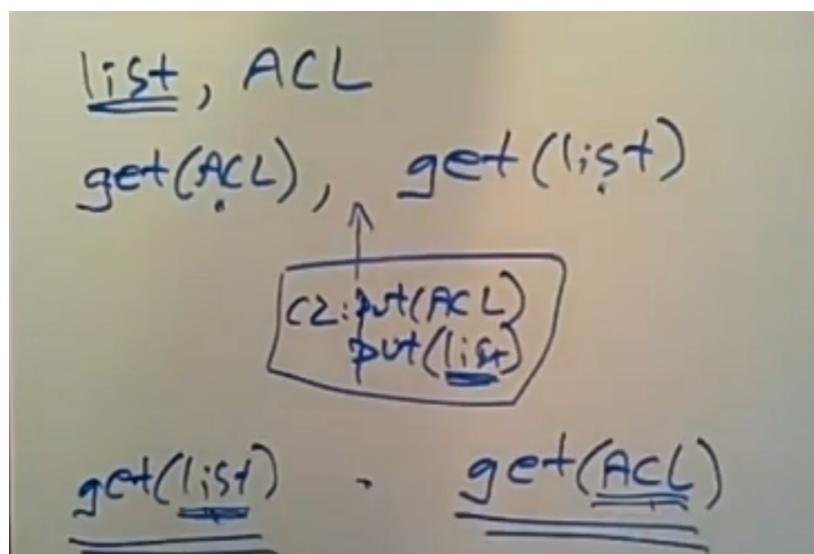
DC1向DC2、DC3分别发送对应的put操作，当DC2和DC3收到该操作后，仅当在dependency列出的一系列key中，key在本机对应的版本号上大于等于dependency给出的版本号时，才能进行写入操作，否则将会暂停，等待对应key被更新到对应的版本。由于DC内部使用shard，因此这个过程需要在同一DC内访问别的存储结点。

- 对于非依赖的两个操作，COPS不维护这两个操作执行的顺序，称它们是**并发的**，并发操作提供良好性能。
- 优化：



client context没有必要维护所有依赖关系。例如在上图中， $\text{put}(y)$ 依赖 $\text{get}(x)$ ， $\text{put}(z)$ 依赖 $\text{put}(y)$ ，形成了级联依赖关系。

- 因果一致性不能解决所有的问题，例如：



在情况一中，C2删除了C1的访问权限ACL，而后在自己的照片list中增加了一张照片，但是C1却仍然能够访问C2的照片，并且这一操作不违反因果一致性；在情况二中，C1原先没有访问权限，而后C2增加了C1的控制权限，并且删除了一张照片，而此时C2却能够看到删除的照片。需要通过COPS-GT来进行控制。

## 5.2. Conflict keys的处理

- conflict keys：当不同的client同时对同一个key进行put操作时，则称这个key为conflict key。
- 处理方式：last-writer-wins
- 不能对抗network partition。需要手动修复该DC。