0.0.1.

# LECTURE 8 CRAQ

## 1. 背景知识

- **object-based storage：** 数据以完整的单元传送给应用程序
  - 提供原语：read/wrtie
  - 由kv 数据库提供支持
  - object-store 更适合扁平命名空间（例如kv数据库），不适合层次结构的命名空间。
  - object-store 在整个object进行修改时会很方便
  - object-store 在推理修改顺序时更容易
  - object-store 更容易保证一致性
- 最终一致性 （eventual consistency）：开始、结尾一致，中间未必一致
- 干净读取：是一种读取操作，读取不会受到正在进行的写入操作的影响。换句话说，干净读取操作不需要等待正在进行的写操作完成，它可以立即读取到数据的最新版本，而不会看到写操作的中间状态或未提交的变更。与脏读形成对比，脏读即为读取了未提交数据。
- **单调读一致性**：如果一个客户端连续地进行多个读取操作，每次读取都返回了不同版本的数据，那么这些版本的顺序应该是单调递增的
- **mini-transaction（小事务）**：是一种处理数据变更的机制，旨在提高系统的性能和效率。传统的数据库事务处理可能涉及多个操作，需要在事务开始和结束时进行一些开销较大的操作，如锁定和日志记录。而 CRAQ 引入了 mini-transaction 的概念，以减少这些开销，从而实现更高的吞吐量。
- One-Hop distributed Hashing table：一跳（one-hop）DHT（分布式哈希表）是一种特殊类型的分布式哈希表，其中每个节点只需要与一个邻居节点通信，而不需要在网络中进行多次跳跃来查找数据或执行哈希操作。这种结构旨在减少通信和延迟，从而提高系统的性能和效率。

## 2. 概述

- CRAQ具有强一致性、改善了读取性能。同时CRAQ天然支持最终一致性（满足强一致性则必然满足最终一致性），可以指定读取过时数据的数目。CR中，链上的一个结点就是集群上的一台计算机
- chain
  - chain 尾用于处理读请求
  - chain 头用于处理写请求
- 存在的问题：
  - hotspot问题：对于某个object的 读处理 需要同一个node来处理，因此可能会产生热点问题。即使采用了一些负载均衡的算法，当某个object过分火热时，仍然会产生不均衡。
  - 当CRAQ应用在多数据中心之间时，对同一条chain的读取可能需要跨数据中心
- 分派询问 （apportioned queries）：apportioned queries为 读操作提供了更低的延时和更高的吞吐量。
  - 通过让读请求分散在整条链的所有结点上，而不是由单一结点来处理读请求

# 3. 系统模型

## 3.1. 接口

- ***write(objID, V)***: The write (update) operation stores the value $V$ associated with object identifier $objID$.
- $V \leftarrow$ ***read(objID)***: The read (query) operation retrieves the value $V$ associated with object id $objID$.

write = put = update

read = get = query

## 3.2. CR （链式复制）

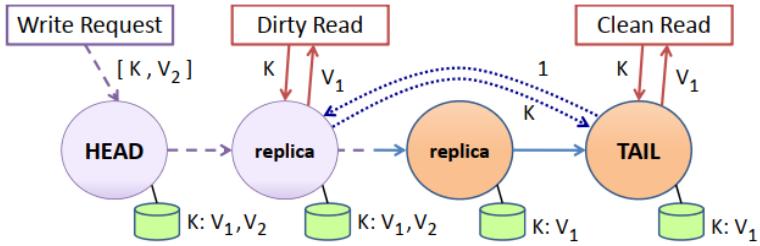
- 写操作：头部负责处理写操作。结点收到写请求之后，将写请求沿着链传播到下一个结点。写操作到达链尾时，则写操作已经应用到了所有的结点上，此时该写操作即为**已提交**。

  当写命令到达尾部结点后，尾部结点需要对命令进行排序，来维护命令到达的全局的顺序（注意，命令到达尾部节点的顺序可能与命令发出的顺序是不一致的，所以需要额外的排序？？？？？？？？？？？？？？？？？？？？？？？？？？？？？）

- 读操作：尾部负责处理读操作，根据写操作的流程，只有已提交的写请求才会被返回，并且由尾部的结点直接返回给client。因此，在传统的CR中，读操作完全由尾部节点完成，因此读操作的吞吐量很小。如果允许链上的所有结点处理读操作，则可能会读到过时的数据，从而违反强一致性

## 3.3. CR + Apportioned Query

### 3.3.1. apportioned query 内容




1. 在每一个node中，为每一个对象都创建了一个版本号。该版本号始终单调递增。每个对象的每个版本还有一个**标记**，为clean/dirty。一开始所有的对象都被标记为clean
2. 每个node为每个对象维护了一个列表，当收到write请求时，node将数据写入对应对象的列表中
    - 若当前node为tail，则当对象写入链尾中后，该对象成为已提交状态。将该对象的对应版本标记为clean，而后通过**确认消息（acknowledge message，似乎是通过TCP协议实现的）** 自后往前传递。确认消息中会包含版本号
    - 若当前node不为tail，则将对象的对应版本标记改为dirty，而后将写操作写到后继结点
3. 若某个node收到了确认消息，则就爱那个对应的node的状态修改为clean，并且删除该对象所有过时的版本
4. 若某个node收到了读请求
    - 若该node中，对象的最新版本的状态为clean，则直接返回
    - 若该node中，对象的最新版本的状态为dirty，则向链尾索要该对象上次提交的版本号（这一过程 称为**version query**），而后该node返回对应版本号的数据

### 3.3.2. 关于Clean Read

传统的，没有apportioned query的CR，会在链尾结点处为所有读写操作定义执行顺序。而CRAQ中，只有写操作是必须在tail处完成的，因此tail处维护了在tail处发生的写读、写写、读读顺序。但发生在非tail结点部分的读操作，CRAQ不会定义它们执行的顺序

#### 3.3.3. Apportioned Query**对吞吐量的影响**

##### 3.3.3.1. **读密集任务**

吞吐量随着集群数目$C$的增长而线性增长

##### 3.3.3.2. **写密集任务**

在写密集任务中，由于写操作频繁，因此读取请求访问dirty node的概率非常之大。因此绝大多数情况下，读操作会需要version query，这一过程需要tail 的参与。但是与传统的CR相比，tail需要处理完整的读请求，而CRAQ只需要tail处理最新的版本号，因此CRAQ在面对写密集任务时，CRAQ中的tail仍然可以比CR中的tail处理更多的读请求。

在长链写密集任务中，可以禁止tail处理完整读请求，只处理版本号来实现优化。

## 3.4. 一致性模型

#### 3.4.1. **强一致性**

对于某一个对象的读写是以某种顺序序列执行的，并且读操作能够读到最新写入的数据，（弱一致性不保证能读到最新的数据）。

在CRAQ中，强一致性也就意味着所有的读取操作必须和对象的最后一次提交的版本完全一致！

#### 3.4.2. **最终一致性 （Eventual Consistency)**

对于某一个对象的读写是以某种顺序序列执行的；在某些数据不一致的阶段可能会读到过时的数据（例如写操作还没有被更新到所有的结点上），但当写操作在所有结点上完成了更新之后，一定能读到最新写入的数据，

在CRAQ中，最终一致性意味着，node将不会检查当前对象最新的version是否为dirty，而是会直接返回。但是最终一致性的情况下，读取操作不满足**单调读一致性。**因为客户可能先与前面的node通信，得到对象的最新版本，而后再与其后继通信，会得到落后的版本，因此不满足单调读一致性。

#### 3.4.3. Eventual Consistency with Maximum-Bounded Inconsistency

也即最终一致性+不一致限制

限制可以是版本号相关的，也可是与时间相关的

# 4. CRAQ部署问题

## 4.1. CRAQ部署策略

不同的使用场景下，CRAQ的部署可能会有不同的策略。CRAQ中的每个对象的命名都由两部分组成：

- chain id
- key id

论文中关于这一部分的内容介绍的并不是很细致，课程中也没有提及。本人的猜测是，数据中心即为链中的一个或多个结点，一个结点的对象分布在该数据中心的所有计算机上。而不是从数据中心中选择出一台计算机作为结点。因此才需要对象命名由两部分组成。chain size则指明了一个数据中心中最多包含**同一条链**的多少个结点，而非链大小?

论文中给出了三种链的配置方式

- 隐式数据中心 + 全局链大小：$\{num\_datacenters; chain\_size\}$，此处数据中心的数目被指定，但是没有指明使用哪些数据中心。因此需要使用**一致性哈希算法**来决定一个链上的结点存在哪些数据中心上。
- 显式数据中心 + 全局链大小：$\{chain\_size; d_{c_1}; d_{c_2}; \ldots; d_{c_N}\}$，此处指定了具体的数据中心。链的头部结点存在$d_{c_1}$中，尾部结点存在$d_{c_N}$中，中间结点则按照上述列表的顺序进行存储。而一个数据中心之中，一个对象存在哪一台计算机上同样需要一致性哈希算法。可以指定$chain_size$为0，表明需要使用每个数据中心的所有结点。
- 显式数据中心链大小：$\{d_{c_1}; chain\_size_1; \ldots; d_{c_N}; chain\_size_N\}$，这一种方式指定了一个数据中心中能存储的链的结点的数目，从而能够更好地实现负载均衡

在上述方法二与方法三中，$d_{c_1}$通常为默认的master。这个master可以被定义也可以不被定义。（推测这种情况是为了对付partition）

- 在定义了master的情况下， 当整条chain发生故障时，写请求只能由master接受。如果master故障了，则master的后继结点会接替master，直到原来的master恢复
- 在未定义master的情况下，仅当某个网络分区包含了chain中的大多数结点时，写请求才能进行
- 在既没有定义master，也没有哪个网络分区包含了chain中的大多数结点时，写请求不能正常进行，chain是只读的

## 4.2. 在单一数据中心内多条链的组织

猜测：

为了优化资源利用、提高并行性等，有时会将多条链映射到同一个头结点。（此时头结点的chain id可能是一个特殊值）。这时多条链之间形成了树形结构，而共用的头结点为这棵树的根节点

### 4.3. 跨数据中心的链

client可能会选择距离自己最近的node进行读操作。（如果对象的当前版本号是clean状态）。但是写操作必须要与头结点进行通信才能完成。因此使用一致性哈希算法可能会导致某个结点在写操作时多次出入某个数据中心。可以通过仔细地选择网络上的结点，确保单向通信时，最多只跨越一个数据中心来进行相关优化

### 4.4. 对于ZooKeeper的利用

CRAQ的实现需要ZooKeeper进行组成员管理和元数据管理的实现。但ZooKeeper本身并没有跨数据中心相关的优化。CRAQ在实现时，对ZooKeeper进行了一定程度上的改造。CRAQ将ZooKeeper分成两层。第二层中，同一数据中心内部可能会有多个ZooKeeper 结点，而后选出其中一个作为代表，同时参与第一层。第一层ZooKeeper Follower相互之间进行跨数据中心协调。

## 5. CRAQ拓展

### 5.1. 小事务

CRAQ提供了Single-Key Operations、Single-Chain Operations和Multi-Chain Operations

### 5.2. 多播降低写延迟

使用多播协议来传输数据，仍然使用链式传输方式来传送元数据，用于保证chain上的结点在tail之前完成写操作。 若某个结点没有收到多播的数据，则只需要在该节点收到commit message时，先向前驱结点获取数据，而后再进一步传播commit message即可。同样，commit message也可以使用多播来发送。若某个结点没有收到多播的commit message，则当它收到读请求时，会询问tail，导致该对象的对应version被更新成clean。

## 6. 实现细节

### 6.1. 集成ZooKeeper

- 组成员管理：维护了每个数据中心内部的结点列表。初始化阶段，创建文件：/nodes/dc_name/node_id。该文件被标记为ephemeral。dc_name是数据中心的名字，node_id是该数据中心内部，该计算机的名字。文件的内容包括该计算机的IP地址和端口号。
- 链元数据管理：当创建一条链时，创建文件/chains/chain_id。chain_id为160位。文件只包含了链的配置信息，而不包含具体的结点。链中的每个结点会对这个配置信息设置watch用于

检测配置信息的变化。同时每个结点都必须要watch 其所在数据中心node的变化。

- 每个结点在加入系统时，会随机生成一个id。one-hop DHT会管理这些id，并且维护结点之间的关系。

### 6.2. 成员变化

当一个新的结点加入链中时，需要它的前驱和后继对它传递数据。这个结点在未与后继达成一致时，会拒绝读请求。结点之间会使用对账算法/一致性检查算法（reconciliation algorithm）来寻找需要传输的最小的数据。在传送某个对象的数据时，会传送这个对象的所有版本的数据，而不是最新提交的（因为可能有未提交的脏数据）。

## 7. 与Raft的对比

对于写请求，在Raft中leader需要一次性发送所有的AppendEntries操作给Follower，而CRAQ只需要发送一次（即传递给后继）。因此在写负载上，Raft比CRAQ负担更大。对于读请求，CRAQ上的所有结点都可以处理读请求，而Raft中只有leader可以处理读请求。

同时，CRAQ不具备处理脑裂的能力。例如头部结点和头部结点的后继失去了联系，两者可能同时都认为自己是头部结点。此时需要**Configuration manager**的参与。Configuration manager会查看CRAQ的结点中哪些已经故障，而后修改Configuration信息。而Configuration manager本身可能是通过Raft、Paxos或者ZooKeeper来实现的。但是由于Configuration只能检查某server是否故障，如果在上述情况下，仅仅是头部结点和头部结点的后继的通信断开，则Configuration会认为两者都是正常工作的。为了应对上述的脑裂情况，Configuration manager不仅需要判断哪些server故障，还需要判断server之间的通信是否正常。

如果不考虑脑裂，则CRAQ的效率通常更高。但是如果要考虑慢服务器，则Raft会更好，因为Raft只需要等待大多数follower 完成AppendEntries即可。同时，Raft还天然地考虑到了地理上相隔很远的数据中心的问题，即等待大多数更近的follower完成操作即可，而CRAQ必须等待链条上每一台server都完成操作才可以。

## 8. 相关论文

- 负载均衡/热点问题 22/29
- chain replication 47

## 9. 历史遗留问题

- one-hop DHT

- 论文中结点增删的问题