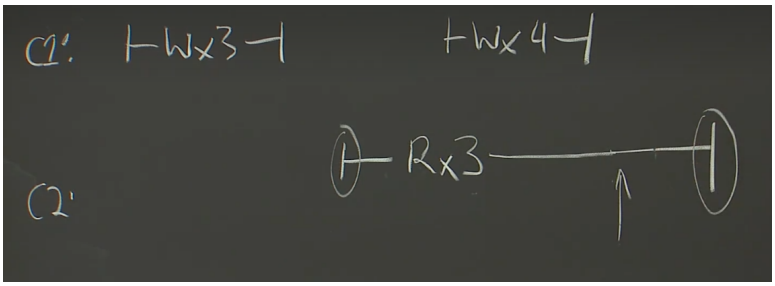
LECTURE 7 ZOOKEEPER

1. 背景知识

- 强同步复制、半同步复制、异步同步复制
  - **强同步**：应用发起数据更新（含insert、update、delete操作）请求，master在执行完更新操作后立即向slave复制数据，slave接受到数据并执行完后才向master返回成功的信息，master必须接收到slave的成功信息后再向应用程序响应。
  - **半同步**：应用发起数据更新（含insert、update、delete操作）请求，master在执行完更新操作后立即向slave复制数据，slave接收到数据并写到replay log中（不需要执行）后才向master返回成功的信息，master必须在接受到slave的成功信息后再向应用程序返回响应。（**raft**）
  - **异步同步**：应用发起数据更新（含insert、update、delete操作）请求，master在执行完更新操作后立即向应用程序返回响应，然后master在向slave同步数据。
- **可线性化（linearizability）**：

  > execution history(record of client operations) is linearizable if：
  - > there exists an order of operations in the history that
    - > matches real time for non concurrent(requests didn't overlap in time)
    - > and each read sees most recent write in the order
  - 注意此处的execution record是从client视角来看的，一条execution record对应了client发出请求的时间点和收到应答的时间点，而内部如何处理这条请求完全不清楚
  - 定义的解释：order of operation是一个操作的顺序。第一条约束的意思为：如果一条operation A在另一条operation B发出请求之前收到回复，则在这个order之中A必须排在B前面。第二条约束的意思为：一条读操作必须在最近的写操作之后
  - 这里的定义是针对执行的历史，而不是分布式系统。只能针对某一段执行历史来判断其是否可线性化，如果需要判断分布式系统是否为可线性化，则必须要知道分布式系统如何工作
  - 可以认为 **可线性化=强一致性**
  - 在下面的情况中，C2发起了Readx，但是没有得到回应，因此在x处重新发送请求（如果请求在箭头处完成，假设应该返回3）。请求在后面完成。由于这个请求持续了很长时

间，并且中间和另一个写请求并发，因此给C2返回3或者4都是正确的（尽管如果不出错，应该返回3）。**此外，leader必须要能够识别重复的请求，不能执行一个请求两次。一种方法是给client的请求加上一个来自client的标志符**



- 可线性化的定义允许请求发出时，过一段时间才回复。（理论上应该在收到请求的时刻就回复）

2. 概述

- 为什么需要使用Zookeeper?
  - 相比于raft，Zookeeper提供了一个通用的协调服务，将其打包并且封装为API，方便分布式系统的开发。而Raft，尽管通常被打包为一个库，但是分布式系统中仍然需要与raft库进行大量的交互
- zookeeper用于提供简单、高性能的内核，该内核可以在客户端建立复杂的协调原语。ZooKeeper实现了一组API，这一API从应用程序的角度出发，允许设计新的原语，而不用修改服务内核。ZooKeeper允许应用程序开发自己的分布式协调方式，而不受固定的原语的局限。ZooKeeper的API可以操纵无等待的数据对象。
- 融合组消息传递（group message）、共享寄存器（shared register）与分布式锁服务（distributed lock service）。其中共享寄存器具有无等待特性（wait-free aspect）
- 保证client的请求以FIFO的顺序执行、保证写操作/更新操作的异步可线性化（linearizability）
- ZooKeeper不使用阻塞原语（例如锁机制），可以保证慢client不影响fast client的运行
- ZooKeeper用流水线结构来实现。这一结构可以允许系统中有大量未完成的请求但与此同时保持低延迟。流水线结构天然地保证了FIFO顺序。FIFO client order使得client可以大量地异步地提交操作
- ZooKeeper实现了基于leader的原子性广播协议Zab，用于保证数据更新操作时的线性化。而对于数据读取操作，在本地进行，因此不需要Zab的参与。此外，ZooKeeper的应用程序，有大量的读请求，少量的更新/写请求

- ZooKeeper使用watch 机制来缓存数据（例如 leader的标识符），而不直接操纵client进行缓存。使用watch机制，client可以监听数据的更新，并且得到通知

3. ZooKeeper服务

- ZooKeeper库：client通过ZooKeeper库来向ZooKeeper提交请求。ZooKeeper库还会管理client和server之间的网络连接
- Zookeeper与Raft有些类似，同样有Zookeeper leader和Zookeeper replica。写操作必须要经过leader，读操作可以直接与replica通信，但是不保证读到最新的数据。因此，随着机器数目的增加，写性能会**下降！**，因为leader需要更大的代价来维护Zookeeper 状态的一致性。但是读操作的性能会上升，因为Zookeeper允许直接读取replica。
- 读到stale数据是Zookeeper在设计时，在强一致性和性能之间的抉择
- 术语：
    - client： ZooKeeper服务的用户
    - server：提供ZooKeeper服务的进程
    - znode：一个在内存中的数据节点。znode以data tree的形式组织。
- znode：ZooKeeper为用户提供数据节点的抽象（也就是znode）。用户可以通过API来控制znode。数据结点使用多级目录的方式进行管理，可以方便元数据的管理。每个znode使用unix表示法来表示其地址。znode有两类：regular以及ephemeral。
    - regular：由client显式进行创建和删除
    - ephemeral：由client显式创建和删除，同时系统也会在创建ephemeral的会话结束时自动删除

  用户可以在创建znode的时候，指定sequential flag 为true，则znode 的文件名后会附加上一个单调递增的数字。这个数字可以保证会比同目录下的其他文件的数字要大
- session：当client与ZooKeeper联系时，即发起一个会话。会话具有timeout。当ZooKeeper超过了timeout规定的时间没有收到来自client的消息时，就会认为client出错/崩溃。当以下两件事件之一发生时，session会关闭：
    - client显式地关闭了session句柄
    - ZooKeeper认为client故障

  在一个会话之内，client可以观察到一系列状态的变化。这些状态变化可以反应请求的执行。

session还能允许client在ZooKeeper集群之内进行转移，这个过程是透明的。同时也能支持持久化操作

- watches：

  是一个与会话相关联的一次触发器。一旦会话终结或者某事件被触发，则watches会取消注册。client可以在发起读取数据请求时，设置watch flag 为true，则ZooKeeper会完成读取，同时server会在这一数据发生变化之后通知client（似乎只会通知数据被修改这一事件！！！！！！）。会话事件（session event）也会被发给回调函数。

  由于getdata（path，flag）可以发给replica，因此当前watch是由replica直接进行处理的。replica内部维护了一张watch表格。当replica 崩溃时，watch 表格会丢失。client重连别的replica server时，新server没有这个表格。但是当旧replica崩溃时，client会收到通知

- 数据模型：ZooKeeper的数据模型本质上类似于一个只支持完全读写的文件系统，或是以层次结构组织的key/value表。这种层次结构可以方便给不同应用分配子树以及分配存取控制权限。然而，znode并非文件，znode是到应用程序的映射，内部存储了用于协调的元数据。某~~个应用程序的结点下，实现了组成员协议（group membership protol）。~~如图所示。应用程序app1的客户端进程p_1, p_2, p_3在/app1结点下创建了子结点。只要client进程没有结束，这些znode就存在。同时，ZooKeeper也允许client存储可以用于分布式计算的元数据。（例如leader相关信息）

3.1. API

`create(path, data, flags):` Creates a znode with path name `path`, stores `data[]` in it, and returns the name of the new znode. `flags` enables a client to select the type of znode: regular, ephemeral, and set the sequential flag;

`delete(path, version):` Deletes the znode `path` if that znode is at the expected version;

`exists(path, watch):` Returns true if the znode with path name `path` exists, and returns false otherwise. The `watch` flag enables a client to set a watch on the znode;

`getData(path, watch):` Returns the data and meta-data, such as version information, associated with the znode. The `watch` flag works in the same way as it does for `exists()`, except that ZooKeeper does not set the watch if the znode does not exist;

`setData(path, data, version):` Writes `data[]` to znode `path` if the version number is the current version of the znode;

`getChildren(path, watch):` Returns the set of names of the children of a znode;

`sync(path):` Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. The path is currently ignored.

在数据更新操作中，需要指定版本号。当znode的版本号与给定的版本号不匹配时，则更新操作返回失败。（但当版本号为1时，不会进行检查）

sync：类似于写操作。由于Zookeeper不保证读一致性，有可能读到过时的数据，因此sync机制弥补了这一缺陷，sync通常用于读取最新数据。通过sync随后紧跟读取的方式，可以保证读取到最新的数据（也即sync之前所有的写操作都会被完成之后，再进行读操作）。

ZooKeeper提供了两套API，分别支持同步和异步的操作。在client不需要进行并发的任务时，可以使用同步API；反之若需要并发操作，可以使用异步API，异步API允许client有多个未完成的任务，以及多个任务并发进行。client需要保证回调函数是按照顺序调用的

3.2. ZooKeeper Ordering Guarantees

3.2.1. **可线性化（linearizable） 写入**

可线性化本文分成两种：

- 同步可线性化：称为 Linearizability。一个client一次只能有一条未完成的请求
- 异步可线性化：称为A-linearizability。一个client一次可以有多条未完成的、多条正在执行的请求

Zookeeper实现的可线性化都是指异步可线性化（A-linearizability）！

Zookeeper的可线性化为**写入的异步可线性化**，并且能保证来自**同一client的的请求**是以先来先服务的方式组织。（即FIFO Client Order Guarantee）如果某结果需要等待同步可线性化的对象，一定也需要等待异步可线性化的对象。（因为一个异步可线性化的系统一定满足同步可线性化，换言之，异步可线性化是同步可线性化的充分不必要条件，能力更强。异步可线性化可以通过限制client一次发出一条未完成请求来变成同步可线性化）。

但Zookeeper不支持读的可线性化！换言之，Zookeeper不能保证client读到最新的数据。因此，client可以在Zookeeper集群上的任何一台server上进行读取请求，而不是与leader通信，这大大提高了读取性能。

写操作的可线性化的含义：即满足了读写可线性化之中的实时性。client可能会发出多个并发异步（也就是说client不会等待写操作完成！）的写请求。如果client发现A在B之前完成，则Zookeeper必定会先执行写操作A，而后再执行写操作B

3.2.2. FIFO Client Guarantee

某一台服务器所发起的所有请求，会以FIFO的顺序进行执行。对于FIFO client Guarantee，必须要分成读写两方面看：

- 对于写操作，其执行顺序是由client发出请求的顺序决定的。由于client是异步发出顺序的，据Moris的推测，实现过程可能是给写请求打上了时间戳来表明写请求执行的时间顺序
- 对于读操作，由于读操作允许在replica上读取，读操作可能会读到过时的数据。但是读操作必定会对应log之中的某一点（log中存的是写操作，这个位置以zxid表示，且这个位置应该是当前replica中最新的位置，详情见后文client interaction部分）。如果读操作A先于读操作B，且读操作A对应日志中的x点，则B在日志中对应的位置必须大于等于x。如果读取A操作之后replica故障了，切换到另一个replica读取时，也必须保证上述性质。

3.2.3. 例：通过一个leader/slave系统，体现上述两点如何保证系统正常交互

通常leader需要一些配置（configuration）参数。这些参数在leader接管系统时被修改。这一过程必须要保证：

- 当leader在修改configuration时，不希望slave读取configuration
- 当leader在修改configuration，但并未完成修改时，leader发生故障，则slave不应该读取configuration

第一点可以通过分布式锁机制来保证，第二点可以通过Zookeeper来保证。Leader可以把一个路径指定为znode，Configuration存在znode之中。当leader 更新configuration时，必须先删除该znode，而后修改configuration，再重新创建包含该Configuration的znode。当slave需要读取configuration时，必须要先检查znode是否存在，而后再进行读取。Ordering Guarantees保证了当leader创建znode之后，configuration已经被完全修改。当leader在完成configuration之前崩溃，slave会无法读取znode，因此不会再去使用configuration。

3.2.4. 两个tricky的问题：

- 在leader/slave架构中，slave先发现了znode 存在，而后发起读取configuration的请求，随后leader开始改写configuration。

  解决：slave可以在读取configuration之前，将watch flag设置为true。则整个过程的时间线如下：slave 使用exist检测Configuration，而后读取configuration，并且设置watch flag，leader开始修改configuration，修改完成，slave接收到通知，知道Configuration发生了修改，会重新读取。（**注意，Zookeeper保证：对某数据设置了watch flag后，若数据发生修改，一定会在后续读取之前先发出通知！**）

- 两个server之间共享configuration，并且存在独立于Zookeeper的communication channel。当A修改了configuration并且通过communication channel通知B时，B会再去读取configuration。但B的Zookeeper replica 可能落后于A，导致B无法读到configuration

  解决：B可以在重新读取configuration之前，发布一个write请求，而后再重新读取，根据Ordering Guarantees就能保证读取到最新的configuration。为了优化上述过程，Zookeeper

提供了一个sync 请求：当一个write请求后面紧跟着一个read请求时，这个read就成为**slow read**，所有在发起sync请求时未执行的write操作都会被server执行，并且时间开销小于一个完整的write操作

3.3. Liveliness Guarantee

若Zookeeper集群之中的大多数server都可用，则通信服务可用

3.4. Durability Guarantee

如果Zookeeper成功地回复了一个更新请求，并且集群大多数server都可以恢复，则这个更新可以持久化，无论有多少次故障

3.5. 使用Zookeeper API可以构建的原语

3.5.1. Configuration 管理

Configuration会被存在znode之中。process在启动时，会读取znode对应的Configuration，并且设置watch flag。如果它接到通知，Configuration被修改，则会再次读取，同时再次设置watch flag。如果在某个进程读取Configuration之前，它已经收到了Configuration被修改的通知，且Configuration被修改了多次，但是由于watches机制是一次性的，所以只会收到一次通知。这并不会影响进程的正确性。因为即使多次通知，也只能表明当前进程内部存储的Configuration是过期的而已

3.5.2. Rendezvous 集合点

有时在leader/slave（master/worker）结构中，client需要创建若干worker和master进程。但是这些进程通常由调度器产生，因此client无法提前知道master的端口和地址是什么。client会创建一个znode，而后将znode 的完整路径名作为参数传递给worker和master。当worker被创建时，会使用read操作并且设置watch flag 来监听这个znode。当master向其中写入了自己的端口和地址之后，worker则会被通知，从而得到master的端口和地址。如果这个znode是一个ephemeral znode，则当它被删除时，master和worker就知道可以结束进程了

3.5.3. 组成员 Group Membership

创建一个znode z_g用于表示整个组。当组内某个进程启动之后，就在z_g下添加一个ephemeral子结点（注意：根据ephemeral子结点的性质，其生命周期与会话的生命周期有关联。若client没有显式地删除ephemeral node，则当client与Zookeeper的会话结束时，ephemeral结点也会被删除），子结点中的数据可以是进程的端口，地址等。若进程有名字，则该子结点可以以进程的名字命名；反之，可以在创建时指定sequential flag为true，则Zookeeper会为该子结点创建一个独一无二的名字（通过递增的编号）。而后，只需要read z_g并且set watch flag，则组成员发生变化时，会得到通知。想要知道所有的组成员，只需要列出z_g的子结点即可

3.5.4. 简单锁机制

这里的锁机制实现只支持互斥锁/排他锁。实现方法为：需要上锁时，创建一个ephemeral znode。若能创建成功说明获得了锁，若创建失败，说明当前的锁有人占有，此时可以read 这个znode，set watch = true，则当锁被释放时，会接收到通知。

然而这种锁存在一些问题：

- 群体效应/羊群效应（herd effect）：可能会有很多的server在等待获得锁，而当锁释放时，所有的server都会收到通知，但是只能有一个server再次获得锁，因此会导致激烈的竞争
- 只支持互斥锁

3.5.5. 无群体效应的互斥锁 / 伸缩锁

Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```

client只会在编号比自己小的前一个znode被删除时收到通知，而后client会再次检查自己是否获得锁（client前一个znode被删除，不代表前面所有的znode都被删除，可能有编号更小的znode获得锁）

此处exist返回false有两种可能性：

- 比当前cilent编号小的client得到了锁然后释放了锁
- 比当前client编号小的client崩溃，然后Zookeeper自动删除了 lock-编号 文件（注意是ephemeral）

之所以需要多次列出C的子结点，而不是只寻找当前client编号-1的client，是因为当前client编号-1的client可能会崩溃退出。

伸缩锁和go routine之中的互斥锁有些不同。互斥锁可以用于保证一个代码段是原子执行的。但是伸缩锁如果当前的server故障，可以释放掉锁。例如在更新某些共享数据结构时，若server故障，且更新未完成，则会自动释放锁，但是后面的server再次更新共享数据结构时，这些数据是未完成更新的，因此可能需要一些机制来识别之前的server更新到了什么位置。这一类锁可以用在保护不是无关紧要的东西，例如MapReduce之中的task，可以用于说明当前的task正在由某个worker执行，但当这个worker崩溃之后，根据MapReduce的设计，会重发task，因此task执行到中途崩溃不会对整体产生大影响。

3.5.6. 读写锁

Write Lock

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock

```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

解锁方式与前文一致。此处读操作会发生herd effect。但是读操作本身就允许并发进行，因此herd effect反而是我们需要的

3.5.7. Double barrier （双重屏障）

barrier有一个阈值，规定了当到达第一个barrier的process数目达到这个阈值之后，才能执行下面的代码。当所有进程都到达了第二个barrier之后，才能离开并执行下一步代码。可以创建znode表示进入第一个barrier，删除znode表示离开第二个barrier。并且配合watches机制即可

4. 实现细节

Zookeeper实现高可用性的方式：将Zookeeper的数据复制在每一台机器上。同时Zookeeper假设某台server在崩溃之后会恢复。每一个server都有一个Zookeeper database，并且完全一致，包含了data tree等信息。Zookeeper把它放在内存之中，默认每个znode存放的数据为$1MB$。对于读写操作，Zookeeper的行为如下：

- 对于读操作，从本地data tree中读取相应数据并且返回
- 对于写/更新操作，先由request processor进行操作执行前的准备，而后由atmoic broadcast负责进行写操作的同步，最后再把写操作**提交**（commit）到整个Zookeeper集群的Zookeeper数据库中。Zookeeper为了数据可恢复，会先把数据写入磁盘，而后再应用于内存中的Zookeeper 数据库。

Zookeeper实现一个重现日志（replay log），保存已经提交了的写操作，同时Zookeeper也会定期记录数据库的快照。replay log 中每一条log entry都有一个编号，这个编号是leader给出的，称为zxid（Zookeeper transaction id）

当client进行写操作时，可能会选择任意一台server，而后这台server需要将这个请求推送给leader进行处理。除了leader之外的server均为follower。followers会收到来自leader关于Zookeeper状态改变的消息。

4.1. Request Processor

尽管有时，有些server可能执行了比其他server更多的事务，但是由于这些事务是等幂的，因此仍然能保证server的Zookeeper replica是满足一致性的。当leader收到来自client 的write request时，leader需要计算这个request完成之后，Zookeeper的未来状态。若client 的write request中的版本号和znode的未来版本号一致，则会生成GetDataTXN（TXN == 事务），否则会生成errorTXN。之所以需要计算未来状态，是因为目前可能存在未完成的事务。

4.2. Atomic Broadcast

Zookeeper通过ZAB（一种原子性广播协议/共识算法）来决定server是否接受proposal。ZAB采用了多数投票算法，因此只有在集群server数目为$2n+1$，且当前仍然available的server数目至少为$n+1$的情况下，才能继续运行。若集群server数目为$2n$，则必须保证有$n+1$台正常工作。与Raft很相似

Zookeeper为了加大request执行的吞吐量，使用了流水线架构（类似于计组中的CPU流水线？）。由于新的状态的改变与旧状态相关，因此Zookeeper保证：

- leader广播状态变化的顺序与请求的顺序一致。这条通过TCP协议实现的。TCP会保证数据传输的顺序。
- 在新的leader发送第一条 **状态变化提议（proposal）** 之前，旧leader会将自己所有的状态变化发送给新的leader

通常ZAB只会给server发送一次消息，但是由于ZAB没有持久化保存消息的标志符，因此在恢复阶段，可能会重复发送。但是这些消息（应该也就是状态变化提议，或是TXN）是等幂的，因此重复发送并不会对执行造成实际的影响，只需要保证消息发送是按照请求收到的顺序进行即可。同时，Zookeeper需要ZAB重发在最后一次快照之后产生的所有消息。

4.3. Replicated Database

如果一台server崩溃了，则其in-memory Zookeeper数据库完全丢失。为了恢复Zookeeper的状态，如果从最开始的消息（状态变化提议）开始重新执行，时间会很长。因此Zookeeper定期拍摄快照。Zookeeper拍摄快照时不会对数据上锁，因此Zookeeper创建的快照被称为**fuzzy snapshot**（混乱快照）。在拍摄快照开始之后，Zookeeper仍有可能发生状态的改变，因此snapshot可能还会包含一部分在开始拍摄快照之后的状态改变。但是由于**状态改变**是等幂的，因此只需要让ZAB重新传送在开始拍摄快照之后所有的消息即可

4.4. Client-Server Interaction

当server处理一个写请求时，同时会负责发送和清除与这个写请求相关的watch。server在处理写请求时，不会处理其他请求（包括写与读），如此则可以保证执行的顺序性。

当server处理读请求时，还会在读请求上附加一个zxid。zxid对应server的log中最后一个事务（也就是log entry的index）。server在本地进行读取操作，并且不需要运行一致性协议，因此可以很好地提高读性能，但是无法保证读取操作的先后次序（i.e. 读取的znode已经被更新了，但是读操作仍然读到了过时的数据，这是由于client可以直接读取server，而不是与leader通信导致的！尽管读操作不保证可线性化，但是吞吐量增大了！）。为了保证这个问题，需要使用sync（sync会保证，使得在sync被调用之前的所有write操作都被执行完毕，仅限于client连接的server）。sync操作并不需要被广播，只需要将其插在server和leader沟通的队列的尾部即可，则当server发现sync时，就知道sync之前的写操作已经完成。上述操作必须要保证server所沟通的leader仍然是leader。当这个队列之中存在已提交但未完成的事务时，server会认同当前leader，若队列中不存在事务，则leader会添加进去一个空事务，然后再把sync插到末尾。

server在回复client的请求时，会附带上zxid（当前log entry中最大的zxid）。如果在两次heartbeat中间没有任何操作产生，heartbeat也会附带上zxid。如果client最新的zxid比server的大，则server不会与client建立session，直到server跟上了最新的zxid。client也会去寻找zxid至少与自己一样的server，这是持久性（durability）的重要保障

在一次会话之中，当server超过timeout 没有收到来自client的请求时，就会认为client故障。若client发送请求很频繁，则自然会满足上面的要求，反之则需要client定期发送heartbeat信息。同时，若client无法向server发送heartbeat或者request，则会向别的server重建会话。

5. 相关论文

- Amazon Simple Queue Service 3
- election selection 25
- Chubby 强同步锁服务 6
- Zab 原子性广播协议 24
- 同步可线性化 15