数据库-系统可用性

Posted on 2021-12-01
Words count in article: 4k | Reading time ≈ 15

数据库-系统可用性

系统的可用性是指：一个系统处在正常工作状态的时间比例

比如说，一台机器的 MTTR(mean time to failure)是81.5年，MTTF(mean time to repair)是1h，那么：

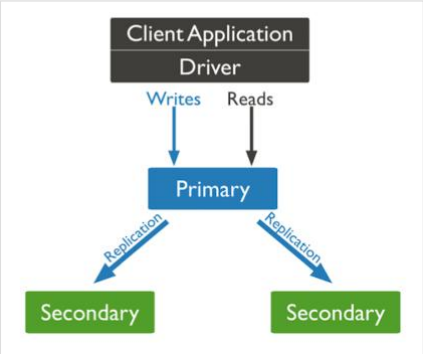
- MTTF in hours = 81.5×365×24 = 713940
- Inherent availability(Ai) = 713940/(713940+1) = 99.99986%

怎么才能让服务器变得高可用呢?

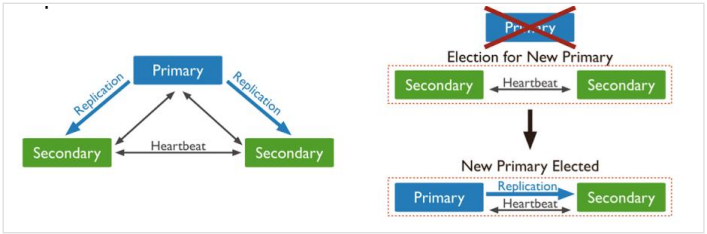
我们可以使用 冗余节点(Redundancy) 和数据复制(Replication)的技术，当某节点发生故障时，切换到冗余节点。

MongoDB的容错方案

MongoDB诞生的时候，就决定部署在云上。那么在部署的时候我们就不能只放在一台机器上，需要做备份，如下图：



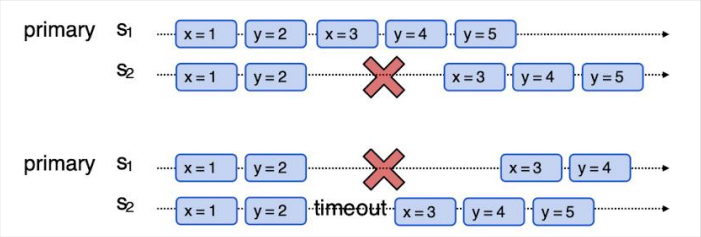
Primary是主机，有两个Secondary是主机的备份。系统要做的就是在主机更新时通知备份的及其来让这三个机器的数据保持一致。为了实现同步，我们需要了解**心跳**的含义，简单来说，心跳就是主机过一段时间(可能是几十毫秒).告诉对方我还在，还有心跳。一旦有Primary机器出故障了，就会影响心跳。这时候Secondary机器会感知到，其中一台机器变成了Primary, 如下图所示：



那么有一个问题：为什么是一主两备，而不是一主一备呢？第三台机器存在的意义是什么

其实，原来的高可用机制就是单机热备：

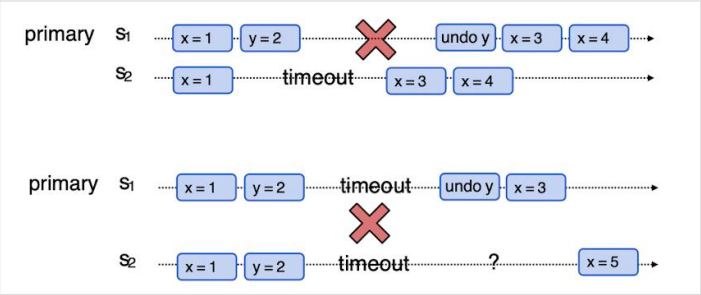
它的期望效果是：只要S1和S2 其中一个在线，系统即可用，如下图：



第一种情况，当S2出问题的时候，Primary S1还可以正常运行，等S2重启之后，再去做更新。

第二种情况，当Primary S1发生故障的时候，S2会宣布自己没有心跳了(Timeout)，然后S2转换身份变成Primary，当S1恢复之后，再把S2的信息同步给S1

这种设计，看起来只要两个主机不同时发生故障，就会一直运行下去，但事实上并非如此。如下图：



比如，最后一种情况，S1只是卡了一下，它并不认为自己是宕机了或者故障了，但此时S2已经收不到S1的心跳了，因此它决定变成Primary，但此时S1却回过神来了。这种情况下到底谁是Primary呢？这便出现了**争议**。有的用户会认为S1是Primary，而有的用户认为S2才是Primary。

解决办法只能是再增加一个S3，当两个人出现争议的时候，可以通过**投票**的方式选出新的Primary。

Raft算法

raft是一个共识算法（consensus algorithm），所谓共识，就是多个节点对某个事情达成一致的看法，即使是在部分节点故障、网络延时、网络分割的情况下

而在分布式系统中，共识算法更多用于提高系统的容错性，比如分布式存储中的复制集（replication），在 带着问题学习分布式系统之中心化复制集 一文中介绍了中心化复制集的相关知识。raft协议就是一种leader-based的共识算法，与之相应的是leaderless的共识算法。

简单概括算法就是：raft会先选举出leader，leader完全负责replicated log的管理。leader负责接受所有客户端更新请求，然后复制到follower节点，并在“安全”的时候执行这些请求。如果leader故障，followes会重新选举出新的leader。

Raft算法可以分为几个子问题：Leader election , Log replication

Leader election

在raft协议中，一个节点任意时刻都处于以下三个状态之一：

- Leader
- Follower
- Candidate

给出状态转移图能很直观的知道这三个状态的区别：

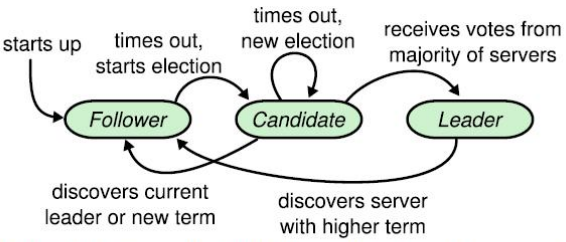


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

可以看出所有节点启动时都是follower状态；在一段时间内如果没有收到来自leader的心跳，从follower切换到candidate，发起选举；如果收到大多数节点的票（含自己的一票）则切换到leader状态；如果发现其他节点比自己更新，则主动切换到follower。

总之，系统中最多只有一个leader，如果在一段时间里发现没有leader，则大家通过选举-投票选出leader。leader会不停的给follower发心跳消息，表明自己的存活状态。如果leader故障，那么follower会转换成candidate，重新选出leader。

term

从上面可以看出，哪个节点做leader是大家投票选举出来的，每个leader工作一段时间，然后选出新的leader继续负责。这根民主社会的选举很像，每一届新的履职期称之为一届任期，在raft协议中，也是这样的，对应的术语叫**term**。

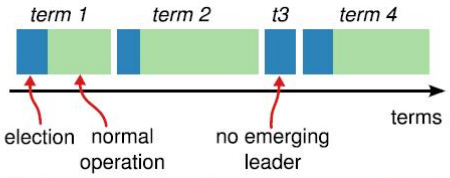


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

term（任期）以选举（election）开始，然后就是一段或长或短的稳定工作期（normal Operation）。从上图可以看到，任期是递增的，这就充当了逻辑时钟的作用；另外，term 3展示了一种情况，就是说没有选举出leader就结束了，然后会发起新的选举，后面会解释这种*split vote*的情况。

选举过程

上面已经说过，如果follower在*election timeout*内没有收到来自leader的心跳，（也许此时还没有选出leader，大家都在等；也许leader挂了；也许只是leader与该follower之间网络故障），则会主动发起选举。步骤如下：

- 增加节点本地的 *current term* ，切换到candidate状态
- 投自己一票
- 并行给其他节点发送 *RequestVote RPCs*
- 等待其他节点的回复

在这个过程中，根据来自其他节点的消息，可能出现三种结果

1. 收到majority的投票（含自己的一票），则赢得选举，成为leader

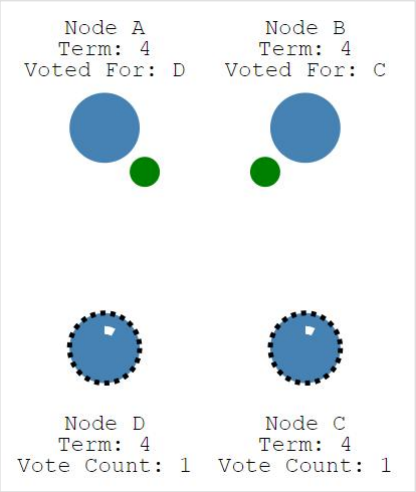
> 第一种情况，赢得了选举之后，新的leader会立刻给所有节点发消息，广而告之，避免其余节点触发新的选举。在这里，先回到投票者的视角，投票者如何决定是否给一个选举请求投票呢，有以下约束：
>
> - 在任一任期内，单个节点最多只能投一票

- 候选人知道的信息不能比自己的少（这一部分，后面介绍log replication和safety的时候会详细介绍）
- first-come-first-served 先来先得

1. 被告知别人已当选，那么自行切换到follower

第二种情况，比如有三个节点A B C。A B同时发起选举，而A的选举消息先到达C，C给A投了一票，当B的消息到达C时，已经不能满足上面提到的第一个约束，即C不会给B投票，而A和B显然都不会给对方投票。A胜出之后，会给B,C发心跳消息，节点B发现节点A的term不低于自己的term，知道有已经有Leader了，于是转换成follower。

1. 一段时间内没有收到majority投票，则保持candidate状态，重新发出选举



第三种情况如上图所示，总共有四个节点，Node C、Node D同时成为了candidate，进入了term 4，但Node A投了NodeD一票，NodeB投了Node C一票，这就出现了平票 split vote的情况。这个时候大家都在等啊等，直到超时后重新发起选举。如果出现平票的情况，那么就延长了系统不可用的时间（没有leader是不能处理客户端写请求的），因此raft引入了randomized election timeouts来尽量避免平票情况。同时，leader-based 共识算法中，节点的数目都是奇数个，尽量保证majority的出现。

Log replication

当有了leader，系统应该进入对外工作期了。客户端的一切请求来发送到leader，leader来调度这些并发请求的顺序，并且保证leader与followers状态的一致性。raft中的做法是，将这些请求以及执行顺序告知followers。leader和followers以相同的顺序来执行这些请求，保证状态一致。

Replicated state machines

共识算法的实现一般是基于复制状态机（Replicated state machines），何为复制状态机：

> If two identical, **deterministic** processes begin in the same state and get the same inputs in the same order, they will produce the same output and end in the same state.

简单来说：**相同的初识状态 + 相同的输入 = 相同的结束状态**。引文中有一个很重要的词 `deterministic`，就是说不同节点要以相同且确定性的函数来处理输入，而不要引入一些不确定的值，比如本地时间等。如何保证所有节点 `get the same inputs in the same order`，使用replicated log是一个很不错的注意，log具有持久化、保序的特点，是大多数分布式系统的基石。

因此，可以这么说，在raft中，leader将客户端请求（command）封装到一个个log entry，将这些log entries复制（replicate）到所有follower节点，然后大家按相同顺序应用log entry中的command，则状态肯定是一致的。

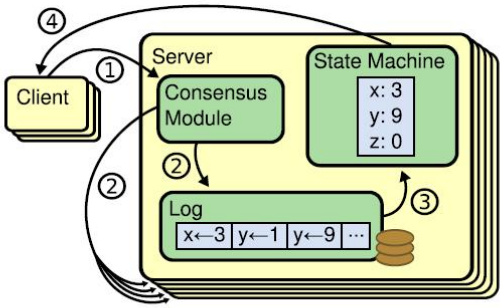


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

请求完整流程

当系统（leader）收到一个来自客户端的写请求，到返回给客户端，整个过程从leader的视角来看会经历以下步骤：

- leader append log entry
- leader issue AppendEntries RPC in parallel
- leader wait for majority response
- leader apply entry to state machine
- leader reply to client
- leader notify follower apply log

可以看到日志的提交过程有点类似两阶段提交(2PC)，不过与2PC的区别在于，leader只需要大多数（majority）节点的回复即可，这样只要超过一半节点处于工作状态则系统就是可用的。

那么日志在每个节点上是什么样子的呢

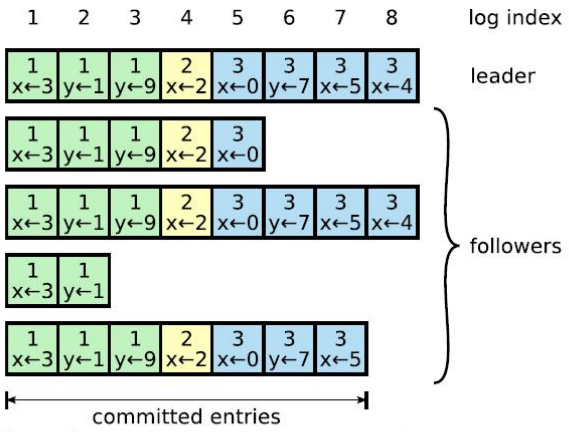


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

不难看到，logs由顺序编号的log entry组成，每个log entry除了包含command，还包含产生该log entry时的leader term。从上图可以看到，五个节点的日志并不完全一致，raft算法为了保证高可用，并不是强一致性，而是最终一致性，leader会不断尝试给follower发log entries，直到所有节点的log entries都相同。

在上面的流程中，leader只需要日志被复制到大多数节点即可向客户端返回，一旦向客户端返回成功消息，那么系统就必须保证log（其实是log所包含的command）**在任何异常的情况下都不会发生回滚**。这里有两个词：commit（committed），apply(applied)，前者是指日志被复制到了大多数节点后**日志的状态**；而后者则是节点将日志应用到状态机，真正影响到**节点状态**。

> The leader decides when it is safe to apply a log entry to the state machines; such an entry is called committed. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers

Safety

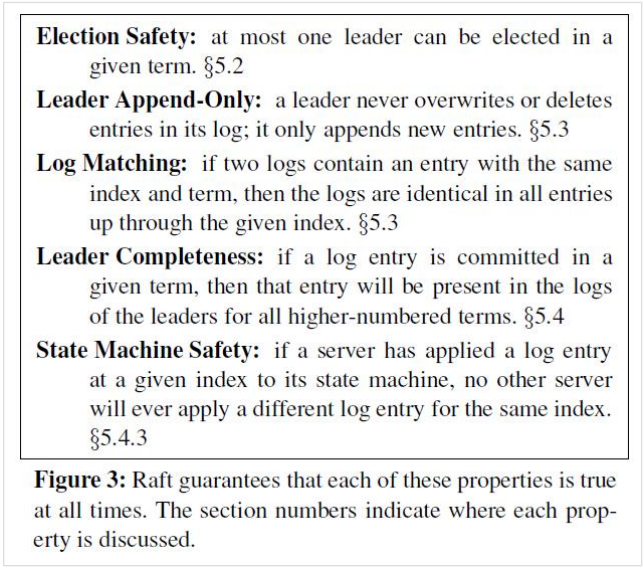
在上面提到只要日志被复制到majority节点，就能保证不会被回滚，即使在各种异常情况下，这根leader election提到的选举约束有关。在这一部分，主要讨论raft协议在各种各样的异常情况下如何工作的。

衡量一个分布式算法，有许多属性，如

- safety：nothing bad happens,
- liveness： something good eventually happens.

在任何系统模型下，都需要满足safety属性，即在任何情况下，系统都不能出现不可逆的错误，也不能向客户端返回错误的内容。比如，raft保证被复制到大多数节点的日志不会被回滚，那么就是safety属性。而raft最终会让所有节点状态一致，这属于liveness属性。

raft协议会保证以下属性



Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

Election safety

选举安全性，即任一任期内最多一个leader被选出。这一点非常重要，在一个复制集中任何时刻只能有一个leader。系统中同时有多余一个leader，被称之为脑裂（brain split），这是非常严重的问题，会导致数据的覆盖丢失。在raft中，两点保证了这个属性：

- 一个节点某一任期内最多只能投一票；
- 只有获得majority投票的节点才会成为leader。

因此，**某一任期内一定只有一个leader。**

log matching

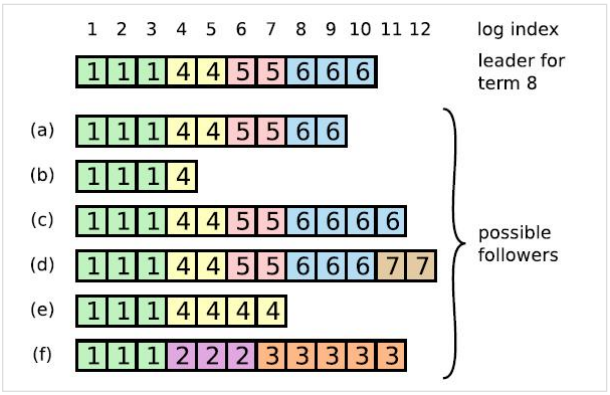
很有意思，log匹配特性， 就是说如果两个节点上的某个 `log entry` 的 `log index` 相同且term相同，那么在该index之前的所有log entry应该都是相同的。如何做到的？依赖于以下两点

- If two entries in different logs have the same index and term, then they store the same command.如果不同日志中的两个条目具有相同的索引和term，那么它们存储的是同一个命令。
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.如果不同日志中的两个条目具有相同的索引

和term，那么日志中的所有前面的条目都是相同的。

首先，leader在某一term的任一位置只会创建一个log entry，且log entry是append-only。其次，consistency check。leader在AppendEntries中包含最新log entry之前的一个log 的term和index，如果follower在对应的term index找不到日志，那么就会告知leader不一致。

在没有异常的情况下，log matching是很容易满足的，但如果出现了node crash，情况就会变得复杂。比如下图



注意：上图的a-f不是6个follower，而是某个follower可能存在的六个状态

leader、follower都可能crash，那么follower维护的日志与leader相比可能出现以下情况

- 比leader日志少，如上图中的ab
- 比leader日志多，如上图中的cd
- 某些位置比leader多，某些日志比leader少，如ef（多少是针对某一任期而言）

最终的结果是日志必须按照顺序记录的如下图：

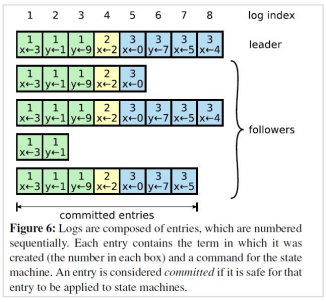


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

因此，当出现了leader与follower不一致的情况，leader强制follower复制自己的log

> To bring a follower’s log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower’s log after that point, and send the follower all of the leader’s entries after that point.

leader会维护一个 `nextIndex[]` 数组，记录了leader可以发送每一个follower的log index，初始化为eader最后一个log index加1， 前面也提到，leader选举成功之后会立即给所有follower发送AppendEntries RPC（不包含任何log entry， 也充当心跳消息）,那么流程总结为：

> s1 leader 初始化nextIndex[x]为 leader最后一个 log index + 1
> s2 AppendEntries里prevLogTerm prevLogIndex来自 logs[nextIndex[x] - 1]
> s3 如果follower判断prevLogIndex位置的log term不等于prevLogTerm，那么返回 False，否则返回True
> s4 leader收到follower的回复，如果返回值是False，则nextIndex[x] -= 1, 跳转到s2. 否则
> s5 同步nextIndex[x]后的所有log entries

leader completeness vs elcetion restriction

leader完整性：如果一个log entry在某个任期被提交（committed），那么这条日志一定会出现在所有更高term的leader的日志里面。这个跟leader election、log replication都有关。

- 一个日志被复制到majority节点才算committed
- 一个节点得到majority的投票才能成为leader，而节点A给节点B投票的其中一个前提是，B的日志不能比A的日志旧。下面的引文指处了如何判断日志的新旧

> voter denies its vote if its own log is more up-to-date than that of the candidate.
>
> If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

上面两点都提到了majority：commit majority and vote majority，根据Quorum，这两个majority一定是有重合的，因此被选举出的leader一定包含了最新的committed的日志。

raft与其他协议（Viewstamped Replication、mongodb）不同，raft始终保证leade包含最新的已提交的日志，因此leader不会从follower catchup日志，这也大大简化了系统的复杂度。

我们可以在 Raft官网看到可视化的Raft，十分有趣 **https://raft.github.io/**

----------本文结束，感谢您的阅读----------

数据管理系统

数据科学算法ch12-子模函数

计算机视觉-神经网络的训练