Raft 协议

摘要

Raft 是一个管理副本日志的共识算法。 它的作用跟 Paxos 一样，并且比 Paxos 更加高效，但是它的架构跟 Paxos 有很大的不同。这使得 raft 比 Paxos 更容易理解，并能更有效的应用在生产系统中。为了让算法更容易理解，raft 将关键组件拆分成了不同部分，比如 Leader 选举，日志副本，安全，和状态机。 结果使得用户可以更加容易的学习使用 raft 算法。Raft 同时提供了一种新的机制来改变集群成员，以此来保证安全性。

简介

共识算法使得一群节点可以作为一个节点组来一起合作对外提供服务，并且在内部允许部分节点的失效。 正因如此，他们在高可用大规模系统中扮演着重要的角色。 在过去的数十年间，Paxos 一直占据着绝对的主导地位。绝大多数的共识算法的实现都是基于 Paxos 或者受到 Paxos 算法的影响。此外在共识算法的学习中，Paxos 也是一门必修课。

不幸的是，Paxos 算法非常难理解，即使后续做了很多简化理解的尝试，但还是未能改变此种现状。 此外，Paxos 复杂的架构设计在实现时也困难重重。这也导致不管是 系统工程师还是共识算法理论的学习者都深受其困。

在跟 Paxos 算法缠斗许久之后，我们终于发现了一种新的共识算法，它不仅可以让系统的构建更加容易，而且也能让学习者更容易理解。我们采取了非比寻常的方式， 只为让共识算法变更容易理解并且更利于在系统架构中使用。所以我们问自己： 我们可以定义一种共识算法，它比 Paxos 更实用，更易于理解。同时，我们也希望此算法能够成为系统构建的核心算法，变成系统架构中最重要的基础设施。 所以不仅仅算法能够正常工作变的至关重要，对算法的观察跟监控也相当重要。

当然，我们做到了，它就是 Raft 共识算法。 在设计 Raft 算法的时候我们使用了很多特殊的技术手段来提高它的可理解性，包括分解（ raft 被分解成了 Leader 选举，日志副本，安全） 以及状态的减少。（相比于Paxos， raft 减少了不确定性的程度以及服务器之间不一致的方式）通过对 43 个学生理解情况的对比，我们得出了 Raft 算法比 Paxos 算法更容易理解的结论: 在学习两种算法之后，43 个人中有 33 人能够理解 Raft 算法，而能够理解 Paxos 算法的人却屈指可数。

Raft 在很多方面跟现存的共识算法都比较类似, 但是它也具有一些新的特性：

- 强壮的 Leader

  raft 使用了比其他共识算法更健壮的领导关系。 比如： 日志只能从 leader 到其他的服务器。 这种策略简化了日志副本的管理，并且让 raft 更加容易理解。

- Leader 选举

  raft 使用了随机时间选举 leader。任何的共识算法都会检测节点的心跳，raft 算法在此基础上添加了一点小小的机制，也就是随机时间，来解决冲突。

- 成员变化

raft 管理集群中成员变化的机制是通过一种称为联合共识的方式进行的。

我们相信 raft 是要比 Paxos 和其他共识算法更优的算法，无论是用于教育还是工程实现。它要更为简单和易于实现。 对于生产系统来说它的描述足够完整。 raft 有几个开源的实现，并且已经被一些公司所采用。它的安全性已经经过了实践证明，同时效率和性能也要优于一般的共识算法。

简介剩余的第二部分介绍了复制状态机，第三部分讨论了 Paxos 算法的优劣，第四部分描述可理解的一般性方法，第 5-8 部分介绍 raft 算法， 第 9 部分对 raft 算法进行评估， 最后第 10 部分描述相关工作。

复制状态机

共识算法通常出现在复制状态机的上下文中。在这种方法中, 服务器集合上的状态机模型会保证集群具有相同的状态跟副本，即使有一些服务宕机也可以继续运作。 复制状态机主要用于分布式状态中解决分区容错的问题。比如: 具有单 Leader 节点的大规模集群，如 GFS，HDFS 以及 RAMCloud，通常使用复制状态机来管理 leader 选举和存储配置信息 ，并且即使在 leader 奔溃之后也能保持。比如在 Zookeeper 和 Chubby 中都使用了复制状态机。

复制状态机经典的实现方式是采用日志副本，正如下图所示，每一个服务存储一个包含一系列命令的日志副本，状态机按次序以此执行。每一份日志在同样的次序下保存了相同的命令 ，所以每一个状态机处理相同的命令序列。 因为状态机是确定性的， 所以每一个计算机具有相同的状态和相同的输出序列。

保持副本日志的一致性是共识算法的工作。服务器上的共识模块从客户端接收命令并将其添加到他们的日志当中。 它与其他服务器上的共识算法进行交互， 确保每一份日志最终都会保证在相同的序列下保持相同的命令，即使一些服务失效了。 一旦命令被复制，每一个服务的复制状态机将在日志序列中处理他们，并且输出会返回给客户端。 结果，服务组成一个单一格式的，高可用的状态机。

经典的共识算法系统都具有以下特性：

- 安全：

  （绝对不会返回不正确的结果）在所有的非拜占庭容错条件下， 包括网络依赖，分区，丢包，复制或者重新排序等情况下都能确保安全。
- 高可用

  只要服务中主要的服务节点可以操作并且彼此通信，那么它就是完整可用的。 因此一个典型的5个节点的集群允许2个节点失效。服务恢复之后又可以重新加入集群。
- 保持日志一致性的时候不依赖时钟

  错误的始终和极端的消息延迟，在最坏的情况下，也只是会影响可用性，并不影响一致性。
- 命令的完成取决于集群中的绝大多数节点：

  一般来说，只要集群中绝大多数的节点作出了响应，则认为此次命令完成。个别表现较慢的服务器， 不会影响到整个系统的性能。

Paxos 算法

在过去的数十年间，Paxos算法几乎编程了共识算法的代名词。 它是最普遍被用来教学的协议。绝大多数的共识算法的实施都是以此为基础。 Paxos首次定义了一个可以就单一节点达成共识的协议。 比如，单节点日志副本。我们将此子集称为单一法令Paxos算法。Paxos将多个实例组合在一起来处理 一系列的决策，如日志(多-Paxos) Paxos确保安全和活跃度, 并且支持集群成员改变，它的正确性已经得到了证明，在使用中也很高效。

不幸的是，Paxos有两个大的缺点。 第一个缺点是Paxos很难理解。 对Paxos算法完整的解释非常不透明。鲜有人能够理解，需要花费很大的经历。 所以在下面的参考目录中有一些文章做响应的解释，但是这些解释也仅仅面向的是单一指令的Paxos，即使如此，也已经很难理解了。

为可理解而设计

在设计raft算法的时候有几个目标： 它必须是一个完整的并且实用的分布式系统基础设施。 所以它减少了很多需要开发者做的设计工作； 无论在任何条件下，它都必须 保证安全性和可用性；对普通使用者来说，还需要保证高效。 但这些并不是我们最重要的目标，我们最重要的目标是可理解性。它必须被绝大多数开发者所理解, 这样系。 者就可以在实际的系统中使用他们。 在设计raft算法的时候，有几个点需要关注，我们必须权衡并作出选择。 在我们评估可理解性的时候，我们会问： 解释起来有多难? 实现起来有多难？

我们意识到，这种分析具有高度的主观性： 尽管如此，我们还是使用了两种一般适用性的技术。第一个就是众所周知的问题分解。我们将问题拆解成了几个可以被解决，被解释， 被独立理解的部分。比如说，我们将raft协议拆分为了leader选举，日志复制，安全和成员管理几部分。

第二个就是做减法，我们通过减少状态的数量来减少状态空间， 使得系统更加连贯并消除一些可能的不确定性。尤其对于日志来说，不允许有遗漏， raft限制了日志彼此可能不一致的方式。 尽管在绝大多数情况下我们尽力去消除不确定性， 但在一些情况下，不确定性反而提高了可理解性。尤其是，随机数本身就具有不确定性，但是他们是非常好的方式去处理所有的选择问题。

我们在raft算法中，通过随机数的方式来进行leader选举。

Raft 共识算法

在第二部分的描述中，我们知道raft是一种管理副本日志的共识算法。 图二对共识算法做了一个概述。 图三列举了算法核心的特性。在后续的内容中，会逐个进行详细的讨论。 raft实现共识算法首先选举一个leader，然后赋予leader完整的能力去管理日志副本。 leader接受客户端的输入，然后将其复制到其他服务器，并且当其将日志复制到其他服务器 之后进行广播。通过Leader的方式可以简化副本日志的管理。 比如，leader可以在不经过跟其他节点共识的前提下决定将新的输入放入到日志的什么位置，并且数据的流向始终是从 leader到其他服务器。Leader也可以失效或者与其他服务器失连，这种情况下，将会选举一个新的Leader节点。

通过Leader的方式，raft算法将一个共识问题，拆分成了三个相关的子问题

- Leader 选举

  当系统中没有leader节点时，必须选举产生新的Leader

- 日志副本

  Leader必须从客户端接受日志的输入并将其复制到其他服务器。

- 安全

  最关键的raft的安全特性是图三所示的状态机安全性。

在展示了共识算法之后，下一部分讨论高可用以及不同时刻节点角色的转换。

5.1 Raft基础

一个raft集群包含几个服务器。5个节点是最常见的数量，这样就可以容忍系统最多出现两个节点失效。在一个集群中任意一个节点的角色只能是leader，flower或者candidate. 在一个运行正常的集群中，只有一个leader节点，其余节点都是follower角色。 followers是被动的: 他们不会主动发起请求，他们仅仅只是响应leader的请和candidate的请求。

Leader处理所有客户端的请求(如果一个客户端连接的是follower， follower会将请求重定向到leader节点)。 第三种角色candidate， 是用于选举的角色，也就是只有在选举的 时候才会出现candidate角色。 下面讨论各个角色之间的转换。

raft算法将时间划分为任意长度， 正如下图5所示的一样。 任期是连续的整数。每一次任期都是由选举开始，选举期间会有很多候选人尝试成为leader。 如果一个候选人赢得了选举， 它将在剩余的任期时间内成为leader节点。有些情况下，会因为选票被分裂，任期会因为没有leader而结束; 紧接着一个新的任期将开始。

在一个任期内，不同的服务可能会在不同的时间观察到事务， 一些情况下，一个服务或许无法观察到一次选举，甚至整个任期内都无法发现。 任期在raft中扮演着逻辑时钟的角色，他们 允许服务器去检测过时的信息，比如之前人气的leaders. 每一个服务都会存一个当前任期数，这个数字随着时间单调递增。 如果一个服务的当前任期比其他服务的当前任期要小，他们 会将自己的任期更新为更大的任期值。 如果一个候选节点或者leader节点发现自己的任期过期了，他会立即恢复到follower的状态。如果服务接受到了过期任期的请求，则会拒绝此次 请求。

Raft服务之间通过远程过程调用(RPCs)进行通信， 最基本的共识算法需要两种类型的RPCs. 请求投票的RPC调用，由候选节点发起， 以及数据写入的RPC，是由leader节点发起，follower 节点追加日志到wal。 如果发出去的RPC请求未获得响应， 服务会进行重试。同时为了提高性能，RPC请求都是并发执行的。

5.2 leader选举

raft采用心跳机制来追踪leader选举。 当服务启动时，都是followe的角色。 只要一个服务可以从leader或者candidate接受到有效的RPCs请求，就会保持follower的状态。 Leader会定期的 发送心跳检测请求到follower节点，以此来确保他们之间的正常通信。 如果一个follower节点在固定的时间内没有接受到leader或者candidate的请求， 他们会嘉定没有有效的leader，并 会变为一个选举节点进行新一轮的leader选举。

开始一次选举时， 一个follower节点首先会将自己的任期值自增，并将自己的状态转换为candidate。 然后为自己投票，并通过RequestVote RPCs的方式对集群中其他节点进行广播。 候选人会一直持续这种状态，直到以下任意一种情况发生: (a) 赢得选举， (b)其他的节点获得选举或者(c) 一段时间之后没有产生owner。 下面分别进行讨论。

如果候选人获得集群中绝大多数节点的投票，那么它将赢得选举。 在一个任期内，每一个服务至多给一个候选人进行投票。 这些策略保证了在一个任期内，至多有一个节点赢得选举。一旦一个节点 赢得选举，它就会编程leader节点。 紧接着会给所有其他节点发送心跳消息，以此来建立自己的权利同时阻止产生新的选举。

当等待投票时，一个candidate可能会接收到来自其他节点的AppendEntries RPC的请求，如果对方的任期值比当前candidate的任期值要大的话，candidate会恢复到follower状态。 如果对方的 任期值要比candidate的任期值小，则保持candidate的状态，并拒绝来自对方的请求。

还有一种可能的情况是，候选者既没赢也没输: 如果有很多的follower节点在同一时间编程候选人，投票可能会发生分裂，从而导致没有节点获得绝大多数选票。 当发生此种情况时，候选将超时并且会 发起新一轮的选举。 但是，如果不采取额外的措施，分割投票的情况可能会一直进行下去。

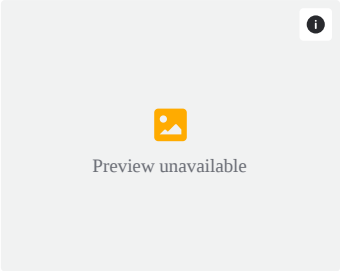
Raft使用随机选举超时时间来确保投票分割不会一直持续。 为了阻止投票分割，选举超时时间会在一个固定的时间区间(150-300ms)随机选择。 这会使得在大多数情况下，只有一台服务器会超时；它会赢得 选举并且在其他服务器超时之前发送心跳到其他节点。 同样的机理被用于选票分割。 每一个候选人重置它的随机选择超时时间，一直等到直到下一次选举开始。 这会减少下一次选举时投票分割的可能性。

选举是一个可理解性指导我们进行架构选择的一个鲜活的例子。 起初我们计划采用一个投票系统: 每一个候选人指定一个唯一的排名，用于在竞争候选人之间作出选择。 如果一个候选人发现了更高排名的候选人， 它会将自己的状态置为follower，以此来让更高排名的候选人赢得选举。 我们发现这种方式创建了一系列子问题，我们也对这种算法进行了一些调节，但是发现总有一些边角料的问题发生。 最终，我们得出结论， 还是随机重试的方式更为使用，也更容易理解。

5.3日志复制

一旦leader被选举出来，它就开始处理客户端的请求。 每一个客户端请求包含的命令将会被复制状态机所执行。 leader将命令作为一个新的输入添加到自己的日志中， 然后并行广播AppendEntries RPCs 到每一个其他的服务器，进行日志复制。当输入被安全复制之后， leader会将输入应用到它的状态机中并且给客户端返回结果。 如果follower节点宕机或者运行缓慢，或者网络丢包，leader会重试AppendEntries RPCs(即使已经对客户端进行了响应)

日志形式组织为图6的方式。 每一个日志输入存储一个带任期值的状态机命令。日志中的任期值用于检测日志间的一致性。 此外每一个日志还会带一个整型的索引值来标记它在日志中的位置。



leader决定了什么时间是安全的去应用日志到状态机中; 这样的一次输入叫提交(committed)。 Raft保证提交会持久化并且被所有的状态机执行。一旦leader创建的日志被复制到绝大多数节点， 日志就会被提交。 这也会提交之前所有的输入到日志中，包括先前节点创建的输入。 下面5.4一节会对leader变更之后的细节进行进一步的探讨, 同时还有提交时候安全的定义。 leader会一直追踪 其提交的最大索引。一旦follower发现一个新的日志输入被提交，他会按次序将其应用在其状态机中。

我们设计的Raft算法日志机制可以在不同的服务器之间保持日志的高度一致性。不仅简化了系统行为，而且更可预测，此外它也是保证安全的一个重要组件。 Raft保持了以下属性， 他们共同构成图三所示的日志 匹配属性：

- 如果在不同日志中的两次输入有相同的索引和任期，他们一定存储了相同的指令。
- 如果在不同日志中的两次输入有相同的索引和任期，先前所有的日志一定是相同的。

第一条是遵循以下规律，一个leader在给定的任期和索引下，至多创建一条输入，输入的日志不会改变其在日志中的位置。 第二条特性被简单的AppendEntries一致性所保证。 当发送一次AppendEntries RPC时候，leader的日志中包含先前的索引和任期，并在新的索引跟任期的前面。 如果follower没有在他的日志中发现相同的索引跟任期，则拒绝写入。 一致性检查扮演着至关重要的作用，初始化的日志满足日志匹配特性。综上，当AppendEntries返回成功时， leader就知道follower的日志跟自己的日志相同。

集群正常运作情况下， leader的日志跟follower日志是一致的， 所以AppendEntries一致性检查不会失败。 但是leader节点异常可能会使得日志不一致。 这种不一致可能会导致一系列的leader或follower奔溃。 图7表示leader和follower日志可能不同的几种可能的情况。 follower可能会缺失leader已经存在的日志， 也有可能会比leader有多余的日志， 或者两种情况都会出现。 这种日志缺失或者多余的情况可能会跨越几个 任期。

在Raft算法中，leader通过强制follower复制自己的日志来处理这种情况。 这也意味着在follower中的冲突日志会被leader强制覆盖。 5.4小节会说明当加上限制的时候，此种做法是安全的。

为了使follower的日志跟自己一致, leader必须找出两个日志达成共识的最新一条日志，并删除follower节点上在这条日志之后的所有日志，并将leader上最新的日志复制到follower节点。 这些所有的行为都在AppendEntries RPCs过程中发生的。 leader为follower保存下一个索引，也就是leader上面下一条日志的索引值将会发送到follower。 如果一个follower节点的 日志跟leader节点不一致。 AppendEntries一致性检查将会在下一次AppendEntries RPC的时候失败。 最终下一条索引值将会是leader跟follower匹配的那个值。此时，AppendEntries 就会成功， 这个过程中删除了follower中跟leader中不一致的日志，并将leader中新的日志应用到follower中。 一旦AppendEntries成功了，也就意味着leader跟follower的日志完全一致 了， 并且在剩余的任期时间内一直保持。

如果必要的话， raft协议可以被优化来减少拒绝AppendEntries RPCs的次数。 比如，当拒绝一次AppendEntries 请求的时， follower节点可以包含冲突日志的任期以及其为此任期存储的第一个索引。 利用这个信息， leader可以减nextIndex的值来绕过该任期内所有冲突的记录。每个冲突记录需要一个AppendEntries RPC，而不是每个记录需要RPC。实际上，我们怀疑这种优化的必要性，因为失效总会发生 并且不可能总是出现不一致的情况。

基于此机制， leader无需使用特殊的手段去保证日志的一致性。 只要它正常运作，日志就会自动保持一致。 leader绝对不会重写或者覆盖自身的日志(leader只能追加)。

日志复制机制展示了第二部分描述的理想的共识特性。 raft可以接收，复制并应用新的记录只要绝大多数的节点正常工作。正常使用中，日志会通过RPC复制到集群中的绝大多数节点。 单一节点的延迟不会影响整个集群的性能。

5.4 安全

前面部分描述了raft算法leader选举以及日志复制的部分。 但是截止目前描述的内容不足以证明此机制可以保证复制状态机在每个节点上以相同次序执行相同的指令。

此部分通过添加约束的方式来完成leader选举，这种约束确保了任何给定的任期内包含所有先前日志的提交。 给定选举约束之后，我们可以让提交规则更加明确。 最终我们呈现了 Leader Completeness Property 并且展示了leader如何修正复制状态机的行为。

5.4.1

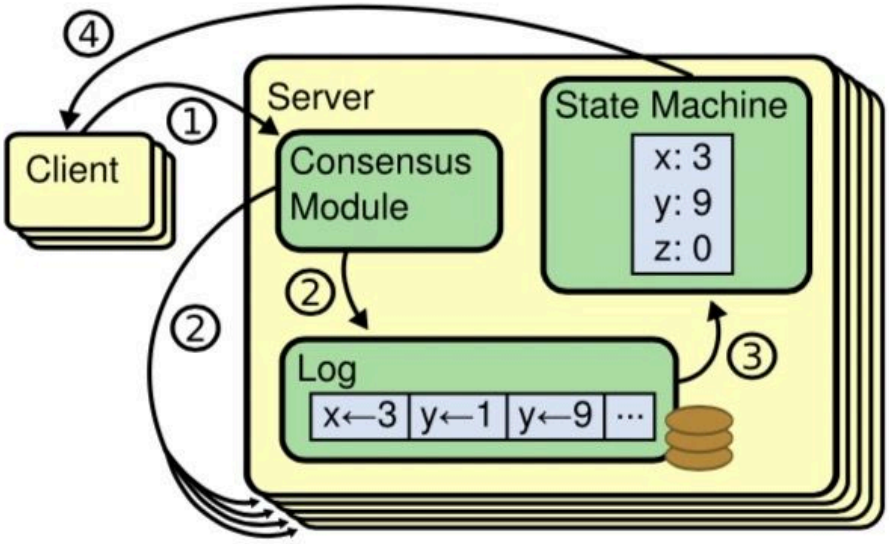
在所有的基于leader选举的共识算法中， leader一定最终存储所有的已提交的日志。在一些共识算法中，即使leader没有包涵所有已存在的日志，也可以被选举，比如Viewstamped Replication 。 这些算法使用额外的机制来确保日志的确实和传输。但是这种措施增加了额外的机制以及复杂性。 相反raft使用了一种简单的方法来保证日志， 先前的记录一定会存在于leader节点中。也就是说日志只能朝着一个方向流动，从leader->follower。

raft通过投票过程来限制没有完整日志的candidate获得选举。 candidate必须得到绝大多数的投票才能赢得选举， 这也就意味着绝大多数服务器上具有已经被提交的日志。 只要发现candidate的日志比其他的节点更老， 那么candidate就没法赢得选票。

Raft通过对比索引和任期的方式来确定，那个日志是更新的日志。 如果具有相同索引的日志有不同的任期， 那么具有更大任期值的日志更新。 如果日志有相同的任期， 那么日志更长的更新。

5.4.2

一旦日志被存储到绝大多数的服务器上， 那么leader就知道其在当前任期下被提交了。如果leader在提交之前发生奔溃，下一任leader将会尝试完成日志复制。



参考

- raft
- raft动画