Raft

23 May 2015

一直很知趣地没敢看过 Paxos。首先 Paxos 本身就难理解；其次，Paxos 作为一个学术上的算法，并没有为工程实现提供指导，最终 Chubby 和 Zookeeper 都并未实现真正的 Paxos，其正确性更多要依赖线上验证，而不能得到形式化证明的可靠性。近两年新出现的 Raft 协议，针对 Paxos 的两个问题做了很大的改善：其一，协议的设计从始至终面向可理解性，使学生容易建立对算法正确性的直观认识；其二，面向工程领域的实际需求，容易在满足正确性的前提下扩展协议，比如 Membership 管理、Snapshot 等。

怎样做到可理解性？Raft 的方法是将问题分解为 Leader 选举、日志同步、安全性约束 三个部分，使单独的部分容易理解；此外控制状态的数量，减少不确定性，乃至善用不确定性减少复杂性。值得一提的是，Raft 论文的叙述结构也十分清晰，非常值得一看。

Leader 选举

与 Paxos 的 P2P 方案不同，Raft 协议采用强 Leadership 方案，其集群中的每个节点处于三个状态之一：

- Leader：当前整个集群的 Leader，接受客户端的请求，通过心跳发送 AppendEntries 消息同步日志，并维持自己的 Leader 地位；
- Follower：被动接受来自 Leader 的日志同步；
- Candidate：尝试成为 Leader 的节点，它们会发送 RequestVote 消息参与选举；

Raft 使用一个单调递增的 term 值作为逻辑时钟，每个节点持久地保存 term 值，且每条 RPC 消息皆携带 term 值。每一次 term 值的增加，都意味着新一届 Leader 的选举。同一个 term 只能有唯一的 Leader，每个节点在每个 term 中只能投一票。

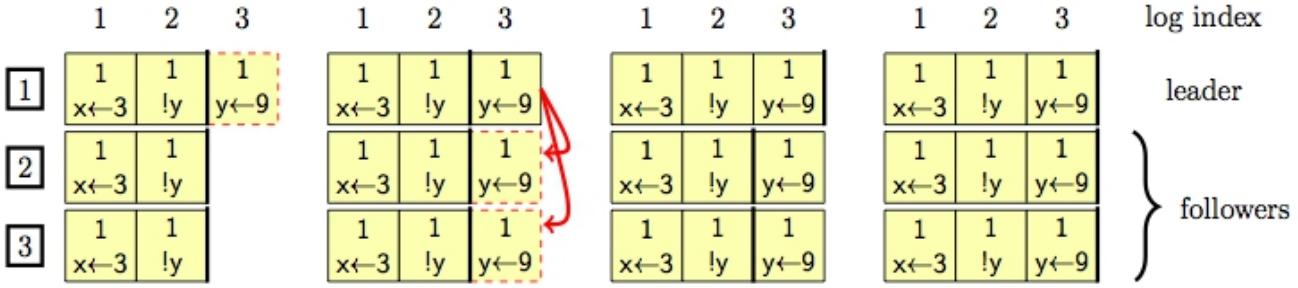
节点新启动时，会作为 Follower，如果超过 FollowerTimeout 仍没有收到来自 Leader 的消息，则把自身转换为 Candidate。

成为 Candidate 时，节点会提升自己的 term 值，启动 CandidateTimer，发出 RequestVote 消息为自己投票。这时如果成功收到多数投票，则成为 Leader；如果收到别人成为 Leader 的消息，则恢复为 Follower；如果 CandidateTimeout 时仍未选出 Leader，则重启选举。

如果是偶数个节点，很有可能出现两个节点同时拿到同样票数的情况，以至于无法选出 Leader。Raft 用了一个聪明的技巧来避免无限的等待：CandidateTimeout 之后，节点随机睡眠一段时间然后重启选举，这时先睡醒的节点会成为 Leader。

日志同步

Leader 向所有 Follower 发送 AppendEntries 请求，收到 多数成功 之后，提升自己的 CommitIndex 表示 Commit 这条日志，并应用该日志中的命令，向客户端返回成功。注意这时 Follower 节点只是保存了日志，但仍未 Commit 这条日志。而要等到下一条 AppendEntries 消息（心跳）时，再 Commit 该日志，并执行日志中的命令。



AppendEntries 请求中携带的参数有日志条目列表、term 值、prevLogIndex，Follower 在收到该请求时执行的流程大约是：

1. 如果 AppendEntries 的 term 值小于本地的当前 term，则返回失败；
2. 如果本地不存在 prevLogIndex 对应的日志条目，则返回失败；
3. 如果 prevLogIndex 匹配但 term 冲突，则删除该 prevLogIndex 之后的所有日志条目，以 AppendEntries 中的参数取代之；
4. 追加日志条目；
5. 将 prevLogIndex 对应的日志条目 Commit；

对于 Leader，如果 AppendEntries 失败，则回退 prevLogIndex，并无限重试。

安全约束

在以上 Leader 选举与日志同步规则之中，有两个安全性约束需要注意。

其一，在选举时，Candidate 有权拒绝来自别人的投票。对方的 term 值的必须大于等于自己，然后对方 log 的长度必须大于等于自己，否则会违反 Leader Completeness 约束 [1]，而导致丢日志。回想，写入时 Raft 只有保证多数节点写入成功时才返回成功，到选举时要求获得多数投票，可以保证能够选出数据最完整的节点。[2]

其二，新选举出的 Leader，需要检查是否存在来自上一个 term 的日志条目仍未 Commit，如果有，则 Commit 到新 term，这意味着日志条目的 term 值可能发生变化。试想这一个场景：Leader 成功执行 AppendEntry 到多数节点，并 Commit 到本地，然后 Leader 挂掉了，重新选举了一个新的 Leader 上台，这个新的 Leader 发现最近的日志条目还没有 Commit，则 Commit 到新 term，开始 AppendEntries 到 Follower，这时旧 Leader 又作为 Follower 复活了，在 AppendEntries 中会发现这两个节点的日志有冲突，旧 Leader 的日志条目的 term 值会被被新 Leader 替换。

持久性

除了日志之外，每个节点需要持久化 currentTerm 和 vote 值；因为 Leader 选举限制每个节点只能投一票，节点需要记住自己投了谁，不然重复投票会导致脑裂而选出多个 Leader。

协议扩展

这里只是基本的 Raft 协议，在工程实践中仍需要一些扩展，比如添加、删除节点，Snapshot 等。这里 adhoc 方法则很容易出错，而需要对协议做一点调整，好在 Raft 的官方在这方面也提供了比较丰富的资料和例子。

也有人说 Raft 协议容易理解是真的，但是实现正确仍不容易。在使用 goraft 这样不够成熟的库时，尤其需要注意。“可理解性” 是相对于 Paxos 而言，而作为工程问题的 Consensus 有它固有的复杂性。

[1]: 只要一个日志条目被 Commit，那么在更大 term 的 Leader 中一定都含有此条日志条目；

[2]: 这和 Quorum 方法中的 R + W > N 公式中：写入超半数，读取超半数，那么加起来大于 N，可以保证数据不丢；

References

- In Search of an Understandable Consensus Algorithm
- Consensus: Bridging Theory and Practice
- http://github.com/goraft/goraft (http://github.com/goraft/goraft)