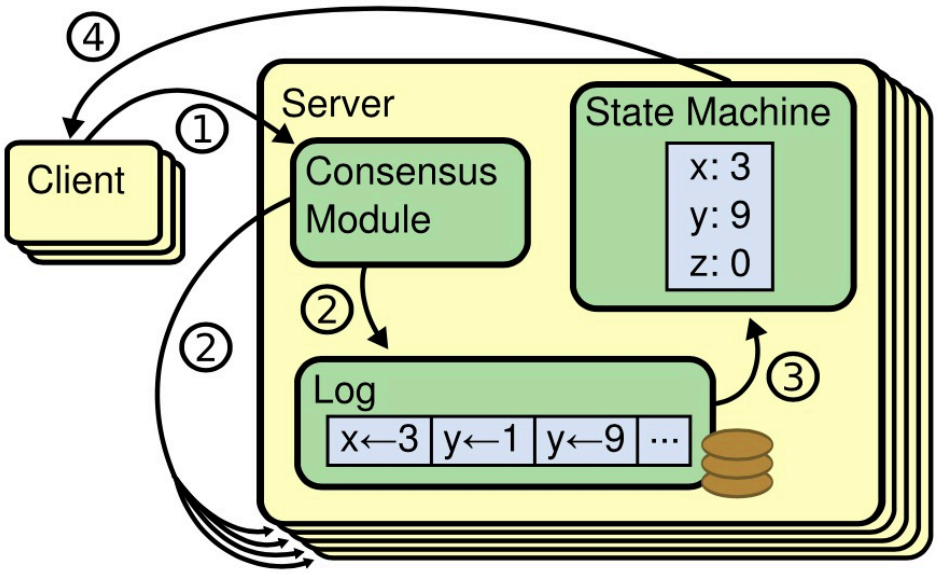
Raft 总结

名词解释

replicated state machines: 分布式系统中每个 server 都会将需要执行的 command 存在本地 log 中，本地的 state machine 按照本地的 log 中存储的执行顺序执行 command，来自 client 的读请求直接从 state machine 中获取数据，具体结构见下图：



leader: raft 集群中的 server，主要负责接收来自 client 的 log entry 并复制给所有 server，并告诉它们何时可以安全地写到 state machine。

follower: raft 集群中的 server，不会主动发起请求，只会响应来自 leader 和 candidate 的请求。

candidate: raft 集群中的 server，当收不到来自 leader 的 RPCs，follower 会将自身当前 term 加一并转为 candidate 状态，并选举自身成为 leader。

term: raft 会将时间分为持续任意长度的任期，用连续的数字来代表。每个 term 会以一次选举为开始，如果某个 candidate 赢得这个选举，它会作为 leader 服务在此次 term 剩余的时间里，某次 term 也会不存在 leader，下一 term 会马上开始。

log entry: 日志条目，记录在每个 server 存储的 log 中的一条记录，对应 state machine 需要执行的一条命令。

RPCs: Raft 用 remote procedure calls 来通信，主要分为两种，一种是 RequestVote RPCs，由 candidate 在选举期间发起，一种是 AppendEntries RPC，由 leader 发起主要用来复制 log entry 和提供心跳功能。

heartbeat: leader 周期性地发起不带 log entry 的 AppendEntries RPC 来确认自己的 leader 身份，如果一个 follower 在一段时间内没有收到来自 leader 的 heartbeat，就会发起 leader 选举。

Raft 算法通过选举一个 leader 来让集群达成共识，这个 leader 负责接收来自 client 的 log entry 并复制给所有 server，并告诉它们何时可以安全地写到 state machine。raft 将共识算法分解为三个相对独立的问题，并分别解决

1.leader election:当 leader 挂掉需要重新为分布式系统选举出新 leader

2.log replication: leader 需要接收来自 client 的请求落日志，并把该日志同步给分布式系统

中其他 follower
3.safety:如果 raft 系统中某个 server 将 log 中某个条目执行写入其 state machine，集群中其他 server 不能在同样位置执行不同命令

leader election (section 5.2)
Raft 用心跳机制来触发 leader 选举，当能按时收到来自 leader 或 candidate 的 RPCs 请求时，follower 会保持 follower 身份，否则在 election timeout 时间没收到 RPCs 请求后就回认为没有 leader，并开始一轮 leader 选举。
follower 会将自身的 term 加一并转为 candidate 状态来开始一轮选举。
当它赢得这轮选举，表明这个 candidate 获得了集群中 majority server 对于某一 term 的投票，对于某一 term，server 最多只会投一个 candidate，限制了一个 term 最多只会有一个 leader
或者别的 server 声明自己是 leader，在 candidate 发出 RequestVote RPCs 并等待投票的过程中，如果收到集群 leader 的消息，并且该 leader 的 term 大于等于 candidate 的 term，candidate 就会认为该 leader 合法并恢复 follower 状态
或者一段时间过后没有 server 赢得选举，当许多 follower 同时变成 candidate 发起 leader 选举，投票会比较分散无法选出一个 leader，当超时后会紧接着开始新一个 term 的选举，为了解决这种情况无限循环，raft 会让每个 server 的 election timeout 在一定范围内随机(e.g., 150ms-300ms)，这样的话在大多数情况下会让一个 server 先 timeout 然后发起选举在别的 server timeout 之前赢得选举成为 leader。

log replication (section 5.3)
选举出一个 leader 后就会开始接受来自 client 的请求。client 的请求包含需要执行的 command，leader 会把这个 command 写入自身的 log，也就是一条 entry，然后并发地发送给所有 follower 要求复制这条 entry，当这个 entry 被安全地复制之后，leader 会把这条 entry 包含的命令真正地执行到 state machine，这个 entry 就是 committed 的 entry，并告诉客户端执行的结果。所谓安全复制就是 leader 成功的复制了一个 entry 给集群中的大多数 follower。这个提交同时也会把 leader 的 log 中之前的 entry 提交了，包括前任 leader 的 log。
Raft 通过下面两个特性来确保 Raft 的日志匹配特性，从而确保数据安全性。
1.不同 log 中，在相同位置上的 command，如果 term 相同，那么这两个 command 相同。这个特性是因为某个 term 的 leader 只会创建一个指定 log 位置的 entry。
2.不同 log 中，在相同位置上的 command，如果 term 相同，那么这两个 log 中该 command 之前的 command 也相同。这个特性则是在 AppendEntries RPC 中通过加入简单的一致性检查来保证。在 AppendEntries RPC 请求中，leader 会加上在 leader log 中当前 entry 的前一个 entry 的位置和 term，follower 如果找不到对应的前一条 entry 就会拒绝复制这条 entry。这样当 leader 收到 follower 复制成功的消息时，就可以知道 leader 和这个 follower 的 log 完全一致。
通常情况下 leader 和 follower 的 log 完全一致，一致性检查不会失败，但是 leader crash 的时候就会出现 log 不一致的情况，follower 相比 leader 可能会缺 entry 也可能会多出一些 entry，Raft 会强制要求 follower 复制 leader 的 log 来保证一致性。为此 leader 需要找到其 log 和 follower 的 log 产生分歧的位置，并把之后的 entry 都复制给 follower。为此 leader 会为每个 follower 维护一个 nextIndex，这个 nextIndex 表示 leader 将会传给 follower 的下一个 entry，如果 AppendEntries RPC 失败，nextIndex 会减一并重试，当 AppendEntries RPC 成功，leader 会将剩下的 entry 都赋值给 follower。(看上去效率很低，但实际中这个情况比

较少见，没必要优化，见 section5.3)。

safety (section 5.4)

到此为止的 leader election 和 log replication 还不能保证所有 follower 会按相同顺序执行相同的 command。比如一个 leader 新加了某些 entry 并下线了，此时没有收到这些 entry 的 follower 成为 leader 后就会覆盖这些已经提交的 entry。为此 Raft 对与能够成为 leader 的 server 作了限制，既新一个 term 的 leader 的 log 必须包含上一个 term 所有提交了的 entry。Raft 在选举过程中就限制了不包含所有提交过的 entry 的 candidate 无法成为 leader，follower 会在收到 RequestVote RPC 时，判断如果 candidate 的 log 不如自身的新就会拒绝这个 candidate 的选举请求。这里新的含义是指 entry 的 term 越大 index 越大就越新。

如下所示，如果一个 leader 可以提交前一个 term 的 entry 的话，可能会导致已经存在于集群大多数 server 中的 log entry 被覆盖，因此 Raft 系统中，leader 不会去提交前任的 entry

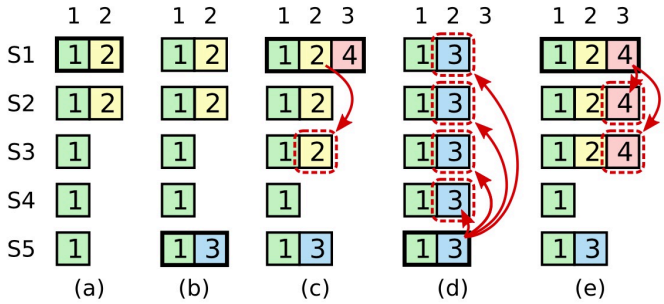


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.