# raft 协议论文阅读笔记

陈超　201922080611

August 30, 2021

# 一致性算法：Raft

Raft 保证如下属性总是成立的，

Election Safety 在任何给定的 term 中，最多只有一个 leader。

Leader Append-Only 一个 leader 不会覆盖写或者删除日志，只会追加新的日志项。

Log Matching 如果两个日志拥有同样的 index 和 term，那么对于从开头到这个 index 点，这两个日志都是完全相同的。

Leader Completemess 如果在一个给定的 term，一个日志项是提交的 (is committed)，那么，对于更高的 term 的 leader，该日志项也在该 leader 的日志上。

State Machine Safety 如果一个服务器在指定的 index 点上，将日志项运用在本机状态机上。那么，不可能存在一个服务器在这个 index 点上，运用其他日志项到它本机状态机上。

# 一致性算法：Raft

**committed**

A log entry(**current term**) is committed once the leader that created the entry has replicated it on majority of the servers。 This also commits all preceding entries in the leader's log, including entries created by previous leaders.

# 一致性算法：Raft

## Log Matching Property

- If two entries in different logs have the same index and term, then they store the same command.

# 一致性算法：Raft

## Log Matching Property

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

# 一致性算法：Raft

## Log Matching Property 证明 1.

The first property follows from the fact that a leader creates at most one entry within given log index in a given term, and log entries never change their position in the log. □

# 一致性算法：Raft

## Log Matching Property 证明 2(不考虑 leaders or followers fails 情况).

The second property is guaranteed by a **simple consistency check** performed by AppendEntries.When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an **induction** step:

- the initial empty state of the logs satisfies the Log Matching Property.
- the consistency check preserves the Log Matching Property whenever logs are extended.

As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries. □

# Raft 保证 State Machine Safety

## 安全选举的意义

Raft uses a simpler approach where is guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election.

**without the need to transfer those entries to the leader。**

This means that log entries only flow in one direction, form leaders to followers, and leaders never overwrite existing entries in their logs.

# Raft 保证 State Machine Safety

## 安全选举的方法

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries.
A candiate must contact a majority of the cluster in order to be elected, which means that every committed entry (has replicated it on majority of the servers。 ) must be present in **at least one of those servers.**
If the candidate's log is at least as up-to-date as any other log in that majority, then it will hold all the committed entries.
RequesVote RPC:

- the voter denies its vote if its own log is more up-to-date than that the candidate

比较 up-to-date 操作:

$$\begin{cases} \text{term}_A > \text{term}_B \\ \text{term}_A = \text{term}_B \;\&\&\; \text{index}_A > \text{index}_B \end{cases}$$

# Raft 保证 State Machine Safety

## 安全选举的方法

However, a leader cannot immediately conclude(推断出) that an entry from a previous term is **committed**, once(以前；曾经) it is stored on a majority of servers.

Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's **current term** are committed by counting replicas.

# Raft 日志复制 (Log replication)

## leader 为每个 followers 维护的 *nextIndex* 变量

The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower.

- When a leader first comes to power, it initializes all *nextIndex* values just after the last one in its log.
- If a follower's log is inconsistent with the leader's, the *AppendEntries consistency check* will fail in the next AppendEntries RPC.

Eventually *nextIndex* will reach a point where the leader and follower logs match. When this happens, AppendEtries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log. Once AppendEtries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

# Raft 日志复制 (Log replication)

## leader 为每个 followers 维护 *nextIndex* 变量的意义

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consitency check. A leader never overwrites or delete entries in its own log.

(Leader Append Only) 一个 leader 不会覆盖写或者删除日志，只会追加新的日志项。

# Raft AppendEntries RPC

Invoked by leader to *replicate log entries*, also used as *heartbeat*。

| **Arguments:** | |
|---|---|
| **term** | leader's term |
| **leaderId** | so follower can redict clients |
| **prevLogIndex** | index of log entry immediately preceding new ones |
| **prevLogTerm** | term of preLogIndex entry |
| **entries[]** | log entries to store(empty for heartbeat;may send more than one for efficiency) |
| **leaderCommit** | leader's commitIndex |

Receiver implementation:

1. Reply false if term $<$ currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one(same index but different term), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit $>$ commitIndex, set commitIndex $=$ min(leaderCommit, index of last new entry)

# Raft 正常流程

1. client sends PUT/GET "command" to k/v layer in leader.
2. leader add command to log.
3. leader sends AppendEntries(*AppendEntries consistency check*) RPCs to followers.
4. leader waits for replies from a bare majority(including iteself) entry is "committed" if a majority put it in their logs, committed means won't be forgetten even if failtures majority -> will be seen by the next leader's vote requests.
5. leader executes command, replies to client
6. leader "piggybacks" commit info in next AppendEntries.
7. followers execute entry once leader says it's committed.

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.
- S3 send an AppendEntries with entry 13 **prevLogIndex** = 12, **preLogTerm** = 5.

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.
- S3 send an AppendEntries with entry 13 **prevLogIndex** = 12, **preLogTerm** = 5.
- S2 replies false (AppendEntries step 2).

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.
- S3 send an AppendEntries with entry 13 **prevLogIndex** = 12, **preLogTerm** = 5.
- S2 replies false (AppendEntries step 2).
- S3 decrements nextIndex[S2] to 12.

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.
- S3 send an AppendEntries with entry 13 **prevLogIndex** = 12, **preLogTerm** = 5.
- S2 replies false (AppendEntries step 2).
- S3 decrements nextIndex[S2] to 12.
- S3 send AppendEntries w/ entries 12+13, **prevLogIndex** = 11, **preLogTerm** = 3.

# after a crash, Raft agreement about log

after a series of leader crashes,

| | 10 | 11 | 12 | 13 | <- log entry # |
|---|---|---|---|---|---|
| S1: | 3 | | | | |
| S2: | 3 | 3 | 4 | | |
| S3: | 3 | 3 | 5 | | |

Raft forces agreement by having followers adopt new leader's log example:

- S3 is chosen as new leader for term 6, set nextIndex[S2] = 13.
- S3 send an AppendEntries with entry 13 **prevLogIndex** = 12, **preLogTerm** = 5.
- S2 replies false (AppendEntries step 2).
- S3 decrements nextIndex[S2] to 12.
- S3 send AppendEntries w/ entries 12+13, **prevLogIndex** = 11, **preLogTerm** = 3.
- S2 deletes its entry 12 (AppendEntries step 3).

# how to roll back quickly

| | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| S1: | 4 5 5 | 4 4 4 | 4 |
| S2: | 4 6 6 6 | 4 6 6 6 | 4 6 6 6 |

S2 is leader for term 6, S1 comes back to life, S2 sends AppendEntries for last 6(AppendEntries has **prevLogTerm**=6).
rejection from S1 includes:

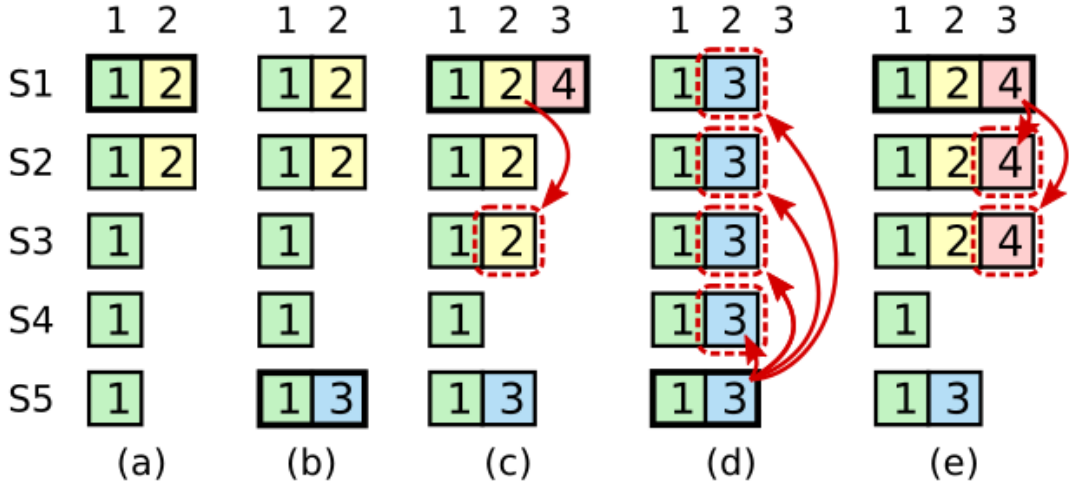
- XTerm: term in the conficting entry (if any).
- XIndex: index of first entry with that term (if any).
- XLen: log length.

Case 1 (leader doesn't have XTerm):
  nextIndex = XIndex
Case 2 (leader has XTerm):
  nextIndex = leader's last entry for XTerm
Case 3 (follower's log is too short):
  nextIndex = XLen

# Raft Committing entries from previous terms I

## Log Entry 什么时候是 Committed

不能简单的认为，一个 Log Entry 是 Committed，如果这个 Log Entry 复制到大多数的 followers 中。
因为，Raft 中的 Log 只能由 Leader 流向 follower，和 Raft 选举的过程。
存在如下情况：



# Raft Committing entries from previous terms II

- 情况一：
  - 在 c 中，S1 被选举成了 leader，现在它同步支持它的 followers。当它复制 term2 到 S3 时，S1 崩溃。
  - d 中，S5 被选举成了 leader，现在它同步支持它的 followers。注意，它会把上一次 term2 的数据进行覆盖，变成 term3 的数据。

矛盾，一个 Log 是 Committed，后续是不能再改变的（性质 State Machine Safety）。因此，不能简单认为，一个 Log 是 Committed，如果它被复制到大多数的 followers 中。

# Raft Committing entries from previous terms III

- 情况二:
  - 在 e 中，S1 被选举成了 leader，现在它同步支持它的 followers。当它复制 term4(当前的 term) 到大多数的 followers 中。
  - 如果，此时 S1 崩溃，而根据 Raft 选举的过程，S5 不能被选择上。

因此，如果当前 term中的 Log Entry 被复制到大多数 followers 中，因此这个 Log Entry 可是被认为是 Committed。根据 Log Matching Property，前面一个 term 的 Log Entry，因此，也是 Committed。

# Raft Committing entries from previous terms IV

## 结论

1. Raft never commits log entries from previous terms by counting replicas。
2. Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property.

## Leader 逻辑

if there exists an N such that N $>$ commitIndex, a majority of matchIndex$[i] \geq N$, and $log[N].term == currentTerm$, set commitIndex $=$ N。

# Raft 选举

## Lamport 向量时钟

//TODO

# Raft Snapshots (Raft 快照) I

FAQ: 什么是**Fuzzy checkpoint ?**

A fuzzy checkpoint is where the DBMS allows txns to continue the run while the system flushes dirty pages to disk.

模糊快照，是一种快照。当生成快照时，数据库可以允许事务继续执行，同时将脏页写入到磁盘上。

坏处:

- 恢复时，比较困难。需要结合日志配合，才能完成一致性的恢复。

好处:

- 对性能的影响小。

# Raft Snapshots (Raft 快照) II

Snapshotting is the simplest approach to compaction(压缩).In snapshoting, the entire current system state is written to a snapshot on stable storage, then the entire log up to that point is discarded.
Raft 快照流程:

- Every server takes snapshots independently,covering just the committed entries in its log.(每个服务器，独立地打自己的快照，只包含已经提交的日志!!!)
- Raft also includes a small amount of metadata in the snapshot:
  - the last included index, the index of the last entry in the log that the snapshot replaces(the last entry the state machine had applied).(每个服务器中，打快照时，最后被应用到状态服务器上的 index).
  - the last included term. the term of this entry.
- Once a server completes writing a snapshot, it may delete all log entries up through the last included index, as well as any prior snapshot.

# Raft Snapshots (Raft 快照) III

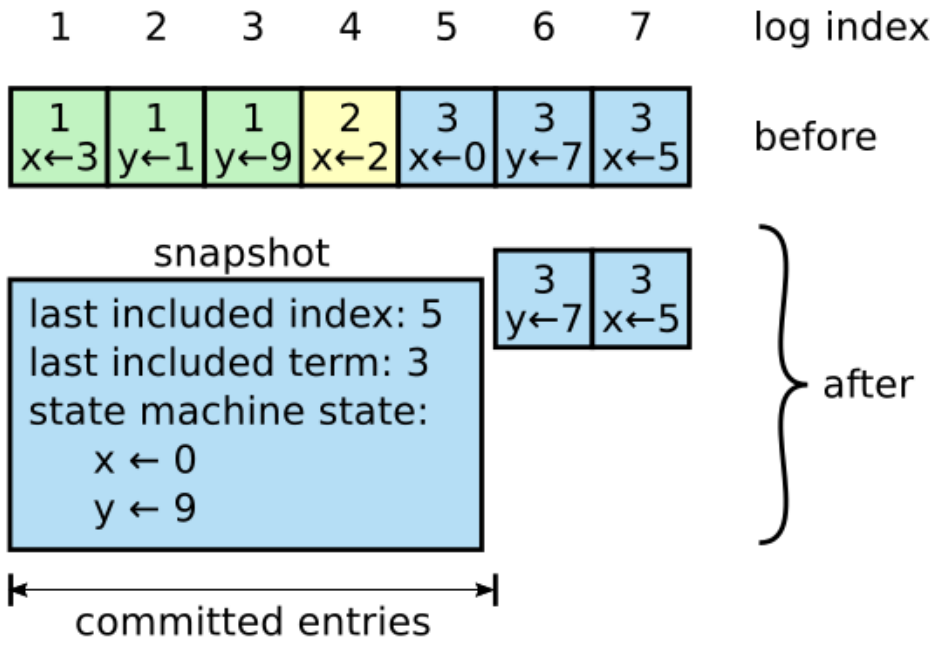



Figure: A server replaces the committed entries in its log with a new snapshot. the snaphsot's last included index and term serve to position the snapshot.

# Raft Snapshots (Raft 快照) IV

FAQ: 当 follower 服务器收到了 Leader 服务器的 installSnapotshot RPC 时，它应该如何办?

- 它应该决定收到的快照和已经存在的日志，如何办?
- the snapshot contain new information no already in the recipient's log (如果快照包含新的信息，但是在日志中不存在。)
  - the follower discards its entire log; it is all superseded by the snapshot and may possibly have uncommitted entries that conflict with the snapshot. (废弃整个日志，用新的快照代替)
- If the follower receives a snapshot that describes a prefix of its log ,(如果快照，只为日志的前缀，比如，被重传了)
  - log entries covered by the snapshot are deleted but entries following the snapshot are still valid and must be retained. (快照包含的日志部分，被删除。快照后的日志被保留!! )

# Raft Snapshots (Raft 快照) V

FAQ: 影响快照的两个问题?

## 第一个问题

First, servers must decide when to snapshot. 一种简单的方案，如果，日志到达了某个大小时，然后，开始进行快照。

## 第二个问题

The second performace issue is that writing a snapshot can take a significant amount of time, and we do not want this to delay normall operations.
一种解决方案是，使用 copy-on-write 技术。so that new updates can be accepted without impacting the snapshot being writen. (新的写入是可以的，但是不影响当前快照的写入!! ) Alternatively, the operating system's copy-on-write support (fork on Linux) can be used to create an in-memory snapshot of the entire state machine.