Consensus Bridging Theory And Practice

wu

June 3, 2024

1 Motivation

1.1 Achieving fault tolerance with replicated state machines

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, they are said to be **committed**. Each server's state machine processes committed commands in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

2 Basic Raft algorithm

2.1 Raft overview

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- Leader election: a new leader must be chosen when starting the cluster and when an existing leader fails
- Log replication: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own
- Safety: the key safety property for Raft is the State Machine Safety Property

Raft **SAFETY**:

- **Election Safty**: At most one leader can be elected in a given term.
- **Leader Append-Only**: A leader never overwrites or deletes entries in its log; it only appends new entries.
- **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.
- **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.
- **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.

2.2 Log replication

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. This also commits all preceding entries in the leader's log, including entries created by previous leaders. The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).

Raft maintains the following properties, which together constitute the 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.

The first

2.3 Safty

This section completes the Raft algorithm by adding a restriction on which servers may be elected leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms.

2.3.1 Election restriction

In any leader-based consensus algorithm, the leader **must** eventually store all of the committed log entries.

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries:

- A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry 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.

Raft determines which of two logs is more **up-to-date** by comparing the index and term of the last entries in the logs.

- 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.

2.3.2 Committing entries from previous terms

A leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers:

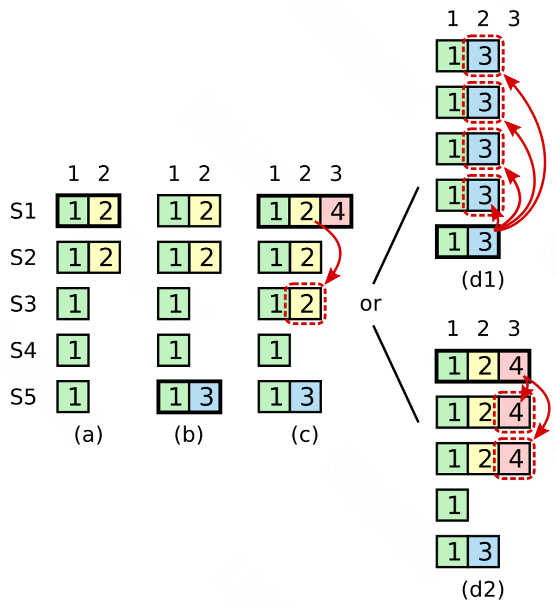

Figure 3.7: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry 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 (d1), 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 (d2), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

To eliminate problems like the one in Figure 2.3.2, Raft never commits log entries from previous terms 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.

3 Client Interaction

3.1 Processing read-only queries more efficiently

Fortunately, it is possible to bypass the Raft log for read-only queries and still preserve linearizability. To do so, the leader takes the following steps:

1. If the leader has not yet marked an entry from its current term committed, it waits until it has done so. The Leader Completeness Property

guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank no-op entry into the log at the start of its term. As soon as this no-op entry is committed, the leader's commit index will be at least as large as any other servers' during its term.

2.