Consensus Bridging Theory And Practice

wu

June 4, 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 property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log.

  If two entries have the same term, then they come from the same leader. If a log is replicated into a specific entry, then the index of that log is the same as the leader. Therefore the two entries have the same command as they come from the same entry from the same leader in the same term.
- The second property is guaranteed by a 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(prev log). 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, and 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.

A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

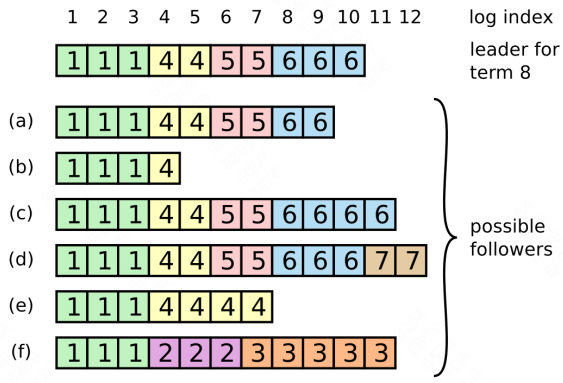

Figure 3.6: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

The leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs:

- 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 to the index 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. After a rejection, the leader decrements the follower's nextIndex and retries the AppendEntries RPC. Eventually the nextIndex will reach a point where the leader and follower logs match.

Until the leader has discovered where it and the follower's logs match, the leader can send AppendEntries with no entries (like heartbeats) to save

bandwidth. Then, once the matchIndex immediately precedes the nextIndex, the leader should begin to send the actual entries.

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs:

- when rejecting an AppendEntries request, the follower can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term;
- the leader can use a binary search approach to find the first entry where the follower's log differs from its own; this has better worst-case behavior.

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:

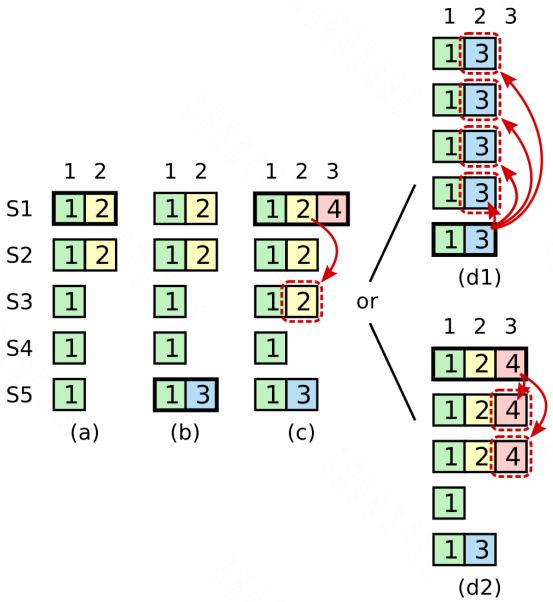

Figure 3.7: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry 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.

2.3.3 Safety argument

Assume Leader Completeness Property does not hold. Suppose the leader for term T $leader_T$ commits a log entry from its term, but that log entry is

not stored by the leader of some future term. Consider the smallest term $U > T$ whose leader $leader_U$ does not store the entry.

1. The committed entry must have been absent from $leader_U$'s log at the time of its election.

2. $leader_T$ replicated the entry on a majority of the cluster, and $leader_U$ received votes from a majority of the cluster. Thus at least one server both accepted the entry from $leader_T$ and voted for $leader_U$.

3. The voter must have accepted the committed entry from $leader_T$ **before** voting for $leader_U$; otherwise it would have rejected the AppendEntries request from $leader_T$

4. The voter still stored the entry when it voted for $leader_U$, since every intervening leader contained the entry, leaders never remove entries, and followers only remove entries if they conflict with the leader.

5. The voter granted its vote to $leader_U$, so $leader_U$'s log must have been as up-to-date as the voter's. This leaders to one of two contradictions.

6. First, if the voter and $leader_U$ shared the same last log term, then $leader_U$'s log must have been at least as long as the voter's, so its log contained every entry in the voter's log.

7. Otherwise, $leader_U$'s last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T. The earlier leader that created $leader_U$'s last log entry must have contained the committed entry in its log. Then by the Log Matching Property, $leader_U$'s log must also contain the committed entry, which is a contradiction.

8. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.

9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly.

Given the Leader Completeness Property, we can prove the State Machine Safety Property from, which states that 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:

- At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry, and the entry must be committed. Now consider the lowest term in which any server applies a given log index; the Leader Completeness Property guarantees that the leaders for all higher terms will store that same log entry, so servers that apply the index in later terms will apply the same value. Thus, the State Machine Safety Property holds.

Finally, Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

2.3.4 Followe rand candidate crashes

2.3.5 Persisted state and server restarts

- current term and vote: prevent the server from voting twice means vote for different candidates in the same term or replacing log entries from a newer leader with those from a deposed leaderterm.
- new log entries before they are committed: prevents committed entries from being lost or "uncommitted" when servers restart.
- *last applied* index

2.3.6 Timing and availability

2.3.7 Leadership transfer extention

To transfer leadership in Raft, the prior leader sends its log entries to the target server, then the target server runs an election without waiting for the election timeout to elaspe.

1. The prior leader stops accepting new client requests.
2. The prior leader fully updates the target server's log to match its own, using the normal log replication mechanism
3. The prior leader sends a *TimeoutNow* request to the target server.

Once the target server receives the TimeoutNow request, it is highly likely to start an election before any other server and become leader in the next term. Its next message to the prior leader will include its new term

number, causing the prior leader to step down. At this point, leadership transfer is complete.

It is also possible for the target server to fail; in this case, the cluster must resume client operations. If leadership transfer does not complete after about an election timeout, the prior leader aborts the transfer and resumes accepting client requests. If the prior leader was mistaken and the target server is actually operational, then at worst this mistake will result in an extra election, after which client operations will be restored.

3 Cluster membership changes

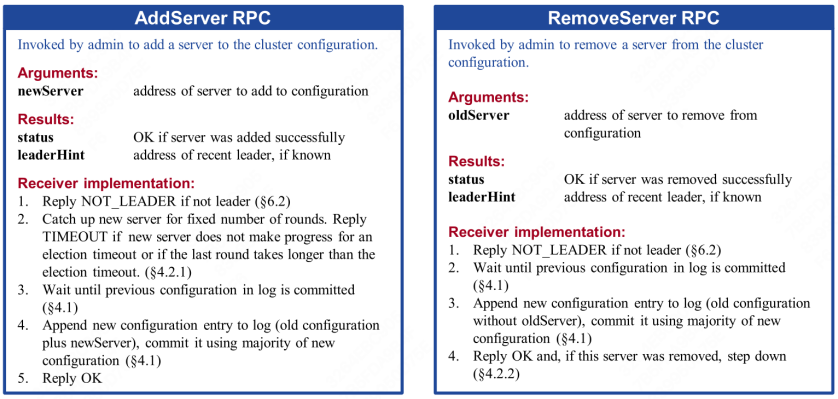

Figure 4.1: RPCs used to change cluster membership. The AddServer RPC is used to add a new server to the current configuration, and the RemoveServer RPC is used to remove a server from the current configuration. Section numbers such as §4.1 indicate where particular features are discussed. Section 4.4 discusses ways to use these RPCs in a complete system.

3.1 Safety

4 Client Interaction

4.1 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.