Distributed Systems Lab 11

Consensus: Log replication

During the previous lab, we introduced Raft and discussed its first part: leader election. This lab continues presenting Raft by discussing its second part: log replication.

This lab features no Small Assignment. Instead, the third Large Assignment will be presented. The lab also has no code examples.

Learning Section

Raft (cont.)

Let us assume that a distributed system, which implements Raft, has elected a leader, as discussed during the previous lab. All clients contacting the system are redirected to the leader, as this is the only process that is allowed to handle their requests.

Log

Each process in the system stores locally a **log**, which consists of **entries**. Each entry is assigned a position in the local log, named an **index** (entries are indexed by consecutive integers), and a **term** (the term when a leader received this entry from a client). Entries that are reliably replicated by the system are said to be **committed**. Each process tracks the index of the last committed entry in its log: a **commit index**.

At the beginning, when a process boots up for the first time, its local log is empty (or, as it is usually done in practice to simplify implementation, it contains a single dummy entry). The first actual entry is expected to have index 1.

Log replication

When a leader receives a request from a client to append a new entry to the distributed log, the leader firstly assigns to the entry the current term number. Then it appends the new entry to its log (but does not advance its commit index yet), and sends *AppendEntry* messages to other processes of the system (i.e., to the followers). The message contains this new entry along with the term number and an index of this entry in the leader's log, and the preceding entry from the leader's log (with its term number and index).

When a follower receives an *AppendEntry* message, it first verifies that the preceding entry contained in the message matches the last entry of its local log (i.e., the entries, their indexes, and terms are the same). If it does, the follower appends the new entry to its log (but does not advance its commit index yet), and replies to the leader.

If, in contrast, the preceding entry contained in the *AppendEntry* message does not match the last entry of the follower's log, the follower rejects the request to append the new entry and informs the

leader about an inconsistency. The leader, in reply to the rejection, sends to the follower another *AppendEntry*, but with an entry that precedes in the leader's log the previously sent entry (i.e., the second-to-last entry). This procedure is repeated until the most recent common position in the leader's and the follower's logs is found (this might not be the last entry of the follower's log, as the follower might have appended some entries that have not been committed eventually). Then, the leader overwrites nonmatching entries of the follower's log by sending to the follower *AppendEntry* messages with subsequent entries from its log starting from the first common entry found, until the logs are identical.

A leader advances its commit index, thereby committing the entry, when it receives acknowledgments from at least a majority of the processes (i.e., when some majority of the processes have replicated the leader's log up to this entry). The leader commits only entries that have their terms equal to the term of the leader (i.e., those that the leader has appended itself). The leader includes the value of its commit index in each *AppendEntry* message, and followers advance their commit indexes following it.

In practice, a leader may send in an *AppendEntry* message multiple log entries at once, as it is more efficient to send more data in one message. Therefore, an *AppendEntry* message is usually implemented to contain a list of entries to be appended to the log.

Revisiting leader election

A *Heartbeat* message broadcast periodically by the leader, as introduced during the previous lab, is actually implemented as an *AppendEntry* message. If there are no new entries to be appended, the message simply contains an empty list of entries. The idea to use an *AppendEntry* message not only for log replication itself but also for other features is exploited in Raft also for client sessions and cluster membership changes, as it simplifies implementation of these features.

Moreover, in the presence of the log, the voting procedure described during the previous lab has to be extended with one more rule: a process votes for a candidate if the candidate's log is at least as up-to-date as the log of the voter. A log is newer than another log if its ends with an entry with a newer term, or—when terms of the terminal entries are equal—it is longer (i.e., the index of the last log's entry is greater). This rule defines a *total order* on logs, so every two logs can be compared this way. To this end, a *RequestVote* message contains also the log index and the term of the cadidate's last log entry.

In a nutshell

In addition to the previous lab's summary:

Every process stores in its local stable storage:

- the current term (to reliably track terms),
- information about its vote (to prevent voting twice in a term),
- the log (i.e., all log entries).

Every process stores in its volatile storage:

- the commit index (after a restart, it is obtained from an *AppendEntry* message received from a leader).

A leader additionally stores in its volatile storage:

- processes' log indices (to track which entries of its log are replicated by which followers).

A *VoteRequest* message contains the candidate's:

- identifier and current term,
- index and term of the last log entry.

A process votes for a candidate if all of the following conditions hold:

- The candidate's term is not smaller than the voter's term.
- The voter has not already voted in this term.
- The candidate's log is at least as up-to-date as the voter's log.

An *AppendEntry* message contains:

- the leader's identifier and current term,
- the index and term of the preceding entry in the leader's log,
- a list of new entries to be appended to the log (may be empty for a heartbeat),
- the leader's current commit index.

When a follower receives an *AppendEntry*:

- It ignores the message if the leader's term is smaller than the follower's term.
- It compares the last entry of its log with the preceding entry of the leader's log:
  - if they do not match, it disregards the message and notifies the leader about the inconsistency.
  - if they match, it appends the new entries to its log, advances its commit index, and replies positively to the leader.

Safety

Let us explain why Raft guarantees safe log replication by proving the following invariants:

> *There is at most one leader in any given term.*

As discussed during the previous lab, in one term, every process has one vote, and a majority of votes is necessary for a process to assume its leadership for this term.

> *Leader never destroys entries in its log, only appends them.*

It stems directly from the algorithm. Only entries in followers' logs can be overwritten.

> *If two logs have an entry with the same term and index, then the logs are identical in all previous entries.*

This can be proved by induction. First, at the beginning all logs are empty (in practice, they contain the same dummy entry). Second, when a follower receives an *AppendEntry* message, it first verifies that the last entry in its log matches the preceding entry of the leader's log (which is supplied in the message), and appends the new entry only if the entries match. Therefore, after appending the new entry this invariant holds for both the leader and this follower. And because it held globally before this *AppendEntry* was sent by the leader (the inductive assumption), then every pair of processes has still consistent logs.

Leader completeness

> *If a log entry is committed in a given term, then this entry will be present in the logs of the leaders for all higher-numbered terms.*

Let us fix some log entry *e* and assume it is committed in some term *t*. Consider a future leader *l*. We will proceed with reasoning by contradiction, so we assume that this future leader *l* is the first leader

after term *t* that does not have *e* as committed in its log.

If the entry *e* was committed, it must have reached some majority of processes. Any future leader received a majority of votes, so these two majorities have at least one common process *p*. The entry *e* must have been accepted by *p* before the process voted for the leader, because otherwise *p* would reject messages from previous leaders (because of the term). Since leader *l* is the first one not to have *e* in its log, and because only leaders can force removal of entries inconsistent with their logs, entry *e* must have been present at *p* when it voted for *l*. Voting for the leader also means the new leader's log is at least as up-to-date as *p*'s log.

There are two cases now. First, the highest term of entries in the new leader's log was *t*. Then its log had to be at least as long as the log of *p*, and thus the leader's log contains *e* (because the leader for term *t* was the last one to contact both of them)—contradiction. Second, the new leader's log had entries with a higher term than *t*. Then these entries must have come from *AppendEntries* issued by some leader *l'*, which had *e* in its log (we assume that *l* is the first leader which does not have *e*). As shown above, if *l'* and *l* had in their logs a matching entry with an index higher than the index of *e* (*l* had *e* in the log, and entries with higher terms must have been further in the log), then all previous entries also had to be the same in both logs, and thus leader *l* has entry *e* in its log—contradiction.

Log replication

> *If a process commits a log entry at some log index, no other process will ever commit a different log entry at the same index.*

Consider the lowest term during which this entry is committed at the process (i.e., the process's commit index is increased to or beyond this entry). When the entry is committed by the process, it has already been committed by the leader for this term. As shown above, every leader for any future term must have this entry in its log. Therefore, processes committing entries in subsequent terms will commit the same entry at this index (if a log entry at some further index is the same, then all previous entries are the same).

This completes the proof of safety of log replication across all processes of the system. To replicate the state machine, the processes have to start with an initial state and execute the committed entries in the order they appear in their logs.

Additional Homework

We recommend reading the whole Raft paper. It discusses also implementation details such as log compaction, client interaction and membership changes in the cluster. Some of the features require introducing additional rules to the algorithm presented above.

If you have problems with understanding Raft, read this blog entry, watch this visualization, or play with a visualization here.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski.