Log Compaction in Raft

Diego Ongaro

April 27, 2013

The Raft log cannot grow without bound. If it did, Raft would have two potential problems:

- The log must be stored on Raft servers. Since each server is assumed to have a finite amount of storage, eventually this storage would be exhausted, and new entries could not be appended to the log.
- Upon starting or restarting, a Raft server must replay the log to ready its state machine. This would take an unbounded amount of time, which could result in availability problems should this server become leader before it finishes.

This document discusses the design choices in preventing the Raft log from growing without bound. As usual in Raft, the primary goals for Raft's log compaction mechanism are correctness, understandability, and efficiency. Safety may only depend on asynchronous assumptions, while liveness and availability may take timing into consideration. The best approach would work in a variety of environments, including:

- A high-performance datacenter network as in RAMCloud
- A slower, over-subscribed datacenter network
- A wide-area network
- A cluster with access to shared storage such as Amazon S3, where access from any server is assumed to be slow but reliable and files are assumed to be immutable after creation.

1 Log compaction approaches

The most general approach to log compaction is log cleaning as in LFS [1]. In log cleaning, the log is split into segments, and segments are the unit of cleaning. To clean a segment, its necessary data is copied to the head of the log, and then the segment may be discarded. Though this allows for the most efficient solutions, it is widely regarded as complex.

A special case of log cleaning is snapshotting. In snapshotting, the entire live state of the system is copied, then the entire log may be discarded. Snapshotting is attractive for its simplicity, but it is only efficient if the log may be snapshotted at very low utilizations.

A special case of snapshotting is a write-ahead log. In the write-ahead log approach, the place of record for data is the snapshot, and the snapshot is mutated according to log entries. As soon as a log entry is applied to the snapshot, it may be removed from the log. A write-ahead log requires less space than snapshotting but at the expense of mutating the snapshot during normal operation.

2 Raft server state

A Raft snapshot consists of the state machine state and the cluster configuration state required for membership changes. The cluster configuration state is very small (100s of bytes).

It is important to understand the expected size of a state machine in order to evaluate alternative solutions. We expect three different size ranges, each with legitimate use cases and distinct requirements:

1. **Small:** these state machines are order KBs or low MBs. Copying them, shipping them, and writing them to disk are all fast enough to be not affect availability.

   Small state machines may encompass uses of Raft for very basic coordination of small clusters. These state machines will not stress any solution to log compaction: naïve solutions are likely to work well, and this discussion is essentially moot.

2. **Medium:** these state machines are order MBs or low GBs. Copying them, shipping them, and writing them to disk are all slow enough to affect availability. One copy fits comfortably in the server's RAM but two may not.

   Medium state machines are likely to encompass most uses of Raft for coordination. These state machines present an interesting challenge.

3. **Large:** these state machines are order GBs or TBs. No copy fits comfortably in RAM; one copy fits comfortably on disk but two may not.

   Large state machines are likely to encompass uses of Raft for replicating data in large storage systems. These state machines may be best suited by a write-ahead log approach, since it requires the least space overhead (presumably space is a precious resource in a storage system).

The remainder of this document focuses on medium and large state machines. Any solution for those will also be sufficient for small state machines. It should also be easy to simplify these solutions for small state machines, though such optimization would be ill-advised if system requirements change and these small state machines grow larger.

3 Fundamental requirements

There are two fundamental requirements of any general solution to log compaction in Raft: servers must interleave snapshotting with normal processing of client requests, and servers must sometimes send snapshots to each other across the network. This section justifies each of these in turn.

First, for medium and large state machines, servers must interleave snapshotting with normal processing of client requests. This is because these state machines are too slow to create, ship, or persist, affecting system availability. For example, copying 10 GB takes about 1 second on today's processors, and persisting a 1 GB snapshot takes about 10 seconds on a magnetic disk. Without interleaving client requests with the creation and persistence of such a snapshot, Raft would be unavailable for too long (for most uses).

Second, servers must sometimes send snapshots to each other across the network. This is because Raft must be fully operational even if a minority of servers are unavailable. Raft can never guarantee that a log entry is "fully replicated" – persisted on every server in the cluster – since a minority of servers might not (ever) be available. Thus, to avoid unbounded logs, Raft servers will need to compact their logs when only a bare majority of the cluster is available. But the unavailable minority must be caught up in the case that they later become available; in this case, they will need to receive a snapshot over the network, since the log entries they're missing are potentially gone forever.

4 When to snapshot

There are three factors that determine how frequently to snapshot: space usage and log replay time push for snapshotting more frequently, while bandwidth overhead pulls for snapshotting less frequently. Different uses of Raft will weigh these factors differently. For example, a write-ahead log approach favors space and recovery time over bandwidth overhead: it snapshots continuously. This section discusses the trade-off and proposes a reasonable solution for medium-sized state machines (which live in RAM).

- **Space:** As RAM is much more expensive per GB than magnetic disk and even solid state disk, the storage space required for a very large log may be cheap and available in many environments. Using an order of magnitude more disk space than RAM may be acceptable.

- **Replay time:** The longer the log, the longer it will take a server to start or restart. This affects servers have been off-line: either after a cluster-wide outage or a minority of servers.

  In the case of a cluster-wide outage such as a datacenter power failure, we expect the outage to have already taken a long time by the time the Raft servers finish booting, so slow replay time may be tolerable. On the other hand, if Raft is used as a coordination service, it may be on the critical path for bringing remaining datacenter services back on-line. Re-loading

a server's complete RAM from disk would take about 1-10 minutes for today's servers; the expansion factor on the log would increase this time proportionally. Additional disks from which to read in parallel would help, but slower state machine operations would hurt.

In the case that an unavailable server in the minority starts up, it's likely been off-line for so long that replay time is unimportant.

- **Bandwidth overhead:** Any bandwidth spent snapshotting can not be used to service new client requests. It also becomes unavailable for co-located processes, consumes energy, and reduces the lifetime of disks. The term 'bandwidth' here is intentionally vague: it refers to memory, network, and/or disk bandwidth. Though it is hard to know how scarce bandwidth is, it is certain that slowing normal client requests is bad and less bandwidth waste is preferred.

The solution we propose for most uses of Raft is to snapshot at very low utilizations, provided that the required space is available. For example, if the log is snapshotted at 10% utilization, the space required is about 10x that of the state machine. At 10% utilization, for every 10 bytes of new client requests, we're willing to write 1 byte towards a snapshot.

Define u as the fraction of snapshot bytes we're willing to write for each byte of client requests. u would be a configuration setting between 0 (never snapshot) and ∞ (snapshot after every client request). A setting of about 0.1 is a reasonable default, using 10% of bandwidth towards snapshots.

A simple approach is to create a snapshot once the size of the log times u exceeds the size of the previous snapshot. The maximum disk space required with this approach is $(1 + \frac{1}{u})p + n$, where p is the size of the previous snapshot and n is the size of the next snapshot (assuming only one next snapshot must be stored). If the snapshot is bounded by the server's RAM capacity r, the disk space required is bounded by $(2 + \frac{1}{u})r$. For example, for $u = .1$ the disk space required will be no more than 12 times the server's RAM.

It is easy to see that this solution satisfies the property that for each bytes of client requests, no more than u bytes will be written towards a snapshots. In some cases, however, it will snapshot when the snapshot achieves no compression over the size of the log. For example, if the log is composed of commands that only create new objects, snapshotting is not profitable, yet this approach will snapshot anyway.

A more complex alternative is to snapshot once the resulting snapshot size would be less than u times the size of the log. However, it is a burden on the state machine to calculate the resulting snapshot size. For example, if the state machine creates a snapshot using a serialization library like Protocol Buffers, it is difficult to predict how large the serialized output will be without essentially creating the serialized form. Moreover, compressing the snapshot results in a nice space and bandwidth savings, but it is hard to predict how large the compressed output will be without actually doing the compression.

In conclusion, snapshotting once the size of the log times u exceeds the size of the previous snapshot is reasonable and easy; doing significantly better comes at a cost in complexity or performance.

Ryan also suggested considering current load in deciding when to snapshot. This seems like it could have benefits for some workloads.

There's also some debate about whether the state machine or the Raft module should decide to initiate the snapshot. I don't think it matters very much: the most informed decision would require information from both, and both need to be involved in the actual snapshot creation (only the state machine knows its state, while only Raft knows the configuration).

5 Possible approaches in Raft

To find the best log compaction solution, we partition the solution space on two axes, consider the local maxima within each partition, and compare the trade-offs between those. The first axis is whether the Raft leader initiates the snapshot or whether servers decide to snapshot independently. The second is whether the snapshot is stored inside the Raft log or whether the snapshot is stored separately (probably next to the log on each server's filesystem).

leader-initiated, stored in log	independently initiated, stored in log
leader-initiated, stored externally	independently initiated, stored externally

In Raft, only the leader can write to its log. The "independently initiated, stored in log" case is not compatible with this fundamental design choice; thus we will not consider it further. The following subsections discuss the "leader-initiated, stored in log", "leader-initiated, stored externally", and "independently initiated, stored externally" approaches in turn, then summarize the trade-offs.

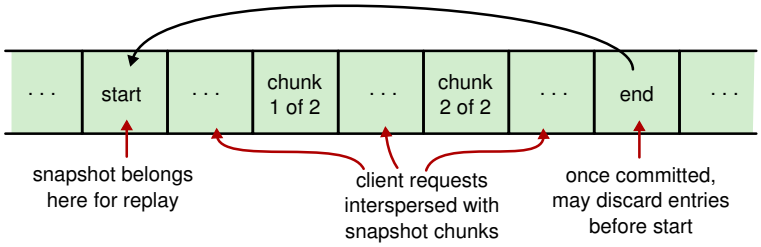
Each approach will address:

- How is the snapshot stored
- How is the log replayed
- What does an up-to-date follower do
- How is a slow follower caught up
- What happens upon server failures

Each of the approaches specifies when a server may discard a log prefix. Implementations may want to hold on to them longer than necessary to reduce the number of "slow" followers. In some cases, this would be more efficient.

5.1 Leader-initiated, stored in log

The following diagram shows a server's log using this approach. When the leader decides to snapshot, it first logs a special *start* entry. The snapshot will cover precisely the prefix of the log ending in the *start* entry. When the leader's state machine processes the *start* entry, it creates a snapshot of its state and places that in a queue to be appended to the log. (For now assume creating the snapshot is instantaneous; Section 6 will describe how this can be done in the background.) The leader then intersperses normal client requests with chunks of this snapshot that are sized small enough to fit inside log entries. Once it completes writing chunks, the leader appends the *end* entry. Upon committing the *end* entry, the leader may discard the prefix of its log prior to the *start* entry.



A sufficiently up-to-date follower will receive the snapshot log entries and add them to its log as usual. The *start* entry and snapshot chunks require no special processing. Upon learning that the *end* entry has been committed, a follower may discard the prefix of its log prior to the *start* entry. An unfortunate inefficiency of this approach is that these servers will receive the snapshot over the network, even though it contains information they already had.

For slow followers, the leader will arrive at a point where it needs to send an AppendEntries request to the follower with an entry that it has already discarded. (This is how a slow follower is defined, and applies to all three solutions.) In this case, the follower should discard its entire state. The leader will send it log entries beginning with the *start* entry. (We haven't worked out the simplest changes to AppendEntries to make this happen yet.) Once the server has received the complete snapshot (and knows it's committed), it can apply that snapshot and begin participating in the cluster as normal.

Log replay now requires a two-pass algorithm: in the first pass, a complete snapshot is identified (it has a matching *end* entry) and replayed. In the second pass, the entries following the *start* entry are replayed.

If the leader fails while snapshotting, it could leave a *start* entry and some large number of chunks in the log. In an unlucky execution, this could happen repeatedly, resulting in a fast-growing log until a leader is maintained long enough to commit a snapshot.

A slow follower presents one more complication: when it begins receiving the snapshot from the leader, it can appear to have a fairly up-to-date log (for leader election) before it has a complete snapshot. This could allow it to become leader

but not be able to advance its state machine beyond the initial state. There are two easy ways to solve this:

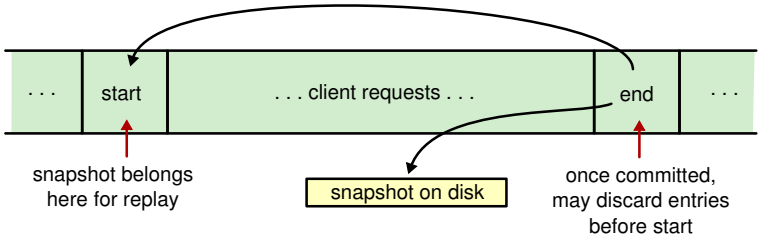
- Disallow servers from becoming candidates when they are missing a snapshot/log entry.
- Have servers report their last log entry as 0 in RequestVote when they are missing a snapshot/log entry.

The *start* entry should contain two additional pieces of information: the term of preceding log entry and the cluster configuration. The term of the preceding entry is needed for future AppendEntries requests that begin with the *start* entry. The configuration is needed to support membership changes.

Interspersing client requests with snapshot chunks is important so that snapshot chunks don't block client requests from being committed. A simple way to rate-limit the snapshot chunks is to append n, wait for them to be committed, append the next n, wait for them to be committed, etc.

5.2 Leader-initiated, stored externally

This approach is similar except that snapshots are not chunked into the log but rather stored externally, for example on a file on disk. The *end* entry contains a reference to the snapshot. The following figure illustrates this:



A follower will reject an *end* entry unless it has a matching snapshot on its disk. It's also a good idea to include the snapshot's checksum in the *end* entry. This serves as a built-in consistency check.

A sufficiently up-to-date follower should create its own snapshot upon seeing the *start* entry. This provides a network bandwidth savings: for these followers, the leader will not need to send them a snapshot over the network. Upon learning that the *end* entry has been committed, a follower may discard the prefix of its log prior to the *start* entry. If a leader gets an *end* entry rejected from an up-to-date follower, this indicates that either the follower is still creating its snapshot or that its checksum does not match. Followers should thus delay responding to the request if they are in the middle of creating a snapshot; the fastest course of action is to simply wait for the snapshot to complete.

For slow followers, the leader will arrive at a point where it needs to send an AppendEntries request to the follower with an entry that it has already discarded. In this case, the follower should discard its entire state. The leader

will send it log entries beginning with the *start* entry. (We haven't worked out the simplest changes to AppendEntries to make this happen yet.) Prior to sending the *end* entry, the leader should send the follower the snapshot file (using any file transfer mechanism). Once the server has received the complete snapshot (and knows it's committed), it can apply that snapshot and begin participating in the cluster as normal.

Log replay now requires a two-pass algorithm: in the first pass, a complete snapshot is identified (referred to from an *end* entry) and replayed. In the second pass, the entries following the *start* entry are replayed.

If the leader fails while snapshotting, it could leave an orphaned snapshot file on disk. Each server needs to store at most two snapshots, however: the last good snapshot, and the next one in progress.

As before, a slow follower that is missing a snapshot/log entry must still not become leader.

As before, the *start* entry or snapshot file should contain two additional pieces of information: the term of preceding log entry and the cluster configuration.

5.3 Independently initiated, stored externally

In this approach, each server may independently decide to snapshot any prefix of its log which is known to be committed; then it may discard that prefix from its log.

For slow followers, the leader will arrive at a point where it needs to send an AppendEntries request to the follower with an entry that it has already discarded. In this case, the leader will send it a snapshot (using any file transfer mechanism). The snapshot must contain:

- The index of the last entry covered by the snapshot
- The term of the last entry covered by the snapshot
- The cluster configuration
- The state machine state

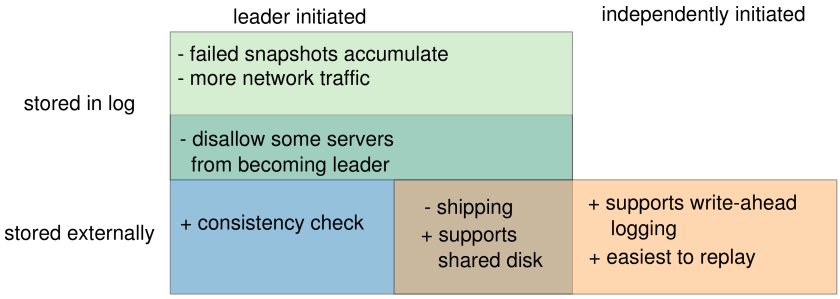
Upon receiving a snapshot from a leader, if the follower's log contains an entry at the snapshot's last index with a term that matches the snapshot's last term, then the follower already has all the information found in the snapshot and can simply ignore it. Otherwise, the follower discards its entire log, then saves the snapshot. This is analogous to truncating the end of the log upon receiving conflicting entries in AppendEntries requests.

Log replay now requires applying the latest snapshot, then applying the entries following the snapshot's last index.

If a server fails while snapshotting, it could leave a partial snapshot file on disk. An implementation could place a checksum in the snapshot, which wouldn't check for partially written files. Each server needs to store at most two snapshots: the last good snapshot, and the next one in progress.

Every server will have either an empty log or at least a valid snapshot, so every server can start elections.

5.4 Comparison of approaches



- Storing in the log has the problem that failed snapshots accumulate.
- Storing in the log sends snapshots over the network in the common case for up-to-date followers.
- Storing outside the log has the disadvantage that some mechanism for shipping files is needed.
- Storing outside the log has the advantage that, in shared-disk environments, much of the redundant snapshotting effort can be skipped.
- Referencing the snapshot from the log has the advantage of a built-in consistency check.
- Independently snapshotting has the advantage that it supports usage as a write-ahead log efficiently.
- Independently snapshotting may have slightly easier replay.
- Leader-initiated approaches have to prevent followers with un-replayable state from becoming leader.

When surveyed, the RAMCloud team unanimously agreed that the independently initiated approach seemed best.

6 Parallel snapshotting

We'd like the state machine should produce a snapshot in parallel with normal operation. This should be reasonably performant and not require much memory. A streaming interface to the snapshot storage is essential to keeping memory waste low.

The obvious solution is to build copy-on-write semantics into the state machine, so that a shallow copy can be snapshotted while the main copy continues

accepting requests. State machines do not otherwise require fine-grained concurrency control or copy-on-write semantics, however, so this has two disadvantages: first, the state machine must designed differently (or redesigned). Second, this slows down the normal-case state machine operations, now requiring some sort of reference counting mechanism and probably more memory allocations.

A cute approach (thanks to Amit) that works on small state machines is to bring up a second state machine for the purpose of snapshotting it. This state machine is fed the same log of operations as the primary state machine, only when a snapshot is started, it falls behind to produce the snapshot. This is a simple and general approach, but it requires twice the memory as normal for the second state machine.

The third approach is to rely on the underlying operating system's memory subsystem to provide copy-on-write semantics for the state machine's memory. The algorithm involves forking a child process which creates the snapshot, then exits. The parent process waits for the child to exit with a successful status to determine when the snapshot has been completed. This approach suffers in portability, but it's fairly general, easy, has low memory overhead, and we expect it to have low performance overhead in practice. However, the memory management overhead may be hard to predict and quantify. The parent and child process would need more communication if storing the snapshot in the log (a producer-consumer ring in shared memory should work).

7 Change log

- 2013-04-27: Initial draft

References

[1] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.