CSC 258: final project

Cole Miller

5 May 2019

Introduction

For my final project, I implemented the Raft consensus algorithm. My implementation is somewhat rudimentary and not written with performance in mind, but it has all the core features described in the Raft paper (configuration changes and log compaction are omitted), and appears to behave correctly. Below, I describe the details of my implementation and how I've attempted to test it.

Note that my code uses a couple of Linux-specific APIs (`prctl(2)`, `mremap(2)`) and won't compile on other systems.

Implementation

My implementation is written in C. The code describing the behavior of a Raft server is in the file `server.c`, whose entry point is the `raft_main` function. This file is intended to be compiled individually and linked against a "driver" program that calls `raft_main` in its own `main` routine. Thus `raft_main` is declared in a header file, `raft.h`, which should be included by the driver code. This header file also declares the functions `apply_command`, a callback that the server code calls when an entry in the Raft log has been replicated to a majority of servers in the cluster, and `resolve_read`, which is similarly used to handle read-only requests. The intention is that the driver code against which the server implementation is linked should define these callbacks; this separates application-specific code from the generic server routines, which are indifferent to the contents of log entries or the nature of read-only requests.

`raft_main` takes as arguments some parameters that determine the behavior of the server. These are:

- A "mode" argument specifying whether the server is being started again after a crash

- The ID of this server, an integer between 1 and the number of servers in the cluster; this is passed as an argument to facilitate using the same configuration file for all servers in a cluster (see next item)
- A path to a configuration file, which specifies the UDP port to be used for server-to-server communication as well as addresses for all servers in the cluster
- A path to a "persistence file", which will be used by the server to record the Raft log and other data that are preserved after a server crash
- A file descriptor for a socket that the server will use for client communication

The remainder of this section describes how `server.c` implements some key parts of the Raft algorithm.

Server communication

In my implementation, servers communicate by exchanging UDP datagrams. UDP is a natural choice in this case because Raft is robust to loss or reordering of server messages and because server messages can be made small enough (512 bytes) to fit in a datagram that will not be fragmented in the network layer. I have made no effort to handle corruption of server messages (which Raft does not allow): a more "production-ready" implementation would no doubt incorporate integrity checks to prevent servers from misbehaving when corruption occurs (and is not detected by the UDP checksum). I also took a purely serial approach to sending server-to-server messages, rather than sending messages in parallel as the Raft paper suggests, because the latter is purely a performance optimization.

The server event loop

At any given time, a Raft server is in one of three states: follower, candidate, or leader. Some transitions between these states are synchronous, in the sense that they happen in response to receiving a message from another server. For instance, a server that receives a message labeled with a higher term number than its own immediately reverts to follower state. Raft also specifies two asynchronous state transitions: a follower that does not grant its vote or receive a valid AppendEntries request for a certain length of time becomes a candidate, and a candidate that does not receive a majority of votes or receive a valid AppendEntries request for a certain amount of time restarts its candidacy in the next term.

In my implementation, a server in follower, candidate, or leader state runs code in the functions `raft_follower`, `raft_candidate`, or `raft_leader`, respectively. To simulate a server's synchronous state transitions, I use the following structure (in the `raft_server` function):

```
while (true) {
```

```
    switch (current_state) {
    case RAFT_FOLLOWER:
        raft_follower(/* arguments */);
        break;
    case RAFT_CANDIDATE:
        raft_candidate(/* arguments */);
        break;
    case RAFT_LEADER:
        raft_leader(/* arguments */);
        break;
    }
}
```

To synchronously transition from one state to another, we set the global variable `current_state` to the appropriate value and return from the function corresponding to the old state. Asynchronous state transitions are achieved using a global POSIX timer, which causes `SIGALRM` to be sent to the server process when it expires. A handler for `SIGALRM`, installed at the beginning of `raft_server`, sets `current_state` to `RAFT_CANDIDATE` and uses `siglongjmp` to return to the top of the `while` loop shown above.

The management of the global timer is somewhat tricky. In response to a message received while in follower or candidate state, we may need to do all three of

1. update some server data;
2. send a message in response to the one just received;
3. reset the timer.

From a correctness perspective, we must ensure that the first and second actions happen atomically: they must either both complete or both fail to complete. For instance, we should not send a "grant vote" message without updating our record of who we voted for in the current term or vice versa. Thus we must ensure that we cannot be interrupted by `SIGALRM` after beginning these actions and before completing them. One way of doing this is to block `SIGALRM` upon receiving a message and restore it after completing any necessary actions, but this approach is problematic because it conflicts with the need to possibly reset the timer (for instance, if we're in follower state and grant our vote to another server). Therefore, I used an alternative strategy: we save the state of the timer on message receipt and either restore or reset it after handling a message. An issue with this approach is that it means that the timer's impression of how much time has passed since it was last reset differs systematically and significantly from the actual elapsed time, since the processing performed while the timer is paused includes some relatively slow operations (`msync`, `sendto`). Thus the parameters controlling the random election timeout (in my implementation, a base value and range size) should be set somewhat *lower* than they would be for an ideal implementation that based timeouts on elapsed real time.

Persistence

To allow servers to resume operation after crashing, Raft requires that some data at each server be periodically written to stable storage. For example, the Raft log must be stored in this way, so that a recovering server can apply entries from before its crash to its state machine (once it has verified that they are committed). In addition, the current term number and the ID of the server voted for in the current term must be stored persistently, the first so that a server's term number is guaranteed to increase monotonically even in the presence of crashes and the second to guarantee that a server never votes for two different candidates in the same term. My implementation also persistently stores the index of the last log entry, since this is needed to manipulate the log and is also used when sending certain messages.

My implementation saves persistent data by performing memory-mapped I/O on a file, the "persistence file". This file contains a C data structure that holds the persistent data described in the previous paragraph. A server starting for the first time opens a new file for persistence, creates a shared (in particular, writeable) mapping of this file into memory with `mmap`, and manipulates the persistent data by performing ordinary C memory accesses. A recovering server similarly maps the file holding the persistent data of its pre-crash instance. As the Raft log at each server grows, more space is needed to store it in memory and on disk; thus, whenever new entries are to be written to the log, TODO

To ensure that restoring data from the persist file after a crash does not cause a server to behave incorrectly, two requirements must hold. First, as noted in the Raft paper, we must ensure that when a server responds to a vote or append request from another server, all of the server's previous updates to persistent data are up-to-date on stable storage, from which they can be recovered after a crash. If this were not guaranteed then a server could grant its vote to another server, crash, recover, and then grant its vote to different server, not having recovered its update to the "voted for" item from the persistence file. Similarly, a server could indicate to another server that it had appended some entries to its log, crash, and then fail to recover the appended entries. My implementation uses the `msync` system call to ensure that its updates to the mapped persistence file are applied to the file itself before responding to any message from another server.

A second requirement for correct behavior of a recovering server is that the persist file is never left in an "incorrect" state after a crash. This problem arises, for example, whenever a server needs to transition to a new term. This involves both updating the "current term" item in the persistence data structure and setting the "voted for" item to its null value (since the server has not yet voted in the new term). If the server crashed after the "voted for" update propagated to the persistence file but before the "current term" update did so, it might vote for a second candidate in the old term upon recovering. To avoid this, my implementation handles term transitions by first performing the term update,

then calling `msync`, then performing the vote update. This prevents the scenario just described. If the server crashes after the term update is propagated but before the vote update is propagated, it will not behave incorrectly on recovery: it may deny a vote request that it would otherwise have granted because the "voted for" item contains a stale value, but it will not grant its vote even to the server with the corresponding ID without checking the other compatibility conditions for voting (based on term numbers and logs). At worst, availability will suffer.

Servers also need to append new entries to their logs: this involves both copying data into the log itself and updating the "last index" item in the data structure. To ensure that such updates do not leave the persistence file in an incorrect state, my implementation calls `msync` after copying the new entries and before updating the index. Thus, a crash partway through the copy operation, or after the copy operation and before the index update, does not cause the recovering server to examine an inconsistent log.

Assuming that the persistence file is left in a consistent state, recovering after a crash is relatively simple. When running in "recovery mode", the `raft_main` function simply opens the already-existing persistence file and maps it at the appropriate size, then proceeds exactly as normal.

Candidates

The candidate state is simpler than the follower and leader states, because candidates do not modify or consult the Raft log. Conceptually, a candidate sends a RequestVote message to each other server, waits for responses to arrive, and acts on them. To guard against the loss of outgoing or incoming messages, candidates in fact send these requests repeatedly based on a timeout, which is distinct from the timeout that triggers elections (this also affects candidates). The vote timeout is simpler to implement than the election timeout: in my implementation, candidates repeatedly call `poll(2)` to await the arrival of either a client request or a server message, and if a `poll` times out (based on a provided parameter), the candidate resends its vote requests. This implementation has a potential problem: a constant stream of arriving messages either from a client or from servers whose votes have already been received could prevent the timeout from triggering, so that outstanding requests are never resent. This does not affect the correctness of the implementation, but could cause elections to end inconclusively when they might otherwise have chosen a leader, hurting availability. As a simple guard against this possibility, candidates might keep a running count of the number of "spurious" messages (client requests or server messages that did not change the candidate's vote tally) received since vote requests were last sent, and resend outstanding requests when this count reaches a certain threshold. (I chose not to add this feature to my implementation for the sake of simplicity.)

Leaders

In leader state, a Raft server attempts to propagate new entries provided by clients to other servers' logs, and also sends "heartbeats" to prevent the other servers from transitioning to candidate state. My implementation allows for the possibility that a client will submit several entries successively without waiting for confirmation that each has been replicated and applied to the state machine (although my actual driver code does not do this, for simplicity); entries will be applied in the order they are submitted. Consequently, the leader may send an append request to a follower without waiting to receive responses to any previous append requests to that follower, even neglecting "heartbeats". This highlights the need for a mechanism for the leader to correlate responses with the append requests they were sent in response to. In fact, since server-to-server messages may be reordered, such a mechanism would be required even if only one append request were allowed to be outstanding at each follower at any given time: the leader must be able to recognize "stale" responses that correspond to an already-completed append request. (This issue does not arise for voting, essentially because all the vote requests a candidate sends to a fixed other server in a fixed term are indistinguishable.) I accomplished this in my implementation by having followers include the "previous index" and "number of entries" fields from a received append request in their responses to that request. The server code can then process incoming append results uniformly, without considering the order in which they arrive; see the comments in the relevant part of the `raft_leader` routine for details of how this works.

Raft's "heartbeat" messages are simply AppendEntries messages containing no entries. Servers do not have to process them specially: they can take the same steps for all AppendEntries messages, and the heartbeats will have exactly the desired effect. Followers will reset their election timers, and all servers will update their `current_term` and possibly revert to follower state; their logs will be unaffected. A trivial consequence of this is that leaders do not need to send heartbeats to any servers to which they have recently sent "real" (non-empty) AppendEntries requests. Thus, in my implementation, leaders deal with the responsibilities of resending potentially lost append requests and sending heartbeats simultaneously: if a leader passes a certain amount of time without receiving a client or server message, it sends each other server an AppendEntries message, which contains the entries that the leader has not yet successfully replicated (to its knowledge) on that server, up to a cap. If there are no such entries for some server, the corresponding message is a "pure" heartbeat. (See the "Candidates" subsection for a potential problem with this approach.)

Clients

Raft leaders act as intermediaries between a locally-maintained *state machine* and clients that aim to modify or consult this state machine; using the formalism

of the Raft log and the associated protocols, leaders ensure that modifications are applied in the same order to all servers' state machines and that read-only requests (see below) are satisfied in a linearizable manner.

In my implementation, a leader listens for client requests on a single socket, which must be set up by driver code. The server code is mostly agnostic to the type of socket used, but assumes that it provides reliable two-way communication. (The second restriction could easily be lifted by passing two sockets to the server, one for reading and one for writing.) My driver code uses an anonymous Unix-domain stream socket pair (`socketpair(2)`); minimal changes would be required to use Internet-domain stream sockets (that is, TCP) instead. In a realistic system, leaders would dynamically accept new client connections and concurrently process request streams from multiple clients; this would be significantly more difficult to implement.

A client submits a write request to a leader by sending a message that includes a fixed-size buffer of arbitrary data. The leader copies this data into a new entry in its log, and when this entry has been replicated at a majority of servers it calls the driver-provided `apply_entry` callback, passing as an argument a pointer to the client data. (Other servers do the same once they have updated their view of which entries are committed based on the leader's messages.) The callback should have the effect of applying the modification represented by this data to the local state machine; in a realistic application, `apply_entry` might connect to a database or key-value store via a socket and modify some of its contents. The server code is completely oblivious to the structure of the state machine.

Read-only requests

The Raft paper describes how leaders can safely service requests from clients that consult but do not modify the local state machine. The difficulty with such requests is that that the Raft log is not available as a tool to control the order in which they are satisfied relative to other events, since it deals only with modifications to the state machine. Instead, after receiving a read-only client request, a leader waits to satisfy it until

1. some entry from the current term has been committed, and
2. append results (successful or not) have been received from a majority of servers since the request was submitted.

My implementation provides for read-only requests via a second callback, `resolve_read`. This is called by a leader after a client submits a read-only request and the two conditions above have been met. The first argument is a pointer to a buffer containing the data from the original request; the second is a fixed-size output buffer, to which the callback should write the "result" of the read. The leader then copies this output into a message and sends it to the client. Again, the server code is completely oblivious to how `resolve_read` goes about consulting the state machine. Implementing the conditions above

is fairly straightforward, but note that the data passed in a read request must be stored by the leader until it is able to satisfy the request (for requests that modify the state machine this storage is provided by the log). For simplicity, my implementation does not accept additional read requests if one is already pending. It would not be especially difficult to lift this restriction: instead of a single "slot" for pending read request data, leaders could keep a bounded queue (probably implemented as a ring buffer) of such slots and only refuse read requests once this queue filled up.

Testing

Below, I'll describe how I've tested my implementation by simulating clients that attempt to apply entries to a rudimentary state machine.

Driver

Besides the server code in `server.c` and `server.h` and the common Raft header `raft.h`, my submission includes a "driver" for my Raft implementation in `driver.c`. The driver both provides an extremely simple state machine for the server to interact with (including defining the callbacks `apply_entry` and `resolve_read`) and simulates client requests for the server to process (when it is a leader). (In a more realistic setup, client handling and state machine maintenance would probably be provided by separate modules.) The `driver` executable is run separately on each machine participating in a Raft cluster; command-line arguments specify the server ID of the machine in question, the location of the configuration file (the same for all machines), a file to use for persistence, and a random seed. Servers can be started (by executing the `driver` executable) in any order. The driver code performs some setup, including opening a Unix-domain socket pair, and then forks a child process to run the `raft_main` function. After this, the parent process acts as a client, sending alternate read and write requests to the child process and receiving responses via the socket pair.

The state machine used by the driver code is simply an in-memory C array of integers, declared in `driver.c`. The data passed in a client write request (which is copied into a Raft log entry) consists of an index and a value; the `apply_entry` callback sets the corresponding entry of the array to that value, and also prints a record of the write operation to standard output. A client read request just contains an index; the `resolve_read` callback reads the corresponding array value and copies it, along with the index, into the passed output buffer. Simple as this setup is, roughly the same approach would be used to connect the server code to a more useful data store.

The client-simulating portion of the driver code runs a simple loop. First, it

waits to receive a message from the server process indicating that it has become a leader. Then it alternates between submitting a write request (for an index that starts at 0 and increments) and a read request (for a randomly-chosen index), waiting for each request to "clear" before sending the next. The results of read requests are not printed. Once the client receives a message indicating that the attached server has lost leadership (even temporarily), it returns to the top of the loop.

Leaders may fail to respond to client requests that arrive shortly before they transition to follower state (after seeing a message with a new term number) or crash. Read-only requests are unproblematic in this respect, but clients have no way of knowing whether a write request that goes unanswered in this way will eventually be applied or not. My implementation does not address this issue since it is essentially the client's concern; the Raft paper suggests attaching unique identifiers to write requests to permit clients to retry these safely in the event of leader crashes, but I chose not to pursue this feature.

Simulating unreliability

In the interest of stress-testing my implementation, I wrote simple code to cause servers to skip processing of a random selection of incoming server messages. This simulates an unusually lossy network that regularly drops messages (simulating message reordering would be more complex, and message corruption isn't handled by my implementation). `server.c` can be compiled including or excluding this code, so we can build either "reliable" or "unreliable" `driver` executables. (When running `make`, the reliable driver is built as `driver`, and the unreliable driver as `driver-flaky`.)

Results

I've run my implementation on a cluster under a variety of circumstances and checked as best I can that its behavior matches Raft's guarantees. In particular, I've tried to check that write requests are applied in the same order at all servers, even when messages are lost and servers may crash and recover. Because of the complexity of checking this and other properties of the implementation in an automated way, my assessments of correctness are based on manually examining the logging output from the driver processes (which records when entries are applied and read requests are satisfied) and the server processes (which records state transitions and sent and received messages). Below, I describe my observations of some scenarios I've tested in this way. In all cases I used a five-machine configuration (`node01`,..., `node05` on the CSUG network).

For the simplest possible test, we can run the driver on just one server. The server process quickly times out and becomes a candidate, then repeats this pattern indefinitely since no vote responses are received from other servers. If

we start the driver on a second machine, this server gets a vote request from the first server, updates its term, and grants its vote, but because a majority of servers are unresponsive the first server still cannot be elected, and thus elections continue to time out indefinitely.

When three or more servers are started up, one server (generally the one that was started first) quickly wins an election and begins to process requests from its attached client. The printed output shows that write requests are applied in the same order at every server. If the driver was compiled without the unreliability-simulating code, the cluster settles into a steady state, and the initial leader continues in this capacity as long as at least two other servers are active. If the leader is killed manually and a majority of servers remain active, a new leader is elected and begins to process its own client requests; otherwise, the cluster comes to a halt again. If the cluster operates with three or four running servers for a time before an additional server starts up, the new server gets a burst of AppendEntries messages from the leader containing the backlog of entries and quickly them to its state machine, in the same order as occurred at the other servers. There may be a delay before this occurs, because the current leader may initially attempt to send entries to the new server starting at a high index (depending on when it was elected); in this case, the leader gradually "counts down" in response to the failures of its append requests until it reaches the beginning of the log, after which the burst of successful appends proceeds as described.

Next, we can add the complication of bringing servers back online after they crash—this is done by passing the `-R` flag to the driver. The situation of a recovering server is similar to that of a server coming online for the first time when the remainder of the cluster has been operating without it, since both are missing some committed entries from their logs, and we can observer a similar delay followed by a burst of appends bringing the server up to date as in that case. Note that the recovering server will "replay" its entire log, applying each entry to its state machine anew. When the cluster eventually settles again into a steady state, the recovered server has applied the same entries in the same order as the other servers, and processes new entries from the leader just as they do.

Finally, we can check that the implementation is robust to lost server-to-server messages by enabling the message drop code. This has two readily noticeable effects: it increases the frequency of elections, and it decreases the throughput of the cluster (the rate at which write requests are satisfied). Nonetheless, the same entries are still applied at each server in the same order. The increased frequency of elections exacerbates the delay seen on starting one server much later than the others, or crashing and recovering a server: because each leader must start the "countdown" process for the affected server anew, when new leaders are installed frequently this server may not catch up with the others for quite a long time (or indeed ever). (However, if the remaining servers constitute a majority, new requests can continue to be satisfied while a server is delayed in this way.)

Limitations of manual testing

Although I believe my implementation successfully provides Raft's core guarantee (the same sequence of state machine operations is eventually applied at each server), much more could be done to verify its correctness beyond the purely manual methods I've used. I think the most promising avenue for automated testing in this case would be to take advantage of the additional guarantees listed in the Raft paper (p. 5), and particularly the "State Machine Safety" property (two servers cannot apply different entries residing at the same log index). These properties could in principle be verified by examining a sufficiently detailed history of each server's operations, without any knowledge of the order of events across different servers. Thus, for example, one could test that my implementation provides the State Machine Safety property by having each server write a record of all the entries it applies, with their log indices, to a file, and then running a simple script to parse the files for each server and verify that they are consistent with this property.

References

The Raft paper, on which I based my implementation:

- Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 305-319. 2014.

The following were useful references for the details of the various Linux APIs used in my code:

- Kerrisk, Michael. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.
- Stevens, W. Richard, Bill Fenner, and Andrew M. Rudoff. *UNIX network programming*. Vol. 1. Addison-Wesley Professional, 2004.