Part 1 Introduction to GO Language

**Infrastructures:**

Good supports for threads, locks, synchronizations between threads, convenient RPC package.

Type safe and memory safe.

Garbage collection. The combination of threads and garbage collection is particularly important.

When there is an error in go, it is much quicker to find by looking the line number provided by go.

**Go memory model:**

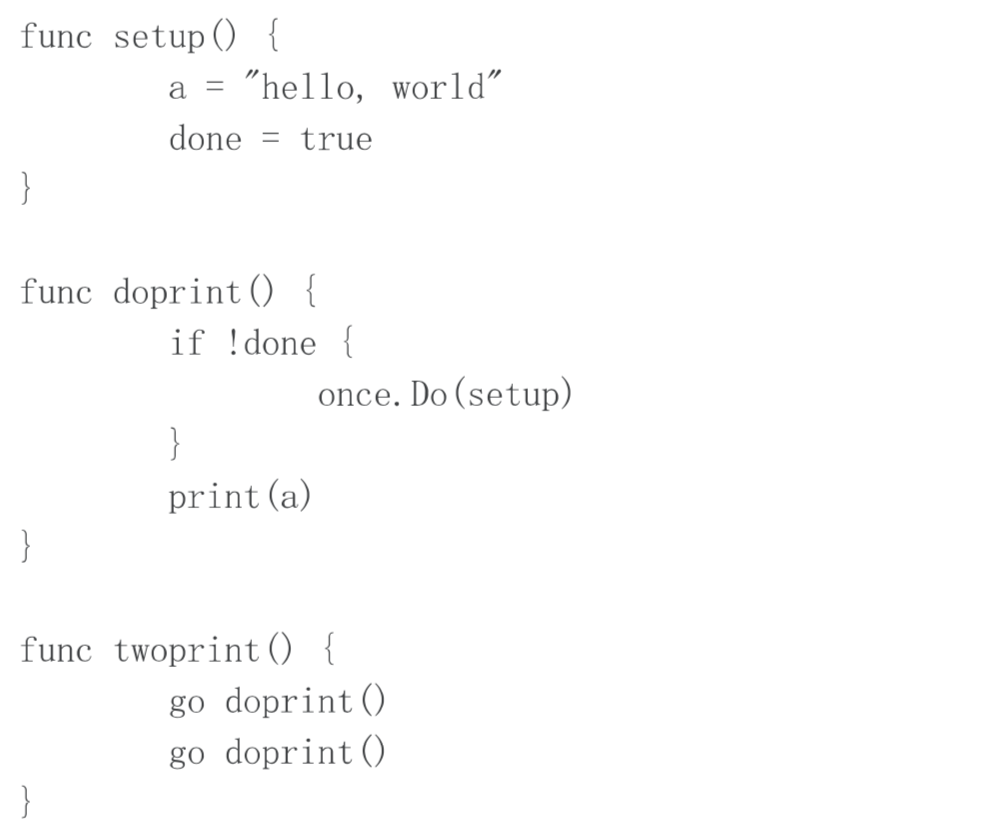
1. Within a single go routine, reads and writes must behave as if they executed in the order specified by the program. When multiple go routines access a shared variable v, they must use synchronization events to establish happens-before condition that ensure reads observe the desired writes.

2. The exit of a go routine is not guaranteed to happen before any event in the program. If the effects of a go routine must be observed by another go routine, a synchronization mechanism is required. (By the way, when the main routine exits, the other go routines also terminate.)

3. Channel communication. A send channel happens before the corresponding receive from that channel completes. The closing of a channel happens before a receive that returns zero. A receive from an unbuffered channel happens before the send on that channel completes (for unbuffed channel, the routine that sends to the channel blocks until the data in the channel is received. Verse visa). As for buffered channel, Kth receive on a channel with capacity C happens before the K + C th send from that channel(routine would not block unless the buffer has already full). This is a common idiom for limiting concurrency.

4. Use once for initialization in the presence of multiple goroutines. Multiple threads can execute once.Do(f) for a particular f, but only one will run f. (We can also run the f before calling these go routines.)

**Incorrect synchronizations:**



In func setup. There’s no guarantee that write to a happens before writes to done.

The solution is easy: use explicit synchronizations.

Part 2 Introduction to GFS

**Why it is hard to build a distributed system that stores large files:**

Performance -> Sharding -> Fault Tolerance -> Replication -> Risk in Inconsistency -> Pay for it with low performance.

The idea to build a distributed store system is starting with performance, to harness the hundreds of computer resources. Use sharding to achieve parallelism.

Thus, the single point failure could be common and automatic fault tolerance system is required. (Using replicas.)

If the replicas are not taken carefully, the system is risking in inconsistency…….

Strong consistency model is like you are talking to a single server running on a single machine, but the cost is too high.

**GFS Assumptions:**

Component failures are the norm rather than the exception. The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.

The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.

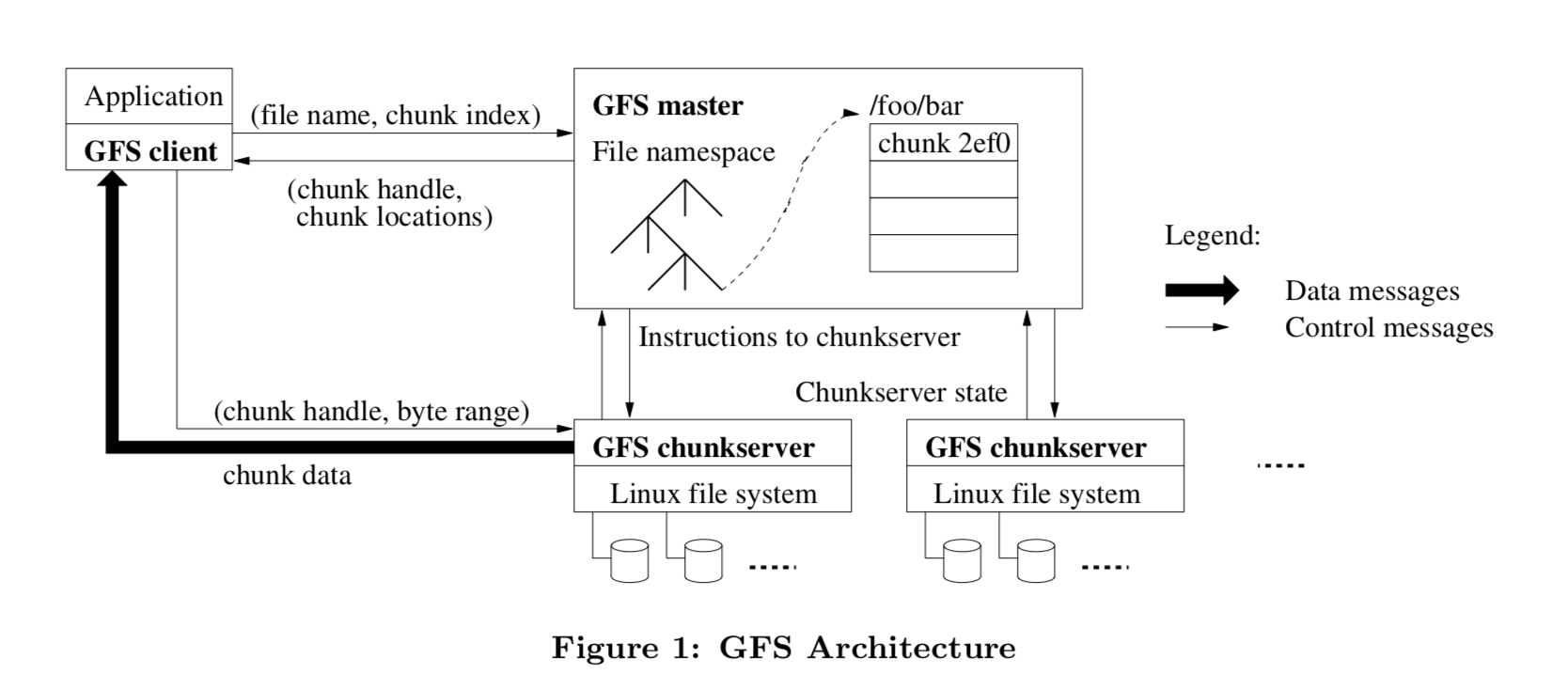
Most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Given this access pattern on huge files, the workloads primarily consist of two kinds of reads: large streaming reads and small random reads. The workloads have many large, sequential writes that append data to files. Small writes at arbitrary positions in a file are supported but do not have to be efficient.

The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.

Co-designing the applications and the file system API. Provide weaker consistency. High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

**Architecture:**

A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients.



Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64 bit chunk handle assigned by the master at the time of chunk creation.

Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks.

The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues.

**Single master:**

We must minimize master's involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk’s replicas. All metadata is kept in the master’s memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an operation log stored on the master’s local disk and replicated on remote machines.

Maintain two tables,

Filenames -> array of chunk handles.

Chunk handles -> list of chunk servers(Having version numbers for each chunk, and a primary server for lease expiration time.)

(Checkpoints and log needs to be stored on disk. Log entries are like set a new chunk for a file, or change the version number.)

**Chunk Size:**

A larger chunk size reduces client’s need to interact with the master. However, it may cause some chucks are required more often, and the chunkservers storing those chunks may become hot spots and thus slow down the performance.

**Operation Log:**

The operation log contains a historical record of critical metadata changes. It is the only persistent record of metadata. The master recovers its file system by replaying the operation log.

**Writes:**

Master find up to date replicas, and pick primary and secondary servers, increment version number on disk. And then tell P, S the new version number.

The client would talk to the primary to append records. The primary server appends the records and ask the secondaries to append records at the same offset. If all secondaries reply “yes” to the Primary, then primary could reply success to the clients. Otherwise the primary reply error to the client.

**Consistency Model:**

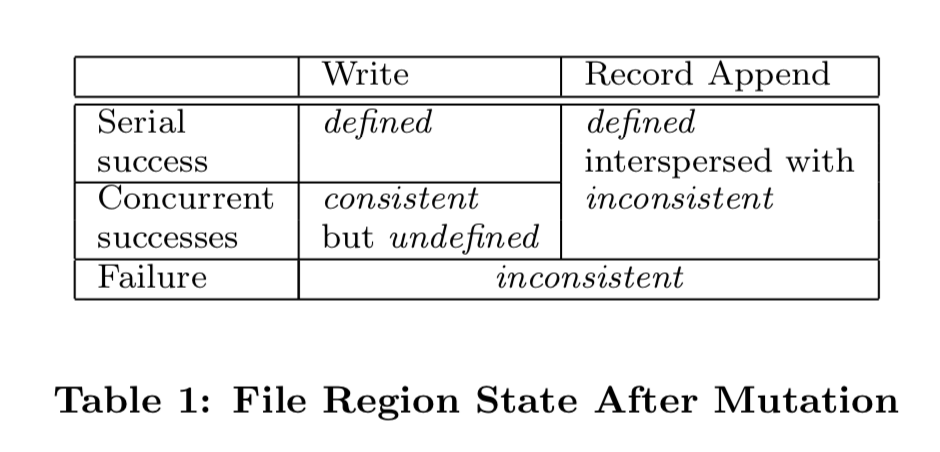
File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness.

A file region is consistent if all clients will always see the same data, regardless of which replicas they read from. A region is defined after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety.

When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written.

Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written.

A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times.



Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming.

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents.

Part 3 Introduction to Raft

**General concepts:**

Single primary server, which will never disagree with itself. In the system fault tolerance view, it pushes the real heart of the failure tolerance machinery into a little corner. (When the leader fails, what should the system do?) The design should be careful about the split-brain problem (When more than two servers think they are primary and serve the clients. There’s an easy but costing way to rule out the problem, make a network which can’t not fail). With a single leader, you can build a efficient system if there is no failure.

**Majority**. The use of majority makes Raft safe from the split-brain problem even in the face of a flaky network. Using an odd number of servers, when the servers are partitioned into several parts, at most one part could form a majority and serve the clients. Majority scheme is used in both leader election and to commit an entry. And there is one subtle thing going on here: at every step, the majority assembles for that step must contain at least one server that in the previous majority. Namely, any two majorities overlap at least one server.

Fault tolerance. A raft system with 2n + 1 servers can tolerate at most n servers failure.

Leader election. The new leader is guaranteed to know about the term number used by the previous leader because its majority overlaps the previous majority.

Entry commitment. Similarly, anything the previous leader could have committed must be present in the majority of the servers in Raft (There is at least one copy in the new majority and can be observed and received by other servers in the new majority, sooner or later. Commitment scheme guarantees that no later leader can overwrite the committed entry.)

Intended to design as a library. Used by applications maintain a deterministic state machine.

**Cooperation between raft and application:**

Hopefully, the clients are unaware that they are talking to replicated servers. Clients send the request to one server’s application layer.

The application layer pushes the request done to the raft.

Rafts chitchat with each other and try to commit the request.

Until the request is committed (followers write entries to disk before reply to leader to make sure that the log does replicate), the raft could tell the application layer that the request is handled successfully.

Application layer in leader server can then safely execute that operation and finally send the reply to the clients. Also, leader has to inform other servers that the commit number has been updated, which can be piggybacked on the next append entry message, or on the heartbeat RPC.

(I have a doubt here. I think, at least, the clients should find the leader among the servers.)

A flow control that throttles leader is required in the practice.

**Reboots:**

When the server reboots, it should not apply operations in its log immediately. Because it doesn’t know how many entries are not committed.

**Leader election:**

Raft divides time into terms of arbitrary length.

There is at most one leader at a term. It is guaranteed by 1) the candidates require to get the majority votes from all servers. 2) A server votes at most once in any term. If nothing goes wrong, the term will not change.

Every raft server keeps an election timer. If the total leader election time expires without receiving any message from the leader. The server assumes that the leader is died, increments its term and starts election. The expiration time is randomized to avoid competing.

When starting an election, the server votes for itself, and makes RPCs to ask other servers to vote for it. If receives votes from the majority (including itself), it becomes leader, and starts to send heartbeat message to inform other servers.

A server votes for the candidate if 1) the candidate’s term is greater than the server’s term (actually this is not declared in the paper. However, I find it perfectly reasonable as it strongly guarantees that a server votes at most once per term); 2) candidate’s log is at least as up-to-date as the server’s log. (More up-to-date: last entry with greater term, or same term, greater index.)

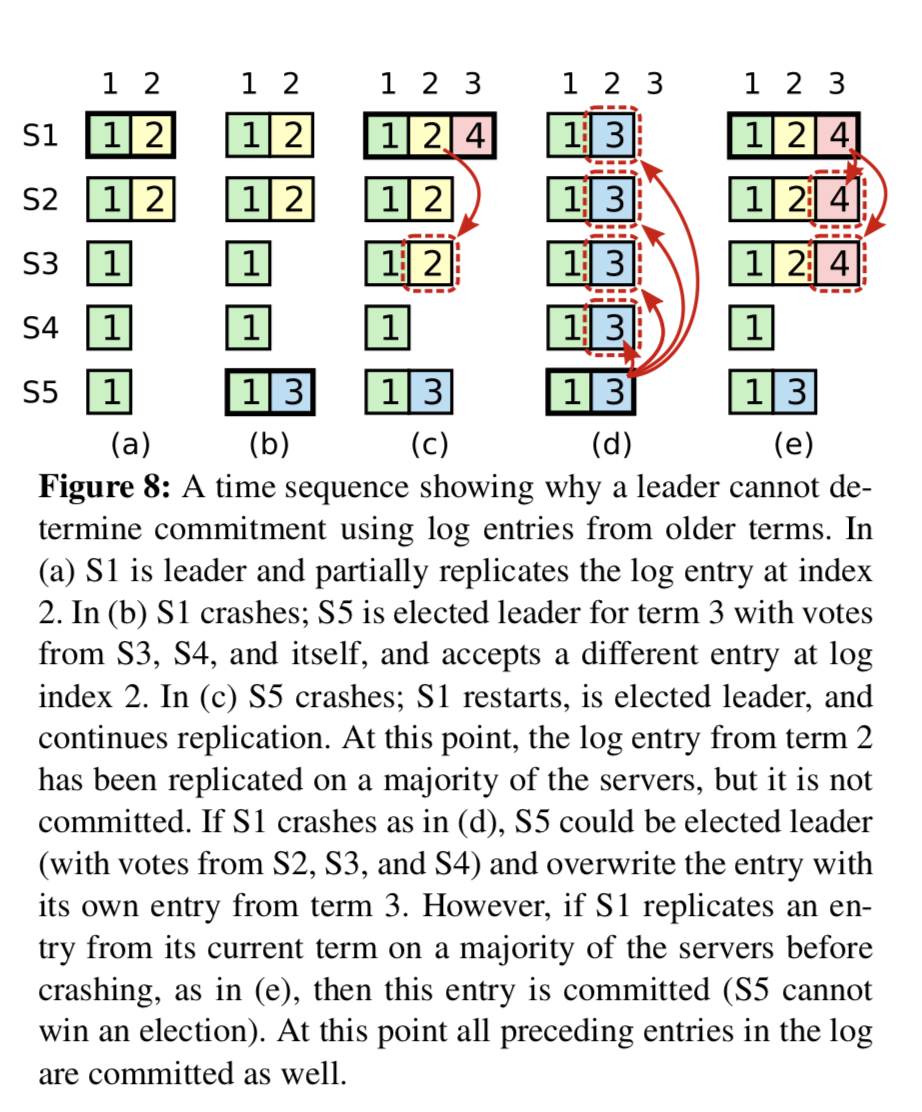
**Log replication:**

What is an entry in log contains? Term, and command.

After a leader comes into power and receives request from application layer, it creates a new entry, appends it to its own log, and sends the entry to its followers. Besides the entry, the leader also sends some info about the entry just ahead of new one. If the majority replies with “OK” for the entry, the leader can commit this entry. Otherwise, it tries to send previous entries.

On the follower side, if info in leaders message matches its own log, it drops all entries after that entry, and appends the new entry. Otherwise, it drops that entry and all entries after that.

However, a leader should not commit an entry which is not appended in its term. Because of the following scenario:



Otherwise it may commit an entry that can be overwritten by later leaders. (In figure 8, if S1 commit entry 2, leader S5 may overwrite it. Similarly, if S5 in (c) sends the entry with term 3 to S3, S4 and then commit it, it can be overwritten by S1.)

A leader only commits entries in its term. And when one entry in its term has been committed, all previous entries are also committed. (Thus, it is possible for an entry that has been replicated on the majority to be overwritten.)

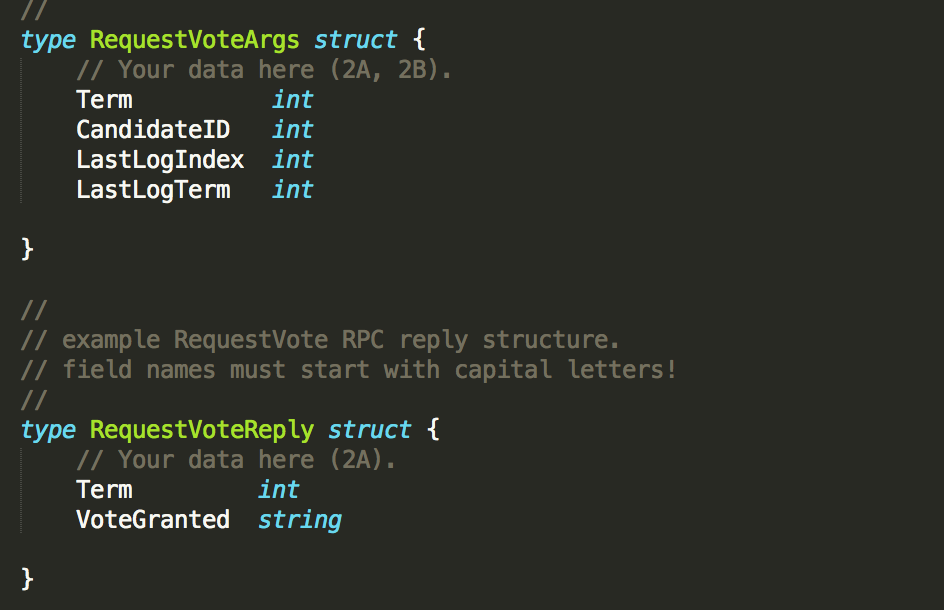
**Some additions:**

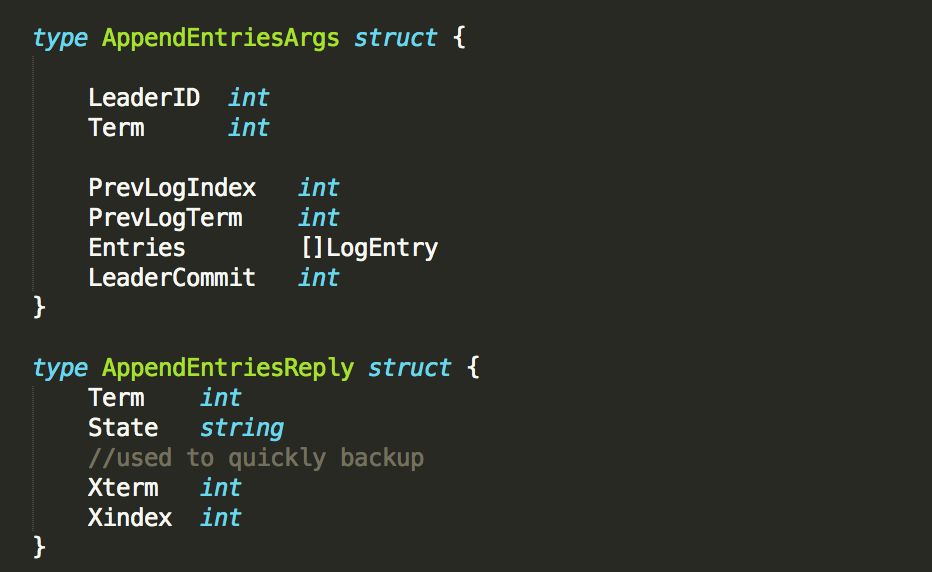
Every time servers communicate with each other, they exchange their term. If a server find someone’s term is greater, it updates its own term to be equal to that, and becomes follower.

Part 4 Raft Lab

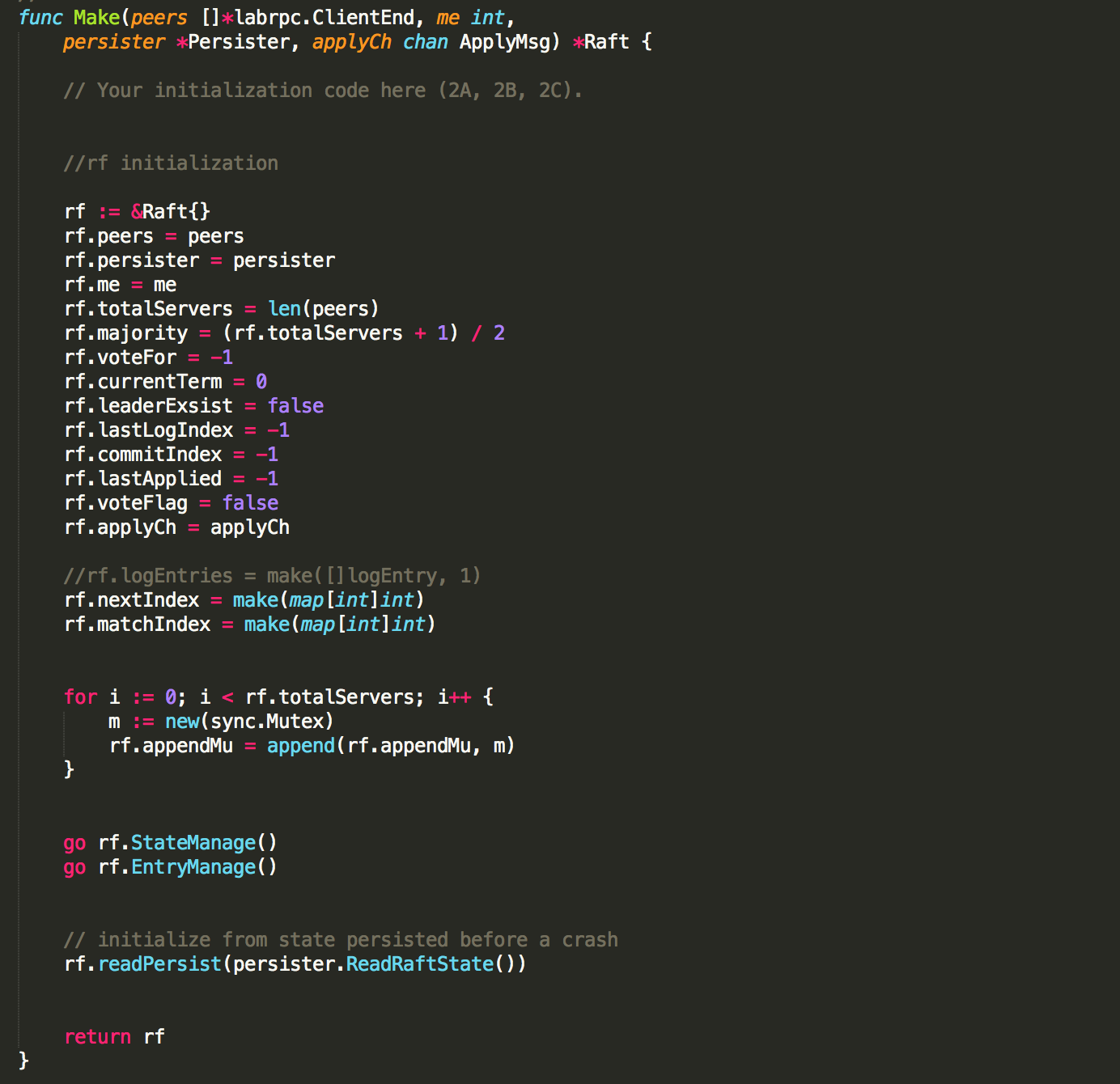
**Data structures:**







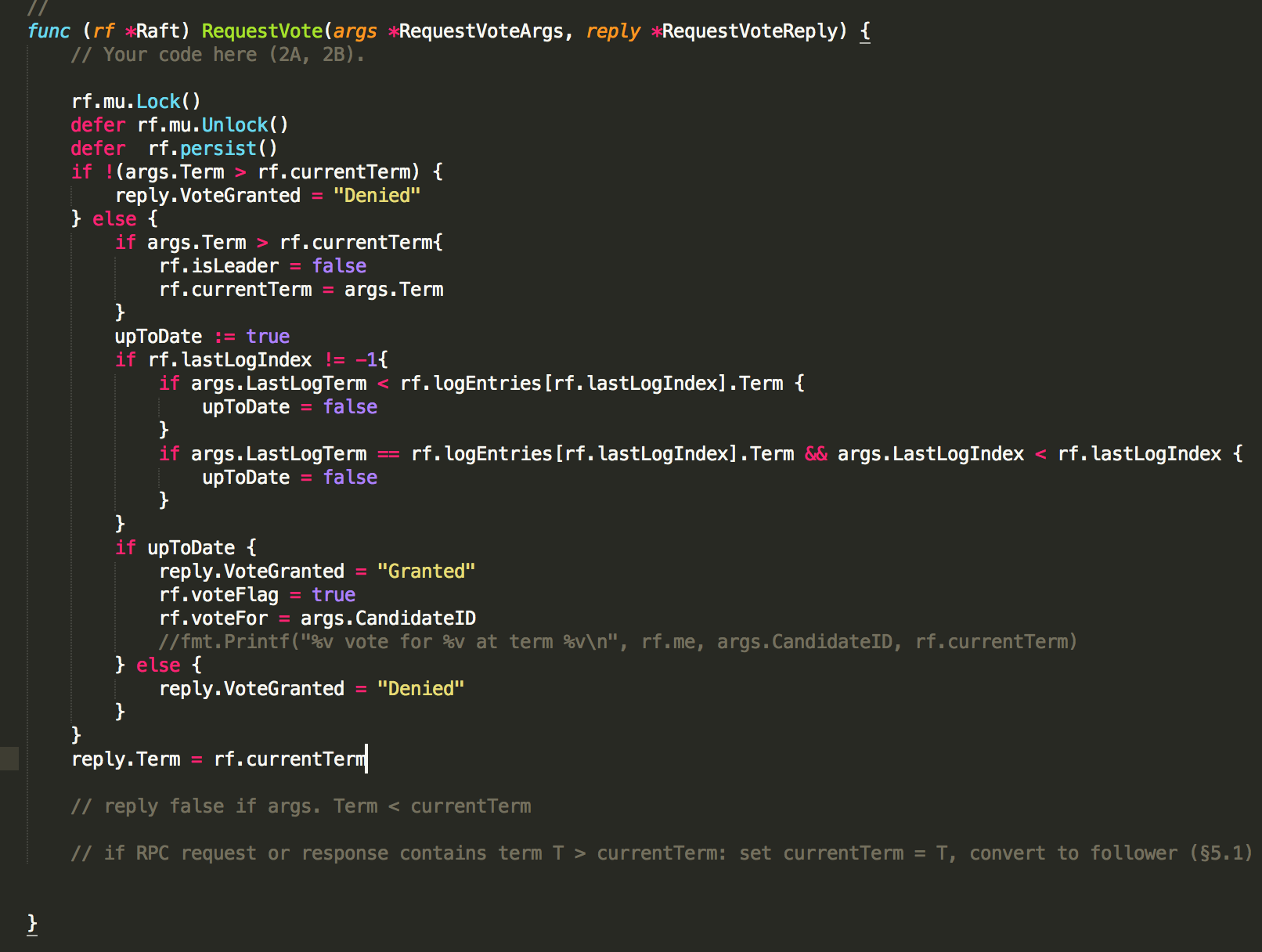
**Initialization:**



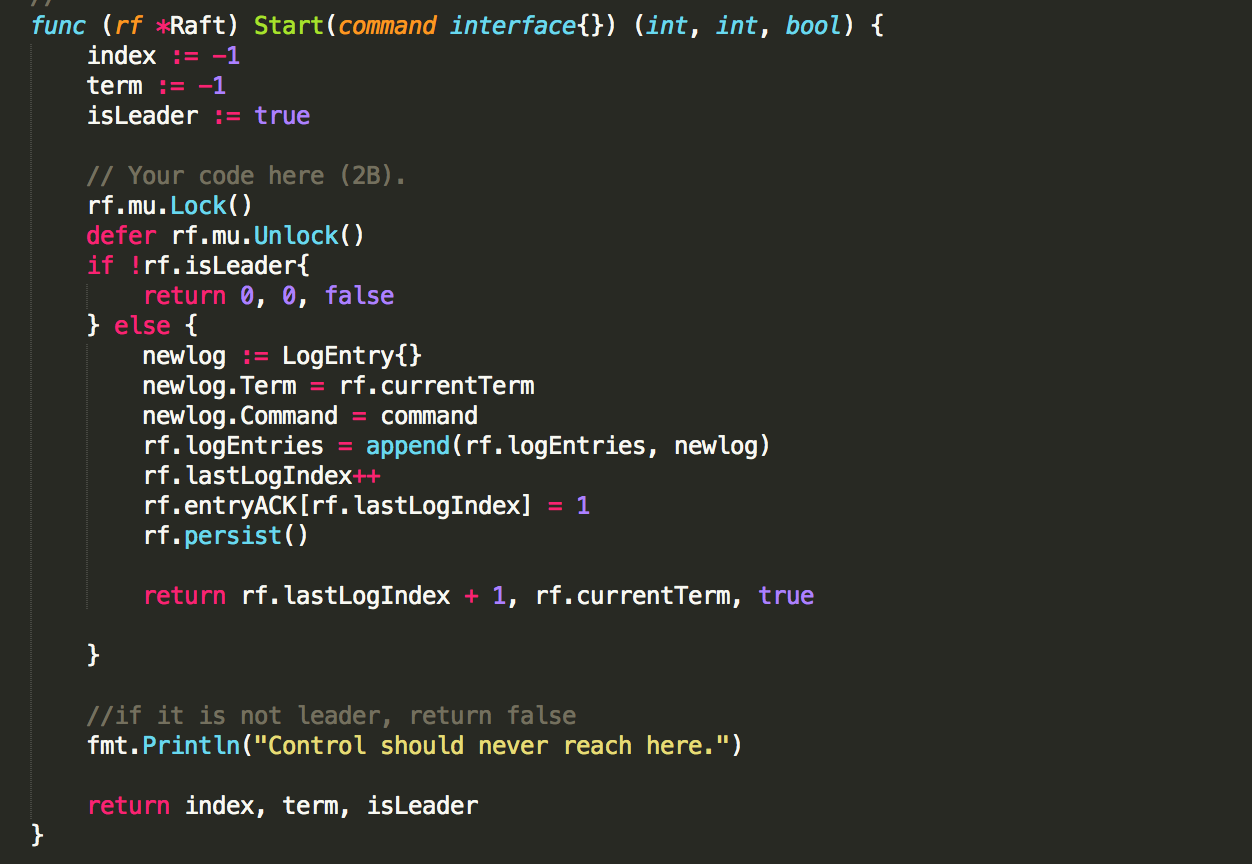
**Leader election:**

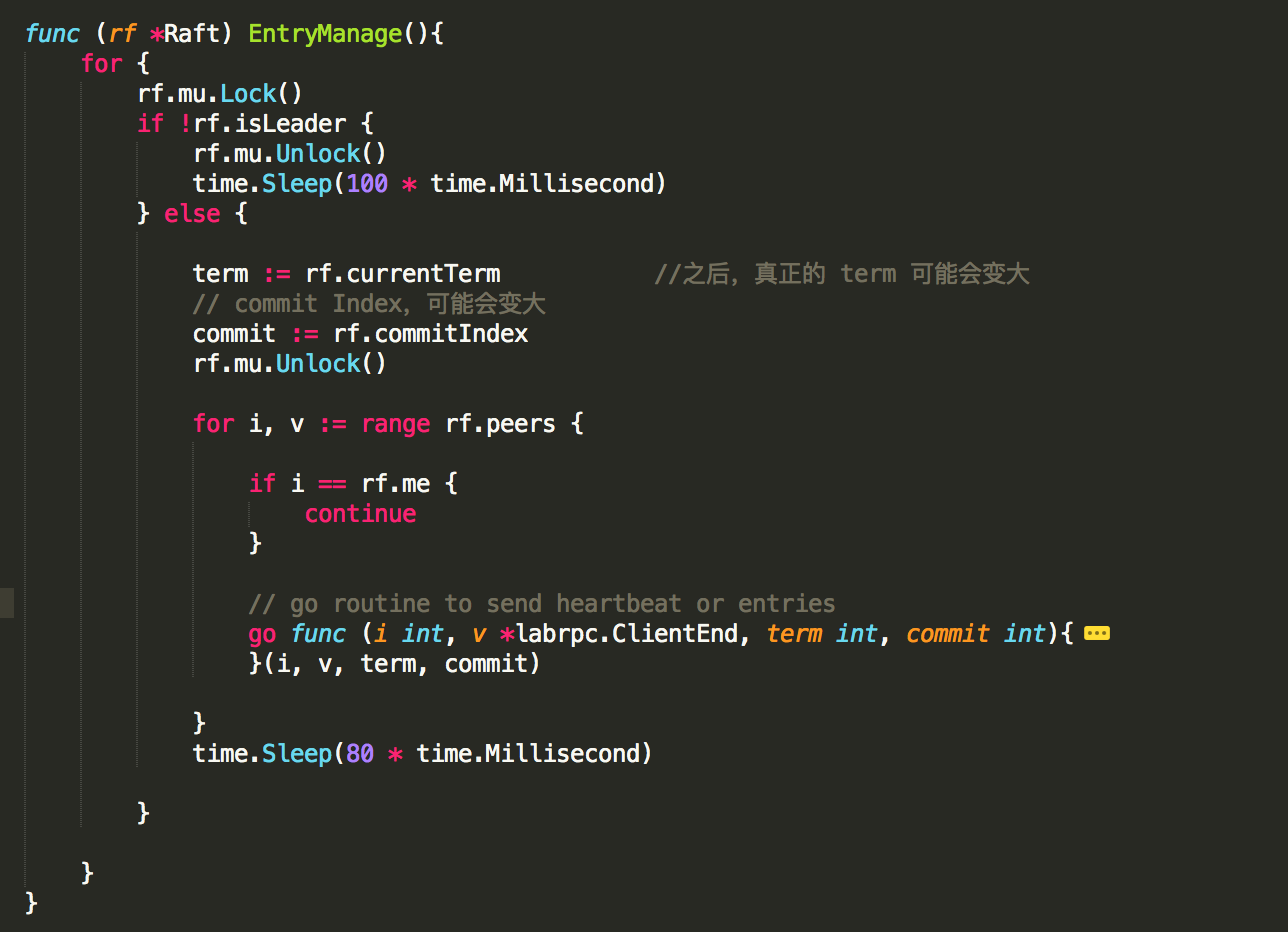


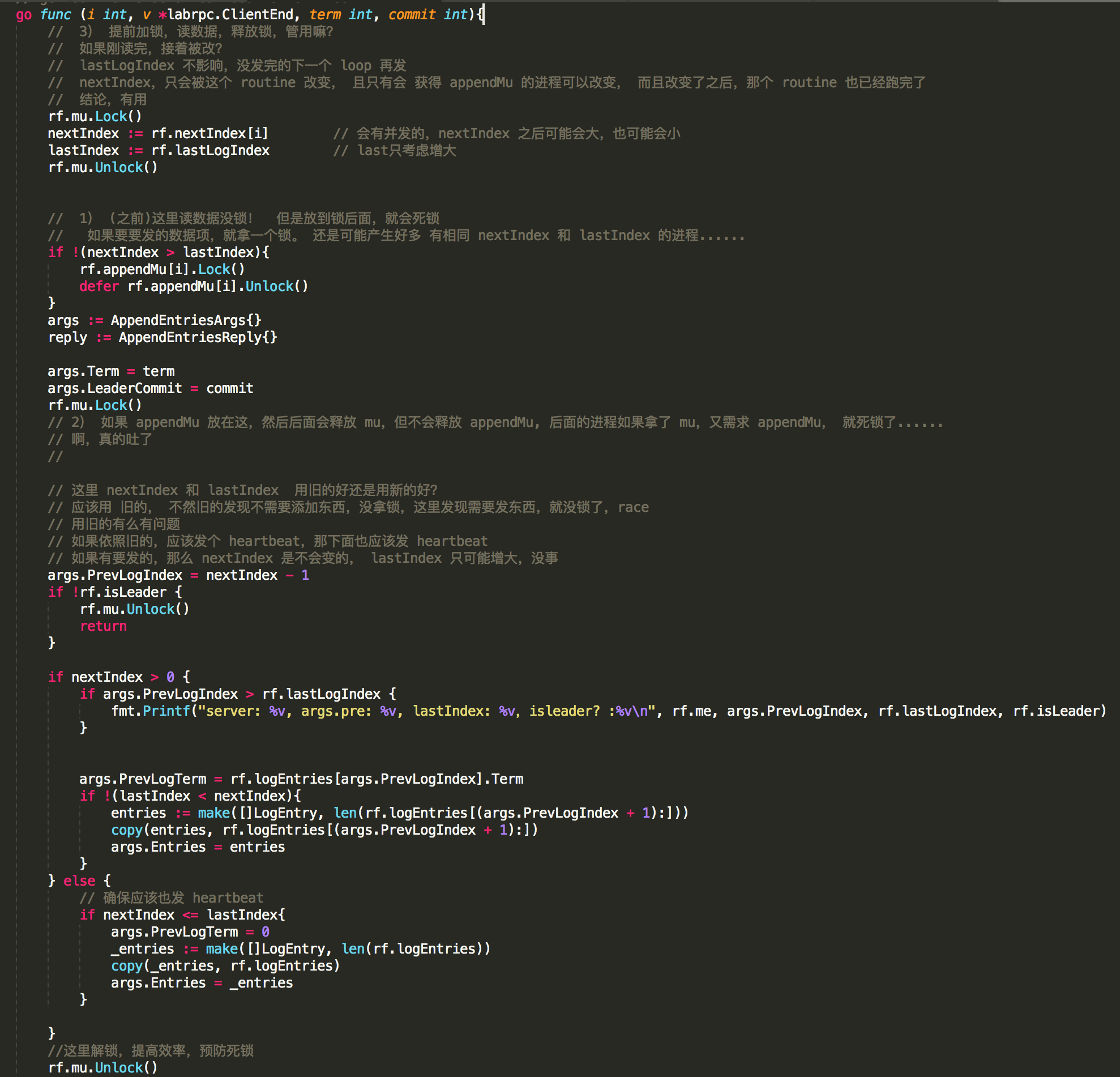




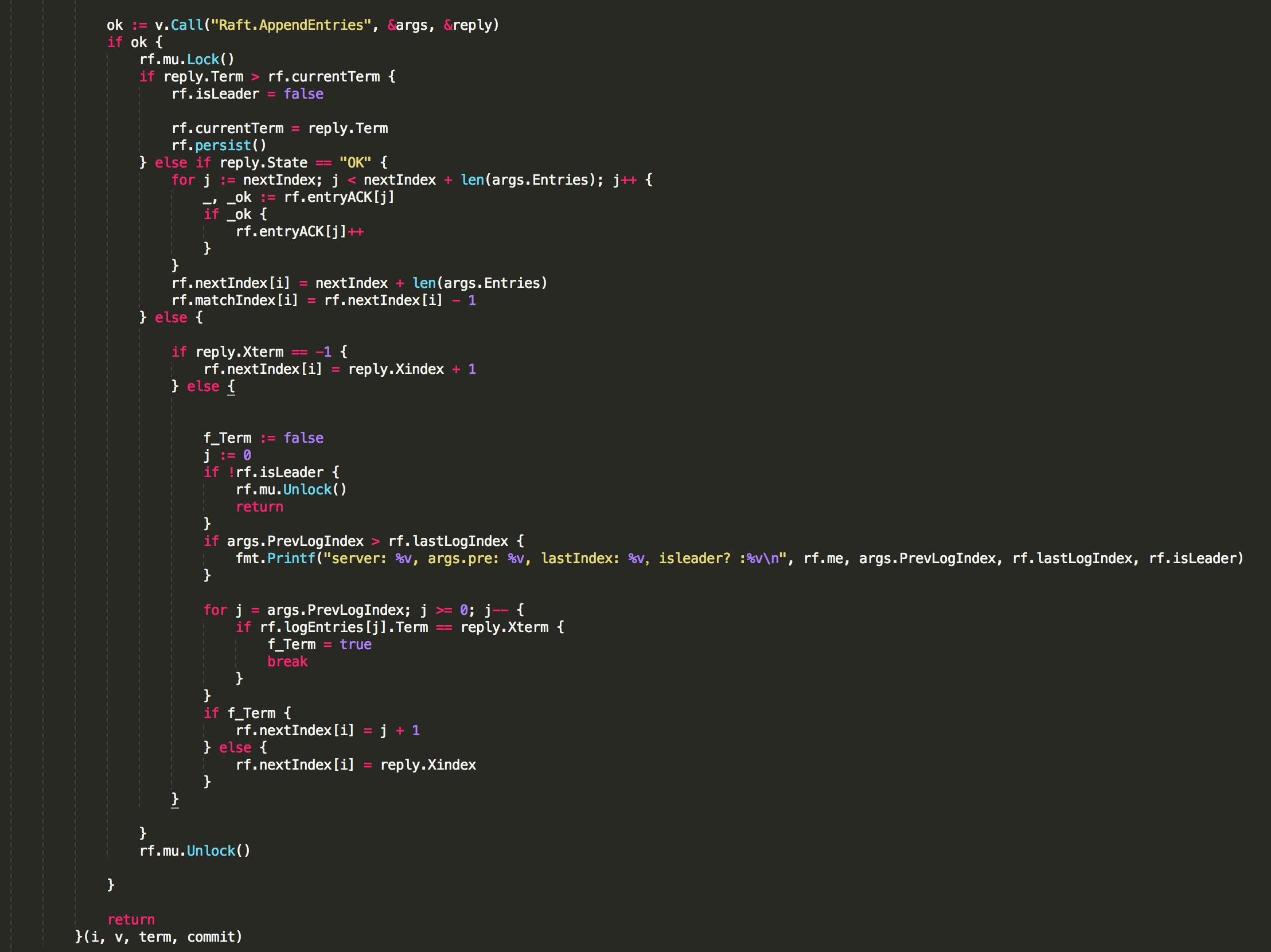
**Log replication:**

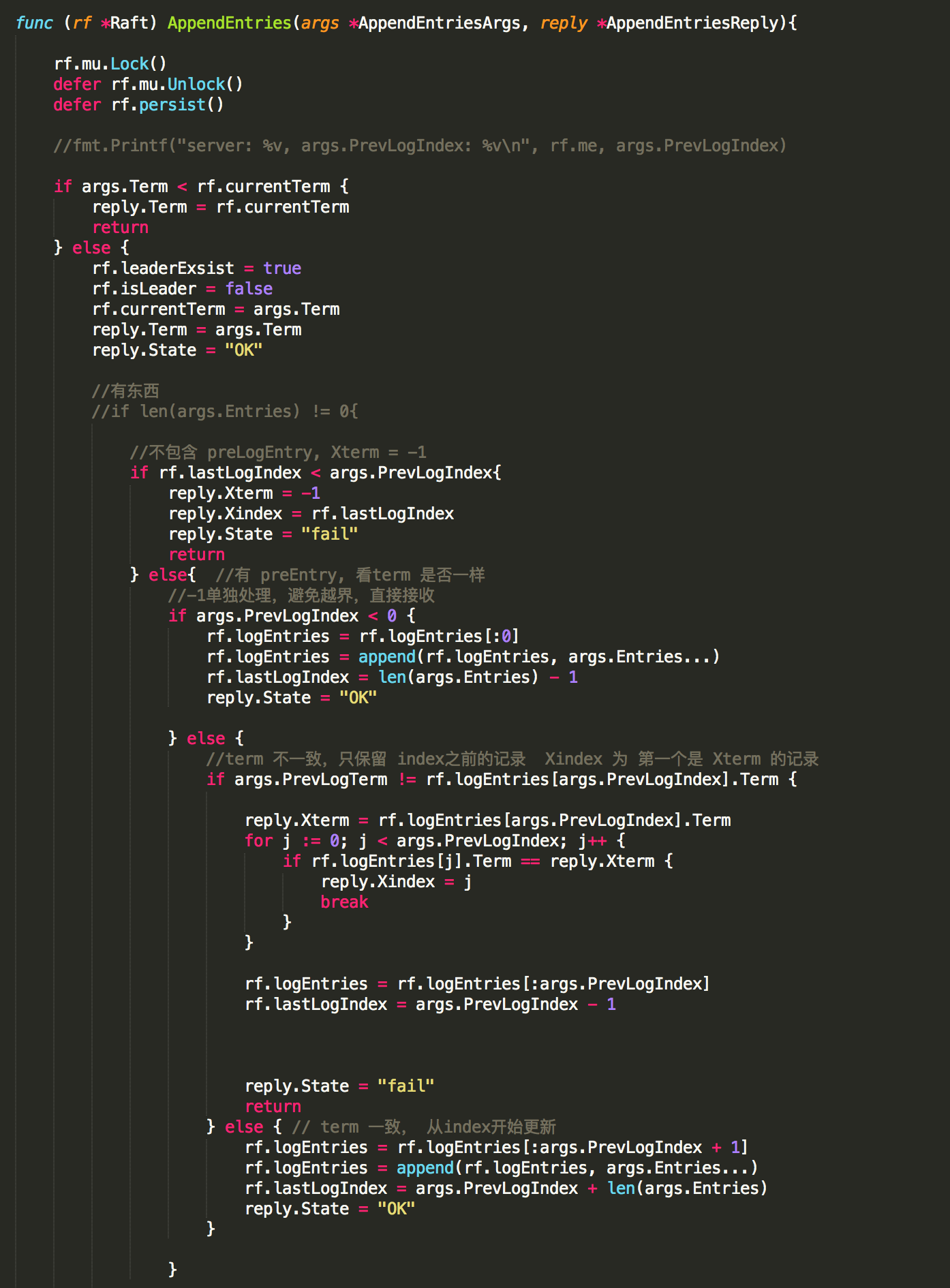






(Before making RPC)



(to be continue)



Part 5 Encountered Bugs

**Dead Lock:**

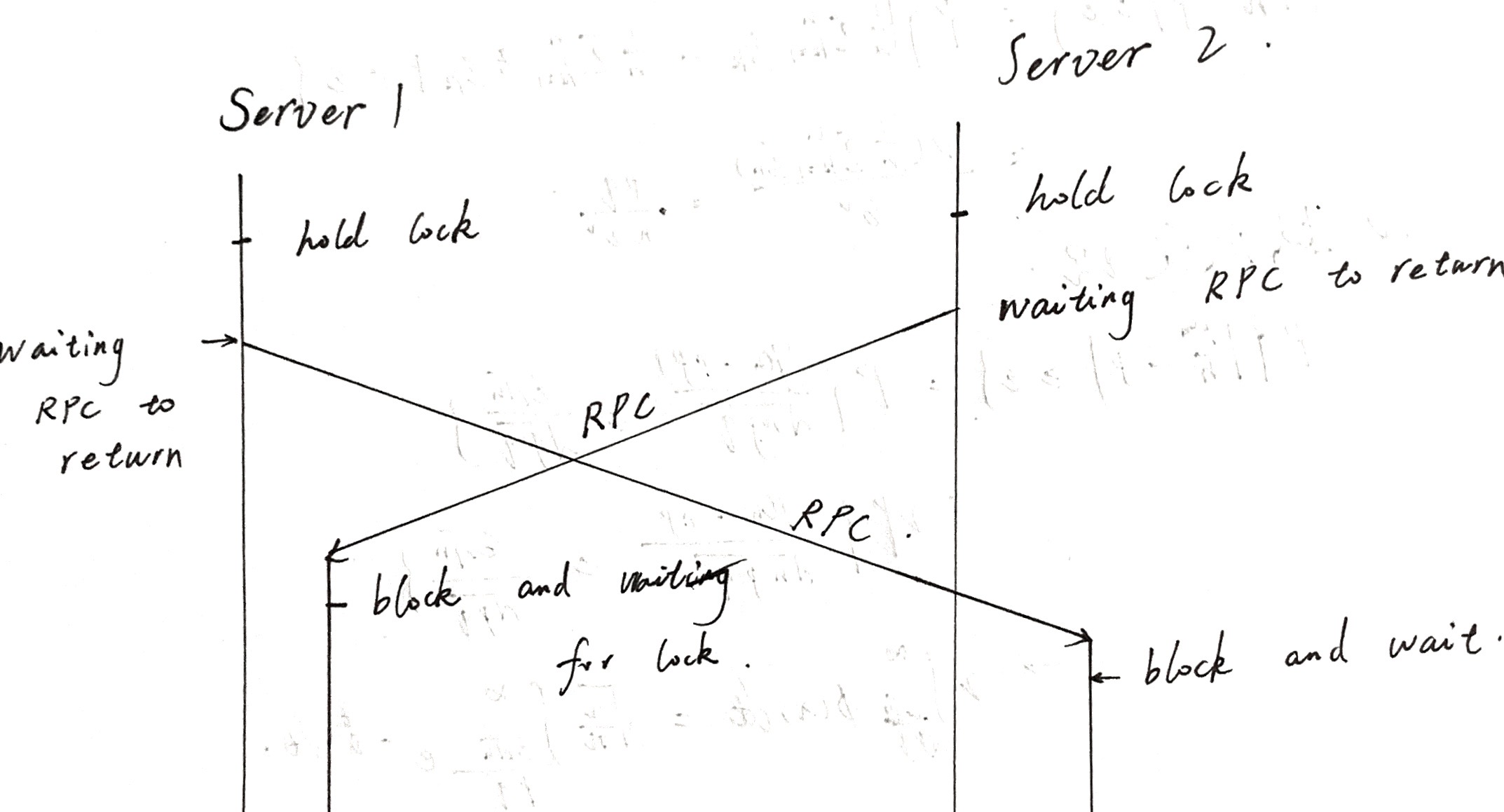
1) If a thread holding a lock across an RPC, it may cause Dead Lock. For example, in 2A, the scenario below might happen:

Server 1 requires lock; Simultaneously, Server 2 requires lock.

Server 1 call Server 2’s requestVote function; Simultaneously, Server 2 call Server 1’s requestVote.

In the requestVote function, the lock is also required.

In Server 1, the thread that calls RPC holds the lock, and waits RPC to return. However, in Server 2, the requestVote function is blocked because the lock is not released. And the thread in server 2 that holds the lock is waiting for Server 1’s reponse.

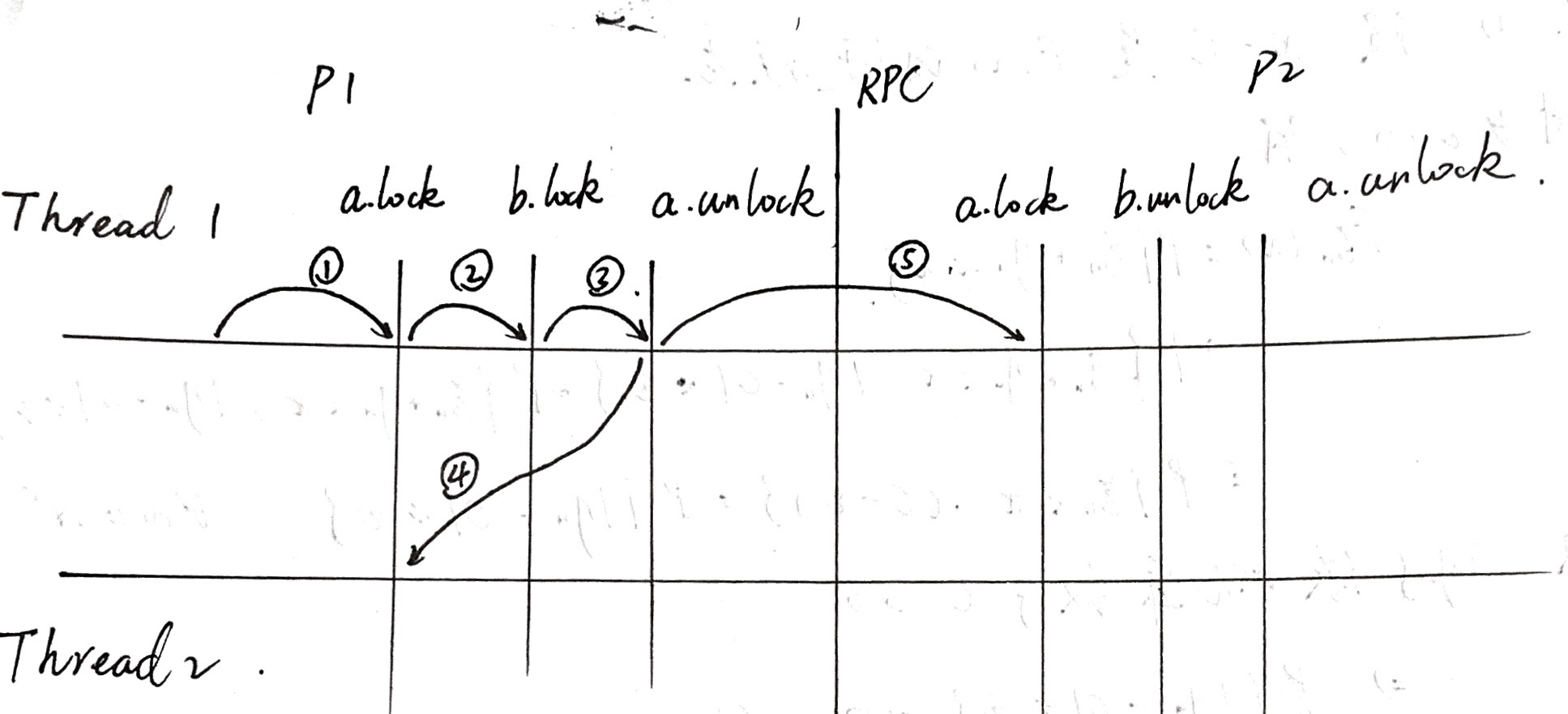


Even if it does not cause a deadlock, holding the lock across an RPC is inefficient. Since other threads can make no progress when this thread is waiting RPC to return.

DO NOT hold the lock across an RPC.

2) If a thread requires two mutex lock, careless release may cause deadlock.

In 2B, to avoid holding the lock across RPC, I released the lock before making the RPC. However, this may cause a race: if another goroutine is created before the RPC returns, rf’s state will be altered by the new goroutine, and then the anterior will update the rf’s state that has already been updated. So I tried to introduce a new mutex lock, which caused deadlock.



**Race:**

If you lock and unlock several times in a thread, the mutex lock may not keep your code from facing of race.

Example. In my entryManage function, there is an infinite for loop that make a goroutine to send new entries or heartbeat message to followers (if the server is leader.) Since I lock and unlock several times, the threads may interleave with each other, and a single thread may span a long period of time.

At the beginning of the thread, it may get the lock, find the server is leader, and then release the lock. Later, when it get the lock again, the server may not be the leader anymore. If it assumes the server is still leader, something might go wrong.