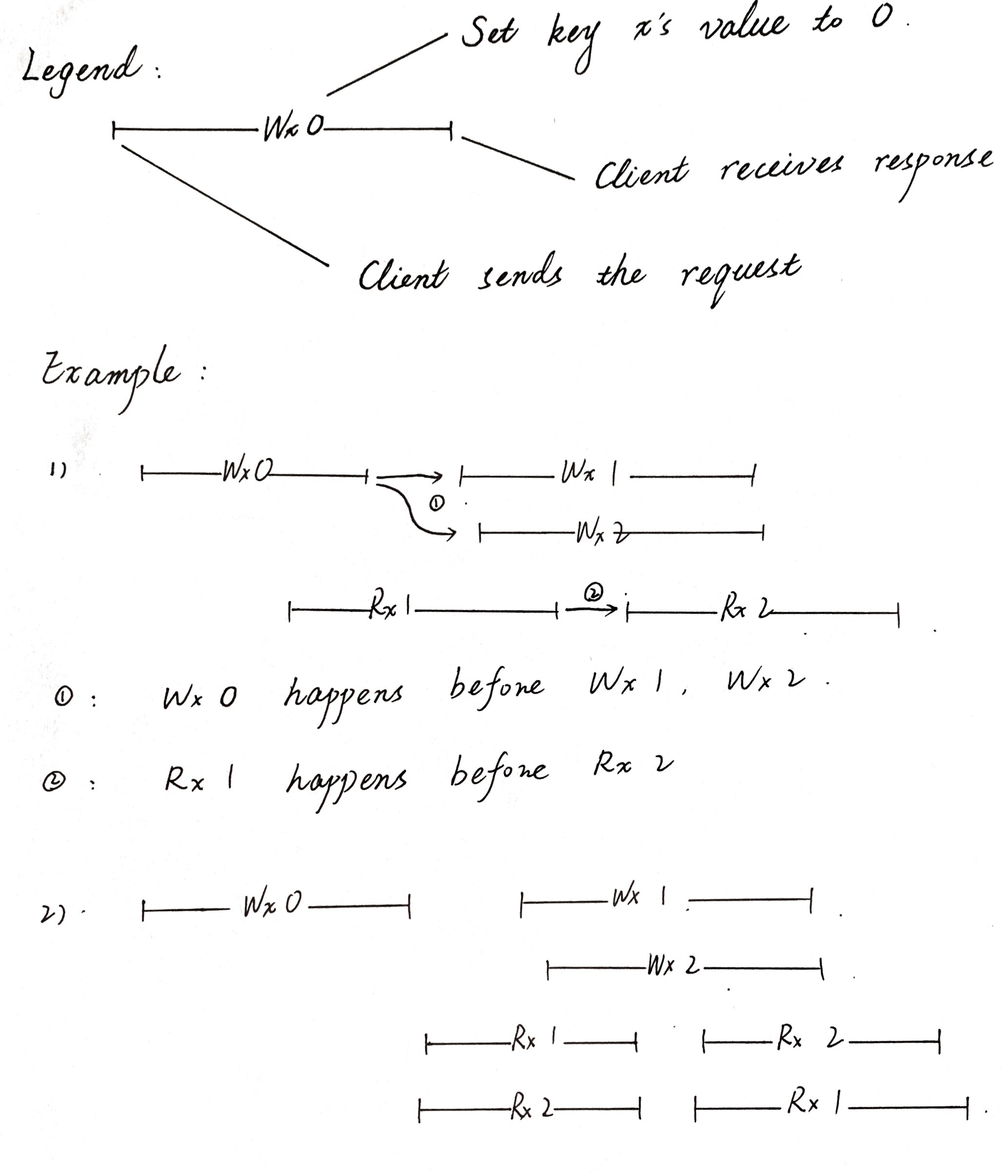
Part 1 Linearizability

It’s a kind of standard definition for what strong consistency means (works like a single server). If a system has a strong consistency, its history should always be linearizable.

Def: a history is linearizable if, ∃ total order of Ops that matches real time reads see preceding write in the order. (It is not something about the design. It’s really history by history.)

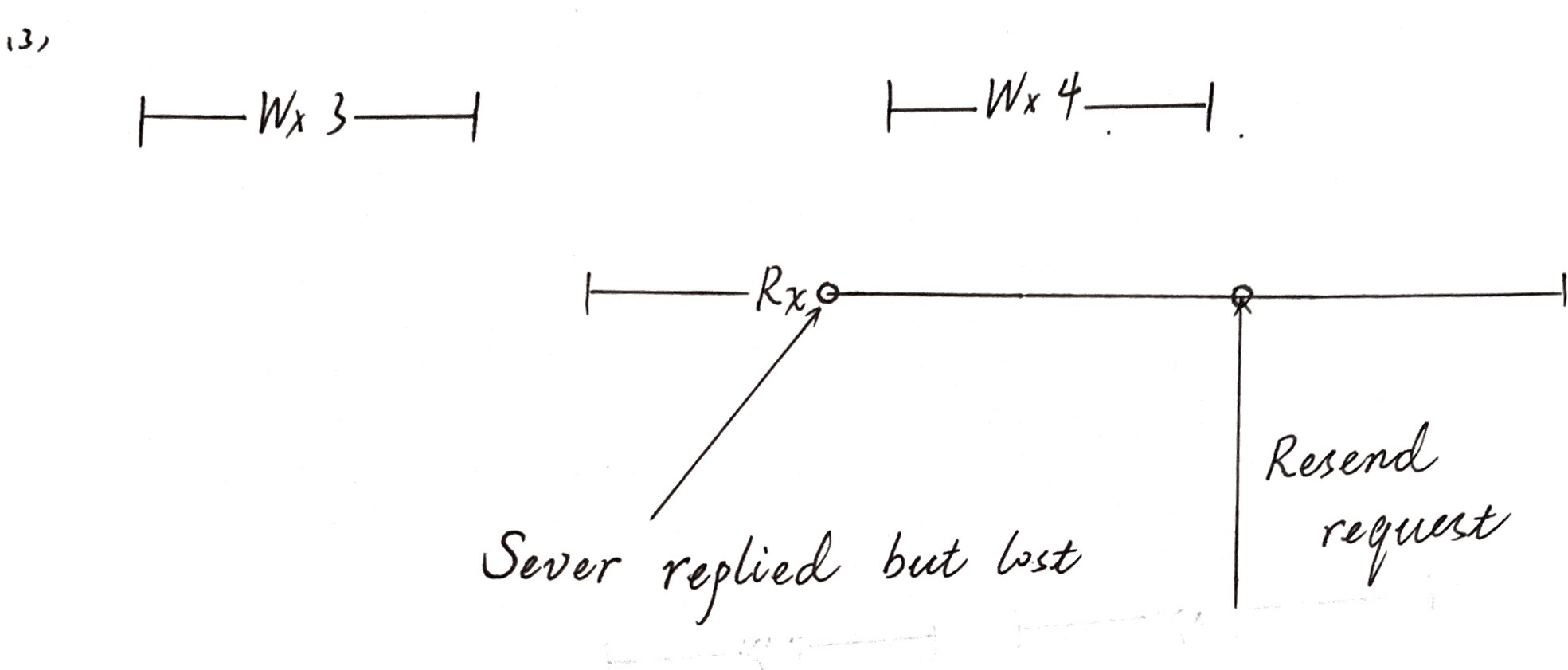


In example 1, Wx1, Wx2, Rx2 happened concurrently. Also, Wx1, Wx2, Rx1 happened concurrently. There is one possible order that guarantees that all reads see preceding write in the order: Wx0, Wx2, Rx2, Wx1, Rx1. So that order 1 is linearizable.

As for example 2, there is a Rx1 happened before Rx2, and also a Rx2 happened Rx1. However, there is only one Wx1 and one Wx2, so that the history is not linearizable.

In a linearizable system, no stale date is allowed. For example, if Wx1 happens before Wx2, and Wx2 happens before a read, then the read should never be able to see the old value (Wx1).

A final example:



The client issues a read of x but doesn’t get a response. Maybe the leader crashes at some point, or maybe the request is dropped by the network. And maybe leader has executed the request and replied but the reply is lost. All in all, in the client’s view, it doesn’t receive a response. And for most system, client will resend the request.

As for the leader, it should not re-execute a duplicated request. It may maintain a table to keep indexes (something like unique number from clients to find duplicated request) to avoid re-execution. And then may remember the original reply and repeat it for the client. However, whether the server repeats a reply or generates a new one is up to the system design.

Back to the client’s side. From the application’s point of view, even with retransmissions, the real time extent is from the very first transmission of the request to final time the application actually got the response. Given this, both Rx3 and Rx4 are valid here in the definition of linearizability.

Part 2 KV Raft Lab

The lab implements a Key-Value service on the top of Raft. 3 operations all allowed:

Get (key): returns its value

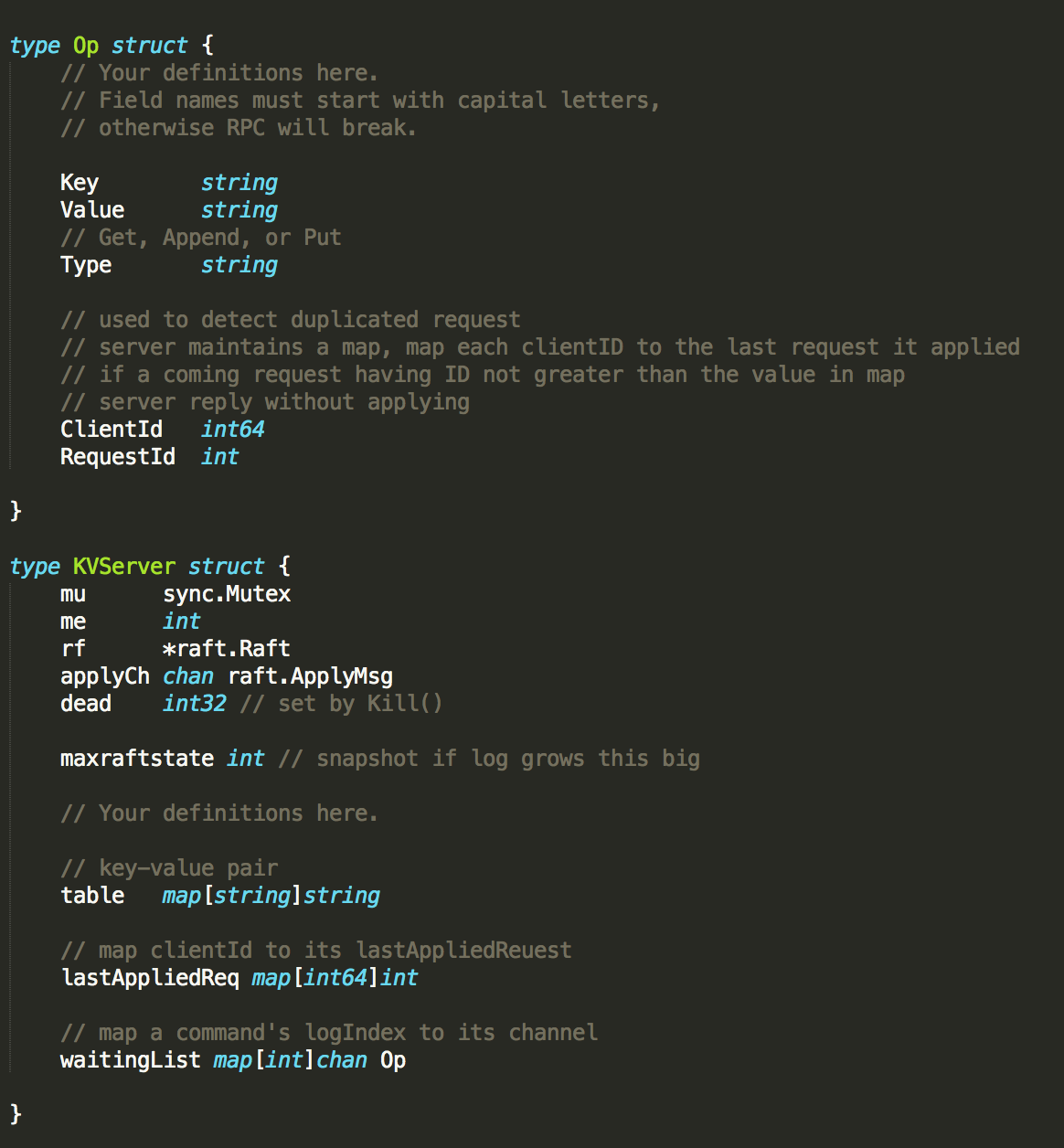
Put (key, value): table[key] = value

Append （key, value）: table[key] += value

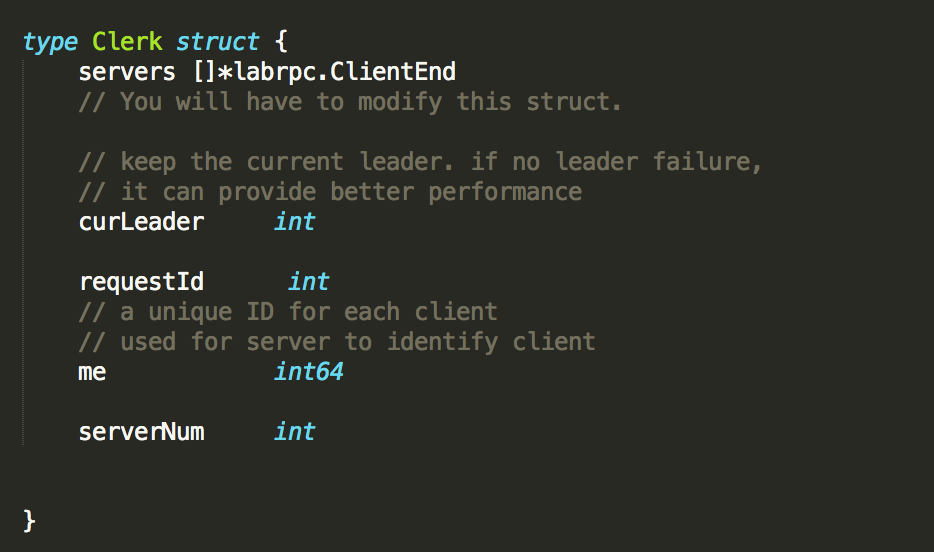
To simplify the system, the server adds the duplicated requests to its log, but will never apply them on its table. The server doesn’t keep the reply for the latest read(get), thus when a duplicated read request comes, the server returns a current value.

Struct

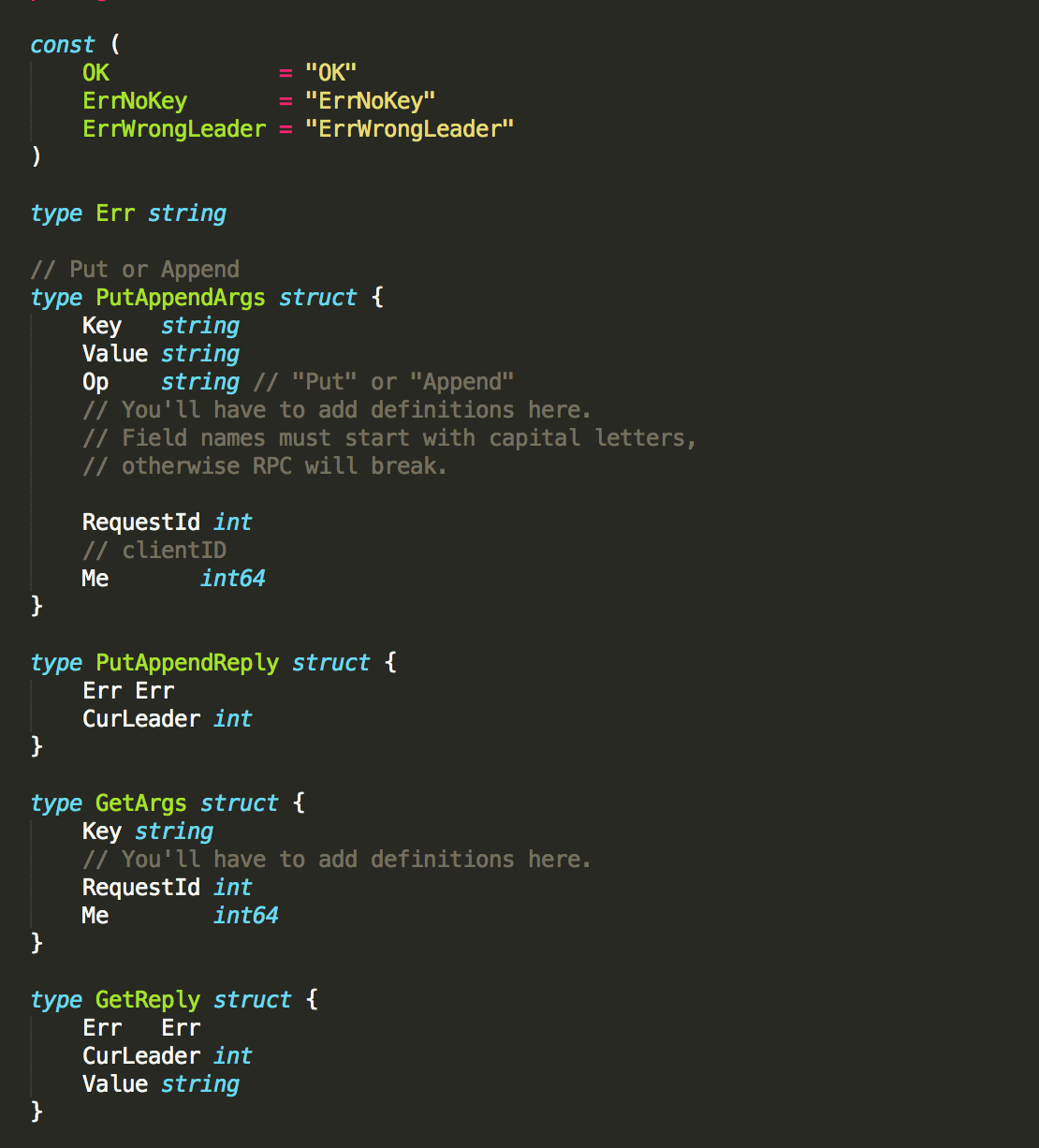
Server side:



Client side:



RPC:

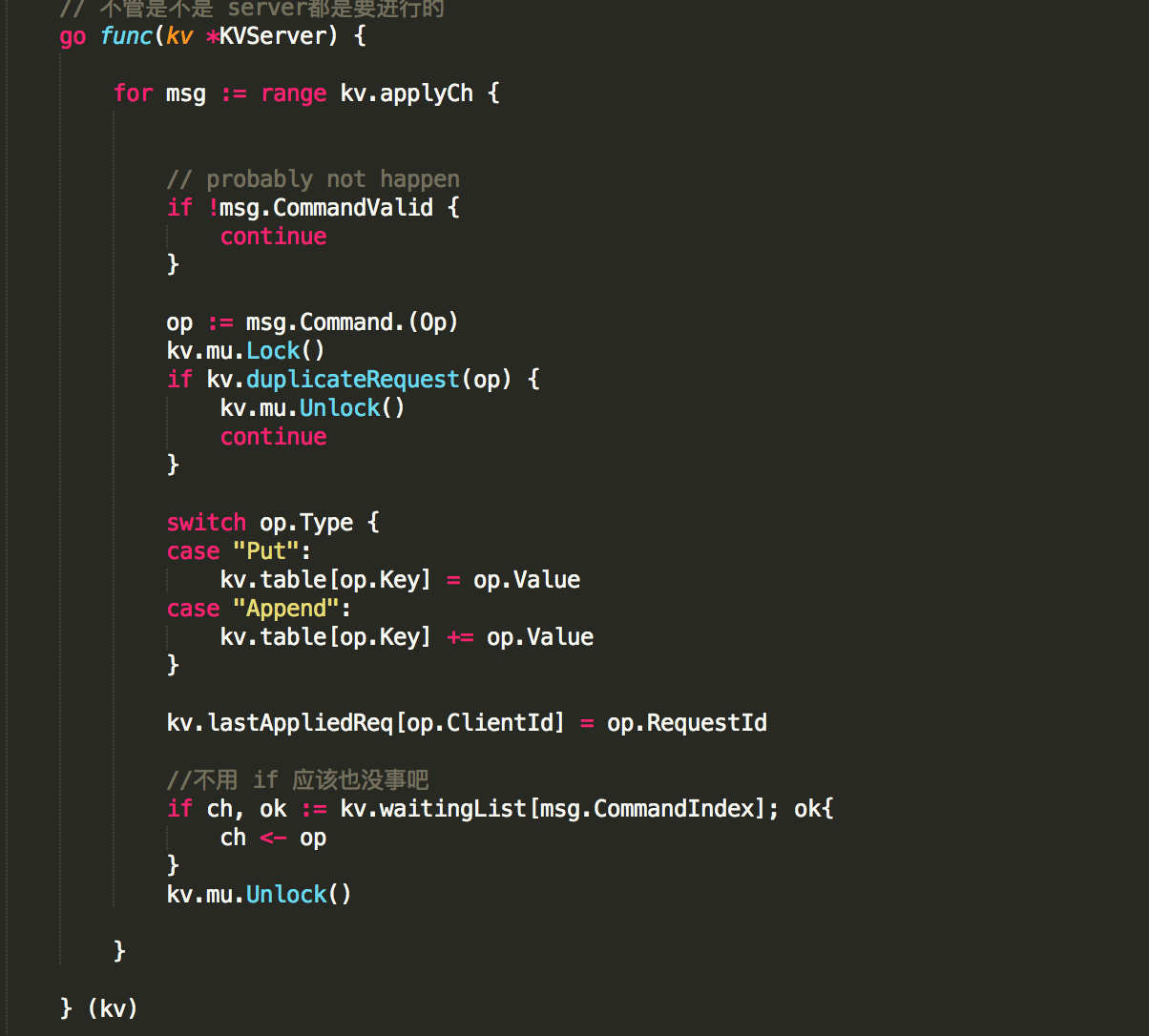


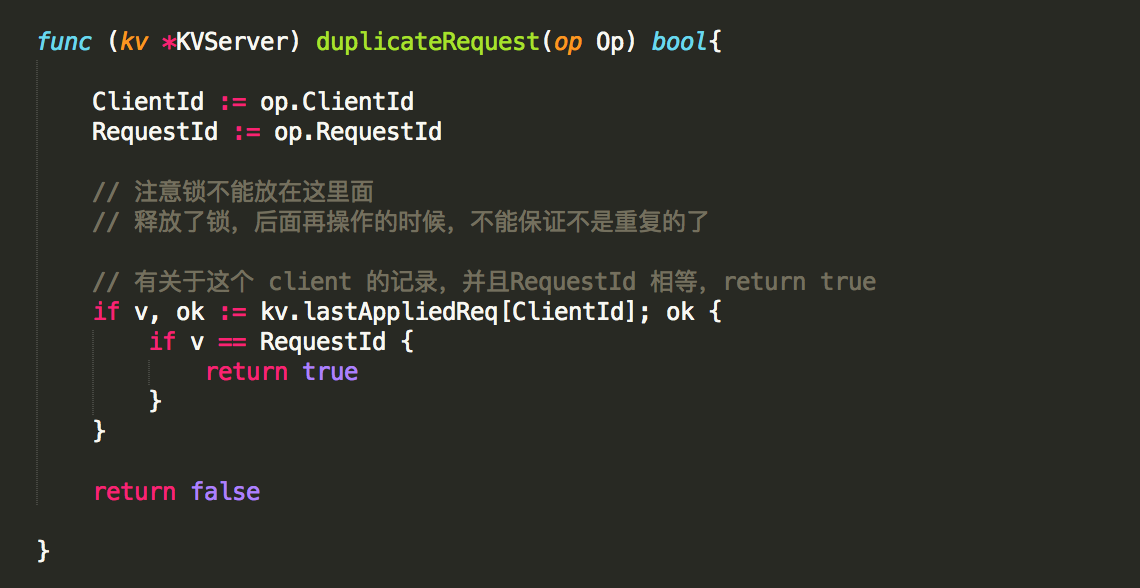
Set up

Server side:

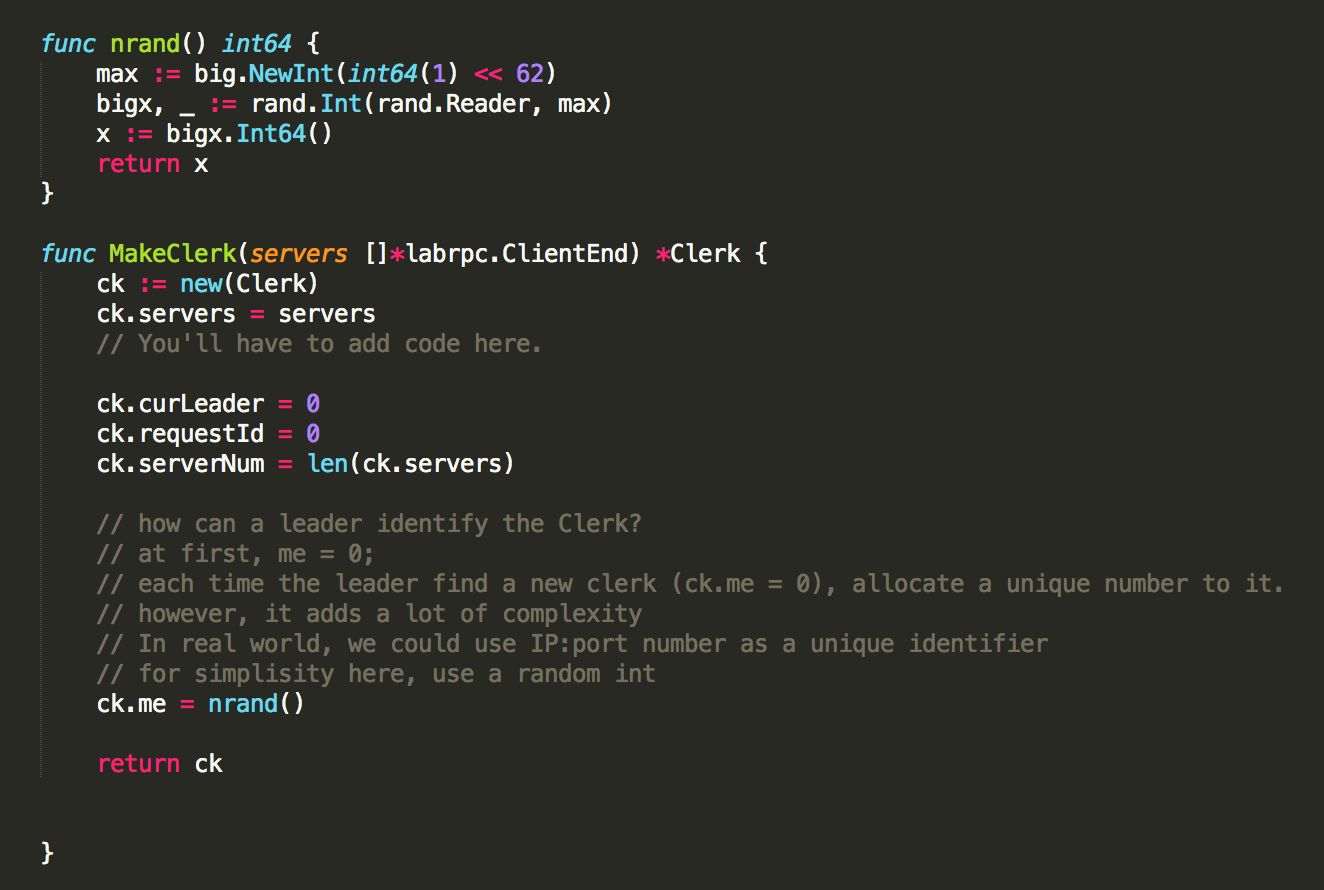


When apply a request, it also informs a control flow that is waiting for the request to be applied. This thread can safely ignore duplicated request, since the server will not create a routine for duplicated request.



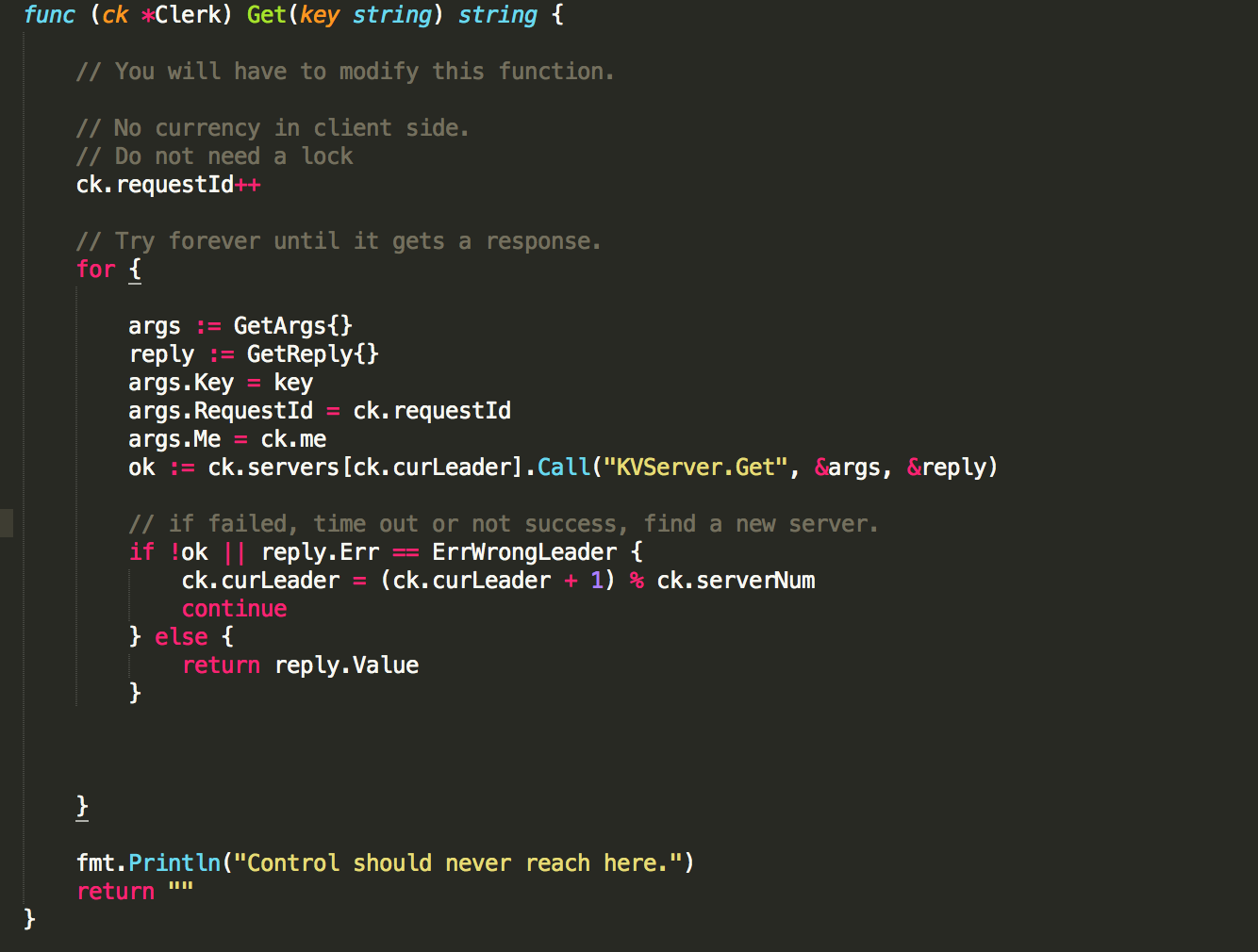


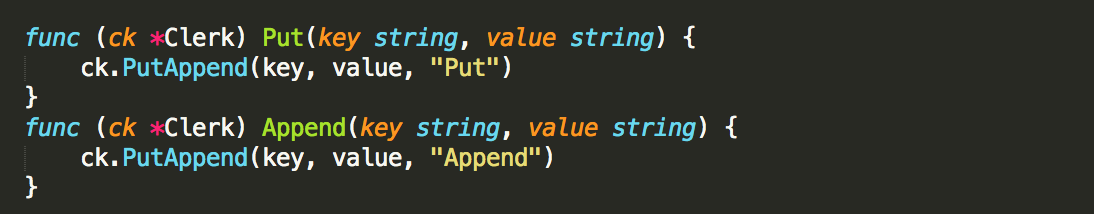
Client side:

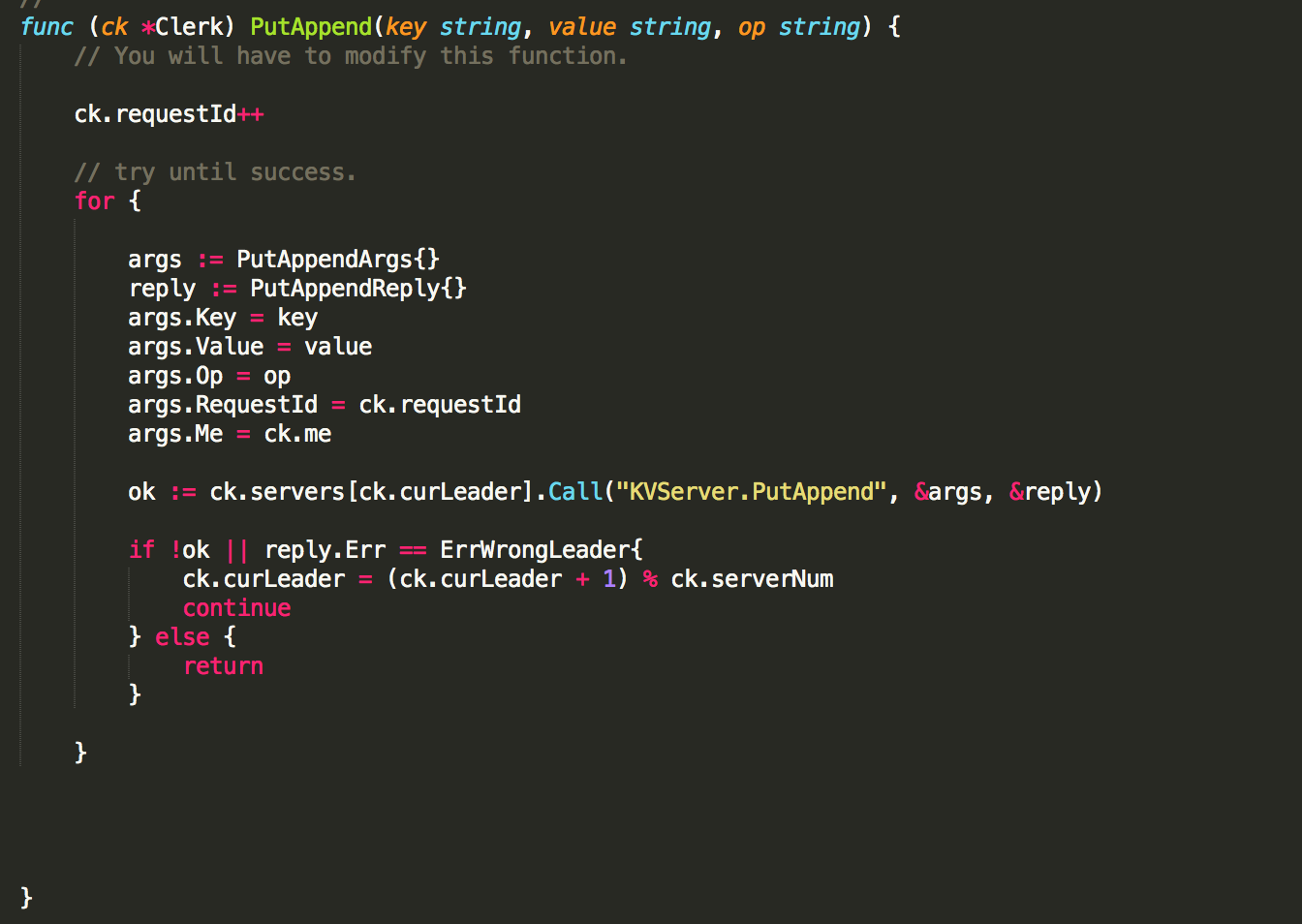


API for applications:

Make request via a Clerk. Clerk will keep trying until succeed.

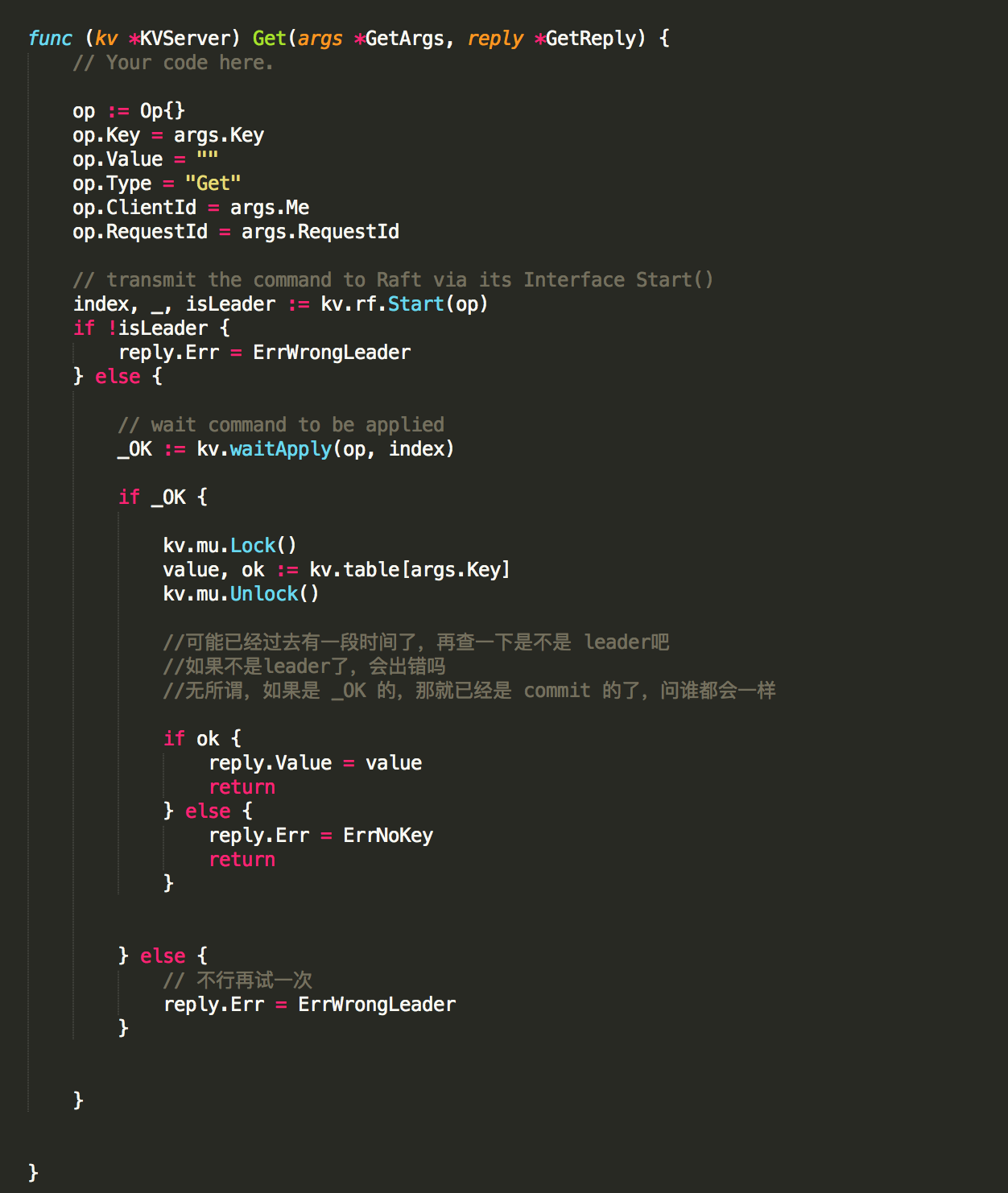


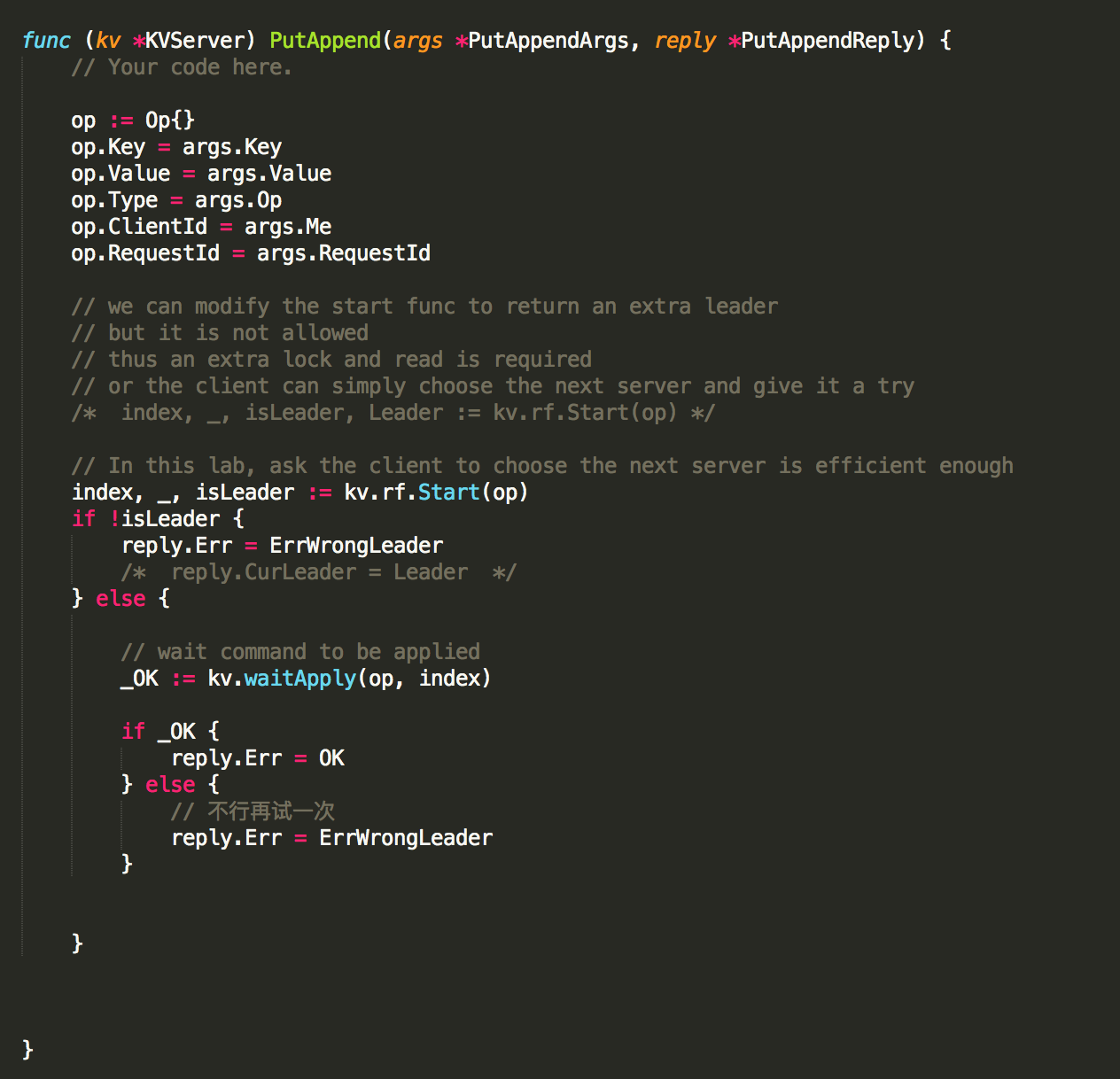




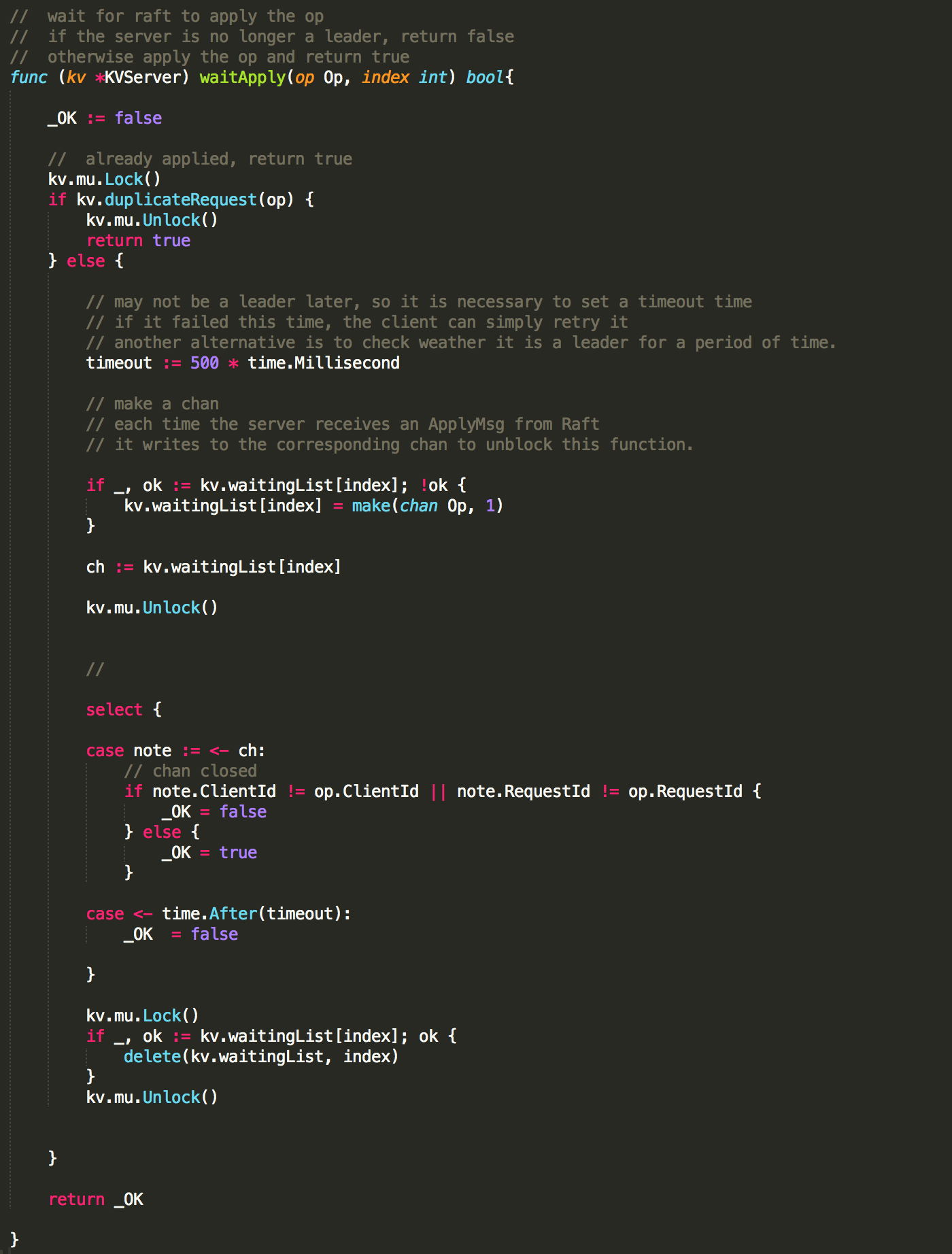
Server implementation:

Server needs to handle RPCs from clerks.





Ignore duplicated request here.



Part 3 Remarks on Raft and a brief introduction to Zookeeper.

Raft is a duplicated system with strong consistency. It’s a linearizable system and provides great fault tolerance. It works as long as the majority of servers can work and communicate with each other.

However, it doesn’t provide better performance. The leader is the only one talk to clients. As we add more servers, the leader needs to communicate with more followers. We are adding more burdens on the bottleneck of the system. More servers, worse performance.

What should we do if we want that: if we n times of servers, then will can get n times, or almost n times of performance? We have to give up the frame: only leader can talk to client.

Zookeeper is another duplicated infrastructure that provides weaker consistency. Clients are only allowed to talk to leader when trying to write data However, they can send a read request to any one of the replicas (Thus providing linearizability only for write commands).

And for each client, it acts in a FIFO manner. If a client sends a request A before a read request B, B is guaranteed to see the state that is as least as up-to-date than the state when server replied request A.

If most of the requests are read, zookeeper can surely provide great performance and not too stale data.