**Introduction**

In this project you'll build two key/value storage systems that "shard," or partition, keys across servers. A shard is a subset of the key/value pairs; for example, all the keys starting with "a" might be one shard, all the keys starting with "b" another, etc. The reason for sharding is performance. Each server handles Puts and Gets for just a few of the shards, and servers assigned to different shards operate in parallel; thus, total system throughput (Puts and Gets per unit time) increases in proportion to the number of servers.

A sharded storage system must be able to shift shards among servers. One reason is that some servers may become more loaded than others, so that shards need to be moved to balance the load. Another reason is that servers may join and leave the system: new servers may be added to increase capacity, or existing servers may go offline.

The main challenge in this project will be handling reconfiguration in the assignment of shards to servers. For example, if a Put arrives at about the same time as a reconfiguration, all servers must agree on whether the Put occurred before or after the reconfiguration. If before, the Put should take effect and the new owner of the shard will see its effect; if after, the Put won't take effect and client must re-try at the new owner.

Reconfiguration also requires interaction among servers. For example, in configuration 10, a group of servers G1 may be responsible for shard S1. In the next configuration 11, another group G2 may be responsible for shard S1. During the reconfiguration from 10 to 11, servers in group G1 must send the contents of shard S1 (the key/value pairs) to servers in group G2.

**Software**

Download the archive [proj4.tar.gz](https://lamport.eecs.umich.edu/eecs491/projs/proj4.tar.gz) and uncompress it to get the skeleton code and tests in paxos, paxosrm, hashkv, common, shardmaster, and shardkv.

Ignore the huge number of "has wrong number of ins" and other miscellaneous warnings.

**Part A: Consistent Hashing**

In this first part of the project, you'll implement a sharded key/value store in src/hashkv. This system uses consistent hashing to partition keys across servers. Key/value data is not replicated in this system. But, we will assume that servers do not die abruptly; instead, servers are always removed gracefully (with the invocation of kill() on a server).

The hash space is divided into a fixed number of shards, and each of the N shards corresponds to (1/N)th of the hash space. Key2Hash and Key2Shard in common/utils.go compute the hash value for a key and the shard that this hash value maps to.

Every server is assigned a unique identifier. The hash value of a server's identifier determines the relative position of this server in the circular hash space compared to other servers, which in turn determines the subset of shards assigned to the server. Every server is responsible for serving data in a continuous sequence of shards that lie between the server's predecessor and itself. Specifically, the range of shards assigned to a server starts at the shard which contains the identifier of the server's predecessor and ends at the shard which contains the server's identifier. This range does not include the shard which contains the identifier of the server's predecessor. The relative ordering between shards or between servers is determined by clockwise traversal of the circular hash space. If the identifiers for two different servers hash to the same shard, the relative ordering between the two servers is determined by the ordering of their identifiers.

In hashkv/server\_impl.go, you'll write code to introduce a new server into the system (InitImpl()) and to remove a server from the system (KillImpl()). When any server is added or removed, only after the addition or removal is complete (i.e., after StartServer or kill returns) will the addition or removal of another server begin. When a server is initialized, it is given the addresses and identifiers of its predecessor and successor in the hash space. In both InitImpl() and KillImpl(), you must have servers appropriately hand off state to each other in keeping with the new assignment of shards to servers. In doing so, you must preserve at-most-once semantics and linearizability, i.e., each Put or Append request is applied to the sharded key/value database at most once, and once applied, the effects of a Put or Append request is visible to all subsequent Gets.

In hashkv/client\_impl.go, you'll implement client-side support for issuing Get, Put, and Append requests. Every client has a list of all servers in the system, but does not know which shards are assigned to each server. A client executes a Get, Put, or Append request on a key by repeatedly issuing the request to different servers until it finds the server to which the key is currently assigned and that server's response makes it back to the client. When a server receives a request for a key that is in a shard not assigned to it, the server returns an ErrWrongServer error.

A few notes:

* When the first server is added to the system, this server will be its own predecessor and successor. Similarly, when the second server is added to the system, the first server will be both the predecessor and successor of the new server.
* When a client receives an error from one server and retries its request at another server, make sure to sleep (e.g., for 100ms) between retries.
* Make sure to garbage collect the server-side state that you maintain to detect duplicate requests.
* To simplify your implementation, it is okay for your hashkv server to leave shard data that it hands off to others undeleted, even though it is no longer responsible for serving requests on that shard.

**Part B: The Shard Master**

The consistent hashing based system from part A has two significant limitations. First, since data is not replicated, the system cannot tolerate failures. A server can only be manually removed from the system by invoking kill() on it. Second, the randomness associated with hashing of server IDs intrinsically leads to load imbalance across servers, with some servers assigned more shards than others.

To overcome these limitations, in parts B and C, you'll implement a different design for a sharded key-value store, which will have two main components. First, servers will be partitioned into a set of replica groups. Each replica group is responsible for a subset of the shards. A replica group consists of a handful of servers that use Paxos to replicate the shards assigned to the group. The second component is the "shard master", a central fault-tolerant service implemented using Paxos. The shard master decides which replica group should serve each shard. Clients consult the shard master in order to find the replica group for any particular key.

This general architecture (a configuration service and a set of replica groups) is patterned at a high level on a number of systems: Flat Datacenter Storage, BigTable, Spanner, FAWN, Apache HBase, Rosebud, and many others. These systems differ in many details from this project, though, and are also typically more sophisticated and capable. For example, this project lacks persistent storage for key/value pairs and for the Paxos log; it sends more messages than required per Paxos agreement; it cannot evolve the sets of peers in each Paxos group; its data and query models are very simple; and handoff of shards is slow and doesn't allow concurrent client access.

In part B, you'll implement the shard master, in src/shardmaster. The shardmaster manages a sequence of numbered configurations. Each configuration describes a set of replica groups and an assignment of shards to replica groups. Whenever this assignment needs to change, the shard master creates a new configuration with the new assignment, and communicates with relevant replica groups to convey changes in shard assignments. Key/value clients and servers contact the shardmaster when they want to know the current (or a past) configuration.

Your implementation must support the RPC interface described in shardmaster/types.go, which consists of Join, Leave, Move, and Query RPCs.

You don't need to implement duplicate client request detection for RPCs to the shard master that might fail or repeat due to network issues. A real system would need to do so, but this project doesn't require it.

The Join RPC's arguments are a unique non-zero replica group identifier (GID) and an array of server ports. The shardmaster should react by creating a new configuration that includes the new replica group. The new configuration should divide the shards as evenly as possible among the groups, and should move as few shards as possible to achieve that goal.

The Leave RPC's arguments are the GID of a previously joined group. The shardmaster should create a new configuration that does not include the group, and that assigns the group's shards to the remaining groups. The new configuration should divide the shards as evenly as possible among the groups, and should move as few shards as possible to achieve that goal.

The Move RPC's arguments are a shard number and a GID. The shardmaster should create a new configuration in which the specified shard is assigned to the specified group. The main purpose of Move is to allow us to test your software, but it might also be useful to fine-tune load balance if some shards are more popular than others or some replica groups are slower than others. A Join or Leave following a Move will likely un-do the Move, since Join and Leave re-balance.

The Query RPC's argument is a configuration number. The shardmaster replies with the configuration that has that number. If the number is -1 or bigger than the biggest known configuration number, the shardmaster should reply with the latest configuration. The result of Query(-1) should reflect every Join, Leave, or Move that completed before the Query(-1) RPC was sent.

The very first configuration should be numbered zero. It should contain no groups, and all shards should be assigned to GID zero (an invalid GID). The next configuration (created in response to a Join RPC) should be numbered 1, etc. There will usually be significantly more shards than groups (i.e., each group will serve more than one shard), in order that load can be shifted at a fairly fine granularity.

Your shardmaster must be fault-tolerant, using your Paxos and PaxosRSM libraries from Project 3. You are free to modify your Paxos and PaxosRSM implementations from Project 3.

Here are a few hints/tips:

* Start with a stripped-down copy of your KVPaxos server.
* One difference in your use of PaxosRSM, compared to project 3, is that PaxosRSM needs to be initialized not only with an applyOp method (which it invokes on decided values) but also requires an equals method (to compare any two decided values).
* Go maps are references. If you assign one variable of type map to another, both variables refer to the same map. Thus if you want to create a new Config based on a previous one, you need to create a new map object (with make()) and copy the keys and values individually.
* In your implementation of the shardmaster, if you need to use any data types that you will also need in your implementation of shardkv in part C, then define such data types in common/types\_impl.go to prevent circular dependencies.
* While Part C will require your shardmaster to communicate changes in the configuration to shardkv servers, you should be able to pass all of the test cases for Part B without doing so.
* After you complete part C, to pass the test cases for part B, you will have to comment out RPCs to ShardKV servers within your shardmaster code. When the autograder runs the part B test cases, it will ensure that all RPCs to ShardKV servers return true without actually executing the RPC. An implication of this is that, when your shardmaster issues an RPC to any ShardKV server, it should only need to know whether the RPC was successful and not have to inspect any values in the reply parameter of the RPC.

**Part C: Sharded Key/Value Server**

Now you'll build shardkv, a sharded fault-tolerant key/value storage system. You'll modify client\_impl.go, rpcs\_impl.go, and server\_impl.go in src/shardkv. You'll also modify your shardmaster implementation from part B so that, when serving Join, Leave, and Move RPCs, the shardmaster communicates the changes in shard assignments to the relevant replica groups.

Each shardkv server will operate as part of a replica group. Each replica group will serve Get/Put/Append operations for some of the key-space shards. Use Key2Shard() in common/utils.go to find which shard a key belongs to. Multiple replica groups will cooperate to serve the complete set of shards. When the shardmaster informs shardkv servers of changes in shard to replica group assignments, replica groups will have to hand off shards to each other without going through the shardmaster.

Your storage system must provide single copy semantics to applications that use its client interface. That is, completed application calls to the Clerk.Get(), Clerk.Put(), and Clerk.Append() methods in client\_impl.go must appear to have affected all replicas in the same order. A Clerk.Get() should see the value written by the most recent Put/Append to the same key. This will get tricky when Gets and Puts arrive at about the same time as configuration changes. The recommended approach is to have each replica group use Paxos to log not just the sequence of Puts, Appends, and Gets but also the sequence of reconfigurations. When a replica group receives a request for a key that is in a shard not assigned to it, it returns an ErrWrongGroup error.

Your implementation must also ensure that, when the shardmaster responds to a Join, Leave, or Move RPC, the reconfiguration caused by the operation is already complete. In other words, once the shardmaster finishes serving any Join, Leave, or Move RPC, every replica group must be ready to serve Get, Put, and Append requests on shards that were assigned to that group in the new configuration.

You will need to ensure that at most one replica group is serving requests for each shard. Luckily it is reasonable to assume that each replica group is always available, because each group uses Paxos for replication and thus can tolerate some network and server failures. As a result, your design can rely on one group to actively hand off responsibility to another group during reconfiguration. This is simpler than the situation in primary/backup replication (Project 2), where the old primary is often not reachable and may still think it is primary.

You are allowed to assume that a majority of shardkv servers in each replica group are alive and can talk to each other, can talk to a majority of the shardmaster servers, and can talk to a majority of servers in other replica groups. Your implementation must operate (serve requests and be able to re-configure as needed) if a minority of servers in some replica group(s) are dead, temporarily unavailable, or slow.

Unlike Part B, you do need to detect duplicate client RPCs to the shardkv service in Part C. Please make sure that your scheme for duplicate detection frees server memory quickly, for example by having the client tell the servers which RPCs it has heard a reply for. It's OK to piggyback this information on the next client request. Here are a few hints/tips to keep in mind:

* For any data types that you need both in shardmaster and in shardkv, add the definitions of such data types in common/types\_impl.go.
* Your server should not automatically call the shard master's Join() or Leave() handlers. The tester will call Join() or Leave() when appropriate.
* To pass the concurrent test cases, you must design a correct protocol for handling concurrent operations in the presence of configuration changes.
* When a test fails, check for gob error (e.g. "rpc: writing response: gob: type not registered for interface ...") in your output, because Go doesn't consider the error fatal, although it is fatal for the project.
* Be careful about implementing at-most-once semantics. When a server transfers shards to another, the server also needs to send the state that it maintains to detect duplicate requests. Think about how the receiver of the shards should update its own state. Is it ok for the receiver to replace its state for detecting duplicate requests with the received one?
* Think about how should the shardkv client and server deal with ErrWrongGroup. When the client receives an ErrWrongGroup response to one of its requests and retries its request, should the client retain or change the request identifier that it includes in the request? On the server-side, when returning ErrWrongGroup to a client's request, should the server add this request's identifier to the cache that the server maintains to detect duplicate requests?
* Like in Part A, after a server has moved to a new configuration, it can leave the shards that it is not owning in the new configuration undeleted. This will simplify your implementation.
* Think about when it is ok for a server to transfer shards to another server during a configuration change.

**Handin procedure**

Clone the repository that we have created for your group on GitHub.

When you submit your project to the [autograder](https://lamport.eecs.umich.edu/eecs491/submit.php?4), it will pull the following files from your repository:

* hashkv/client\_impl.go
* hashkv/rpcs\_impl.go
* hashkv/server\_impl.go
* paxos/paxos\_impl.go
* paxos/rpcs\_impl.go
* paxosrsm/server\_impl.go
* common/types\_impl.go
* shardmaster/server\_impl.go
* shardkv/client\_impl.go
* shardkv/rpcs\_impl.go
* shardkv/server\_impl.go

So, please ensure that a) your repository has directories called hashkv, paxos, paxosrsm, common, shardmaster, and shardkv containing these files, and b) all modifications that you make to the code handed out are restricted to only these files.

Among the unit tests included in the handout, if you find that you pass some of them locally on your computer but not on the autograder

* Check that you have only modified the \*impl.go files
* Check that you pass those tests on CAEN
* Run the tests repeatedly and make sure you pass them every time
* If you still pass all runs of the test cases, post privately on Piazza

In addition to your score on the autograder, 5% of the 20 points for this project will be based on hand grading. Among all of your submissions with the highest score on the autograder, we will use your last submission for hand grading. We will use the timestamp of your **last** submission for the purpose of calculating late days.