CS 162
Project 3: Key-Value Store
Design Document

Fill in the following information:

GSI:
Group Number: 102
Andrew Blatner <ablatner@berkeley.edu>
Bryan Tran <btacju@gmail.com>
Noah Ruderman <noahc66260@berkeley.edu>
Kevin Lam <kel018@berkeley.edu>

1) Please write down any miscellaneous comments or notes for your GSI.

KVCache and KVCacheSet
2) Copy here the declaration of each new or changed `struct' or
`struct' member, global or static variable, `typedef', or
enumeration for KVCache and KVCacheSet. Briefly identify the purpose of each.

```
/* A KVCacheSet. */
typedef struct {
  unsigned int elem_per_set;
  pthread_rwlock_t lock;
  int num_entries;
  struct UT_hash_table hash_table;
  struct kvcacheentry *eviction_queue; // for our FIFO replacement policy,
queue as doubly linked circular list, using UTlist
} kvcacheset_t;

struct kvcacheentry {
   struct kvcacheentry *next; // for replacement policy, DLC UTlist
   struct kvcacheentry *prev; // for replacement policy, DLC UTlist
}
```

3) How will you ensure that cache accesses are very fast while also implementing the second-chance replacement policy?

Let n be the cache capacity and let k be the capacity of each cache set.

The first thing we can do is have each cache entry contain a next and prev pointer to other cache entries to form a circular queue. New cache entries are added right before the position of the next item to evict in this queue. Adding, removing, or replacing an element in a circular doubly linked list is an O(1) operation, as the cache set has bounded size, so this adds only constant-time overhead to the operations we perform on the cache. This way cache accesses are bounded at O(k) (because in the worst case we must inspect each element in the cache set), and a cache miss followed by an eviction is also O(k).

Socket Server and Networking
4) Copy here the declaration of each new or changed `struct' or
`struct' member, global or static variable, `typedef', or
enumeration for networking. Briefly identify the purpose of each.

No additional declarations are necessary for socket server and networking.

5) To make server_run handle multiple requests simultaneously, you will need to create new threads. However, there is a limit on the number of threads you can have running at any one time, so you cannot just fork a new thread for each incoming request. How will you achieve this?

We will have a pool of threads created before we handle any requests. When we need to handle a request, we will use one of the available threads to perform the task. If all threads are busy, the a new task will be unable to be executed until some task currently running finishes and frees its associated thread.

6) Why can you not just fork a new process for every request? What are the two failure modes when this occurs? Note that we are asking about forking new processes rather than spawning new threads (which is what is actually done in the project).

If we simply forked a new process, there would be much more overhead than there is for same number of multithreaded requests. Each process would also have its own address space, so it would be difficult to have shared state between the server and its request handlers. Locks and CV’s need the same address space to ensure concurrency. The first failure mode for fork is EAGAIN, for when the system could not create the child process due to resource limits or self-imposed limits. The other failure mode is ENOMEM, for when insufficient storage space is available.

7) Describe how you might test if your socket server's server_run function was working correctly

(calling the appropriate handle method of kvserver/tpcmaster).

We could put a print statement inside of handle_slave and handle_master which prints the name of the function and the hostname of the server. Since hostnames are unique there will be no ambiguity which server is calling which functions due to calling the server_run function. Alternatively, we could use the callback parameter of server_run to print this information as well.

Single Server Key-Value Store
8) Copy here the declaration of each new or changed `struct' or
`struct' member, global or static variable, `typedef', or
enumeration for the single server kv-store. Briefly identify the purpose of each.

```
typedef struct wq {
   ...
   pthread_mutex_t wq_mutex;
   pthread_cond_t wq_cond; // Is the queue empty? If so we must block threads.
} wq_t;

/*
Here create each thread and run the task
while (1)
  tp->mutex.acquire();
  num_avail++;
  while (NULL == cur_msg)
    tp->cond.wait(&lock);
  getTask(); // Also sets cur_msg to NULL
  num_avail--;
  tp->mutex.release();
  runTask();
// This would be part of the method each thread runs

When adding a task the code is something like
tp->mutex.acquire();
while (num_avail == 0)
  tp->cond.wait(&lock);
tp->cur_msg = cur_msg;
tp->cond.signal();
tp->mutex.release();

Basically threads are waiting. When a task is available exactly one thread is
woken up which acquires the lock, acquires the task, releases the lock, and
then executes the task.
*/
struct threadpool {
  int num_avail;
  pthread_t threads[MAX_THREADS]
  pthread_mutex_t tp_mutex;
  pthread_cond_t tp_cond;
  kvmessage_t *cur_msg;
};
```

9) Describe at least one end-to-end test you will add. What is the scenario you will test and how will you achieve this in the test?

A test we can add is we can first PUT a key. We can then attempt to GET the key numerous times. This checks to make sure that there are no memory errors with how the values are stored and makes sure that the server can work over time and not just the first time we GET.

Distributed Key-Value Store
10) Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration for the distributed kv-store. Briefly identify the purpose of each.

```
typedef struct tpcslave {
      ...
  kvmessage_t *cur_msg; // Message for current TPC operation
  kvserver_t server;
      ...
} tpcslave_t;

struct kvcacheentry {
      ...
 int version_num; // Current version number of the requested data to use when
                  // handling crashes
      ...
}
```

11) Discuss, in your own words (no copy-pasting), what you will do to handle a PUT request within TPCMaster.

When the TPCMaster receives the PUT request, it will send out the requests to the relevant slave nodes. The slaves will first log this request before searching their kvstores using the key and value -- checking that the data is correct. If it is, the slaves will send a VOTE_COMMIT message back to the master. Otherwise it will send a VOTE_ABORT message. In the master, if any VOTE_ABORT messages are received, it will send ABORT messages to all the relevant slaves which get logged. The slaves resend an acknowledgement which ends the request. This also applies in the case that the master doesn't receive back the amount of votes equal to the number of requests it sent out. Any node that doesn't send a vote within the timeout period will be counted as an VOTE_ABORT. However, if all of the relevant nodes send back a

VOTE_COMMIT, the master will then send out a global COMMIT back to those nodes which gets logged. The key and value which was temporarily stored in the node will then be committed to the kvstore. The kvcache will need to be updated to reflect these changes.

12) Explain, at a high level, how you will modify kvserver_t to allow you to handle two-phase commits.

Each kvserver_t will store the message of the current two-phase commit operation while it waits for the vote result. We will store the request in memory and create a log once we have received a global commit. If we receive a global abort, we just discard this information and do not write any log.

13) Explain, at a high level, how you will implement kvserver_rebuild_state. How will you deal with the situation where a COMMIT message was put into the log following a PUT or DEL request, but it was not actually committed?

When the kvserver comes back up from a crash, it will check the last message in its log using tpclog_iterate_hasnext and tpclog_iterate_next. If the message type was of PUTREQ or DELREQ, when the node restarts, it will send out its VOTE_COMMIT or VOTE_ABORT depending on the state of the requested data (like if it is corrupted or not). If the type of the last message is ABORT, it will end the request as soon as the node comes back. If the message is COMMIT, the kvserver will need to check the version number of data requested. If the versions are the same, it means that the changes have already been committed. If the version of the stored data is lower than that of the data to be committed, it means that the changes have not yet been committed and it needs to be committed. Then the acknowledgement will be sent back to the master.

14) In this project, we did not ask you to consider that TPCMaster might ever fail. What would you need to add to handle this type of failure?

In the case that the TPCMaster fails, extra redundancy will be needed to ensure that the client is able to complete its request. (Somehow detect that the TPCMaster is down. Maybe a timeout.) Slave nodes should have a copy of the key and values. The client will send a request to the slave which will then send commit operations to the other nodes.

We could also have a helper process for TPCMaster that can restart it if it fails. The helper and TPCMaster can periodically signal each other that they are still alive. The helper possibly could have a copy of TPCMaster's work queue so that requests are not lost.

15) This project is optimized for a read heavy system (since writes are so slow). How would you modify this project (while keeping all the slaves) for a write heavy system?

We wouldn't implement the write-through policy since it would require a disk access for most accesses (since most operations are writes). We could either implement a write-around policy (bypassing the cache) or a write-back policy (don't write to disk immediately). The downside to the first choice is that it would still involve a lot of I/O. The downside to the second choice is that a cache failure will mean losing data. That is, we lose durability. But upside to the second choice is that writes will be extremely fast, since no I/O will be involved (until cache eviction).

Maybe we would also have more possible places to store a given value. That is, more buckets in our cache, possibly to the point where our cache is fully associative, so any two writes could take place simultaneously (assuming there is space). For a fully associative cache, we can speed up our performance by a factor of as many threads as are available if the hardware can support it.

We could also change the cache replacement policy to require as little processing as possible. Something like FIFO. This would only be useful if the time spent on the replacement policy could be used to do other useful tasks.