Design Document: asgn4

Husain Adam Askari

Cruzid: haskari

1 Goals

The goal of this assignment is to implement a persitent link between httpnames and Alias names in the Key Value Store.

2 Changes from assignment 3

Realized a bug in my update entry in my kvs store where I left my offset_entry field unintialized when updating an entry. So if I update an entry with that was already updated, errors would occur.

3 Removing slashing

If the first character of the request is a '/' then we remove the slash, else, we leave it be. Already did a similar thing in assignment 1.

4 Patch Command

So our server will now recognize a PATCH command along with get and put. Alias will now map an httpname to an alias name which can be used on GET requests. The struct we will use to write into the map file so that we can have a persistent hashtable will have the httpname and the alias name. They will contain 128 bytes combined. If the existing name, whether it be an httpname or an alias name doesn't exist, then we throw an error. A Patch command looks like in the client

```
snprintf(patch, 4096, ”PATCH httpname HTTP ALIAS httpname new name”);
send(sock, patch, strlen(patch));
```

4.1 Server handle

```
if strcmp(response, ”PATCH”) == 0 then
    char* httpname;
    char* alias;
    char httpname[128];
    char aliasarray[128];
    httpname = strtok_r(NULL, ” ”, saveptr1);
    remove lead slash from httpname
    alias = strtok_r(NULL, ” ”, saveptr1);
    strncpy(httparray, httpnae, strlen(httpname));
    strncpy(aliasarray, alias, strlen(alias));
    if (httparray[0] == ’/’) then
        httpname = strtok_r(alias, ”/”, saveptr2);
    end
    if aliasarray[0] == ’/’ then
        alias = strtok_r(alias, ”/”, saveptr2);
    end
    /*Check if the alias already exists, for updating*/
    if search(alias) then
        update(alias, httpname);
    else
        if kvsinfo(httpname, -1) == -2
        send token 404;
    end
    insert_map(alias, httpname);
end
send(sock, patch, strlen(patch));
```

An alias entry looks like this

```
struct name entry
char [128] alias
char[128] key;
int16_t offset_entry
```

Algorithm 1: map entry

If the string lengths of the key and alias are greater than 128, return a FAIL, If the key name exists and makes sure the length of the two does not surpase the 128 character limit, return a SUCCESS. Multiple aliases can point towards the same httpname as well but can also be overwritten.

5 Key Value Store

-m will dictate the name of the name file store that we will use to store all of our names with. If there is no map file specified, then our program will forcefully exit. The reading and writing from the file will be very similar to the assignment 3 methods where we have a persistent hashtable that can be accessed when rebooting the server.

```
Input 1: ssize_t fd
Input 2: map_entry entry
    alias_insert(entry); if pwrite(fd, entry,
     $sizeof(map_{e}ntry), alias_{s}een * (map_{e}ntry))$¡0 then
        perror("Alias write error");
          return;
    end
    alias_seen;
```

Algorithm 2: Writing an entry

```
Input 1: ssize_t fd
Input 1: map_entry new entry
Input 2: map_entry old entry
    if pwrite(fd, entry, $sizeof(map_{e}ntry), alias_{s}een * (map_{e}ntry))$¡0 then
        perror("Alias write error");
          return;
    end
```

Algorithm 3: Updating an entry

5.1 Hashing

The hashing will be similar the one in assignment 3. We will preallocate 10000 structs inside the map file and update the hashtable anytime a new entry comes along.

```
    for i to SIZE do
        map[i] = (map_entry *)malloc(sizeof(map_entry));
          map[i]→ $httpname = "NULLCHARACTER"$; end
```

Algorithm 4: Filling the hash table on startup of new file

This function will check if an httpname exists for an alias, if it does return true or SUCCESS, if not, return failure and return a 404 error.

```
Input 1: struct map_entry
if strcmp(map_entry→ httpname, map[key]− > httpname) == 0 then
    return SUCCESS;
else
    /*while there are no spots to the right open, loop*/
    while (
    strcmp(map_entry→ httpname, map[key]− > httpname)! = 0) do
        key = (key+1) mod MAPSIZE
        count++;
        /*if every spot in the hashtable is full*/
        if count == total_entries or map[key] != NULL then
            return FAILURE;
        end
    end
    return SUCCESS;
end
```

Algorithm 5: search hash function

```
Input 1: map_entry
allocate memory for entry;
int key = hash_function(string);
int count = 0;
/*If nothing occupies this spot, take it*/
/*Or if the hashtable contains same key and block number, update it
with new data*/
if strcmp(map[key] → httpname, "NULLCHARACTER") == 0) then
    hash[key] == kvs_entry;
else
    /*while there are no spots to the right open, loop*/
    while
    strcmp(map[key]→ httpname, "NULLCHARACTER")! = −0 do
        key = (key+1) mod SIZE
        count++;
        /*if every spot in the hashtable is full*/
        if count == total_entries then
            return;
        end
    end
    map[key] == map_entry
end
```

Algorithm 6: insert hash function

```
Input 1: struct map_entry
  if strcmp(map_entry→ alias, map[key]− > httpname) == 0 then
    return map[key] → httpname; else
      /*while there are no spots to the right open, loop*/
      while (strcmpy(map[key] → alias, map_entry → alias)! = 0) do
        key = (key+1) mod MAPSIZE
        count++;
        /*if every spot in the hashtable is full*/
        if count == total_entries or hash[key] != NULL then
          return map[key] → httpname; end
        end
        return map[key] → httpname; end
```

Algorithm 7: GET hash function

```
Input 1: map_entry
Input 2: fd
  allocate memory for entry;
  int key = hash_function(string);
  int32_t count = 0;
  /*Similar checking of entries as get*/
  if strcmp(map[key] → alias, update_entry → httpname) == 0) then
    update_entry → offset = map[key] → offset;
    kvs_update_entry(fd, update_entry, map[key]);
    map[key] = update_entry;
    returnhash[key] → offset; else
      /*while there are no spots to the right open, loop*/
      while strcmp(map[key] → alias, update_entry → alias)! = 0) do
        key = (key+1) mod SIZE
        count++;
        /*if every spot in the hashtable is full*/
        if count == SIZE then
          return 0;
        end
      end
      update_entry → offset = map[key] → offset;
      kvs_update_entry(fd, update_entry, map[key]);
      map[key] = update_entry;
      returnmap[key] → offset; end
```

Algorithm 8: Update hash function

5.2 Looking for Alias

We will have a recursive Alias look up function. Once we found a name that is 40 characters long, we know we have found an httpname. If we have searched for a name for more than 8 times, we can exit so we don't fall into an infinite loop.We also need to make sure this is thread safe since we are using a global variable to counter the number of recursions.

```
Input 1: Char array name
    pthread_mutex_lock(lock);
    char* httpname = name_resolve(name);
    pthread_mutex_unlock(lock);
    return httpname;
```

Algorithm 9: Look up

```
Input 1: Char array name
    counter++;
    if strlen(name) == 40 then
        counter = 0;
        return name;
    end
    if counter > 8 then
        counter = 0;
        return NULL;
    end
    if search(name) == false then
        char* alias = get(name);
        count++;
        name_lookup(alias);
    else

    end
    return NULL;
```

Algorithm 10: Recursive name look up

6 Dispatch changes

6.1 GET/PUT

```
    if strlen(httpname) != 40 then
        httpname = lookup(httpname);
    end
    if httpname == nullptr then
        send(404) error;
    end
```

Algorithm 11: Look up