Design Document

Because each peer should be both a server and a client. So I created two thread functions server_talk() and peer_talk() serving as server and client.
As a server, it accepts queries from other peers, checks for matches against its local hash table, and responds with corresponding results. To match such requirements, every time the process is initialized, it could firstly serve as a distributed hash table server point with a specific port. If we want it to act as a client, we can type “conn” to server thread, then it will invoke the client thread to server as client.

In server_talk(), I created a TCP socket and set the socket option to SO_REUSEADDR and bing the assigned port. As we know, there are many clients will connect to this server. So I define a customized struct htpeer, it has socket fd, client's name , the IP address and massage data. I chose select() fuction to do with the client's requests and the input and output in standard I/O. Set STDIN_FILENO and listen fd into fd_set allfs.

In the server thread while loop, if listen fd is read ready, we can accept new connection and assign the new connectfd to a new htpeer client struct. If total clients is equal to 1024, we will print a warning, then set the maxfd to the maximum file descriptor.

FD_ISSET(STDIN_FILENO, &rdfs) means if we can type some commands to server thread, such as quit as a server peer or conn to act as a client.

As the server's public services, at last, we need to handle the clients' requests. Parameter maxi is used to store the connecting client's number. After checking client[i].fd, server read buffer from the socket, if the buffer does contain “quit”, so that server will close the corresponding client's socket and FD_CLR from the fd_set allfs. Otherwise, server could create a new thread with a structured argument procli_info with the server's hash table, the received buffer string, and the client's connection socket to allow process_cli() to do with the hash table operation.

In process_cli() function, we use sscanf() to identify client's command and invoke the hash table's general interfaces. For example, if client type “put 10000 string10000”, server knows the key data style is integer and invoke add_str_by_int() and write the function's result back to client. As well, if client type “get 10000”, server knows the key data style is integer and invoke get_str_by_int() and write the function's result back to client. To be mentioned, if the result is not HASHOK, we need to return the error code, if it is, we return the real string value. del_by_int() is used to delete hash entry in integer key. For this assignment, I do not complete other hash table public interfaces for distributed hash table.

As a client, it provides interfaces through which users can issue queries and view search results, so I created peer_talk() function. At the beginning, I invoke the read_cfg_file(), open the servers.cfg file and initialize different hash table servers every line. If we do not allow the server and client run in the same machine, we could enable HASHTEST to prevent the

conflict, at the end of read_cfg_file(), it return the total server count. For these servers, client create an iterator to connect with them. In the while loop, I also use select to do with all file descriptors. For input from STDIN_FILENO, we read commands into buf, and identify them. If it is put, get or del, then I invoke the hashInt() to calculate the str_key for this assignment. When I use hashInt(), reference to the public hash algorithm from stackoverflow. For string key, I will enhance it and support other features in next iteration. After writing command to buffer and checking the result, at last, it use for(;;) loop to check every server fd, and read the result, check if the server quit or invoke translate_result() function to translate the result.

static inline void timersub() function is used to help me get the put/get/del average time cost.

For the basic hash table functions, I research from the github and other reasonable APIs for table CRUD operations. Struct csHashEntry is used for table hash node, it has key union in string/double/int value, enum valtag in Pointer/Numeric/String, and value union string/double/int/pointer, and the linked pointer connecting to next csHashEntry. CsHashTable owns pointers array point to many buckets. The integer buckets and bucketsinitial are used to store the current buckets num. If we are testing in multithreads, I define volatile int array of locks to lock a bucket and the entire table.

create_hash()/delete_hash() are used to create or delete hash table. add_str_by_int()/add_dbl_by_int()/add_int_by_int()/add_ptr_by_int()/get_str_by_int()/get_int_by_int()/get_dbl_by_int()/del_by_int() are designed to put key-value pair with integer key in hash table, get the value or result of specific key, del the key-value pair in hash table. add_str_by_str()/add_dbl_by_str()/add_int_by_str()/add_ptr_by_str()/get_str_by_str()/get_int_by_str()/get_dbl_by_str()/del_by_str() are designed to put key-value pair with string key in hash table, get the value or result of specific key, del the key-value pair in hash table.

In the implementation of these functions of hash table, especially for adding and deleting because we need to add a hash entry or delete, we should use the struct csHashTable's locks to lock the bucket against changes, referred to multithreaded simple data type access and built-in functions atomic variables, I choose __sync_lock_test_and_set(), set &table→locks[hash] to 1function and replace with pthread_mutex for improving performance. After completing adding, updating or deleting, use __sync_synchronize() set memory barrier and reset table->locks[hash] to 0 for coming access.