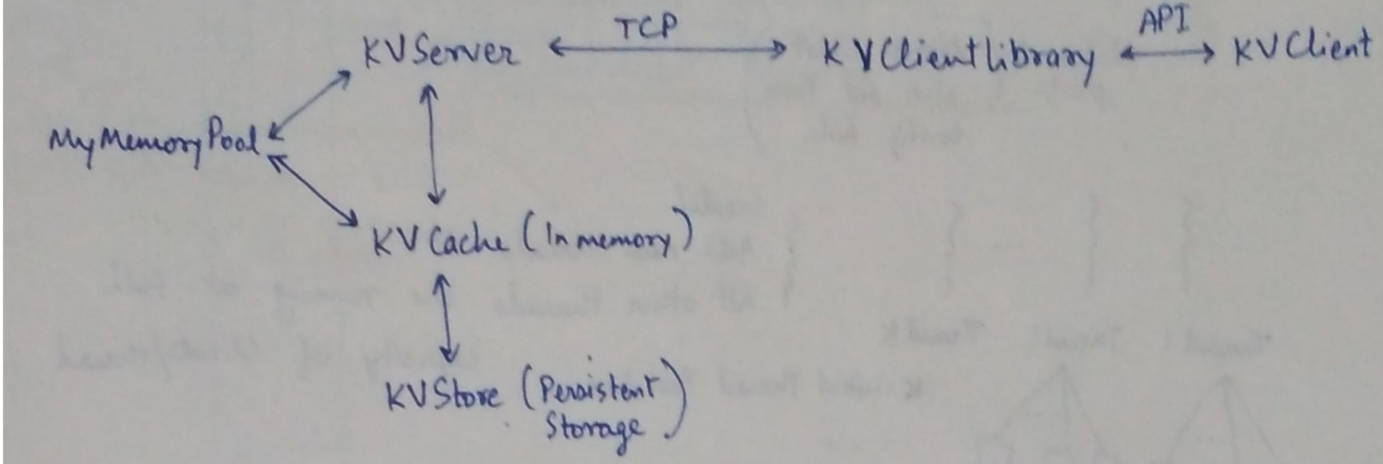
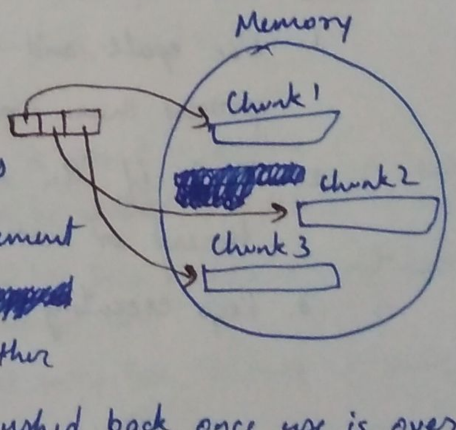
Multi Threaded Key Value Store in C++



KVMessage - Client and Server communicate using the message format defined in this header.
- Used by KVServer, KVCache, KVStore, KVClientLibrary, KVClient
- Structure

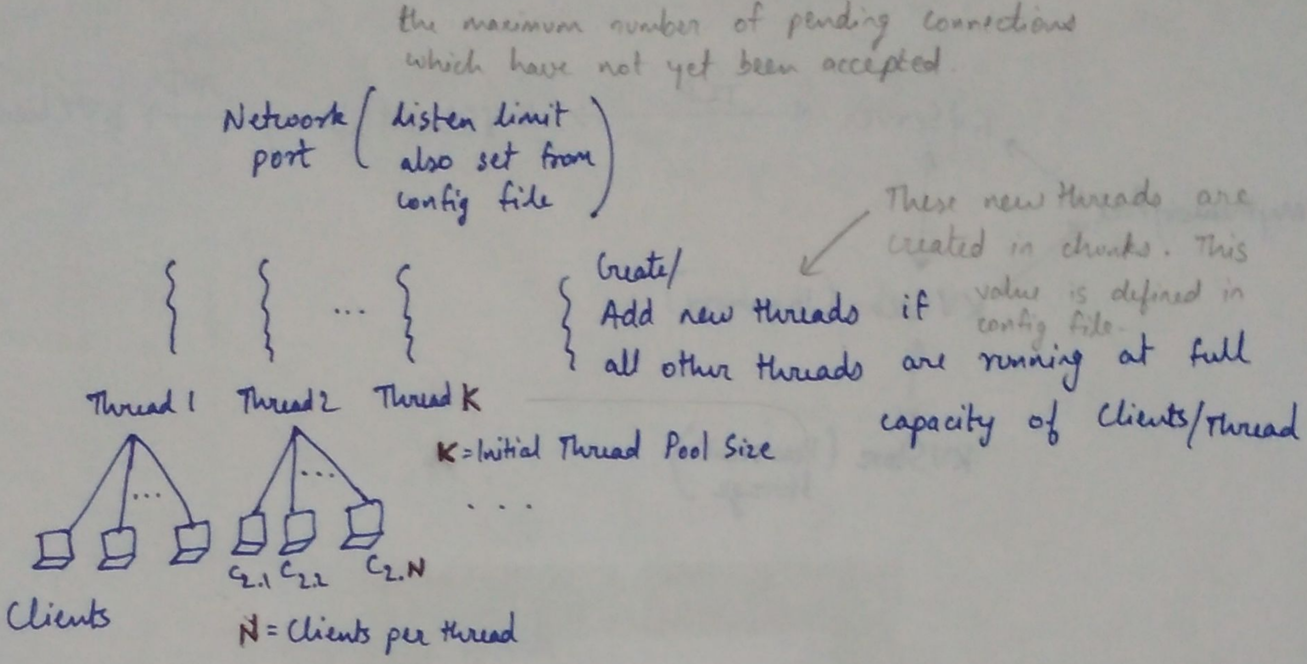
    uint8 status_code ⟶ Get, Put, Del, Success, Error
    char key[256], value[256]
    uint64 hash1, hash 2

My MemoryPool - Thread safe faster alternative to multiple calls of malloc/new.
- Allocates memory in chunks of large size and stores the pointer to each element of the chunk for future use.
- We use two vectors
    1. To store pointer to chunks
    2. To store pointer to each element of each chunk. These are popped and given to the other components for use. And, pushed back once use is over



My Debugger - Custom header for printing logs (with various colours :-)) and some functions work with multiple threads as well.

KV Server - Accept client connections via. KVClientlibrary and serve them



the maximum number of pending connections which have not yet been accepted

Create/ Add new threads if all other threads are running at full capacity of Clients/Thread

These new threads are created in chunks. This value is defined in config file

K = Initial Thread Pool Size

N = Clients per thread

Steps

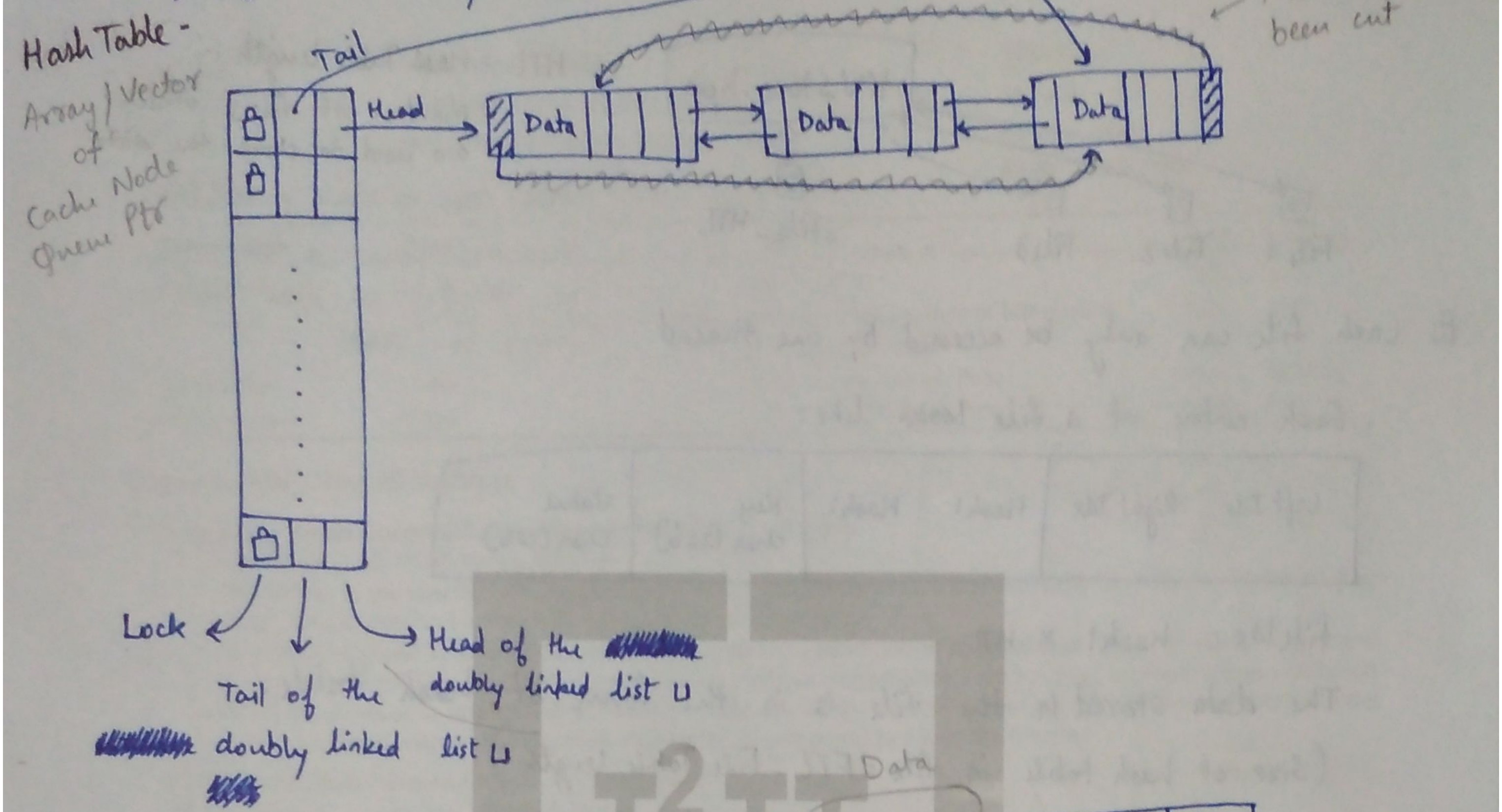
1. Read config file
2. Initialize memory pool
3. Initialize KVStore (Persistent Storage) and KVCache
4. Create and launch threads
5. Create socket, bind it to IP interface and Port, start listening
6. Start accepting client connection and assign them to the created threads in round robin fashion. Create new threads (based on the thread-pool-growth paramiter of the config file) if all existing threads have reached their limit of clients/thread.

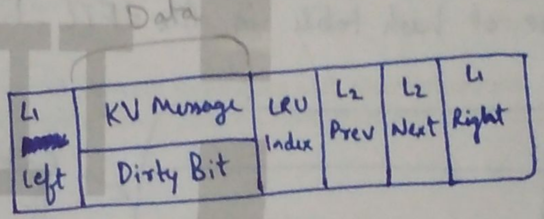
Work by each thread of the server:

1. Use epoll and wait for client messages. Once received, process them and reply back
2. Check if the main thread has assigned new threads to this thread or not. If yes, include them in the epoll list.
3. Keep executing step 1 and 2.

- API Get, Put, Delete, Clean

KVCache - In memory cache

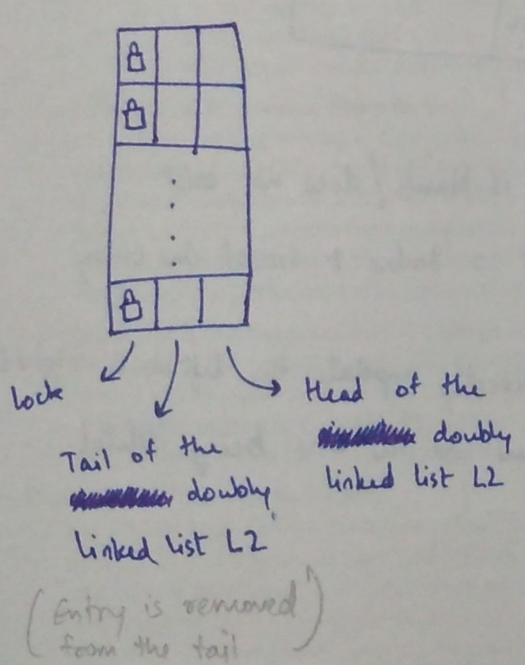

Hash Table -
Array/Vector of Cache Node Queue Ptr
Tail
Head
Data
Data
Data
This thing has been cut
Lock
Tail of the [redacted] doubly linked list L1
Head of the [redacted] doubly linked list L1



LRU Eviction Table

$$Size = \begin{cases} 128 & \text{cache size} \geq 10240 \\ 32 & 1024 \leq \text{cache size} < 10240 \\ 1 & \text{cache size} < 1024 \end{cases}$$

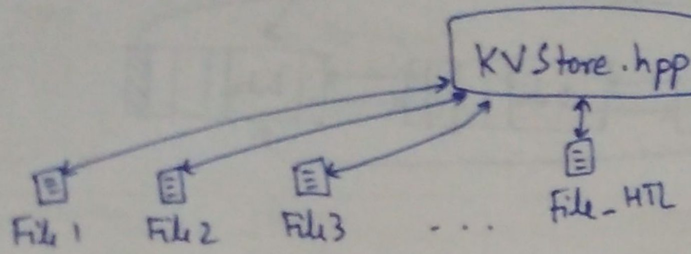
Array/Vector of Cache Node Queue Ptr


Lock
Tail of the [redacted] doubly linked list L2
Head of the [redacted] doubly linked list L2
(Entry is removed from the tail)

- Round Robin fashion is used to evict elements from the eviction table.
- Multiple eviction queue's are used to allow multiple evictions in parallel.
- Elements of same queue of hash table can be in different queue in this eviction table.
- Lock is acquired in Hashtable & LRU Eviction Table to perform eviction.
- Dirty Bit
  - 0 = latest value is present in KVStore
  - 1 = It needs to be updated in KVStore (i.e entry is modified)
  - 2 = Cache Node has been invalidated & put back in the memory pool
  - 3 = Delete this entry from KVStore on eviction

API

KVStore - Get, Put, Detele KVMessage to & from harddisk (persistent storage)



HTL = Hash Table Length
= Number of files which are used to store the data

More files ⇒ More parallelism
However, there is a limit on max number of open files a process can have

Each file can only be accessed by one thread

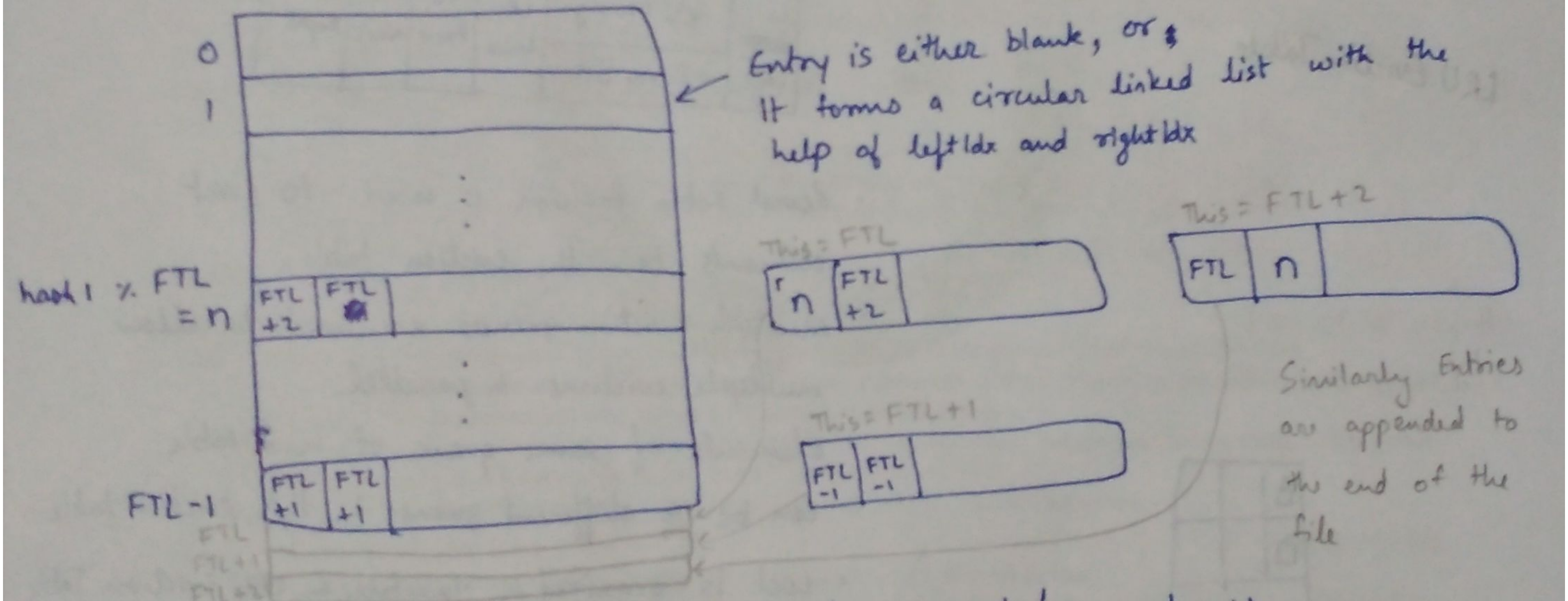
Each entry of a file looks like:

Left Idx	Right Idx	Hash1	Hash2	Key char[256]	Value Char[256]

fileIdx = hash1 % HTL

The data stored in the file is in the form of hash table.

(Size of hash table in file = FTL = File Table Length)



Entry is either blank, or it forms a circular linked list with the help of leftIdx and rightIdx

Similarly Entries are appended to the end of the file

LeftIdx = RightIdx = MAX_UINT64 ⇒ entry is blank / does not exist

Location inside file using "seek" method = Index * sizeof One Entry

Note: During data/Entry deletion, we directly update the leftIdx & rightIdx of Entries adjacent to the one being deleted.

NO compaction is performed.