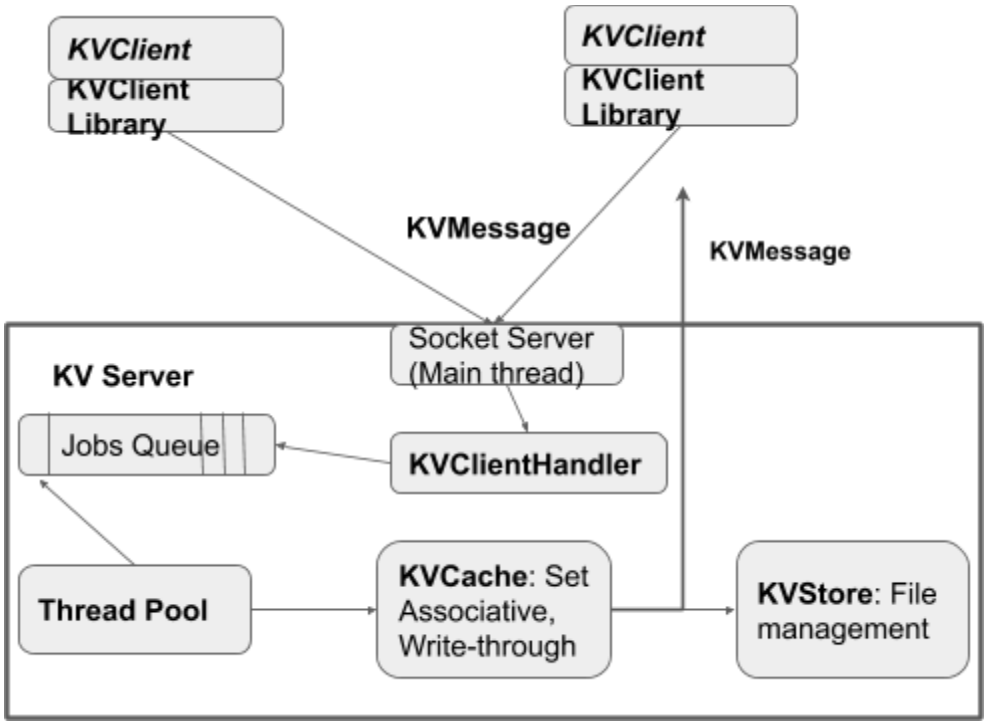
# CS 744: Project 1 - Building a Key Value Store




Multiple clients will be communicating with a single key-value server (**KVServer**) in a given messaging format (**KVMessage**) using a client library (**KVClient Library**). Communication between the clients and the server will take place over the network through sockets (**SocketServer** and **KVClientHandler**). The server must use a threadpool (**ThreadPool**) to support concurrent operations across multiple sets (key sets) in a writeback set-associative cache (**KVCache**), which is backed by a store (**KVStore**).

## Requirements

- Your key-value server will support 3 interfaces:
    1. **Value GET (Key k)**: Retrieves the key-value pair corresponding to the provided key.
    2. **PUT(Key k, Value v)**: Inserts the key-value pair into the store.
    3. **DEL(Key k)**: Removes the key-value pair corresponding to the provided key from the store.

- You will use the **exact** request/response formats defined later in the specification for communication between external and internal components.
- 'Keys' and 'values' are always strings with non-zero lengths. They cannot be nulls either.
- Each key can be no greater than 256 **bytes** and each value can be no greater than 256 **kilobytes**. If the size is breached, return an error. (Assume each character to be 1 byte in size)
- When PUTing a key-value pair, if the key already exists, then the value is overwritten.
- When GETing a value, if the key does not exist, return an error message.
- Make sure that the server can handle concurrent requests across key sets.
- For all networking parts of this project, you should use only sockets and select/poll/epoll system calls.
- For this project, you will have to use Conditional Variables and Locks with regular linked list to simulate a request queue rather than depend upon synchronized implementations. We want you to learn how to build thread-safe data structures by using the basic synchronization building blocks (Locks, ReadWriteLocks, Conditional Variables, etc) that you learned in assignment 2.
- You should ensure the following synchronization properties in your key-value service:
    1. Reads (GETs) and updates (PUTs and DELETEs) are atomic.
    2. An update consists of modifying a (key, value) entry in both the KVCache and KVStore (eventually).
    3. All operations (GETs, PUTs, and DELETEs) must be parallel across different sets in the KVCache, and they cannot be performed in parallel within the same set.
- Each set in the cache will have a fixed number of entries, and evict entries when required.
- For all operations, you must use an operation - **`KVCache.getSetId(key)`** to determine which unique set in the cache each key belongs to.
- You should bulletproof your code, such that the key-value server does not crash under any circumstances.
- You will run the key-value service on port 8080.

## KVMessage Format:

Since these are all in XML, you MUST write a simple XML parser

- **Get Value Request:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="getreq">
  <Key>key</Key>
  </KVMessage>
- **Put Value Request:**
  <?xml version="1.0" encoding="UTF-8"?>

<KVMessage type="putreq">
<Key>key</Key>
<Value>value</Value>
</KVMessage>

- **Delete Value Request:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="delreq">
  <Key>key</Key>
  </KVMessage>
- **Successful Get Response:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="resp">
  <Key>key</Key>
  <Value>value</Value>
  </KVMessage>
- **Successful Put Response:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="resp">
  <Message>Success</Message>
  </KVMessage>
- **Successful Delete Response:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="resp">
  <Message>Success</Message>
  </KVMessage>
- **Unsuccessful Get/Put/Delete Response:**
  <?xml version="1.0" encoding="UTF-8"?>
  <KVMessage type="resp">
  <Message>Error Message</Message>
  </KVMessage>

## KVCache

The KVCache is an implementation of a set associative cache with the second chance (clock algorithm) eviction/replacement policy. You also needs to implement the **`toXML`** function to return data in the XML format to the client. The cache of course will store it in some other more compact format of your choosing (probably a hashtable).

## `toXML()` Return Format

<?xml version="1.0" encoding="UTF-8"?>
<KVCache>
  <Set Id="*id*">
    <CacheEntry isReferenced="*true/false*" isValid="*true/false*">

```
    <Key>key</Key>
    <Value>value</Value>
   </CacheEntry>
  </Set>
</KVCache>
```

There should be as many Set elements as there are sets, and within each set, there should be as many CacheEntry elements as there are entries in each set. The actual sizes of the sets and number of cache entries per set must be read from a configuration file.

## KVStore:

The KVStore must implement at least two functions:

a. **dumpToFile**: That will write out a file containing some set of keys and values. Which keys go into which file is your design decision.
b. **restoreFromFile**: reads a file and has data ready for the cache to populate it when needed.

### `KVStore.dumpToFile(filename)` Output Format

```
<?xml version="1.0" encoding="UTF-8"?>
<KVStore>
   <KVPair>
    <Key>key</Key>
    <Value>value</Value>
   </KVPair>
   <KVPair>
    <Key>key2</Key>
    <Value>value2</Value>
   </KVPair>
</KVStore>
```

## Error Messages (Case-Sensitive)

- "Success" -- There were no errors
- "Network Error: Could not send data" -- If there is an error sending data
- "Network Error: Could not receive data" -- If there is an error receiving data
- "Network Error: Could not connect" -- Could not connect to the server, port tuple
- "Network Error: Could not create socket" -- Error creating a socket
- "XML Error: Received unparseable message" -- Received a malformed message
- "Oversized key" -- In the case that the submitted key is over 256 bytes (does not apply for get or del requests)

- "Oversized value" -- In the case that the submitted value is over 256 kilobytes
- "IO Error" -- If there was an error raised by KVStore
- "Does not exist" -- For GET/DELETE requests if the corresponding key does not already exist in the store
- "Unknown Error: error-description" -- For any other error. Fill out "error-description" with your own description text.

For network errors arising on the server when it is not possible to return the error to the client, you can drop this silently. For errors generated on the client (KVClient), you can drop this silently as well.

# Evaluation:

1. *(20%)* Implement the conversion functions to go from user input strings to XML messages at the client and back to plain strings as the output. These functions can also be used at the KVCache. Implement the message parsing functions to parse the XML - keep this simple rather than using a full blown DOM parser.
2. *(10%)* Implement the **KVClient** such that it will issue requests and handle responses (generated using **KVMessage**) using sockets. It should pick up requests from a file and write all outputs to a file. The inputs and outputs to the client will be strings.
3. *(15%)* Implement a **ThreadPool** -- you are not allowed to use existing implementation of thread pools. The threadpool should monitor a queue of tasks and when a thread is free and returned to the pool should pick up the next task in the queue. Ensure that the addition of tasks to the queue is **non-blocking**.
4. *(15%)* Implement dumpToFile and restoreFromFile as a part of **KVStore**.
5. *(15%)* Implement a write-through set-associative KVCache with the SecondChance eviction policy within each set. You should follow a **write-through** caching policy. Also, implement **toXML** to return results.
6. *(25%)* Implement **KVServer** that brings it all together. Use the set-associative cache you implemented to make key lookups faster.