Technical Design Document

Mini-MemCached

A distributed in memory caching mechanism

-Chiddu bhat V.1.0

Table of Contents

Revision History

Name	**Date**	**Reason for changes**	**Version**
Chiddu Bhat	May 23 2020	Initial version	1.0

1. Introduction

The purpose of this document is to outline the design of a distributed Mini-MemCached server. This will include a view of the high level architecture as well as the breakdown of the internal subsystems. UML class and sequence diagrams will be provided to show how the system will be put together and how data will flow through the system. There will also be discussion on the technologies that we will be using throughout this project. This document provides an outline of the user interface to demonstrate how it will be formatted. Additionally, there is a section that makes use of a requirements traceability matrix, which will make it easier to trace system features and designs back to the requirements. Finally I am ending this document with additional future enhancements that can be made to memcached server for higher performance and better usability.

2. System Overview

2.1 High Level Description: The distributed Mini-memcached provides effective way of storing key , value pairs in memory. It can store small chunks of arbitrary data from database query results, API calls . This caching mechanism prevents making these calls frequently and there by drastically improving your web application performance.It can be easily deployed as a standalone server in itself or cluster of servers if it is a huge application.

High level functional requirements:

- Constant time lookup
- Constant time store
- Suitable cache eviction policy (LRU)
- Follow memcached text protocol standards for all forms of request/response handling.

High level Non functional requirement:

- Availability -
    a. Replicating to one more server.
- Scalability
- Low latency
- High throughput
- High Fault tolerant.
    a. Regular interval snapshot
    b. Log reconstruction.

2.2Tools and Technology Stack used

CachedThreadPool : It helps to dynamically scale threads required as per the request traffic. It automatically scales up or scales down on the number of threads that are required to serve requests at any given point in time based on request traffic. With FixedThreadPool we are bound by a number of threads that can serve requests at any given point in time. This will be a huge bottleneck when I/O through sockets.

ExecutorService : Executor service makes it extremely easy to manage threads. If regular threads are spawned with each request the system is not effectively scalable, we would be limited by the max number of threads that can be spawned and new requests would be dropped, if all threads are busy serving requests.In Executor service threads return to thread pool after the service, waiting to serve the next request. If all the threads are already busy serving requests, new requests are queued until threads are available.

Scanner : This is used to receive requests from clients(Multiple server). In our project we scan incoming data from port 11211.

OutputStream: This is used to write data back to the client. Thread which handles the client request is also responsible for I/O operations

HashMap : Internal data structure used to store <Key> <Value>. Here is value is going to be the Node which is represented in LRU list. The Node contains key, data and metadata.

POJO Plain Old Java Object. We use them for routine operations and book keeping.

Tools

There are a number of tools and technologies that this project will utilize for development.

- Open JDK14 ,Java version 14
- IntelliJ java IDE for development
- JUnit for unit testing.
- Memtier_benchmark for integration and performance testing.
- Git repository (on Github).
- Telnet for functional testing.
- Cacoo for system, class and sequence diagrams.

3. System Architecture

3.1 High Level Architecture

The Mini-memcached server comprises of five main components :

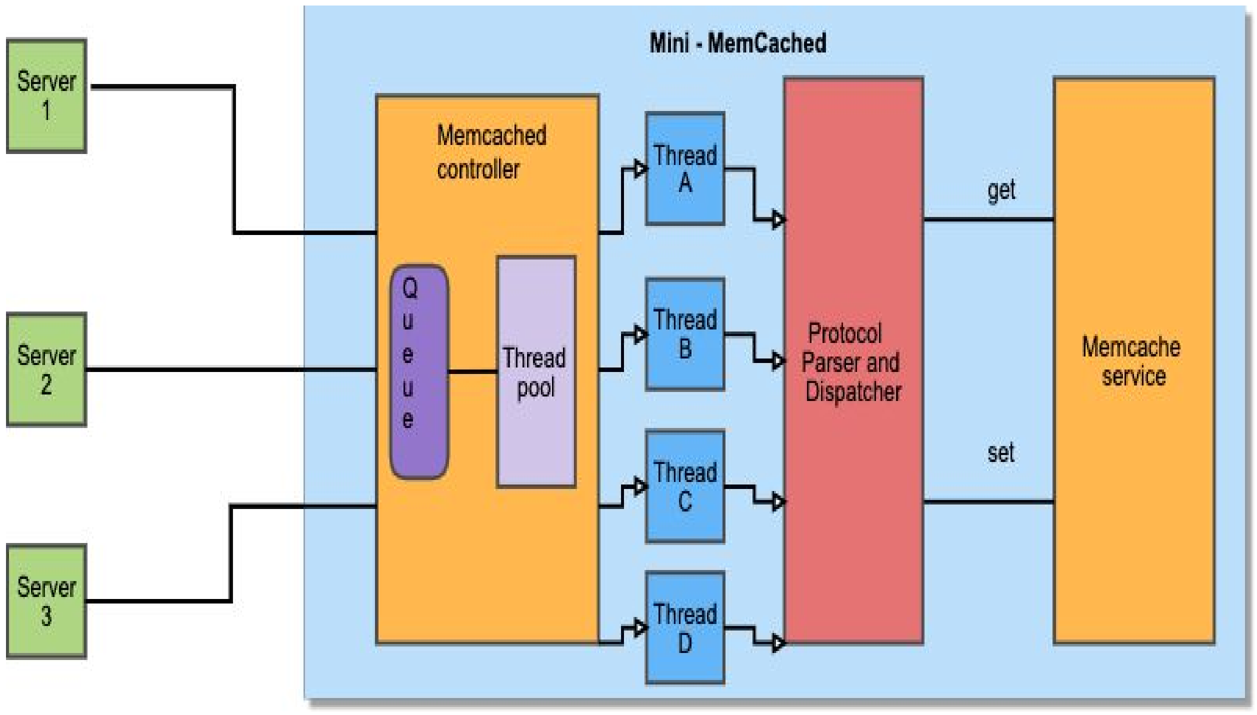
- Controller
- Protocol parser and Dispatcher
- MemCache service.
- Error Handling:

Controller: The controller is responsible for all the configurations of the memcache server. It is also responsible for maintaining dynamic thread pool to throttle cache performance based on the external demand.

Protocol parser: This module is responsible for parsing the incoming request from the client. It is responsible for verifying if the incoming request adheres to memcached text protocol. It also checks the parameters of the request for their validity. It responds to clients following standards of memcached text protocol.

Dispatcher: Once the request is parsed by the protocol parser and verified. The dispatcher forwards the request to the concerned service provider(SET or GET). once the service is completed, results are sent back to clients using standard memcached text protocol.

Cache service : This is the core module of the server, responsible for serving requests sent from the dispatcher. Internally it uses LRU caching policy to keep the cache updated. It's capacity is initialized during the server configuration by the Controller.



3.2 SubSystem Design :

A.Controller: The control The controller is the entrypoint for all the requests from various servers. This method is responsible for creating a server socket based on the port that is configured in Configuratios.properties file , it also initializes the memcached to capacity specified in config file.It is responsible for hosting executor service, which is a neat way of managing threads in Java.

ExecutorServicePool Executor service makes it extremely easy to manage threads. If regular threads are spawned with each request the system is not effectively scalable, we would be limited by the max number of threads that can be spawned and new requests would be dropped, if all threads are busy serving requests.In Executor service threads return to thread pool after the service, waiting to serve the next request. If all the threads are already busy serving requests, new requests are queued until threads are available

CachedThreadPool : CachedThreadPool helps to dynamically scale threads required as per the request traffic. It automatically scales up or scales down on the number of threads that are required to serve requests at any given point in time based on request traffic. With FixedThreadPool we are bound by a number of threads that can serve requests at any given point in time. This will be a huge bottleneck when I/O through sockets. memtier_benchmark - The system was not able to scale with fixedThreadPool of 32 when memtier_benchmark was configured with 8 threads having 10 connections each with 100 requests per thread and data length of 5.

B.Protocol parser and Dispatcher: This module is responsible for parsing the incoming request from the client. It is responsible for verifying if the incoming request adheres to memcached text protocol. It also checks the parameters of the request for their validity. It responds to clients following standards of memcached text protocol. Once the request is parsed by the protocol parser and verified. The dispatcher forwards the request to the concerned service provider(SET or GET). once the service is completed, results are sent back to

Protocol validator:

This method is used to validate requests that are sent to Memcached, this design only supports get and set requests as of now. The key length is restricted to 150. This also checks if the length of set request is 5 along with numeric validity for flags, ttl and bytes parameters.

Valid memcache protocol set request :

Request: set <key> <flags> <ttl> <bytes>

<datachunk>

Response: STORED

Valid memcache protocol get request :

Request: get <key>

Response: VALUE <key> <flags> <bytes>

<data chunk>

END

(source https://github.com/memcached/memcached/blob/master/doc/protocol.txt)

Dispatcher: Once the request is parsed and validated by protocol parser and validation methods , it's forwarded to dispatcher. Dispatcher forwards the request to appropriate interface methods which communicate with Memcache service. There are two interface methods which are implemented today for set and get operations.

C.MemCache service : This is the core module of the server, responsible for serving requests sent from the dispatcher. Internally it uses LRU caching policy to keep the cache updated. It's capacity is initialized during the server configuration by the Controller.

High Level MemCache service design: This Memcached is built with two important components

- HashMap
- Doubly linked list.

HashMap is used for constant time storage and retrieval of key and data.We also maintain a doubly linked list which are mapped to a hashmap. The reason for mapping doubly Linked List Nodes to maps is for achieving constant time insert and delete. This doubly linked list is arranged based on elements which are most recently used. Least recently used elements will be at the end of the list. When we hit MAX_CAPACITY we will start removing elements from the tail of the list which represents LRC cache eviction policy. Since adding and removing elements from the tail of Doubly linked list is also constant time.

MemCache Get service : This is a public method to get elements based on the key provided. If the element exists we will return the value and modify the LRU list by removing element from its current position and adding it to the front of the list (representing most recently used). This method is designed to support concurrency while manipulating LRU List. This method returns Node which contains key, value, flags, ttl, bytes stored.This method is called from dispatcher when it receives a get request from the controller.

MemCache Set service:

This is a public method to put data based on the key provided along with flags, timetolive(TTL) and length of data(flags and TTL are not supported in this design). Three conditions to consider in this method

1) If the key exists we will modify the data and other metadata to the latest data and metadata as per the request. We will modify the LRU list by removing elements from its current position and adding it to the front of the list (representing most recently used).

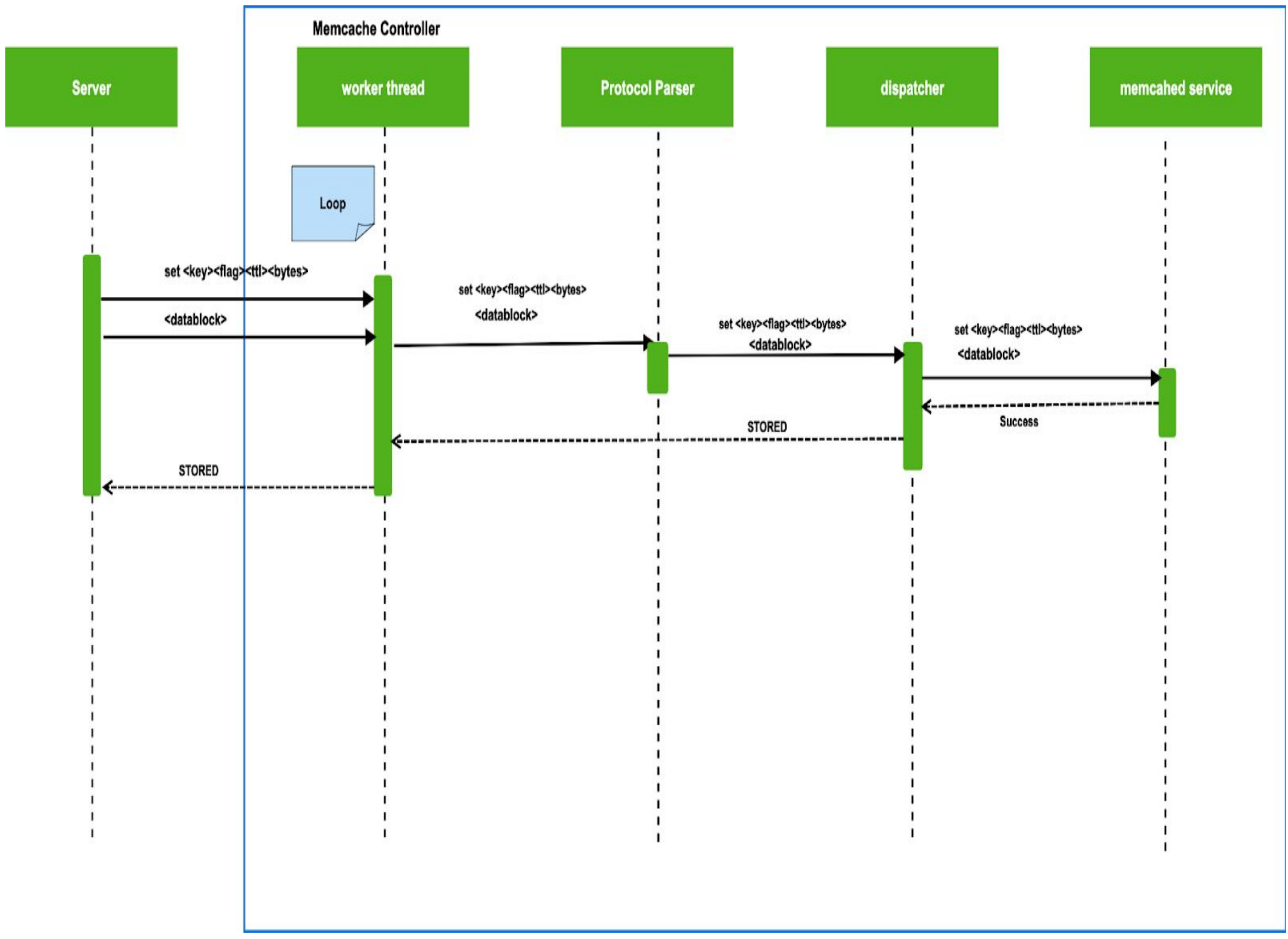
2) If the key doesn't exist we will create a node and store all the metadata information and add it to the front of the LRU list.

3) If we hit the max capacity of the MemCache,

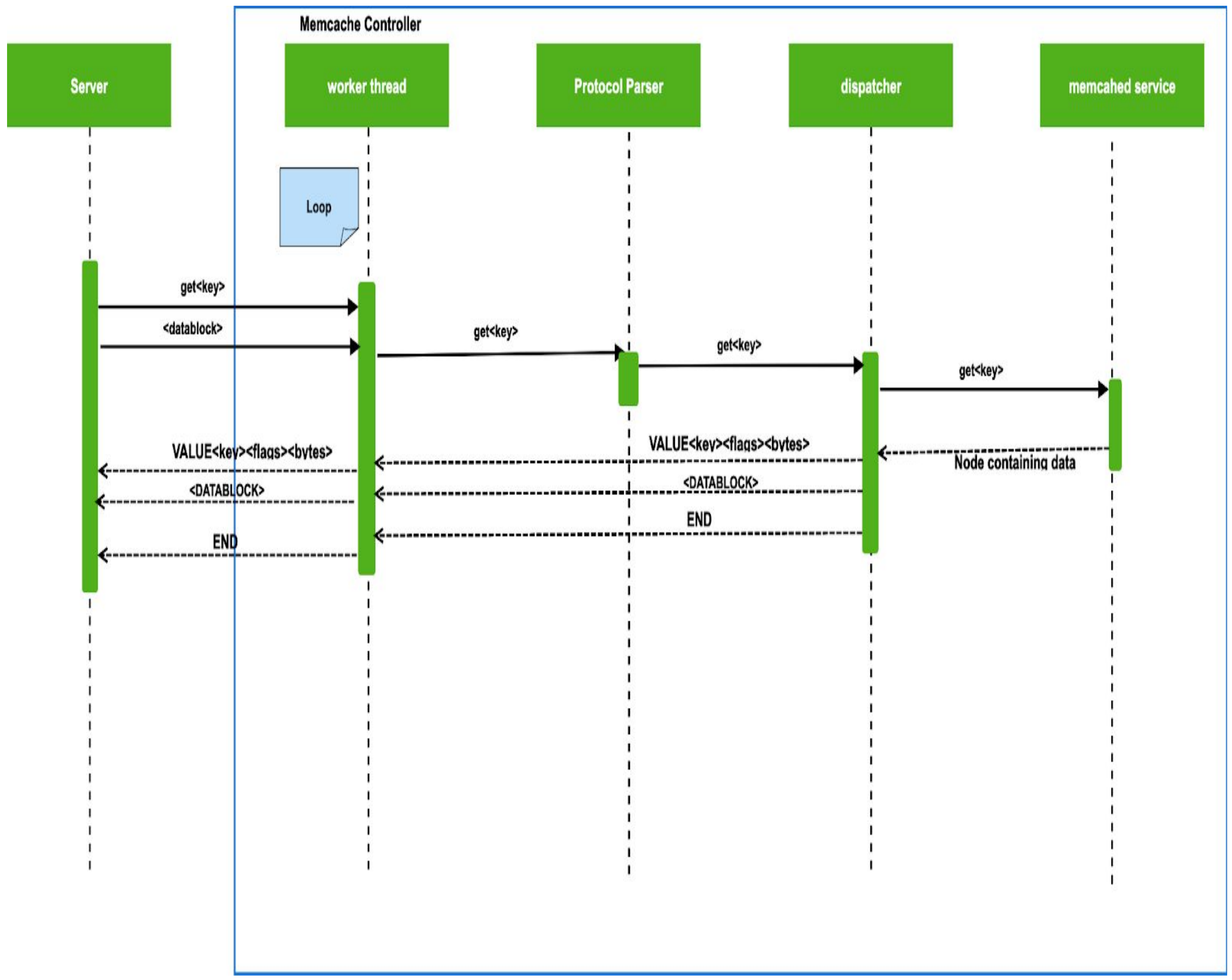
- we will evict items from the MemCache by following the least recently used cache eviction policy. Which means we will remove elements from the tail of LRU list and it's entry in the hashmap.
- Create a new node and add it to the front of LRU list and add that entry into our hashmap. This method is designed to support concurrency while manipulating LRU List. This method returns Node which contains key, value, flags, ttl, bytes stored. This method is called from the dispatcher when it receives a get request.

4. Sequence Diagrams

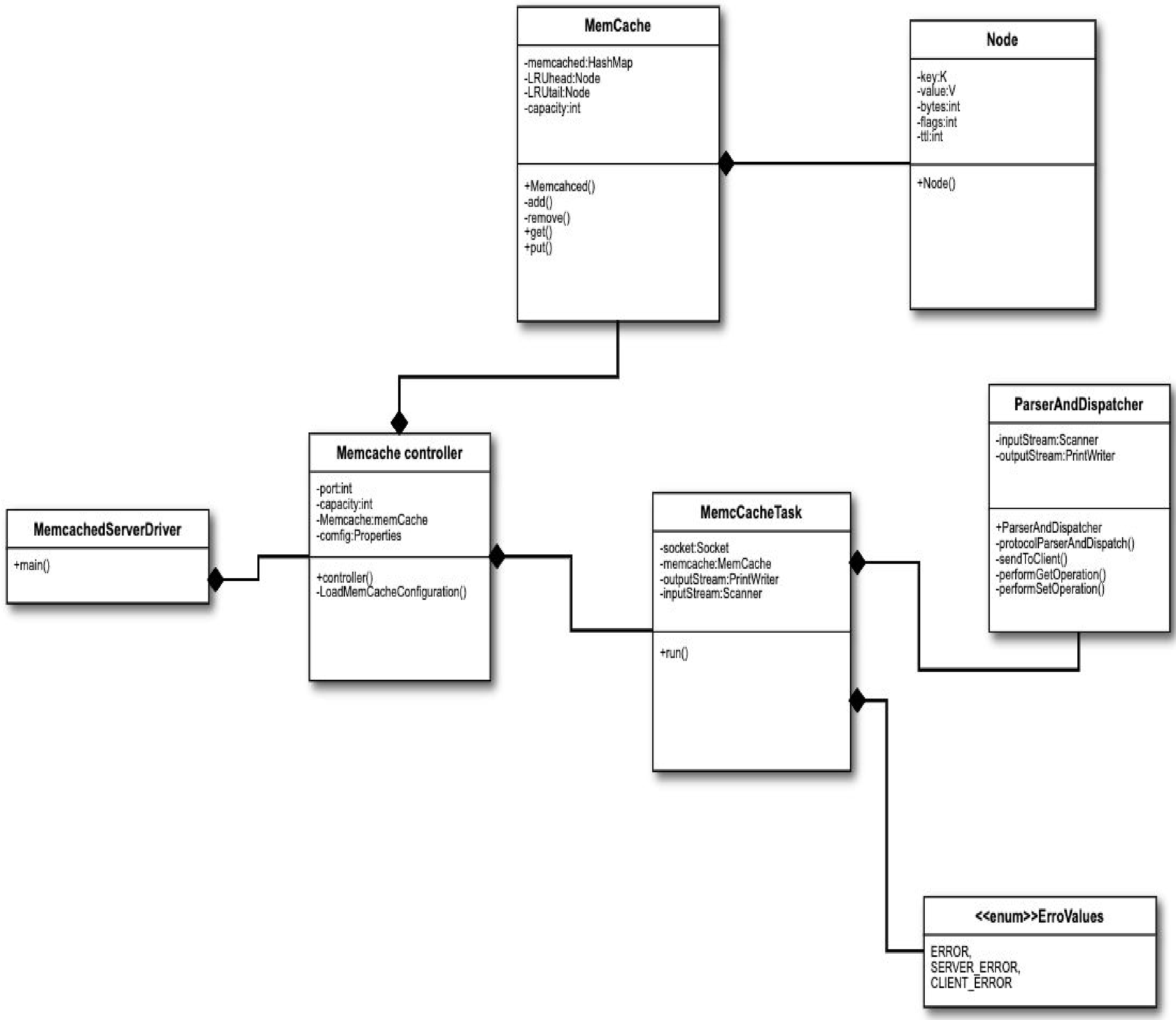
4.1 Set request



4.2 Get Request



5. Class Diagram



6. Deployment and Running:

Project is available here https://github.com/cnbscience/Mini-MemCached.

Prerequisites:

- This project is built with IntelliJ Idea IDE with openJDK14.
- Also have Junit installed in your class path to run the test cases.
- Telnet
- Git installed.

To deploy using command line and git.

1. Make a directory : “mkdir memcached”
2. type “cd memcached”
3. git clone https://github.com/cnbscience/Mini-MemCached.git
4. cd /path/to/Mini-MemCached/Memcache/main/java/MemCachedServer
5. Java -classpath /path/to/memcached/Mini-MemCached/target/classes/MemCachedServer MemCachedServer.MemcachedServerDriver
6. If the server is up you will see this message.

   ***************Memcached Server started****************

   Waiting for clients to connect

7. Open one more terminal and type:
    a. “telnet localhost <port_no>” (defualt pot_no is 11211).
8. Start sending memcached text protocol requests.

To deploy Using IDE like IntelliJ:

1. Download the project from https://github.com/cnbscience/Mini-MemCached.
2. Import the project into IntelliJ IDEA and run build and run MemcachedServerDriver.

7. Testing Coverage:

I have covered two primary modules in unit testing

- Memcached core services
- Memcached protocol parser

Memcached core services test cases :

TestCases included here are

All the hash keys and values are generated randomly.

- TestMemCachePutRequest() : This covers the test case of simple put operation to memcache and validates it by doing a get operation with key.
- TestMemCacheWrongKeyGetRequest() : This does negative testing to check if we are fetching values after providing wrong keys.
- LRUEvictionTest(): This test covers MemCache eviction policy. We verify that the least recently used item is evicted once we hit MAX_CAPACITY.
- MemCacheCapacityTest() : We verify that the Memcache functions well even after hitting MAX_CAPACITY by running an eviction policy.

Memcached protocol parser test cases :

This is the Test class for Memcached protocol testing.This is class performs various protocol testing including negative testing

TestCases included here are :

- TestMemCacheProtocolSet():checking if Set request is parsed properly
- TestMemCacheProtocolGet():checking if Get request is parsed properly
- TestMemCacheProtocolBadRequest():Negative testing by providing all wrong inputs to the protocol parser .

8. Integration testing with memtier_benchmark :

There are a lot of third party tools that are used to benchmark memeched. One such tool is memetier_benchmark.

(source : https://github.com/RedisLabs/memtier_benchmark)

- Download and install memtier_benchmark as per the instructions mentioned in the README.
- Start the memcached server either through the command line or IDE as mentioned above.
- Use various tuning parameters to check the performance of this memcached.
- Here is one such performance benchmark which was captured.

Running memtier_benchmark with 20 threads, 10 connections per thread an 10 requests per client with data size of 10.

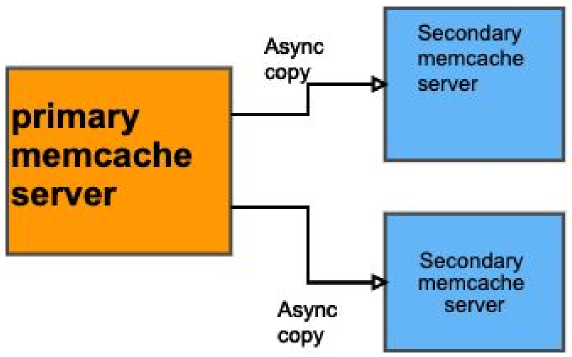
```
 >> memtier_benchmark -s localhost -p 11211 -c 10 -t 20 -n 10 -d 10
--ratio=1:1 --pipeline=1 --key-pattern S:S -P memcache_text

[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
[RUN #1 100%,   0 secs]  0 threads:       2000 ops,       0 (avg:   26683) ops/sec,
0.00KB/sec (avg: 1.22MB/sec),  0.00 (avg:  7.49) msec latency

20        Threads
10        Connections per thread
10        Requests per client

ALL STATS
========================================================================
Type        Ops/sec     Hits/sec   Misses/sec     Latency      KB/sec
------------------------------------------------------------------------
Sets       15988.74        ---         ---           9.18300      655.79
Gets        15988.74    15988.74    0.00             5.79400      843.16
Waits          0.00        ---         ---           0.00000        ---
Totals     31977.49    15988.74    0.00             7.48800      1498.94
```

9. Future enhancements.

Caching policy : The caching policy implemented here is an extremely simple LRU, it may not be the best caching policy for all the workloads. In future it is good to have a tuning mechanism to tune caching policies as per the workload. Allow users to use different caching policy as per the workload. One caching policy which has outperformed LRU is **Adaptive replacement Cache(ARC)**. (source:https://www.usenix.org/legacy/events/fast03/tech/full_papers/megiddo/megiddo.pdf) .

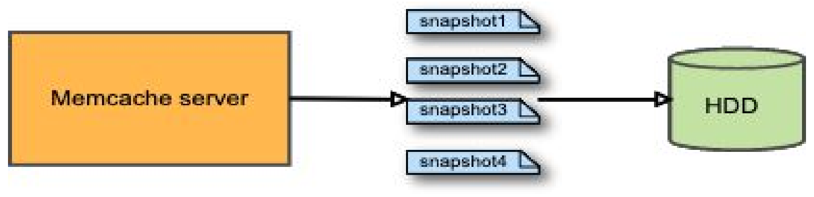
We can incorporate other caching mechanisms along with LRU and ARC for suitable consumer workloads.

High Availability: The system designed here is a single standalone memcache server. If the server goes down, it takes all the cached key value pairs with it. To make the system highly available and prevent single point of failure, we can replicate cached data from memcache server to another. We can provide shadow replication or passive replication, in which we don't reduce speed in which get/set request is processed, by doing asynchronous replication.



Increased Fault tolerant: To make the system more fault tolerant in following two ways

1. Taking consistent snapshots : we can take consistent point in time snapshots of our memcache server and persist it into external hard disk.
2. Log reconstruction : we can keep log of every single operation that is done to memcache server and persisting it to external disk. In times of crisis we can replay the log to recreate the memcache server.



Consistent hashing : This is not particularly related to server side enhancement , but definitely related to efficient implementation of distributed caching mechanism. This falls under load distribution and discoverability. This responsibility falls under clients. Clients should be able to effectively route the load such that it's equally distributed among multiple memcached servers. One simple way of doing this is using a modulus of number of servers and then writing the key value pair to the server.

Hash(Key) % number of memcached servers -------> server number. This is not a very effective way of distributing load, since scale this would be a nightmare.

We can use consistent hashing where distribution of data across multiple memcached servers in such a way that it **minimizes moving/reorganizing data** when a server is added or removed. Making it extremely suitable for **horizontal scaling.**

(sources: https://dl.acm.org/doi/10.1145/258533.258660)