Google
Summer of Code

Swift

Google Summer of Code & Swift

GSoC - 2023

Swift on Server - Memcached Client

Shehab Adel

Supervised by
Franz Bunsh

Acknowledgements

I would like to express my gratitude to my parents, friends, and my mentors who have supported me throughout my journey in Software Engineering. Your unwavering support has been invaluable to me, and I am truly grateful for it. I would also like to extend my thanks to the mentors at GSoC and Swift for their assistance and for giving me the opportunity to present this proposal for a project that I really am interested in. With your guidance, I hope to make you all proud.

Abstract

The goal is to implement the MetaCommands of Memcached to send commands and receives responses. To achieve this we need to implement the request encoding and response decoding. Furthermore, our client should support request pipelining to improve its performance.

Keywords

Memcached, Swift, SwiftNIO, Server

Contents

1 Personal Information

Name:	Shehab Adel
Age:	23
Current Location:	Cairo, Egypt
University:	Ain Shams University - Faculty of Engineering
Email:	shehabeldinadel@gmail.com
Phone Number:	+201011575136
Linkedin:	https://www.linkedin.com/in/shehabadel
GitHub:	https://www.github.com/shehabadel

2 About me

My name is Shehab Adel. At the moment I am writing this proposal, I am 23 years old. I am currently a Computer Engineering and Software Systems student in Faculty of Engineering - Ain Shams University, and there are only three months to go until graduation.

Currently, I am working as a Software Engineering intern at Siemens DISW, where I am developing internal web tools using Typescript, React.js, Node.js, and GraphQL. I have been involved in two projects so far - one where I acted as a maintainer to fix bugs and implement new features, and another where I started the project from scratch.

In addition, I have also founded a platform called Stack Info. It shows the technology stacks used by companies mainly in Egypt, and will expand to include EMEA in the future. This platform has helped over 50 software developers in finding job vacancies that match their preferred tech stack. At present, it has over 250 companies listed and receives an average of 30K views per month.

Although I have not worked with Swift as a programming language before, I am interested in it, particularly its Swift on Server team, and I was hoping to join them as an intern. That's why I am very interested in this project - I believe it will provide me with the opportunity to work with something completely new and explore a project that I am passionate about, especially since I have a massive passion for server-side development.

I am eager to make a meaningful contribution to this project and to learn a lot from it.

3 Overview

3.1 Memcached Overview

Memcached is considered to be an open-source in-memory key/value cache store. It simply works by storing chunks of data in the RAM in order to reduce the database reads, and thus enhance the reading process performance. Memcached works by running a server that could be connected to by establishing a TCP/UDP connection, and it can be used to transmit higher level text-based or binary protocol messages which defines the commands sent and responses received from the server as well. Our project revolves around the Meta commands which is a text-based protocol that is used to communicate with the memcached server.

The Meta commands are considered as a set of new ASCII-based commands [1] representing a request or a response from the memcached server. It basically consists of text strings separated by newline characters.

All meta commands follow a basic syntax:

```
<cm> <key> <datalen*> <flag1> <flag2> <...>\r\n
```

Where `<cm>` is a two character command code, while `<datalen>` is only for commands with payloads, like set commands. [1] For example in order to set a key and value in the memcached server we may use the command below.

```
ms foo 3 T20 \r\n
bar\r\n
```

```
ms                  :  Set Command
foo                 :  Key name
3                   :  Length of the data block
T20                 :  Expiration time flag with a token 20 seconds TTL.
\r\n                :  A seperator indicates end of the headers block
foo                 :  Value being set
\r\n                :  A seperator indicates end of the value block
```

As seen in the code snippet above, we have used the meta commands in ASCII protocol to set a key ‘foo‘ with value ‘bar‘ in the memcached server. The advantage of this protocol that it is human-readable unlike the binary protocol. In addition it supplies generic commands with the help of flags that modify the behavior of each command.
Now looking at flags in meta commands. Essentially, these flags are one-character codes that alter the behavior of the command, in contrast to the opaque numbers passed by the client as client bitflags. meta commands flags are not stored in the cache. Flags in general may also take tokens, which are strings following the initial character for the flag.

Mainly the flags can be supplied in a request and returned in the response in the same order they were originally supplied in the request. So, if we used the same meta command syntax mentioned above like this

```
<cm> <key> <datalen*> <flag1> <flag2> <...>\r\n
```

The response coming back from the memcached server will look like this.

```
<RC> <datalen*> <flag1> <flag2> <...>\r\n
```

Where <RC> is a 2 character return code. The number of flags returned are based off of the flags supplied in the request. To illustrate, suppose we include the ‘t‘ flag in a get request for a key that has a TTL of 20 seconds. In response, we will receive ‘t20‘, which represents the expiration time of 20 seconds. The following example may provide further clarification on flags, specifically related to the set command example mentioned earlier.

```
mg foo t v \r\n
```

mg	:	Get Command
foo	:	Key name
t	:	Flag to include the expiration time in the response
v	:	Flag to include the value of the key in the data block
\r\n	:	A seperator indicates end of the headers block

The response to the `mg` command will now include the value for key `foo`, including the expiration time as well. So, it is going to look like this.

```
VA 3 t v\r\n
bar\r\n
```

VA	:	Value status for get command
foo	:	Key name
t	:	Flag to include the expiration time in the response
v	:	Flag to include the value of the key in the data block
\r\n	:	A seperator indicates end of the headers block
bar	:	Value for key foo
\r\n	:	A seperator indicates end of the data block

In summary, the documentation lists several flags used by various commands such as get, set, and delete. Some flags can accept tokens to perform a specific action. Below are some of the most important flags for `mg` and `ms` commands mentioned in the documentation.

```
ms(set command) most important flags.
```

```
- b: interpret key as base64 encoded binary value
- c: return CAS value if successfully stored.
- C(token): compare CAS value when storing item
- F(token): set client flags to token (32 bit unsigned numeric)
- I: invalidate. set-to-invalid if supplied CAS is older than item's CAS
- k: return key as a token
- O(token): opaque value, consumes a token and copies back with response
- q: use noreply semantics for return codes
- T(token): Time-To-Live for item
- M(token): mode switch to change behavior to add, replace, append, prepend
--------------------------------------------------
```

```
mg(get command) most important flags.
```

- b: interpret key as base64 encoded binary value
- c: return item cas token
- f: return client flags token
- h: return whether item has been hit before as a 0 or 1
- k: return key as a token
- l: return time since item was last accessed in seconds
- O(token): opaque value, consumes a token and copies back with response
- q: use noreply semantics for return codes.
- s: return item size token
- t: return item TTL remaining in seconds (-1 for unlimited)
- v: return item value in <data block>

3.2 SwiftNIO Overview

SwiftNIO is considered an asynchronous non-blocking event-driven framework for developing server-side applications. SwiftNIO mainly uses reactor pattern to handle I/O events, it has main building blocks which are:

- Channel -> Connects network socket with our the application's code.
- ChannelHandler -> An action that is triggered when a network event occurs.
- ChannelPipeline -> Sequence of ChannelHandler that handle or intercept inbound events and outbound operations of a Channel.
- EventLoop -> Responsible for handling I/O events for Channels by firing callbacks.
- EvenLoopGroup -> Handles threads and Eventloops.

3.3 Project Overview

The goal of the project is to implement a Swift client for memcached which is built on SwiftNIO framework. This will allow Swift developers to easily connect to a memcache server and access or modify their data in their application code asynchronously. So, we will be able to set a key ‘foo‘ with value ‘bar‘ for example like this.

```
let client = MemcachedConnection(host: "localhost", port: 11211, group: eventLoopGroup)
do {
    try await client.set(key: "foo", value: "bar")
} catch {
    print("Error: \(error)")
}
```

In the given code snippet, the first step is to establish a connection with the memcached server by passing the host and port number as arguments to the `MemcachedConnection` actor. Once the connection is established, we can use the `MemcachedConnection` to set a key-value pair, where the key is "foo" and the value is "bar", in an asynchronous manner. Yet in the underlying `MemcachedConnection` class, there are several steps that are conducted whenever we create an object from the `MemcachedConnection`, which are:

- Creating a `ClientBootstrap` that establishes a connection with the memcache server.
- Creating a `Channel` to handle communication with the memcache server File Descriptor.
- Creating `ChannelDuplexHandler` to handle requests and responses from the memcache server.
- Creating the Storage and Retrieve commands methods in the `MemcachedConnection` class.
- Creating structs to define memcached request and response with methods to encode and decode.

So, here is a simple prototype for the `MemcachedConnection`.

```
import NIO
actor MemcachedConnection{
    private var host:String?
    private var port:Int?
    private let channel: Channel?
    private let group: EventLoopGroup?
    private let bootstrap: ClientBootstrap
    /*
        Takes memcached server host, port, and EventLoopGroup as constructor parameters
        Initialize a bootstrap and add the handlers to the channelpipeline
        Initialize the channel and connect to the memcache server asynchronously
    */
    init(host:String, port:Int, group:EventLoopGroup) async throws{
        self.host=host
        self.port=port
        self.group = group
        self.bootstrap = ClientBootstrap(group: self.group)
            .channelOption(ChannelOptions.socket(
            SocketOptionLevel(SOL_SOCKET), SO_REUSEADDR),value: 1
        )
        .channelInitializer { channel in
                channel.pipeline.addHandlers([
                    MemcachedChannelHandler()
                ])
            }
        self.channel = try! await bootstrap.connect(host: host, port: port).get()
    }
```

```
    /*
        Set command
    */
    func set(key:String, value:String,options:String?)->EventLoopFuture<Void> {
        let promise: EventLoopPromise<Void>=self.group.next().makePromise()
        /*
            Construct a setCommand of type MemcachedRequest before
            passing it to the MemcachedChannelHandler to be converted
            into array of buffers using encodeRequest() method which
            is going to be a method inside MemcachedRequest struct
        */
        let setCommand = MemcachedRequest(type:"ms",key:key,val:value,options:options)
        /*
            Send the setCommand across the ChannelPipeline to
            be encoded before being sent out to the server
        */
        channel?.writeAndFlush(setCommand).flatMap{
            promise.futureResult
        }
        return promise.futureResult
    }
}
```

The project will involve creating two structs named `MemcachedRequest` and `MemcachedResponse`. Each of them will have attributes related to how we can define a memcached request and response. In addition to `encodeRequest` and `decodeResponse` methods in `MemcachedRequest` and `MemcachedResponse` respectively. This approach will enable the creation of generic commands as memcached requests, following the meta commands approach and allow the decoding of responses from the memcached server.

Another task is to develop a unified `ChannelHandler` named `MemcachedChannelHandler` that is a `ChannelDuplexHandler` and is capable of handling both encoding and decoding of `MemcachedRequest` and `MemcachedResponse` correspondingly.

3.4 Project's Architecture

This could be a basic top-level sequence diagram to demonstrate the communication and creation of structs and objects. Further, as the implementation process commences, more comprehensive UML diagrams and changes will be incorporated into the project documentation.
Here is a prototype sequence diagram.

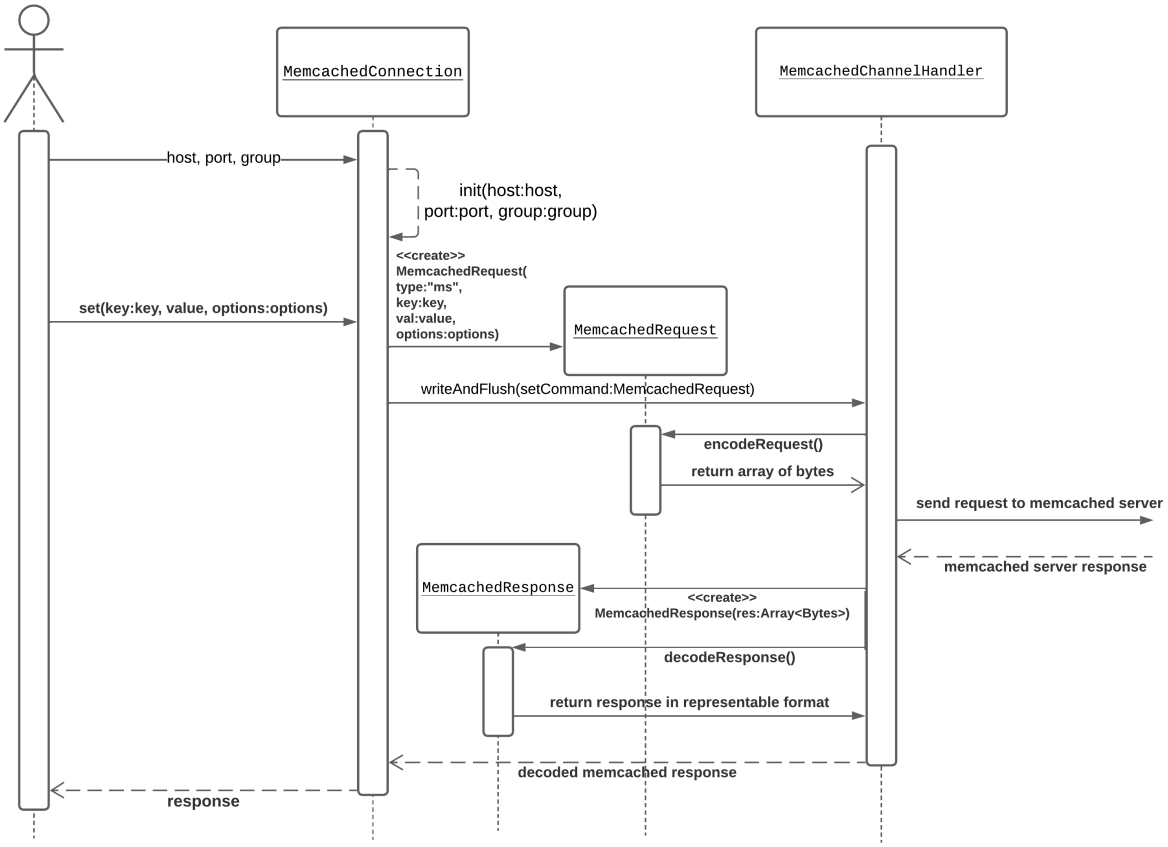

Figure 1: Prototype Sequence Diagram

The figure above illustrates the process of creating an object from `MemcachedConnection`, initializing a `ClientBootstrap`, and making a call to the set method by passing key, value, and options as parameters. This triggers the creation of a `MemcachedRequest` object with a type of "ms" and the specified parameters, which is then sent to the `MemcachedChannelHandler`. The `encodeRequest()` method is applied to the `MemcachedRequest` object to encode it into an array of bytes that are then transmitted to the memcached server. Upon receiving the memcached response, a `MemcachedResponse` object is created using the received array of bytes. The `decodeResponse()` method is then used to convert the memcached response into a presentable format, after which the response is returned to the caller.

4 Project Plan

As seen above in the prototype, I plan to create a `Memcached Client` that supports the most fundamental memcached commands that can perform basic storage and retrieval using generic MetaCommands. To accomplish this, I need to create several actors, classes and structs, including

- `MemcachedConnection` - Actor
- `MemcachedChannelHandler` - Class
- `MemcachedRequest` - Struct
- `MemcachedResponse` - Struct

In this prototype, each command will be represented as a separate method in the `MemcachedConnection` that takes multiple parameters flags for the command and constructs a `MemcachedRequest` that will be sent to the `MemcachedChannelHandler`. The `MemcachedChannelHandler` will be responsible for encoding the `MemcachedRequest` and to be converted into array of bytes, which will be sent to the memcached server. The `MemcachedChannelHandler` will be responsible for decoding the response coming back from the server by passing the response to `MemcachedResponse` to be decoded into a presentable format. If there was a failure, an error will be thrown.

The `Memcached Protocol`[1] specifies that clients communicate with servers using TCP connections.

> Clients of memcached communicate with server through TCP connections. (A UDP interface is also available; details are below under "UDP protocol.") A given running memcached server listens on some (configurable) port; clients connect to that port, send commands to the server, read responses, and eventually close the connection.

In the end there should be the main basic blocks of the `MemcachedConnection` for Swift, so that we can build the project on them later.

5 Project Goals

1. Develop a simple Swift package that connects to a memcached server using TCP connection.
2. Support most fundamental and basic storage and retrieval commands. (To be discussed with the mentor)
3. Document implemented features in the project.
4. Write tests for the implemented features in the project.

6 Project Timeline

There is no estimation for number of hours required for this project so far, and I always try to give a task the longest period it may need. But, I can imagine that I will do following before starting GSoC.

- Learn Swift for the intended objectives of the project.
- Read the memcached protocol documentation.
- Study SwiftNIO and read the documentation.
- Study the PR related to Channels with async/await. (See Appendix.8)
- Communicate with the mentors regarding the project and its objectives.

Following the commencement of GSoC, I had a conversation with my mentor and we mutually agreed to break down my goals into achievable and realistic that I can accomplish on a weekly basis.

7 Commitment

I am scheduled to complete my final exams and graduate sometime between mid-June and the first week of July. Due to this, I may require a slight extension of the start **OR** end date until I have finished my graduation. I plan to discuss changing the standard project length with my mentor to 20-22 weeks. Currently, I am employed at Siemens DISW as a part-time Software Engineer. However, after graduation, I may work full-time until I know my mandatory military status for January 2024. For the time being, I intend to devote 20-25 hours per week to contribute to this project. If there are any changes to my current employment status, I may be able to work as a full-time contributor to the project.

References

[1] M. Contributers, "Memcached protocol," 2022, line 82. [Online]. Available: https://github.com/memcached/memcached/blob/master/doc/protocol.txt

8 Appendix

A. Recommended Pattern

Upon discussion with my mentor, I was given a link to a pull request in the `SwiftNIO` repository, which is expected to be merged soon. The PR demonstrates a recommended pattern for structuring `SwiftNIO` applications that use async/await while operating as servers. I acknowledge the PR and its impact on the project implementation, and will base my work around it. The relevant approaches and techniques used in the PR will be incorporated into my future work on the project.
Checkout the PR here [WIP] Channel sequence, and an example of async/await structure #2067

B. Defining `MemcachedConnection` as an Actor

Upon discussion with my mentor, We thought that `MemcachedConnection` should be an actor. After reading about actors, I found that it guarantees that only one request will be processed asynchronously by placing a request in a message queue that will be processed one by one in the order they were enqueued. This will help us to prevent data races, and guarantees executing a method at a time and ensures that only one caller interacts with the actor at a given time. I plan to study more about actors in the time I am learning Swift, so that I can use it for `MemcachedConnection`.