Kafka Client Package

Swift — Google Summer of Code 2022

Personal Details

Name	Felix Schlegel (he/him)
Languages	English, German
Course	Bachelor of Science in Computer Science
Semester	4
University	Technical University of Munich (TUM)
Email	[redacted]
GitHub	/felixschlegel
LinkedIn	/in/schlegel-felix
Swift Forums	/u/felixschlegel
Phone	[redacted]
Current location	Munich, Germany (UTC + 2)
Link to resume	[redacted]

Contents

1 About me

In the beginning, I want to use this opportunity to introduce myself. My name is Felix, a 20-year-old Computer Science undergrad at the Technical University of Munich in Bavaria, Germany. However, my interest in computers has lasted for much longer. At age 12, I started to program and have not stopped ever since.

Although I gained quite some experience in other programming languages such as C or Java, Swift is my "mother tongue" in terms of programming languages I use.

My entire Swift journey started by developing iOS apps in my spare time. I even had a workout app called Gymstructor that the App Store featured on its homepage as part of WWDC18. Furthermore, I have won two WWDC scholarships in 2018 and 2019, where I met a lot of fellow Swift developers and got my first glimpse of the community. Since I enjoy being involved in the community, I gave a talk about motion capturing in ARKit at the German iOS developer conference Macoun in 2019.

Besides tinkering on personal projects at home, I have been working at a company called Vectornator for nine months now. Vectornator is an illustration app for iOS and macOS that consists of a large Swift codebase. At Vectornator, I am responsible for in-app tutorials, a feature that is already available on the App Store for iPad and Mac. My core responsibilities for this feature were managing the client/server communication for tutorial content via OpenAPI, implementing tailor-made UIKit components and developing the feature's core functionality using Frameworks such as Combine.

I am particularly interested in contributing to Swift as part of GSoC because I have been working with Swift since its release in 2014. It was one of the first programming languages I learned and still excites me today because of its readability and language features. In this sense, I think that GSoC is a great way to become a contributor to the Swift project. GSoC allows me to work alongside an experienced mentor while hopefully becoming a great addition to the entire Swift community!

Finally, I think that Swift is a beautiful programming language that has more potential than just iOS and macOS development. I would therefore love to be a part of the Swift on Server efforts and the mission to open up Swift to more use cases such as server development and augmented reality applications.

2 The Project

The Kafka client package project aims to provide a native Swift Package for communicating with Apache Kafka servers. This shall be achieved by wrapping an already existing C library librdkafka[1] and leveraging the new concurrency features available in Swift.

2.1 Apache Kafka

Due to the limited scope of GSoC, this chapter will only cover the very basics of Apache Kafka. This will hopefully help to understand what Apache Kafka does and why it plays an important

[1] https://github.com/edenhill/librdkafka

role in the modern world of distributed services.

Apache Kafka usually runs on a cluster and contains so-called topics. One could imagine a topic as a message queue dedicated to a certain type of event that occurs. However, the key difference between a topic to a message queue is that messages are not removed from the queue when consumed. In addition, Kafka incorporates reactive programming as clients of the Kafka server can both subscribe to a topic and publish events to it. In practice, this means that when a publisher publishes a new event to a topic, all the subscribers of the topic get notified and react to this event.

An example use case of Apache Kafka could be a weather app that always shows the current temperature for a certain location. Both the thermostat and the app itself are clients to the Kafka server which contains a topic called “temperature”. When the temperature changes, the thermostat publishes the new value to the “temperature” topic of the Kafka server. All the clients running inside of the apps of the end-users will then get notified about the temperature change and will then update their displayed data accordingly.

Kafka clusters consist of one or multiple brokers, which are essentially instances or containers that run a Kafka process. These brokers are responsible for handling read and write requests, storing partitions and managing replication of partitions.

Partitions are chunks of a topic that are distributed among all brokers to allow for parallel reading and writing of messages resulting in better overall performance and more reliability. To increase the fault-tolerance of the Kafka cluster, even more, partitions can be replicated and stored on other brokers. When replication is applied, there are two types of partitions: leaders and followers, where leaders are the primary data store and followers are the replicated partitions.

To achieve parallel reading, data is consumed by so-called consumer groups. A consumer group is a cluster of so-called consumers that all read the same topic, only consuming each message once per group. There can be as many active consumers in a consumer group as there are partitions being responsible for the topic. When the number of consumers in a group changes, the distribution of partitions among the consumers is rebalanced so that work is always split equally and no partition is starving.

2.2 Project Goals

1. Developing a Swift Package that provides a Kafka Client that is able to produce and consume messages
2. Accomplishing 1. by making use of the new Swift concurrency features
3. Making it possible for the user to configure the Kafka Client with the standard Kafka configuration options
4. Creating tests to reduce the number of potential bugs
5. Writing extensive documentation for the newly created Swift Package

3 Proposed Solution

For this proposed solution it is important to mention that it is inspired by the SwiftKafka[2] package that is already available on GitHub. The package gives a broad overview of how the different Kafka entities are pictured in Swift and serves as a good example of how to wrap the librdkafka C library. However, this package has not been actively maintained for more than 2 years now and therefore lacks the latest Swift features such as async/await support.

3.1 Wrapping librdkafka

A key component of this proposal is communicating with the C library for Kafka. This approach was also used in SwiftKafka and Perfect-Kafka and requires the user to install librdkafka separately e.g. using brew.

Brew is a package manager that is mainly used for macOS. Once brew is installed (see installation instructions[3]), the user has to open Terminal and type the following in order to install the librdkafka C library:

```
$ brew install librdkafka
```

Assuming that the librdkafka C library is installed, an LLVM ModuleMap can be used to expose the C headers of the library to our Swift package. This method was also used in existing Kafka libraries for Swift such as SwiftKafka.

Library headers are located in different places on different operating systems. Therefore SwiftKafka uses a shim header that includes the right path to the `rdkafka.h` header in regards to the current operating system. Implementing the module map is straightforward and could look like this:

```
module LibKafka {
    header "shim.h"
    link "rdkafka"
    export *
}
```

The last thing that has to be done to be able to access the C functions is to include our mapped LibKafka library in our `Package.swift` file.

In a further iteration, we would ideally vend librdkafka as a binary dependency of our Swift Package. Support for binary dependencies has been added in Swift 5.3 (SE-0272)[4] but is still limited to Apple platforms. Librdkafka could be integrated as a sub-module of the repo this Package and added as an SPM target. Although this needs some more investigation, it is the direction we should be going before replacing librdkafka entirely with a custom Swift implementation for communicating with Apache Kafka.

[2]https://github.com/Kitura/SwiftKafka

[3]https://brew.sh/

[4]https://github.com/apple/swift-evolution/blob/main/proposals/0272-swiftpm-binary-dependencies.md

3.2 Public Interface

3.2.1 Configuration

Librdkafka exposes two types of configurations[5]: global configurations and topic configurations. In this proposal, those will be implemented as two separate types `KafkaConfig` and `KafkaTopicConfig`, although the implementation of mapping the configuration options to librdkafka will remain the same.

There are two options on how to implement the Config types, which will be elaborated on in the following:

Option 1

Use a string-based dictionary that maps the configuration options to their corresponding (string) values. This method is also suggested by the librdkafka community itself[6].

Pros

- the Configuration type of this package will automatically evolve with new versions of librdkafka
  → if there are new configuration options available in librdkafka, there is no need to change anything about this package's Configuration implementation

Cons

- configuration values are not type-safe as they are all represented as strings
  → the user of the package can propagate any value to librdkafka which will result in errors thrown at runtime

Option 2

Implement a struct that contains each configuration option as a strongly typed member.

Pros

- struct will contain type-safe values for each key
  → in most cases, configuration errors will be caught at compile time rather than at runtime

Cons

- when the underlying librdkafka library gets upgraded, the configuration options of this package have to be updated too in order to expose the newly available options

3.2.2 Message

Even though the properties of Kafka messages are very similar, this proposal suggests using two separate types for messages that are produced and consumed. This decision was made as we want to have strong types and avoid using too many optionals. In general, a Kafka message consists of the following properties:

[5] https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md

[6] https://github.com/edenhill/librdkafka/blob/master/INTRODUCTION.md#recommendations-for-language-binding-developers

`topic`	The topic in which the message is stored.
`partition`	The partition the message is/will be stored in. Partitions are subsets of a topic that are stored on different machines (so-called brokers) in the Kafka Cluster.
`key / value`	Key and Value of the message. Here it is important to mention that the key is hashed to determine what partition a message will be stored in. This guarantees that messages with the same key will be stored on the same partition. That again makes it possible to determine the chronological order of messages that were published with the same key.
`offset`	(Only in `KafkaConsumerMessage`) The offset is the index of a message inside of the topic.

Furthermore, the designated initialiser for both message types requires the `key` and `value` properties to conform to the `ContiguousBytes` protocol. This makes sense as Kafka does not handle explicit data types but rather raw data. Conforming to `ContiguousBytes` allows the user to use `Data` objects, byte arrays (`[UInt8]`) or `UnsafeBufferPointers` for keys and values, just to name a few.

Although conversion between `ContiguousBytes` implementations and common data types such as `Int` or `String` is straightforward, it is planned to create even more convenient initialisers and getters for the Kafka message types that cover all sorts of common data types for keys and values.

KafkaProducerMessage

```swift
struct KafkaProducerMessage {
    let topic: String
    let partition: Int32
    let key: ContiguousBytes?
    let value: ContiguousBytes

    init(
        topic: String,
        partition: Int32? = nil,
        key: ContiguousBytes? = nil,
        value: ContiguousBytes
    ) {
        // Initialisation
    }
}
```

Please note that the `partition` property is optional in the initialiser. If no explicit partition is defined, the message will be published to the default partition.

Additionally, an idea taken from SwiftKafka is to provide another initialiser for `KafkaProducerMessage` that allows the user to conveniently define keys and values as strings:

```swift
extension KafkaProducerMessage {
    init(
        topic: String,
        partition: Int32? = nil,
        key: String? = nil,
        value: String
    ) {
        self.init(
            topic: topic,
            partition: partition,
            key: key?.data(using: .utf8),
            value: Data(value.utf8)
        )
    }
}
```

KafkaConsumerMessage

Apart from having the `offset` property and `partition` being non-optional, the `KafkaConsumerMessage` type is not much different from its counterpart:

```swift
struct KafkaConsumerMessage {
    let topic: String
    let partition: Int32
    private(set) var key: ContiguousBytes? = nil
    let value: ContiguousBytes
    let offset: Int64
}
```

3.2.3 Client

We differentiate between two types of clients, producers and consumers. In the implementation, they will be named `KafkaProducer` and `KafkaConsumer` respectively. Apart from the network functionality, these classes have different responsibilities that shall be separated in a clean manner. On one hand, the producer has to handle message callbacks, the creation of topics and polling the server to get delivery callback updates. On the other hand, the consumer comes with properties relevant to consumer groups and methods for accessing the data with the new async types. Besides this, both clients share a common superclass called `KafkaClient` responsible for connecting to the Kafka server. The first implementation of the `KafkaClient`'s public interface can be imagined as follows:

```swift
class KafkaClient {

    enum `Type` {
        case consumer
        case producer
    }
    init(type: `Type`, config: KafkaConfig) {}

    func start() {}

    func connectAdditional(brokers: [String]) {}

    func stop() {}
}
```

The Type enum is necessary because the underlying librdkafka client needs to know if the initialised client is a producer or a consumer.

As mentioned before, a broker is a server instance that is part of the Kafka cluster that makes up the entire Kafka server. In the already existing implementations, broker specification is done with strings that match the "<host>:<port>" pattern. It is important to mention, that the initial list of brokers is passed to the KafkaClient class via a KafkaConfig object. When the start() method is called, the KafkaClient connects to the brokers specified in the KafkaConfig, which are also known as bootstrap servers. On the other hand, the stop() function ends all broker connections of the KafkaClient.

In addition, the KafkaClient provides a function called connectAdditional(brokers: [String]) that enables the user to connect to even more brokers during its runtime.

3.2.4 Producer API

In the Kafka ecosystem, the producer is responsible for publishing messages to topics of the Kafka cluster. Besides that, it is in charge of creating a topic if not yet been created. For this reason, it takes the additional topicConfig parameter. This parameter is the topic configuration used when a new topic has to be created.

```
class KafkaProducer: KafkaClient {

    let topicConfig: KafkaTopicConfig

    init(
        config: KafkaConfig = KafkaConfig(),
        topicConfig: KafkaTopicConfig = KafkaTopicConfig()
    ) {
        self.topicConfig = topicConfig
        super.init(type: .producer, config: config)
    }

    func sendAsync(
        message: KafkaProducerMessage,
        completionHandler:
            ((Result<KafkaProducerMessage, KafkaError>) -> Void)? = nil
    ) {
        // Put message into the buffer and continue execution
    }

    @discardableResult
    func sendSync(message: KafkaProducerMessage) async throws
    -> KafkaProducerMessage {
        // Put messagee into buffer
        // Await delivery report before returning
    }
}
```

Sending a message can happen in two ways: either by using the `sendAsync()` function or the `sendSync()` function. Generally, both functions use librdkafka's `rd_kafka_produce()` function, which is non-blocking and stores the new message in a buffer of messages that will be sent to the Kafka cluster. However, both functions have some differences that will be elaborated on in the following:

sendAsync()

The `sendAsync()` function puts the message into the message buffer and returns immediately afterwards without blocking. It also has the option to add a `completionHandler` which is a closure that is invoked once the delivery report of the message is sent back from the Kafka cluster.

sendSync()

The `sendSync()` function is technically `async`, but meant to be used synchronously by using `await`. It returns once it received the delivery report of the Kafka cluster or an error. Because it does not have a completion handler with a `Result` return type, it is marked as `throws` so that any errors will be propagated to the calling method.

Reliability

In both cases we set up a delivery report callback that is invoked once the Kafka cluster received the message or our client failed after `message.send.max.retries` (configuration option) retries to send the message. It is important to mention that when our message counts as delivered depends on a configuration option called `request.required.acks`. Here `acks` mean the number of acknowledgements of in-sync replica brokers that are needed to return a successful delivery report. Common values for the `request.required.acks` configuration option are:

`-1 / all`	wait until all in-sync replica brokers have acknowledged the message (most reliable)
`0`	do not send any delivery report to the client, also known as "fire and forget"
`1`	wait for the leader broker to acknowledge the message

3.2.5 Consumer API

A consumer is the counterpart of the producer. After subscribing to one or more topics of the Kafka server, the producer can receive and process messages that were published by producers.

`KafkaConsumer`s are always part of a consumer group, whose ID is stored in the `groupID` property of the class. These consumer groups are a collection of multiple consumers that all read from the same topic in parallel, while only consuming each message once per group. It is important to mention that a topic can only have as many running parallel consumers as it has partitions. When a consumer leaves or joins the group a so-called rebalance happens. Rebalancing describes the process of reassigning each partition to a dedicated consumer so that every partition gets read by a member of the group.

As integrating the new Swift concurrency features into this Package is a key component of this proposal, the `subscribe()` function of the `KafkaConsumer` is returning a custom implementation of `AsyncSequence`, which wraps an `AsyncStream` and enables the user to iterate over incoming messages using the `for await ... in` syntax. To see this in action, please see the Consumer API Example in this proposal.

```swift
struct KafkaConsumerSubscription: AsyncSequence {
    typealias Element = KafkaConsumerMessage

    var topics: [String]

    let _internal = AsyncStream<Element> {
        // ...
    }

    func makeAsyncIterator() -> AsyncStream<Element>.AsyncIterator {
        _internal.makeAsyncIterator()
    }
}

class KafkaConsumer: KafkaClient {

    let groupID: String?

    init(config: KafkaConfig, groupID: String? = nil) {
        self.groupID = groupID
        super.init(type: .consumer, config: config)
    }

    func subscribe(topics: [String]) -> KafkaConsumerSubscription {
        KafkaConsumerSubscription(topics: topics)
    }
}
```

3.2.6 Error

Luckily, librdkafka provides a function `rd_kafka_get_err_descs()`[7] that exposes all available Kafka errors with their corresponding code, name and description. Out of user-friendliness, these errors shall be mapped to a struct called `KafkaError` either by using some automatic code generation tool or manually.

3.3 Producer API Example

For our examples of the Producer and Consumer API, we come back to our example with the thermostat. This is how a new value would be published to the temperature topic:

```swift
let producer = KafkaProducer()
producer.start()

let message = KafkaProducerMessage(
    topic: "temperature",
    key: "MUC",
    value: "22"
)

// Sending via async interface
producer.sendAsync(message: message) { result in
    switch result {
    case .success:
        print("Message sent successfully!")
    case .failure(let error):
        print("Message send failure: \(error)")
    }
}

// Sending via sync interface
do {
    try await producer.sendSync(message: message)
    print("Message sent successfully!")
} catch {
    print("Message send failure: \(error)")
}

producer.stop()
```

3.4 Consumer API Example

As mentioned before, the `subscribe()` function of the consumer returns an implementation of `AsyncSequence`, which allows for the `for await ... in` syntax to be used. In our example, the

[7]https://docs.confluent.io/3.1.1/clients/librdkafka/rdkafka_8h.html#a0475de10b4ad6c20f722fcacbd85aacd

usage could look somewhat like this:

```
let consumer = KafkaConsumer(config: KafkaConfig())

consumer.start()

let subscription = consumer.subscribe(topics: ["temperature"])

for await message in subscription {
    // Process message
}
```

3.5 Logging

It is very common for server applications to include some sort of logging. This enables the administrator to troubleshoot errors or keep track of the overall behaviour of the system.

Therefore this library aims to include the `SwiftLog` API Package[8]. `SwiftLog` itself provides a common API that is then used by custom or third-party logging backends that can then handle the log data by e.g. printing them to `stdout` or saving them to a file.

Moreover, the implementation of the Swift Kafka Client Package shall follow the Swift on Server Log-Level Guidelines[9].

3.6 Documentation

Documentation is a crucial part of every software project. Especially libraries such as this one rely on documentation as other software developers will adopt our provided API.

At WWDC21 we saw the introduction of the Documentation Compiler (DocC)[10]. This allows us to write markdown-flavoured documentation directly in our source code using documentation comments that use the following syntax:

```
/// Single documentation comment

/**
Block-style documentation comment
/*
```

After compiling the documentation can be viewed in the Xcode Developer Documentation window or exported to any other format. Furthermore, DocC allows for the creation of articles and interactive tutorials.

This proposal will certainly include standard API documentation generated using DocC. If time allows for it, we should also create articles and interactive tutorials to make this library also

[8]https://github.com/apple/swift-log
[9]https://github.com/swift-server/guides/blob/main/docs/libs/log-levels.md
[10]https://developer.apple.com/documentation/DocC

accessible to beginners that may require a more project-oriented way of learning how to use this library.

4 Preliminary Timeline

As of the Swift GSoC Project list[11], this project is estimated to take 175 hours to complete, which roughly translates to 4 weeks. However, I will do my best to start the working period as prepared as possible. Although I cannot do any affirmations about my exam dates, I am sure that I will be able to start working in the middle of August.

4.1 Major Milestones

1. Prototype with custom currency types
2. Consumer interface that makes use of Swift concurrency features
3. Publishing documentation for all features

4.2 Weekly Tasks

Douglas Hofstadter once said in his book “Goedel, Escher, Bach: An Eternal Golden Braid” (1979):

“Hofstadter’s Law: It always takes longer than you expect, even when you take into account Hofstadter’s Law.”

For this reason, I am currently planning on using the 4th week as a buffer for tasks that will still be outstanding. However, I imagine the rest of the tasks to be laid out as follows:

- **Before GSoC**
  - read more about Apache Kafka
  - work on a small side-project involving Kafka to know better what a Package user will expect from the end product
- **Week 1**
  - set up a new Swift Package
  - create trivial currency types such as `KafkaProducerMessage` and `KafkaConsumerMessage`
  - expose librdkafka functions to Swift Package
  - make it possible to produce and consume messages (without concurrency yet)
  - write tests for every feature implemented
- **Week 2**
  - implement rather complex currency types such as `KafkaConfig` and `KafkaTopicConfig`
  - creating a consumer interface based on `AsyncSequence`
  - make the new Swift Package conform to the Swift on Server Development Guide
  - write tests for every feature implemented
- **Week 3**
  - write extensive documentation

[11] https://www.swift.org/gsoc2022/

    - create an interactive tutorial for the new Swift Package
- **Week 4**
    - Buffer Time

5 Outlook

Coming to an end, I hope you enjoyed reading my proposal.

I could well imagine future iterations of this project that get along without the C library and implement the communication with the Kafka server themselves.

The ideas mentioned in this proposal are just scratching the surface of what the entire Kafka ecosystem offers. However, I hope that it lays the foundation for a project that will be long lasting and also exciting for other contributors to work with.

In this regard, I would be very happy to join you on the journey of making this project come to life!

Best regards

Felix