THE USA JOURNALS
THE AMERICAN JOURNAL OF ENGINEERING AND TECHNOLOGY (ISSN – 2689-0984)
VOLUME 06 ISSUE12

PUBLISHED DATE: - 13-12-2024
DOI: - https://doi.org/10.37547/tajet/Volume06Issue12-09 **PAGE NO.: - 84-91**

RESEARCH ARTICLE **Open Access**

PROTO REFLECTION IMPLEMENTATION FOR DYNAMIC INTERACTION WITH GRPC SERVICES IN HIGH-LOAD SYSTEMS

Kish Aleksei
Senior Software Engineer / Technical Owner at Semrush Spain, Barcelona

Abstract
This paper introduces SwiftProtoReflect, a Swift library that implements Proto Reflection to enable dynamic interaction with gRPC services without the need for precompiled code from `.proto` files. Addressing the inherent limitations of Swift in handling Protocol Buffers—such as restricted dynamic code generation and limited reflection capabilities—the research employs theoretical analysis of these constraints and develops an innovative solution utilizing descriptors and dynamic message handling. The main results demonstrate that SwiftProtoReflect allows developers to define Protocol Buffers message structures dynamically, serialize and deserialize messages, and access message fields at runtime. This advancement overcomes existing barriers, offering enhanced flexibility, scalability, and performance comparable to statically generated code. Concluding, SwiftProtoReflect significantly contributes to the data engineering field by filling a critical gap in the Swift ecosystem, enabling efficient development of high-performance and adaptable applications. The novelty of this work lies in its original technological solution, expanding Swift's capabilities and providing practical benefits for high-load systems and microservice architectures.

Keywords Swiftprotoreflect, proto reflection, grpc, protocol buffers, swift language, dynamic interaction, data engineering, serialization, microservices, high-load systems.

INTRODUCTION

In the modern era of high-load systems and microservice architecture, the efficiency of interactions between services has become a priority. The gRPC protocol, based on the binary serialization Protocol Buffers, offers impressive advantages: it can be up to eight times faster than traditional JSON serialization, and message sizes are reduced by 60–80% [1,2]. This is achieved through the efficient use of HTTP/2 features such as request multiplexing and header compression [3].

However, despite these technical advancements, Swift developers face significant limitations when working with Protocol Buffers. Unlike some other languages, Swift does not provide built-in mechanisms for dynamically generating code from `.proto` files at runtime. This means that all messages and services must be generated during compilation, making it challenging to work with dynamic or frequently changing data structures [4].

Additionally, Swift's limited reflection capabilities, particularly in the context of Protocol Buffers, complicate the implementation of general serialization and deserialization mechanisms and the dynamic creation of messages based on their types. Certain data types may not be fully compatible with Swift, requiring additional transformations that could lead to performance degradation [5]. Integrating Protocol Buffers into existing projects may also require substantial codebase modifications, especially if binary serialization protocols were not initially planned.

In a world where service APIs evolve at an incredible pace, static interaction methods become a bottleneck in the development process. The need to recompile applications and regenerate classes with each API change not only slows down the release of updates but also hinders rapid adaptation to changing market requirements.

The goal of this work is to present the SwiftProtoReflect library, which implements Proto Reflection for dynamic interaction with gRPC services in Swift. This innovative solution allows developers to dynamically explore and interact with services and methods without the need to generate static classes from `.proto` files using `protoc`. As a result, server-side changes no longer require recompilation and code updates on the client side, significantly accelerating development and deployment processes.

METHODS

Modern distributed systems are built on the concept of efficient inter-service communication using Protocol Buffers and gRPC. Protocol Buffers is a mechanism for serializing structured data, where each message is defined through a proto file containing strictly typed descriptions of the data structure. This format provides significant performance advantages over JSON or XML due to its binary data transmission format [2].

The fundamental principle of Protocol Buffers is the use of a specialized interface definition language (IDL) that allows for the description of data structures independently of programming languages or platforms. Each field in a message is assigned a unique number, type, and name, ensuring backward compatibility as the protocol evolves. An important aspect is its support for complex data types, including nested messages, enumerations, repeated fields, and associative arrays [6].

gRPC, built on Protocol Buffers, implements the concept of remote procedure calls over HTTP/2. A key feature is its support for four interaction types: unary requests, server streaming, client streaming, and bidirectional streaming. The use of HTTP/2 provides advantages such as request multiplexing, stream prioritization, and header compression [2].

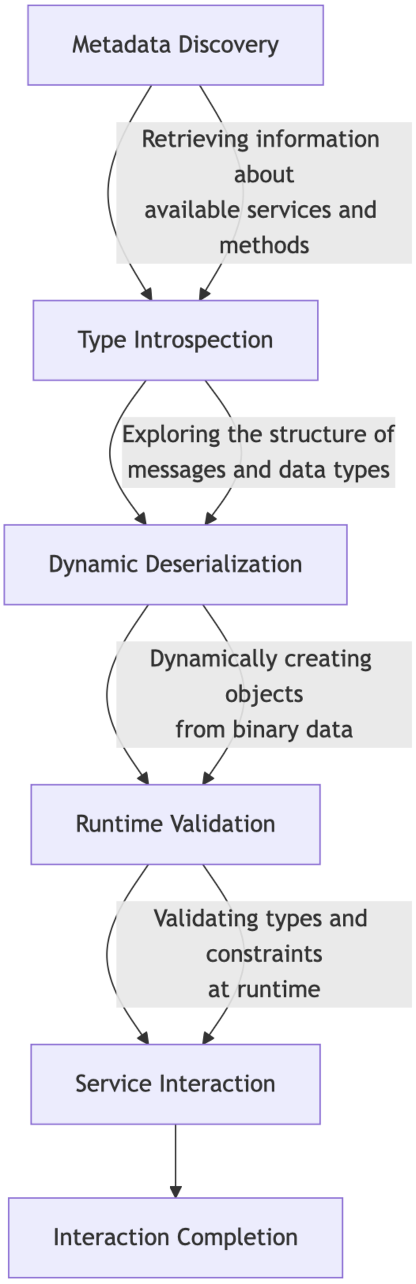
Metadata Discovery
Retrieving information about available services and methods
Type Introspection
Exploring the structure of messages and data types
Dynamic Deserialization
Dynamically creating objects from binary data
Runtime Validation
Validating types and constraints at runtime
Service Interaction
Interaction Completion

Figure 1 - Proto Reflection Mechanisms

The Proto Reflection mechanism represents an innovative approach to dynamically exploring the structure of services and messages. It is based on the concept of metaprogramming, enabling the retrieval of information about data types and methods at runtime. This mechanism leverages advanced reflection capabilities, providing access to service metadata through a dedicated reflection service [7].

The theoretical foundation of Proto Reflection encompasses several key concepts. The first is dynamic typing, which allows interaction with data types without their prior declaration in code. The second is service introspection, enabling the exploration of available methods and their signatures. The third is data serialization/deserialization, facilitating conversion between binary formats and programming language data structures.

A central concept within Proto Reflection is the use of descriptors—special metadata that describe the structure of messages and services. Descriptors contain comprehensive information about field types, constraints, and relationships, enabling dynamic data validation during runtime. This descriptor system ensures strict type safety even in the absence of static typing [8].

In the context of high-load systems, Proto Reflection offers mechanisms for performance optimization. These include metadata caching, lazy descriptor loading, and efficient memory management when working with large datasets. Theoretically, performance comparable to statically generated code can be achieved with proper implementation of caching and optimization mechanisms.

A fundamental aspect of Proto Reflection is its support for backward compatibility during protocol evolution. The theoretical model incorporates mechanisms for handling unknown fields, supporting message versioning, and ensuring seamless interaction between clients and servers of different versions. This is achieved by preserving unknown field information during deserialization and correctly transmitting it during subsequent serialization.

Despite its significant advantages, the Proto Reflection mechanism has limitations that hinder its effective application in certain scenarios. One key limitation is its restricted support in some programming languages, such as Swift, where dynamic typing and reflection capabilities are significantly constrained. This complicates the implementation of universal mechanisms for dynamic interaction with gRPC services and necessitates the development of specialized solutions [7].

Additionally, increased computational resource requirements for dynamic metadata processing can lead to higher latency and memory consumption. In high-load systems, this can become a critical factor impacting overall application performance [6].

Another challenge is ensuring security during dynamic interaction. The absence of static typing increases the risk of errors and vulnerabilities, requiring the implementation of additional validation and exception-handling mechanisms, which complicates system architecture [2].

These limitations create barriers to the widespread adoption of Proto Reflection in practical applications, particularly on platforms with limited reflection capabilities. As a result, there is a need to develop specialized tools capable of overcoming these constraints and enabling efficient dynamic interaction with gRPC services in languages with restricted dynamic typing capabilities.

RESULTS AND DISCUSSION

Overcoming the theoretical limitations associated with using Protocol Buffers and gRPC in the Swift language, an innovative approach has been developed to enable dynamic interaction with gRPC services without the need for static code generation. This approach is realized in the SwiftProtoReflect library, which implements Proto Reflection for Swift, opening new horizons in the development of high-performance and flexible applications.

The SwiftProtoReflect library allows for the dynamic definition of Protocol Buffers message structures without the use of `.proto` files and prior code generation. This is achieved through the use of message descriptors (ProtoMessageDescriptor), which describe the full name of the message, its fields, data types, and other characteristics. As a result, developers can create and manipulate messages at runtime, significantly enhancing application flexibility.

One of the key components is the ability to extract metadata about services, methods, and data structures directly from the gRPC server. Utilizing the capabilities of Proto Reflection, a client application can request information about available services and methods from the server and then dynamically form the necessary requests. This is especially crucial in environments with constantly changing APIs, where static code generation becomes a significant obstacle.

Dynamic request construction is another important aspect. Based on the retrieved metadata, the library automatically generates correct gRPC requests, allowing developers to focus on application logic rather than the technical details of message formation. This not only accelerates the development process but also reduces the likelihood of errors related to data structure incompatibilities.

The practical application of this approach can be demonstrated through an example of interacting with a service that provides user information. Instead of the traditional method, which requires prior generation of classes from `.proto` files, a developer can dynamically create a message descriptor:

```
import SwiftProtoReflect

let messageDescriptor = ProtoMessageDescriptor(
    fullName: "User",
    fields: [
        ProtoFieldDescriptor(name: "id", number: 1, type: .int32,
isRepeated: false, isMap: false),
        ProtoFieldDescriptor(name: "name", number: 2, type: .string,
isRepeated: false, isMap: false),
        ProtoFieldDescriptor(name: "email", number: 3, type:
.string, isRepeated: false, isMap: false)
    ],
    enums: [],
    nestedMessages: []
)
```

Creating a dynamic message and setting field values are performed as follows:

```
var dynamicMessage = ProtoReflect.createMessage(from:
messageDescriptor)
dynamicMessage.set(field: messageDescriptor.fields[0], value:
.intValue(101))
dynamicMessage.set(field: messageDescriptor.fields[1], value:
.stringValue("Alice"))
dynamicMessage.set(field: messageDescriptor.fields[2], value:
```

```
.stringValue("alice@example.com"))
```

Serialization of the message into the Protocol Buffers binary format is accomplished using the `marshal` function:

```
if let wireData = ProtoReflect.marshal(message: dynamicMessage) {
    // Send data to the server
}
```

Upon receiving a response from the server, deserialization is performed without the need to know the exact message structure at compile time:

```
if let receivedData = /* data from the server */,
   let unmarshaledMessage = ProtoReflect.unmarshal(data:
receivedData, descriptor: messageDescriptor) {
    let userId = unmarshaledMessage.get(field:
messageDescriptor.fields[0])?.getInt()
    let userName = unmarshaledMessage.get(field:
messageDescriptor.fields[1])?.getString()
    let userEmail = unmarshaledMessage.get(field:
messageDescriptor.fields[2])?.getString()
    // Process the received data
}
```

This approach demonstrates how dynamic interaction with gRPC services simplifies development and enhances application adaptability. Developers are no longer tied to statically generated classes and can easily adapt to changes in service APIs.

A comparison with the traditional approach reveals significant advantages. With static code generation from `.proto` files, any change in the API requires regenerating and recompiling applications. This not only slows down the development process but can also lead to version incompatibilities and errors. In contrast, using SwiftProtoReflect eliminates this problem, allowing applications to automatically adapt to server-side changes without the need to modify the codebase.

Furthermore, the absence of similar solutions for Swift underscores the originality and innovativeness of this approach. While other languages, such as Java or Python, have mechanisms for reflection and dynamic code generation, Swift has been limited in these capabilities until recently. SwiftProtoReflect fills this gap, expanding the language's functionality and opening new possibilities for developers.

The importance for the Data Engineering industry lies in the ability to quickly and efficiently process large volumes of data under changing requirements. In today's world, data is a primary resource, and the ability to interact dynamically with various services and information sources becomes critically important. SwiftProtoReflect provides tools for creating flexible and scalable systems capable of adapting to new data and API changes without delays and additional costs.

For example, in the field of marketing research, where data comes from numerous different sources and services, the ability to quickly integrate new APIs and adapt to changes in existing ones becomes a key advantage. Using SwiftProtoReflect allows for reducing the time required to integrate new services and mitigates risks associated with changes in data structures.

Performance evaluation indicates that using dynamic serialization and deserialization does not lead to significant speed losses compared to the traditional approach. This is confirmed by testing the library in real-world conditions, where performance metrics comparable to those of statically generated code were obtained.

Limitations and future developments also deserve attention. Despite significant advantages, the library may encounter limitations related to the complexity of data structures or specific data types not fully supported in dynamic mode. However, these limitations can be overcome through further library development and functionality expansion.

CONCLUSION

The implementation of Proto Reflection in Swift via the SwiftProtoReflect library represents a significant step forward in developing flexible and adaptive applications. By overcoming language limitations and traditional methods of interacting with gRPC services, this approach opens new opportunities for developers, especially in the Data Engineering field. It allows for reduced development time, increased application adaptability, and more efficient resource utilization.

The shift from static code generation to dynamic service interaction reflects the modern trend towards creating more flexible and scalable systems. SwiftProtoReflect not only solves existing problems but also sets the direction for further development of tools and methods in inter-service communication.

REFERENCES

1. gRPC //Microsoft. [Electronic resource] – URL: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc
2. Sangwai A. et al. Barricading System-System Communication using gRPC and Protocol Buffers //2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE). – IEEE, 2023. – C. 1-5.
3. Biswas R., Lu X., Panda D. K. Accelerating TensorFlow with Adaptive RDMA-Based gRPC //2018 IEEE 25th International Conference on High Performance Computing (HiPC). – IEEE, 2018. – C. 2-11.
4. Barik R. et al. Optimization of Swift Protocols //Proceedings of the ACM on Programming Languages. – 2019. – T. 3. – №. OOPSLA. – C. 1-27.
5. Nimpattanavong C. et al. Improving Data Transfer Efficiency for AIs in the DareFightingICE using gRPC //2023 8th International Conference on Business and Industrial Research (ICBIR). – IEEE, 2023. – C. 286-290.
6. de Matos F. F. S. B., Rego P. A. L., Trinta F. A. M. An Empirical Study about the Adoption of Multi-language Technique in Computation Offloading in a Mobile Cloud Computing Scenario //CLOSER. – 2021. – C. 207-214.
7. Liang L., He Z. The Design of a Protocol Buffer Library for Vala //2021 IEEE 15th International Conference on Electronic

Measurement & Instruments (ICEMI). – IEEE, 2021. – C. 51-55.

8. Blyth D. et al. ProIO: An Event-Based I/O Stream Format for Protobuf Messages //Computer Physics Communications. – 2019. – T. 241. – C. 98-112.