International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019 ISSN No: 2348-9510

EFFICIENT ENCODING USING PROTOCOL BUFFERS

Nilesh Jagnik
Mountain View, USA
nileshjagnik@gmail.com

Abstract

Protocol buffers are a mechanism for runtime handling and serialization/deserialization of structured data. Protocol buffers are a compact and performant storage format that have several advantages over traditional formats like JSON and XML. In addition, Protocol buffers provide autogenerated boilerplate that creates runtime constructs for handling data in memory. Protocol buffers are supported in multiple languages and platforms. In addition, they provide ways to make backwards compatible changes to the data structure. However, there are certain scenarios where Protocol buffers are not the ideal choice. In this paper, we look at Protocol buffers, their features and advantages, how to use them and also limitations to be aware of when using them.

Keywords: Data encoding, Data serialization, Data structure, Runtime data handling

I. INTRODUCTION

Software systems often process a large amount of data and need to store, read and transfer data between services. An important consideration here is to ensure that the services that read data interpret it in the same way as when it was written or sent. This is because written data is just bytes, and it is up to the reader to interpret what those bytes mean. This is especially true if the data has a complex structure. This structure must be maintained when the data is stored and subsequently read. So, there is a need for a standardized storage and transfer format so that complex data can be easily interpreted by different services in the same way.

There are several storage formats available that ensure data can be read and written is a consistent format. Out of these, the most commonly used ones are JSON and XML. These formats define the structure in which data should be written (when stored as bytes). They also offer parsers that allow reading back from wire format (stored format) to form the original structured data. JSON and XML formats are simple and can be easily interpreted by humans and machines both.

However, human readable formats tradeoff compactness and efficiency for simplicity. Often stored data can be pretty large to accommodate formatting. Protocol buffers are an elegant solution to this problem. They solve many of the problems associated with other storage formats with minimal drawbacks.

II. PROTOCOL BUFFERS FEATURES

Protocol buffers are a way for serialization of structured data introduced by Google. They have the following features:

A. Usage

Protocol buffers are a format suitable for both short-term network traffic and also long-term

storage of data up to a few megabytes in size. This format is well-suited towards inter-server communication.

B. Structure

The structure in which data is stored is defined by service owners. This means that the reader and writer need to know the agreed upon structure. This structure is defined in a .proto file.

C. Cross-Language Compatibility

Protocol buffers offer library for reading and writing protocol buffers, including C++, C#, Dart, Go, Java, Kotlin, Objective-C, PHP, Python and Ruby. This means that different services written in different languages can easily transfer data between each other.

D. Auto Generated Classes

Protocol buffers automatically generate classes according to the data structures defined in .proto files. Instances of data can be represented as instances of these classes. This allows easy data generation and handling during runtime. This also means that there is no need to create data classes which are generally needed for runtime management of data. This reduces boilerplate code that developers need to manage.

E. Compact Storage

Traditional data storage formats store a lot of extra information to allow readers to parse data correctly. In Protocol buffers, the reader and writer are assumed to be aware of the format. Protocol buffers can store data in a compact manner. This allows faster transfer rates and high transfer efficiency since less data needs to be sent and stored.

F. Performance

Protocol buffers parsing is faster than traditional methods leading to improved application performance.

G. Backwards Compatibility

When there is an update to the proto format defined in the .proto file, different processes may have a different understanding of the format. However proto parsers are able to parse data in a backward compatible manner. Note that this requires service owners to make updates to the proto format in a backward compatible way. Protocol buffers have clear guidelines on what changes are safe in terms of backward compatibility.

III. WORKING WITH PROTOCOL BUFFERS

Let us take a deeper look to understand how to use Protocol buffers.

A. Proto Definition

The first thing that needs to be done is to define the structure of data. This definition should be known to both the sender/writer and receiver/reader side. The proto definition should be imported as a code dependency on both sides. Fig. 1 shows an example of a proto definition.

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019 ISSN No: 2348-9510

```
message Owner {
optional string name = 1;
optional string email = 2;
}

message Vehicle {
optional string make = 1;
optional string model = 2;
optional int32 year = 3;
optional Owner owner = 4;
}
```

Fig. 1. Proto definition inside .proto file

B. AutoGenerated Classes

Protocol buffers automatically generate classes according to the proto definitions inside the .proto file. The generated classes should be imported in application code. Then classes can be used as data carriers. Generated classes have builder interfaces which can be used for easy construction of objects. Fig. 2 shows can example usage of the messages defined in Fig. 1.

```
Owner owner = Owner.newBuilder()
  .setName("Nilesh Jagnik")
  .setEmail("nileshjagnik@gmail.com")
  .build();

Vehicle vehicle = Vehicle.newBuilder()
  .setMake("Tesla")
  .setModel("S")
  .setYear(2018)
  .setOwner(owner);
```

Fig. 2. Data classes automatically generated from proto definition (Java).

C. Serializing Data

The autogenerated classes also provide helpers which can assist with serializing data. Serialized data is written following the structure defined in the .proto file. Fig. 3 shows an example of how convenient it is to serialize a proto (short for Protocol buffer) message.

```
OutputStream out = new FileOutputStream("vehicles");
vehicle.writeTo(out);
```

Fig. 3. Seralizing contents of a proto message to a file (Java).

D. Deserializing Data

Similar to serialization, there are also helpers that allow easy deserialization and parsing of data. This can also be done in a different language. Fig. 4 shows a C++ program reading the data serialized by Fig. 3.

```
Vehicle vehicle;
streams input("vehicle");
vehicle.ParseFromIstream(&input);

Owner owner = vehicle.owner();
std::string make = vehicle.name();
```

Fig. 4. Deserializing data using proto buffer helpers (C++).

IV. PROTOCOL BUFFERS AND GRPC

In addition to simple inter-service transfer of data, Protocol buffers can also be used for defining structured RPC interactions between services. The gRPC framework uses Protocol buffers for defining service interfaces. This framework extends the benefits of Protocol buffers to create a cross-language RPC framework. gRPC automatically handles transfer of structured data between client and server. This enables clients and servers to simply focus on developing application logic.

V. LIMITATIONS OF PROTOCOL BUFFERS

There are certain scenarios which Protocol buffers are not ideal for.

A. Large Data Size

Protocol buffers are load messages fully in memory. In practice, it is possible to create several copies of the same data due to the way code is written. In this case, Protocol buffers force the usage of a lot of memory. For data larger in size than a few megabytes, Protocol buffers are not the best choice.

B. Comparison of Serialized Data

A Protocol buffer can be serialized in different ways even when the data represented by it remains same. Due to this, serialized data should first be parsed into memory before comparison.

C. No Compression

Protocol buffers are not meant for saving storage space. They have no built-in support for compression.

D. Scientific Data

Protocol buffers are not ideal for representing scientific data involves matrix operations on multi-dimensional matrices. The speed and size of this type of data using Protocol buffers is not ideal.

E. Object Oriented

Protocol buffers are not ideal for use in languages that do not use object-oriented programming. Protocol buffers has limited support for languages like Fortran and IDL.

F. No Self-Description

There is no way to interpret a serialized Protocol buffer without access to the proto definition. This limits the usage to cases where proto definition can be shared between services.

G. Formalization

Protocol buffers are not suited to legal data requirements since they are not regulated by any regulatory bodies.

VI. CONCLUSION

In conclusion:

1. Protocol buffers are a great solution for handling structured data, including runtime handling as well as serialization and deserialization for storage and transfer. They are compact, fast and efficient.

2. They provide a lot of boilerplate code for runtime management and reading/writing data.
3. They provide many other features such as cross language support and backwards compatibility in case of a schema change.
4. They are easy to work with but do have a learning curve for beginners. There are certain best practices that must be followed when using Protocol buffers.
5. There are certain scenarios and use cases where Protocol buffers are not the ideal choice.

REFERENCES

1. Sasha Rezvina, "5 Reasons to Use Protocol Buffers Instead of JSON for Your Next Service (Jun 2014)," https://codeclimate.com/blog/choose-protocol-buffers
2. "Protocol Buffers Documentation (Jun 2018)," https://protobuf.dev/overview/
3. Marty Kalin, "How to use Protobuf for data interchange (Oct 2019)," https://opensource.com/article/19/10/protobuf-data-interchange
4. Bruno Krebs, "Beating JSON performance with Protobuf (Jan 2017)," https://auth0.com/blog/beating-json-performance-with-protobuf/
5. John Doak, "Protocol buffers: Avoid these uses (Aug 2017)," http://www.golangdevops.com/2017/08/16/why-not-to-use-protos-in-code/
6. "Protocol Buffers vs. JSON (Nov 2018)," https://bizety.com/2018/11/12/protocol-buffers-vs-json/
7. Blake Smith, "A Primer on Protocol Buffers (Sep 2012)," https://blakesmith.me/2012/09/05/a-primer-on-protocol-buffers.html
8. "gRPC | Guides (Sep 2019)," https://grpc.io/docs/guides/