Design Doc: gRPC based protocol buffer message Differential Service

Status: Active

Author: [jinhuangzheliu@google.com](mailto:jinhuangzheliu@google.com)

Reviewers: [xihan@google.com](mailto:xihan@google.com), [xuexuan@google.com](mailto:xuexuan@google.com) wzong@google.com

Last Update: 06/22/2020

[1. Objective:](#_yi0ktl7p773o)

[2. Background](#_iyu0we9dk7yp)

[3. System Design:](#_955eo1xz5cru)

[3.1 Prerequisites:](#_auzzc27bs085)

[3.2 Client-Side Process](#_ozy9356atf3y)

[3.3 Server-Side Process](#_xoaw0swyxnvf)

[3.4 Differential Configuration and Libraries](#_6vrlkiw3w5bo)

[3.4.1 Ignore Criteria](#_bvi75z23cyq2)

[3.4.2 Repeated Field Comparison:](#_v6g7xjc116rc)

[3.4.3 Map-Value Comparison:](#_ngsbmoo4cyqi)

[3.4.4 Float Number Comparison:](#_bixx3ez20dp9)

[4. Project information](#_va28a2a4ofzz)

[4.1 Caveats](#_p0lcv36hxtkm)

[4.2 Security Considerations](#_u7ez28r0pjyh)

[4.3 Privacy Considerations](#_9kudr1ouqlro)

[5. Testing Plan](#_gtk1xoogdmly)

[6. Work Estimates](#_77vxku2jxhm6)

[7. Protintial Patents](#_qztbjbye0ixh)

# 1. Objective:

Creating an open source gRPC based differential service to enable users to differentiate the differences between paired protocol buffer messages. By comparing the user inputs to find the abnormalities that may lead to unexpected behavior of a software system, this service has great potential use cases in debugging, testing, or monitoring software systems. The service is based on gRPC framework and MessageDifferencer libraries.

# 2. Background

The differential service is implemented in multiple developing areas including runtime error monitoring, experiment ro debugging tools, and end-to-end testing. As a high performance RPC framework supporting the cross-platform service and binding for multiple languages, the [gRPC](https://grpc.io/) framework is widely used by software developers to exchange the data among services.

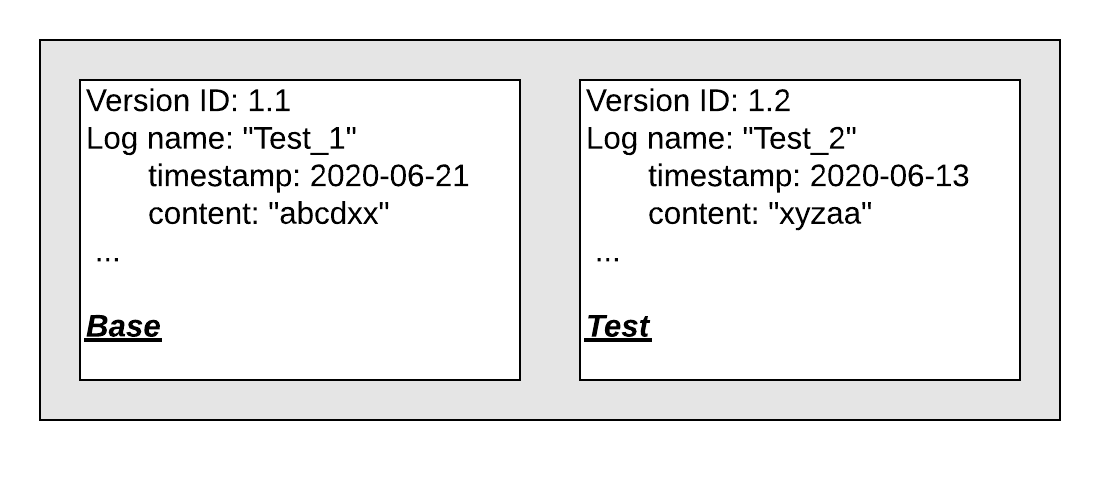
Although gRPC has been well designed and developed by Google and the open-source community, it still lacks a service to implement the end-to-end differential service for protocol buffer messages.

In our design, we propose a gRPC based differential service leveraging the Protocol Buffers message. Because of the variety of the user interfaces it's a challenge to deal with all the various user inputs from different usage scenarios. Therefore, the users have to structure their inputs as followed by a specific format regularized by our system. [Protocol Buffer](https://en.wikipedia.org/wiki/Protocol_Buffers)(protobuf) is a method for serializing the structured data and provides the APIs for multiple languages. Because of the simplicity and is available to multiple languages, the protocol buffer is used by gRPC framework as the definition language and message format.

In our system, we will use the protobuf API, [message\_differencer](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.message_differencer), to implement the differential service.

As a practice of differential service, [differential testing](https://en.wikipedia.org/wiki/Differential_testing) is a testing technique that attempts to overcome the challenges of testing real production pipelines. It executes two versions of a software system, base and test, on the same test input and compares resulting outputs to identify unexpected behaviors. In this document, we employ gRPC framework to offer a differential service that is capable of providing a comprehensive comparison for two versions of resulting outputs(the input of differential service) to exploit non-identity fields between each other.

For using the protobuf message as the input for our system, the users have to define the .proto file for their protobuf message. Listing 1 shows a snippet of a diff testing log that introduces two results from different versions of a software system. In this log snippet, the fields “version Id” and “output name” are parallel each other and the “timestamp” and “content” fields are hierarchically nested under the field “output name”.





Listing 2(a) presents an abstract .proto file template, and a concrete example for the example in Listing 1 shown in Listing 2(b).

A screenshot of a cell phone

Description automatically generated A screenshot of a cell phone

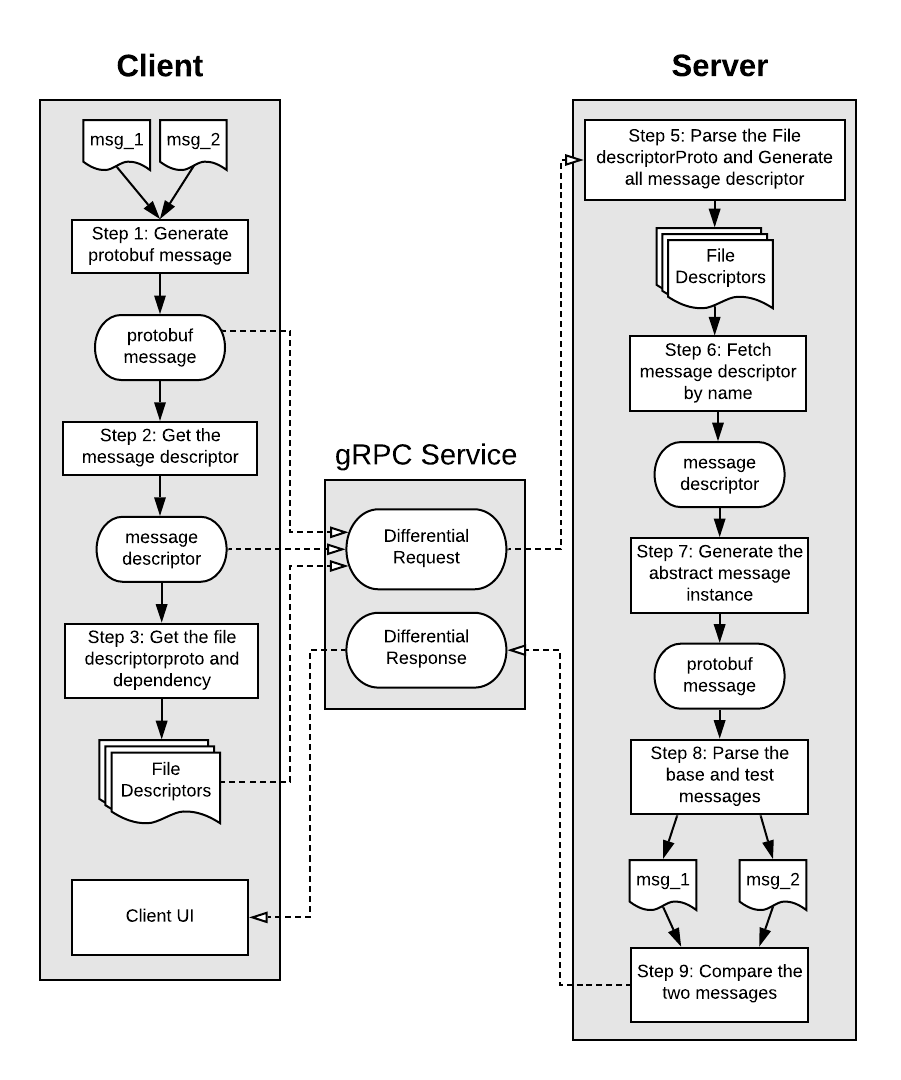
Description automatically generated





# 3. System Design:

Take advantage of the gRPC framework our system provides a comprehensive and customized differential service for end-to-end testing or runtime monitoring. Because the differential service will be used in many platforms, and the users will structure the protobuf message following their specific requirements. Therefore, It’s impossible to cache all of the input messages(e.g. .proto file) in the server. To answer this challenge, this server is able to dynamically resolve the user’s input messages with the concrete contents no matter how complex the structure of the message. We will illustrate our design literally and graphically in the remaining of this document.





Client Side:

* Generate the protobuf message as the inputs of the service.
* Get the message Descriptor and FileDescriptor of the message with all its dependency.
* Send the messages, message descriptor, and FileDescriptorProto to the server using the gRPC request message(Differential Request).6

Server Side:

* Parse all FileDescriptorProtos and convert it to a set of cross-linked Descriptors
* Fetch the specific message descriptor by its name.
* Create an abstract message instance by the message descriptor.
* Using the abstract instance to reconstruct the input messages from the client.
* Compare the two messages and output the differential result.

Figure 1 presents the architectural overview of the differential service. On the client side, the user first needs to structure the input message, msg\_1 and msg\_2, as the protobuf message.

Once the two messages are created without any errors, the user needs to obtain the [Descriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor) of the input message that is a class describing a specific instance of protobuf message(i.e. Step 1 and 2 in Figure 1). Next, for each message we collect all their [FileDescriptorProtos](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.pb) and the dependency of them, which is illustrated as Step 3 in Figure 1. For the differential request messages, the user has to encapsulate the original input messages, message descriptor, and FileDescriptorProtos with all its dependency (i.e. generated from Step 1, 2, and 3) into a protobuf message to communicate with the differential server.

On the server side of our service, the [DescriptorPool](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor#DescriptorPool) will convert all FileDescriptorPool to a set of cross-linked Descriptors. The system will next fetch the Descriptor of the input message by its name. The system will then yield an abstract message instance by leveraging this message descriptor. This abstract message precisely represents the input message including i) the structure and ii) the field type of this message. After that, the system will parse the original input message and construct the message with concrete values for the upcoming differential comparison. These activities are illustrated as Step 4, 5, 6, and 7 in Figure 1. The system will next compare these two input messages. Such the comparison process will leverage the [message\_differencer](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.message_differencer) to compare the messages and send the result to the client using the protobuf message.In our differential service, we propose two kinds of differential services that will be illustrated in the remaining of this session.

## 3.1 Prerequisites:

Our differential service is established on the gRPC framework by C++ language. To implement our service, you have to install the compiler for C++(e.g. gcc), the process building tool(e.g. cmake), and Protocol Buffer in the system. You could follow the gRPC [official reference](https://grpc.io/docs/languages/cpp/quickstart/) to install all these prerequisites step by step.

## 3.2 Client-Side Process

As mentioned in the beginning of this session, it's a challenge to have all message definitions for different software systems. Therefore, we decide to parse and reconstruct the input message dynamically on the server at runtime. As the official reference of Protobuf said: “You can use a message's descriptor to learn at runtime what fields it contains and what the types of those fields are.” The library google::protobuf::Descriptor supports these following classes, [FieldDescriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor#FieldDescriptor), [Descriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor), and [FileDescriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor#FileDescriptor), to describe the field of message, message, and .proto file as presented at Figure 2.

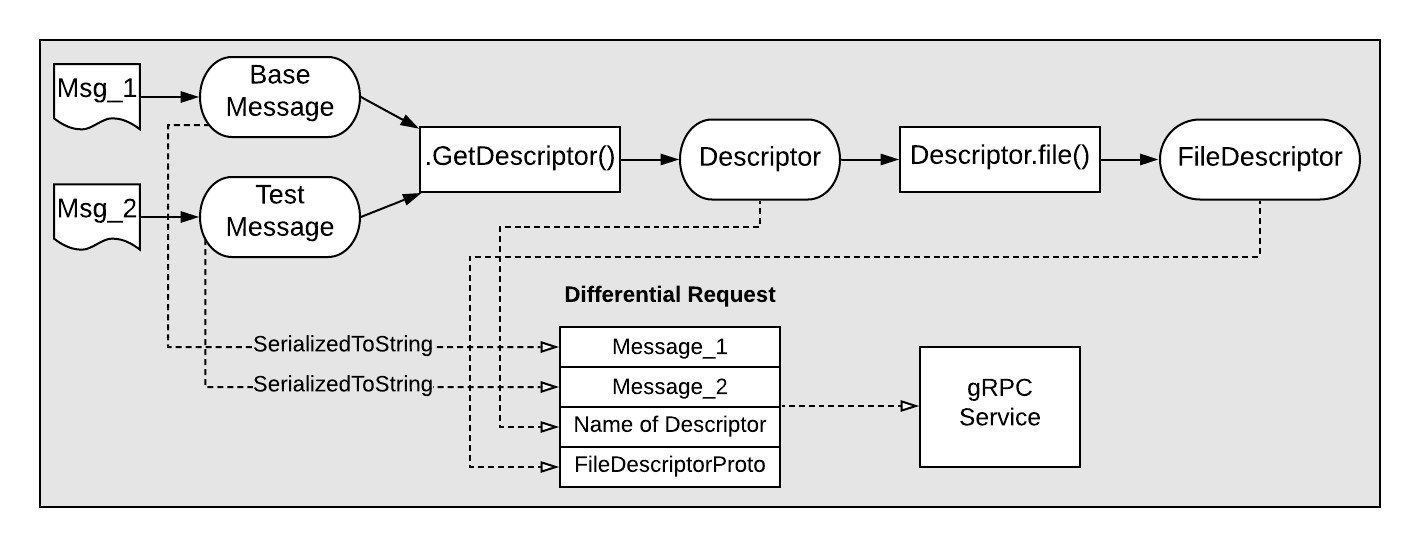
A close up of a map

Description automatically generated



The class FileDescriptor represents a compile .proto file; the class Descriptor corresponds to a specific message that is defined in the .proto file; and FieldDescriptor describes a concert field in a message. Therefore, the FileDescriptor and Descriptor could be utilized for us to build a specific message at runtime dynamically. Transmitting the FileDescriptor and its dependency through the gRPC service often consumes more space rather than only passing a specific message Descriptor. However, passing a message with a complex structure will extremely increase the developers workloads because they have to bind all nested message descriptors that may raise unexpected errors or bugs while the developer does not obtain all necessary descriptors for the input message.

Given this trade-off, we decide to encapsulate all FileDescriptors, name of the message descriptor, and the two testing messages into a rpc message. And then send it to the server as presented in Figure 3.

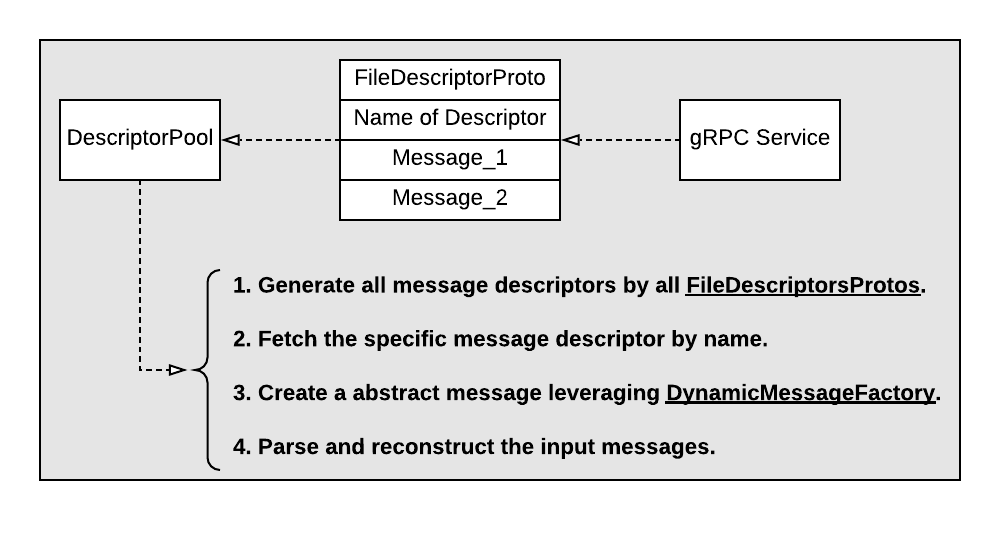




In order to format the user’s input and reduce their workload, we also developed a series of methods to get over the non-necessary mistakes or potential bugs that were defined in the class [client\_util.h](https://github.com/jinhuangzheliu/gRPC-Differential-Service/blob/master/differential_client/client_util.h). By using the method “DiffRequest WriteMsgToDiffRequest()” users could generate the request for the differential service. The method will automatically capture the message Descriptor and FileDescriptorProtos for the input messages.

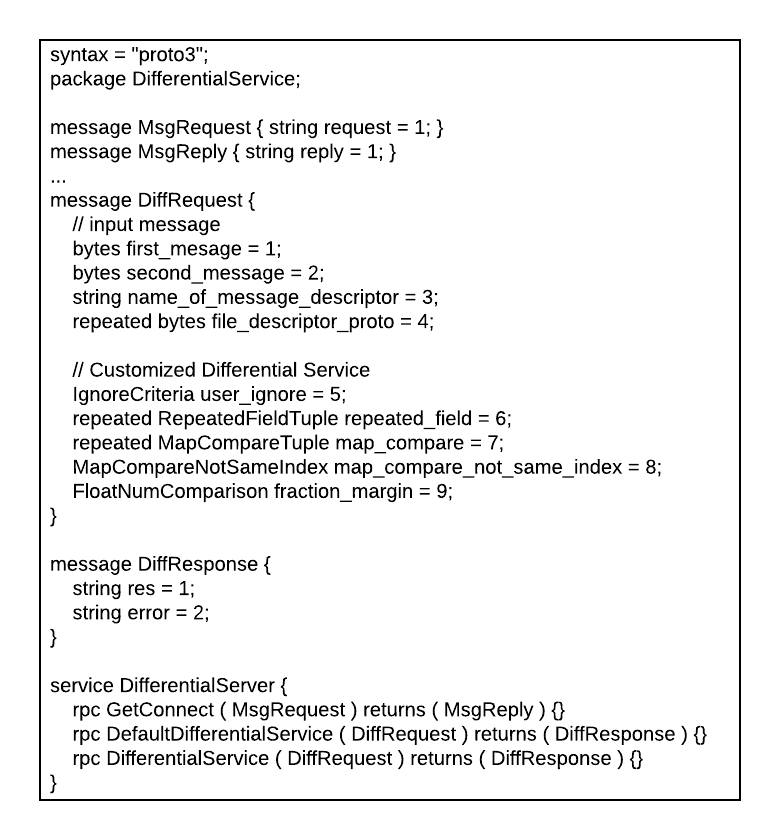
## 3.3 Server-Side Process

In the server-side process, the system will dynamically reconstruct the input messages. It first uses the class DescriptorPool to construct all message descriptors by parsing the all FileDescriptors and its dependency. Next, for acquiring a specific message descriptor, the process will exploit the descriptor by its name. The process will then create an abstract message instance and reconstruct the input messages, msg\_1 and msg\_2. These steps are presented in Figure 4. The server-side process will next compare whether thinputt messages are different.



## 3.4 Differential Configuration and Libraries

In this system, we use gRPC to communicate with the client and server. The configuration of the differential service was shown in Listing 3, where defines the differential service with the specified name “DifferentialServer'' and others three rpc methods.





The method GetConnect() specifies the request and response types to check the connection between the client and server that can help the user to check whether the gRPC connection is established or not. The method DefaultDifferentialService() will directly leverage the default API of [message\_differencer.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.message_differencer) to compare the user’s input messages without any customization.

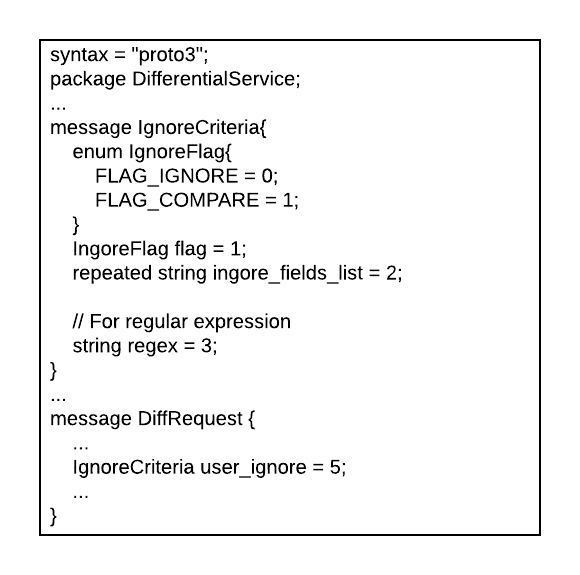
Beside the default differential service we designed a customized differential service(rpc DifferentialService() in Listing 3). By leveraging this method users can compare the messages according to their personal specifications. In this method we design four kinds of interfaces:

1. **IngoreCriteria user\_ignore**. Users could ignore one or more specific fields during the messages comparison.
2. **RepeatedFieldTuple repeated\_field**. Users could compare the elements in a repeated field as a list-based comparison or a set-based comparison.
3. **MapCompareTuple map\_compare**. Users could also compare the repeated field as a Map-Value pair.
4. **FloatNumComparison fraction\_margin**. Users could set the fraction and margin when the float or double type of numbers were compared.

As the declaration of our differential services users have to use the “DiffRequest”(in Listing 3) to pass their input messages to our server. We also support five optional fields (field number 5-9) to customize the user’s request as presented in Listing 3.

### 3.4.1 Ignore Criteria

Ignore criteria is used by users to offer the specific fields to be ignored during the messages comparison. To implement the ignore fields, we created a “IgnoreCriteria” interface in the differential service .proto file to offer this operation as presented in Listing 4. It is worth mentioning that it handles “IgnoreFlag” as an enumerated type field with two elements “IGNORE” and “COMPARE”. Users could implement the “IngoreFlag” to ignore some specific fields or only compare them during the message differentiation. The element “ignore\_field\_list” is used to save the full name of the specific fields. Beside setting by the IgnoreFlag, we also support ignoring fields by regular expression. Users could ignore fields by regular expression match(regex in Listing 4).





In the class [clinet\_util.h](https://github.com/jinhuangzheliu/gRPC-Differential-Service/blob/master/differential_client/client_util.h) we support three methods to the user for setting the ignore criteria.

* void IgnoreFields(DiffRequest\* diff\_request,

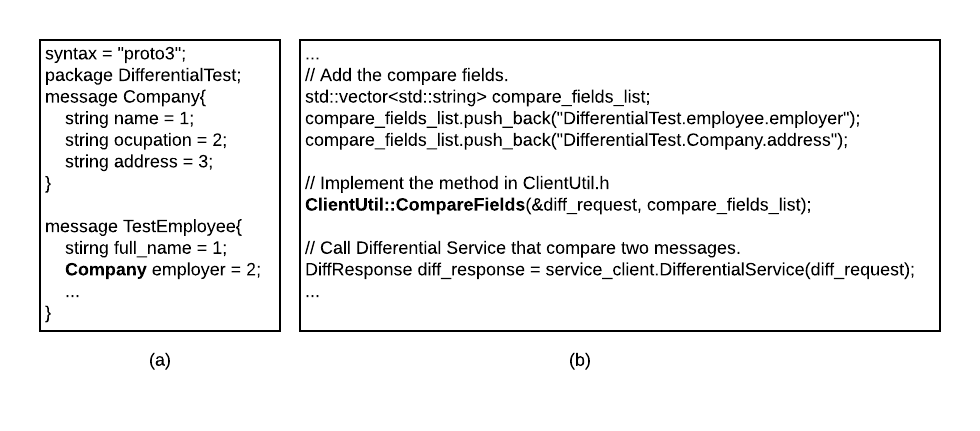
const std::vector<std::string>& field\_list)

* void CompareFields(DiffRequest\* diff\_request,

const std::vector<std::string>& field\_list)

* void RegexCriteria(DiffRequest\* diff\_request,const std::string& regex)

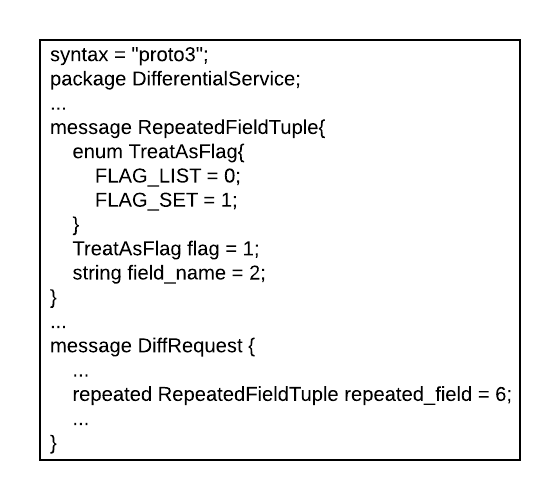
Listing 5 presents an example, where the method void CompareFields() was implemented to compare the differential of the company’s address and ignore all other fields in the TestEmployee message. From the .proto file, presented as Listing 5(a), we know that the company message is nested under the employee message declared as “employer” field. Therefore, as presented as Listing 5(b), before adding the full name of the “DifferentialTest.company.address” users have to add the full name of its parent field, “DiffereentialTest.employee.employer”, into the “compare\_fields\_list” previously. We use the full name of the field to eliminate the bug caused by the duplication of the name from different fields. The method void CompareFields() will add the compared fields to the differential request and the server will only compare the difference between the company address field.





### 3.4.2 Repeated Field Comparison:

Rather than the scalar type of field, the repeated field could be compared as list or set based elements. We support an interface to compare the repeated field according to the user’s specifications. As presented in the Listing 6, when the repeated\_field is set in message log it will work in two ways:





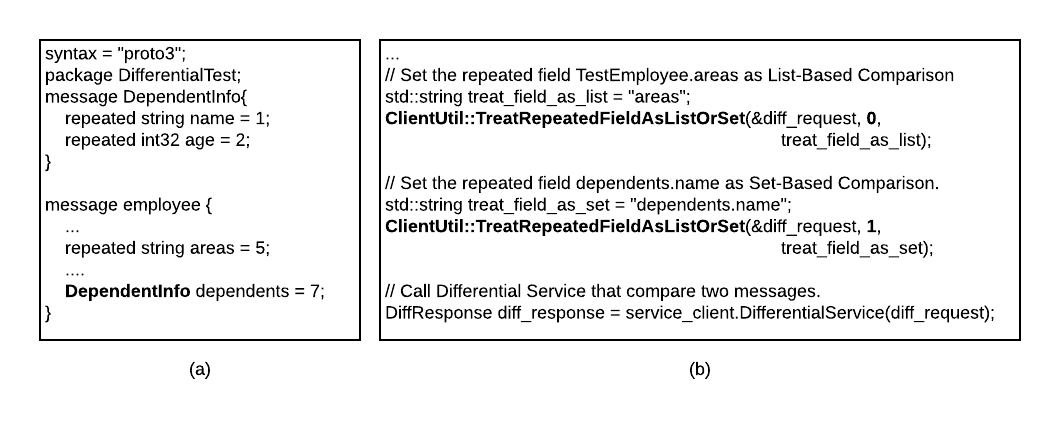
1. When the flag is set as FLAG\_LIST, the repeated field will be compared as the order supported by the user. It will stop to compare if a differential is detected between two messages, which means the differential after the first differential will not be compared. This differential will be reported to the result and return the differential to the user.
2. When the flag is set as FLAG\_SET, all elements in the repeated field will be differentiated by an unordered set that means no matter the order of the user input, if the value of each element is similar with each other, the service will return “SAME” to the user.

In the class [clinet\_util.h](https://github.com/jinhuangzheliu/gRPC-Differential-Service/blob/master/differential_client/client_util.h) we support the method, TreatRepeatedFieldAsListOrSet, to the user for setting the repeated field comparison.

* void TreatRepeatedFieldAsListOrSet(DiffRequest\* diff\_request,const int flag,

const std::string& field\_name);

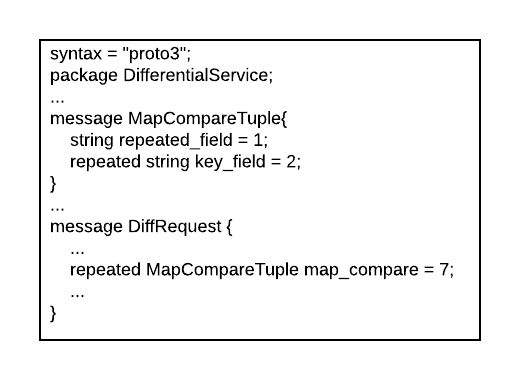
In the method ClientUtil::TreateRepeatedFieldAsListOrSet(), we declare the second parameter as an integer type value named flag. When the “flag” is passed by integer zero(integer 0), the field passed by the third argument will be treated as list-based comparison. if integer 1 passed to the flag the field will be treated as set-based comparison. Listring 7 shows an example, where the user plans to compare the repeated field “employee.areas” as set-based comparison and the field “employee.dependents.name” as a list-based comparison. It is worth to attention that users must guarantee all elements defined in this repeated field as the same data type(e.g. Listing 7(a)). In this example, all elements were defined as string type. If the elements are not the same type, users have to compare the repeated field as Map-Value comparison. Listing 7(b) shows an implementation by the method we supplied in client\_util.h for comparison of these two repeated fields.





### 3.4.3 Map-Value Comparison:

We provide the map-value comparison for the repeated field in our differential service. Users could select one or more fields as the “Key” field. The interface was presented in Listing 8. By leveraging this interface users could set the repeated field in “repeated\_field” and determine the key fields by “key\_field”.





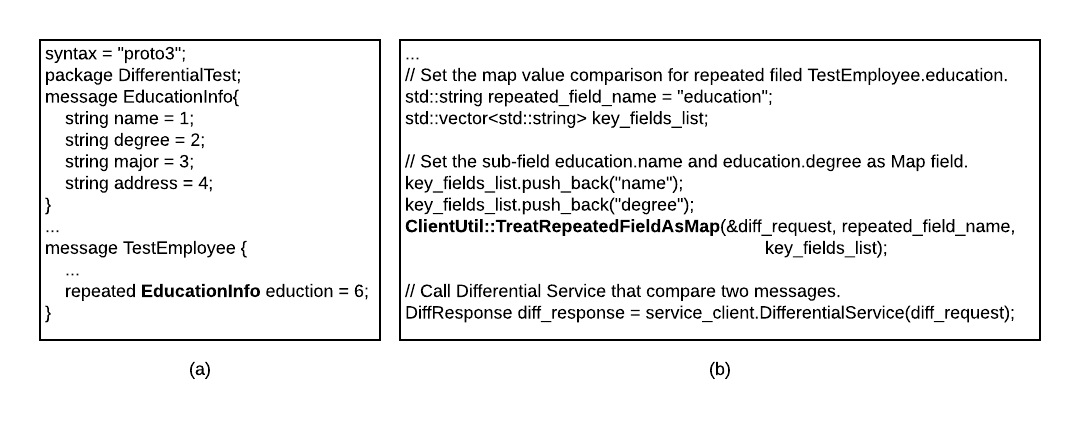
In the class [clinet\_util.h](https://github.com/jinhuangzheliu/gRPC-Differential-Service/blob/master/differential_client/client_util.h) we support the method, TreatRepeatedFieldAsMap, to set the map-value comparison.

* void void TreatRepeatedFieldAsMap(DiffRequest\* diff\_request,

const std::string& field\_name,

const std::vector<std::string>& sub\_field\_name);

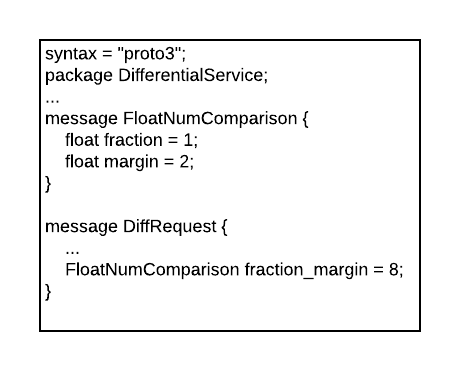
Here is an example for map-value comparison as presented in Listing 9. In this example, users want to compare the repeated field “employee.education” with two sub-fields “education.name” and “education.degree” as the map field.



****

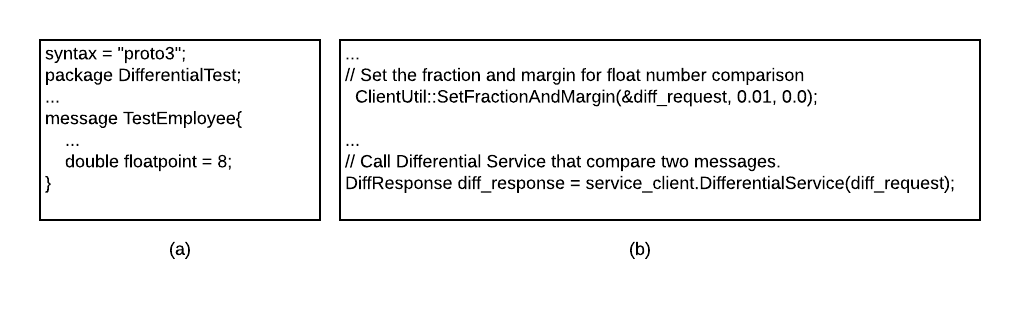
### 3.4.4 Float Number Comparison:

The float or double type of number could be compared by the fraction and margin. Users can set them on the interface “fraction\_margin = 8” in message log, as presented in the Listing 10.





In the class [clinet\_util.h](https://github.com/jinhuangzheliu/gRPC-Differential-Service/blob/master/differential_client/client_util.h) we support the method, SetFractionAndMargin, to set the fraction and margin during the differential service. Listing 11 presents an example that sets the fraction as 0.01 and margin as 0.0 for comparing the double number between the two messages.





# 

# 4. Project information

Project Association: EngProd - Search Ads Quality Team

Primary: jinhuangzheliu@

Reviewers: xihan@, xuexuan@, wzong@

GitHub repository: <https://github.com/jinhuangzheliu/gRPC-Differential-Service>

## 4.1 Caveats

N/A

## 4.2 Security Considerations

Transport security:

gRPC messages are sent and received using HTTP/2. We recommend:

* [Transport Layer Security (TLS)](https://tools.ietf.org/html/rfc5246) be used to secure messages in production gRPC apps.
* gRPC services should only listen and respond over secured ports.

TLS is configured in Kestrel. For more information on configuring Kestrel endpoints, see [Kestrel endpoint configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-3.1#endpoint-configuration).

Exceptions:

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, gRPC doesn't send the details of an exception thrown by a gRPC service to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example, in development or test) with [EnableDetailedErrors](https://docs.microsoft.com/en-us/aspnet/core/grpc/configuration?view=aspnetcore-3.1#configure-services-options). Exception messages shouldn't be exposed to the client in production apps.

Message size Limits: Incoming messages to gRPC clients and services are loaded into memory. Message size limits are a mechanism to help prevent gRPC from consuming excessive resources.

gRPC uses per-message size limits to manage incoming and outgoing messages. By default, gRPC limits incoming messages to 4 MB. There is no limit on outgoing messages.

## 4.3 Privacy Considerations

Client certificate validation:

[Client certificates](https://tools.ietf.org/html/rfc5246#section-7.4.4) are initially validated when the connection is established. By default, Kestrel doesn't perform additional validation of a connection's client certificate.

We recommend that gRPC services secured by client certificates use the [Microsoft.AspNetCore.Authentication.Certificate](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/certauth?view=aspnetcore-3.1) package. ASP.NET Core certification authentication will perform additional validation on a client certificate, including:

* Certificate has a valid extended key use (EKU)
* Is within its validity period
* Check certificate revocation

# 5. Testing Plan

We leverage the [Google Test](https://en.wikipedia.org/wiki/Google_Test) to perform a total of 50 unit-tests for our system. In these unit-tests include:

* 10 Unit-tests testing the ignored field criteria during the message differentiation.
* 7 Unit-tests testing the functionality of treating the repeated field as LIST.
* 11 Unit-tests testing the functionality of treating the repeated field as SET.
* 8 Unit-tests testing the functionality of treating the repeated field as MAP.
* 6 Unit-tests testing the functionality of comparing the float/double number with the fraction and margin.
* 8 Unit-tests testing the differential service comprehensively.

The unit test shows that the four interfaces we supported could be implemented isolated. The test file was uploaded to the repo. of GitHub.

# 6. Work Estimates

**Phase 1, Prerequisites:**

* Installing the gRPC, protocol buffers and realize the example in gRPC tutorial. (Finished)
* Generating a simple gPRC service and realizing the message\_differencer.(Finished)

**Phase 2, gRPC Service:**

* Getting the Descriptor and FileDescriptor of the client-side message. (Finished)
* Defining a gRPC service that communicates with the client and server. (Finished)
* Reconstructing the message from Descriptor and FileDescriptors in the server-side process. (Finished)

**Phase 3, Message Comparison**

* Implements the APIs in message\_differencer to compare the difference from the input messages.(Finished)

**Phase 4, User Feedback:**

* Collect feedback and make changes as appropriate.

# 7. Protintial Patents

N/A