gRPC Technology White Paper

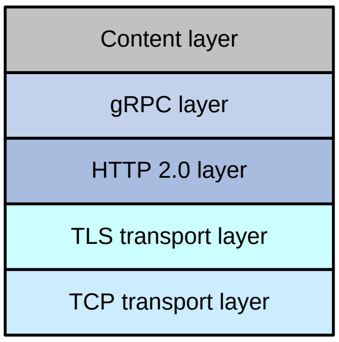
Contents

Overview

gRPC is an open source remote procedure call (RPC) system initially developed at Google. It uses HTTP 2.0 and provides network device configuration and management methods that support multiple programming languages.

gRPC protocol stack layers

Figure 1 gRPC protocol stack layers



The following describes the gRPC protocol stack layers from the bottom to the top:

- **TCP transport layer**—Provides connection-oriented reliable data links.
- **TLS transport layer**—Provides channel encryption and mutual certificate authentication. This layer is optional.
- **HTTP 2.0 layer**—Carries gRPC. This layer features in header field compression, allowing multiple concurrent exchanges on the same connection, and flow control.
- **gRPC layer—**Defines the interaction format for RPC calls. Public proto definition files such as the **grpc_dialout.proto** file define the public RPC methods.
- **Content layer**—Carries encoded service data. This layer supports the following encoding formats:
  - **Google Protocol Buffer (GPB)**—High-efficient binary encoding format. This format uses proto definition files to describe the data structure for encoding. Compared with JSON, GPB has a higher data transmission performance.
  - **JavaScript Object Notation (JSON)**—Lightweight data exchange format. It uses a text format independent of the encoding language to store and represent data, which can be easily read and compiled.

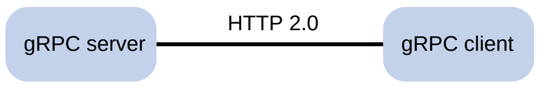
    If service data is in JSON format, you can use the public proto flies rather than the protocol files of the service modules to decode the data.

  For service data to be correctly decoded, make sure the device and the collectors use the same proto definition files

Network architecture

As shown in Figure 2, the gRPC network uses the client/server model. It uses HTTP 2.0 for packet transport.

Figure 2 gRPC network architecture



The gRPC network uses the following mechanism:

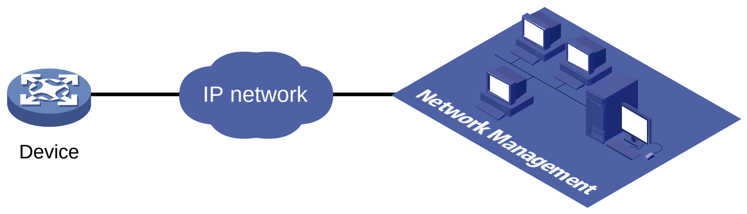
1. The gRPC server listens to connection requests from clients at the gRPC service port.
2. A user runs the gRPC client application to log in to the gRPC server, and uses methods provided in the .proto file to send requests.
3. The gRPC server responds to requests from the gRPC client.

The device can act as the gRPC server or client.

Telemetry technology based on gRPC

Telemetry is a remote data collection technology for monitoring device performance and operating status. H3C telemetry technology uses gRPC to push data from the device to the collectors on the NMSs. As shown in Figure 3, after a gRPC connection is established between the device and NMSs, the NMSs can subscribe to data of modules on the device.

Figure 3 Telemetry technology based on gRPC



Telemetry modes

The device supports the following telemetry modes:

- **Dial-in mode**—The device acts as a gRPC server and the collectors act as gRPC clients. A collector initiates a gRPC connection to the device to subscribe to device data.

  Dial-in mode applies to small networks where collectors need to deploy configurations to devices.
- **Dial-out mode**—The device acts as a gRPC client and the collectors act as gRPC servers. The device initiates gRPC connections to the collectors and pushes device data to the collectors as configured.

  Dial-out mode applies to larger networks where a large number of devices need to be monitored.

Protocol buffer code

Protocol buffer code format

Google Protocol Buffers provide a flexible mechanism for serializing structured data. Different from XML code and JSON code, the protocol buffer code is binary and provides higher performance.

Table 1 compares a protocol buffer code format example and the corresponding JSON code format example.

Table 1 Protocol buffer and JSON code format examples

Protocol buffer code format example	Corresponding JSON code format example
	{
{	"producerName": "H3C",
1:“H3C”	"deviceName": "H3C",
2:“H3C”	"deviceModel": "H3C Simware",
3:“H3C Simware”	"sensorPath": "Syslog/LogBuffer",
4:“Syslog/LogBuffer”	"jsonData": {
5:"notification": {	"notification": {
"Syslog": {	"Syslog": {
"LogBuffer": {	"LogBuffer": {
"BufferSize": 512,	"BufferSize": 512,
"BufferSizeLimit": 1024,	"BufferSizeLimit": 1024,
"DroppedLogsCount": 0,	"DroppedLogsCount": 0,
"LogsCount": 100,	"LogsCount": 100,
"LogsCountPerSeverity": {	"LogsCountPerSeverity": {
"Alert": 0,	"Alert": 0,
"Critical": 1,	"Critical": 1,
"Debug": 0,	"Debug": 0,
"Emergency": 0,	"Emergency": 0,
"Error": 3,	"Error": 3,
"Informational": 80,	"Informational": 80,
"Notice": 15,	"Notice": 15,
"Warning": 1	"Warning": 1
},	},
"OverwrittenLogsCount": 0,	"OverwrittenLogsCount": 0,
"State": "enable"	"State": "enable"
}	}
},	},
"Timestamp": "1527206160022"	"Timestamp": "1527206160022"
}	}
}	}
	}

Proto definition files

You can define data structures in a proto definition file. Then, you can compile the file with utility protoc to generate code in a programing language such as Java and C++. Using the generated code, you can develop an application for a collector to communicate with the device.

H3C provides proto definition files for both dial-in mode and dial-out mode.

Proto definition files in dial-in mode

Public proto definition files

Dial-in mode supports the **grpc_service.proto** public proto definition file, which defines the public RPC methods in dial-in mode.

The **grpc_service.proto** file is provided by H3C. The following are the contents of the file:

```
syntax = "proto2";
package grpc_service;
message GetJsonReply {  // Reply to the Get method
   required string result = 1;
}
message SubscribeReply { // Subscription result
   required string result = 1;
}
message ConfigReply {   // Configuration result
   required string result = 1;
}
message ReportEvent {   // Subscribed event
   required string token_id = 1;    // Login token_id
   required string stream_name = 2; // Event stream name
   required string event_name = 3;  // Event name
   required string json_text = 4;   // Subscription result, a JSON string
}
message GetReportRequest{ // Obtains the event subscription result
   required string token_id = 1; // Returns the token_id upon a successful login
}
message LoginRequest {  // Login request parameters
   required string user_name = 1; // Username
   required string password = 2; // Password
}
message LoginReply {  // Reply to a login request
   required string token_id = 1; // Returns the token_id upon a successful login
}
message LogoutRequest { // Logout parameter
   required string token_id = 1; // token_id
}
message LogoutReply { // Reply to a logout request
   required string result = 1; // Logout result
}
message SubscribeRequest { // Event stream name
   required string stream_name = 1;
}
message CliConfigArgs { // Sends a configuration command and the parameters to the device
   required int64 ReqId = 1; // Request ID of the command
   required string cli = 2;  // Command line of the configuration command
}
message CliConfigReply { // Reply to a configuration command execution request
   required int64 ResReqId = 1;  // Request ID, which corresponds to that in CliConfigArgs
   required string output = 2; // Output from the command
   required string errors = 3; // Command execution result
}
message DisplayCmdArgs { // Sends a display command and the parameters to the device
   required int64 ReqId = 1; // Request ID of the command
   required string cli = 2;  // Command line of the display command
}
message DisplayCmdReply { // Reply to a display command execution request
   required int64 ResReqId = 1;  // Request ID, which corresponds to that in DisplayCmdArgs
   required string output = 2;  // Output from the command
   required string errors = 3;  // Command execution result
}
service GrpcService {   // gRPC methods
   rpc Login (LoginRequest) returns (LoginReply) {}  // Login method
   rpc Logout (LogoutRequest) returns (LogoutReply) {}   // Logout method
   rpc SubscribeByStreamName (SubscribeRequest) returns (SubscribeReply) {} // Event
subscription method
   rpc GetEventReport (GetReportRequest) returns (stream ReportEvent) {} // Method for
obtaining the subscribed event
   rpc CliConfig (CliConfigArgs)  returns (stream CliConfigReply) {} // Method for
executing a configuration command and returning the execution result
   rpc DisplayCmdTextOutput(DisplayCmdArgs)  returns(stream DisplayCmdReply) {} //
Method for executing a display command and returning the execution result
}
```

Proto definition files for service modules

The dial-in mode supports proto definition files for the following service modules: Device, Ifmgr, IPFW, LLDP, and Syslog.

The following are the contents of the **Device.proto** file, which defines the RPC methods for the Device module:

```
syntax = "proto2";
import "grpc_service.proto";
package  device;
message DeviceBase { // Structure for obtaining basic device information
   optional string HostName = 1;    // Device name
   optional string HostOid = 2;     // sysoid
   optional uint32 MaxChassisNum = 3;    // Maximum number of chassis
   optional uint32 MaxSlotNum = 4;  // Maximum number of slots
   optional string HostDescription = 5; // Device description
}
message DevicePhysicalEntities {     // Structure for obtaining physical entity
information of the device
   message Entity {
      optional uint32 PhysicalIndex = 1;    // Entity index
      optional string VendorType = 2;       // Vendor type
      optional uint32 EntityClass = 3;      // Entity class
      optional string SoftwareRev = 4;      // Software version
      optional string SerialNumber = 5; // Serial number
      optional string Model = 6;            // Model
```

```
    }
    repeated Entity entity = 1;
}
service DeviceService { // RPC methods
    rpc GetJsonDeviceBase(DeviceBase) returns (grpc_service.GetJsonReply) {} // Method for
obtaining basic device information
    rpc GetJsonDevicePhysicalEntities(DevicePhysicalEntities) returns
(grpc_service.GetJsonReply) {}  // Method for obtaining physical entity information of
the device
}
```

Proto definition file in dial-out mode

The **grpc_dialout.proto** file defines the public RPC methods in dial-out mode. The following are the contents of the file:

```
syntax = "proto2";
package grpc_dialout;
message DeviceInfo{ // Pushed device information
    required string producerName = 1; // Vendor name
    required string deviceName = 2;   // Device name
    required string deviceModel = 3;  // Device model
}
message DialoutMsg{  // Format of the pushed data
    required DeviceInfo deviceMsg = 1; // Device information described by DeviceInfo
    required string sensorPath = 2; // Sensor path, which corresponds to xpath in NETCONF
    required string jsonData = 3;  // Sampled data, a JSON string
}
message DialoutResponse{  // Response from the collector. Reserved. The value is not
processed.
    required string response = 1;
}
service gRPCDialout {  // Data push method
    rpc Dialout(stream DialoutMsg) returns (DialoutResponse);
}
```

Obtaining proto definition files

To obtain other proto definition files, contact H3C Support.

Data collection supported by gRPC

In dial-in mode, the device can provide data of multiple service modules, including device management, interface management, IP forwarding, LLDP, and syslog. Support for the data of service modules depends on the proto definition files provided by H3C.

In dia-out mode, the device supports the following data sampling types:

- **Event-triggered sampling**—Sensors in a sensor group sample data when certain events occur. For sensor paths of this data sampling type, see *NETCONF XML API Event Reference* for the module.

- **Periodic sampling**—Sensors in a sensor group sample data at intervals. For sensor paths of this data sampling type, see the NETCONF XML API references for the module except for *NETCONF XML API Event Reference*.

  If the simultaneously collected is huge, the CPU workload might spike and then cause the sampling interval to increase temporarily.

You can execute the **`sensor path`** command to view all data types that can be sampled in dial-out mode.

gRPC configuration examples

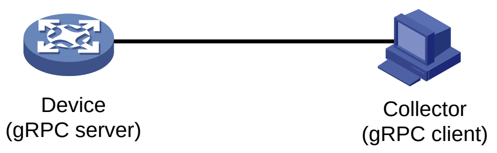
These configuration examples describe only CLI configuration tasks on the device. The collectors need to run an extra application. For information about collector-side application development, see "gRPC collector-side application configuration example."

Example: Configuring the gRPC dial-in mode

Network configuration

As shown in Figure 4, configure the gRPC dial-in mode on the device so the device acts as the gRPC server and the gRPC client can subscribe to LLDP events on the device.

Figure 4 Network diagram



Procedure

1. Assign IP addresses to interfaces on the gRPC server and client and configure routes. Make sure the server and client can reach each other.
2. Configure the device as the gRPC server:

   # Enable the gRPC service.

   ```
   <Device> system-view
   [Device] grpc enable
   ```

   # Create a local user named **test**. Set the password to **test**, and assign the network-admin user role and HTTPS service to the user.

   ```
   [Device] local-user test
   [Device-luser-manage-test] password simple test
   [Device-luser-manage-test] authorization-attribute user-role network-admin
   [Device-luser-manage-test] service-type https
   [Device-luser-manage-test] quit
   ```
3. Configure the gRPC client.
   - **a.** Prepare a PC and install the gRPC environment on the PC. For more information, see the user guide for the gRPC environment.
   - **b.** Obtain the H3C proto definition file and uses the protocol buffer compiler to generate code of a specific language, for example, Java, Python, C/C++, or Go.
   - **c.** Create a client application to call the generated code.
   - **d.** Start the application to log in to the gRPC server.

Verifying the configuration

When an LLDP event occurs on the gRPC server, verify that the gRPC client receives the event.

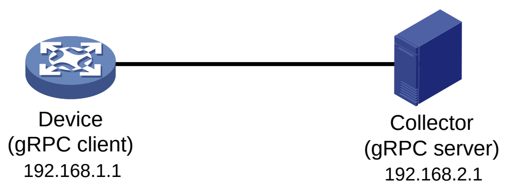
Example: Configuring the gRPC dial-out mode

Network configuration

As shown in Figure 5, the device is connected to a collector. The collector uses port 50050.

Configure gRPC dial-out mode on the device so the device pushes the device capability information of its interface module to the collector at 10-second intervals.

Figure 5 Network diagram



Procedure

# Configure IP addresses as required so the device and the collector can reach each other. (Details not shown.)

# Enable the gRPC service.

```
<Device> system-view
[Device] grpc enable
```

# Create a sensor group named **test**, and add sensor path **ifmgr/devicecapabilities/**.

```
[Device] telemetry
[Device-telemetry] sensor-group test
[Device-telemetry-sensor-group-test] sensor path ifmgr/devicecapabilities/
[Device-telemetry-sensor-group-test] quit
```

# Create a destination group named **collector1**. Specify a collector that uses IPv4 address 192.168.2.1 and port number 50050.

```
[Device-telemetry] destination-group collector1
[Device-telemetry-destination-group-collector1] ipv4-address 192.168.2.1 port 50050
[Device-telemetry-destination-group-collector1] quit
```

# Configure a subscription named **A** to bind sensor group **test** with destination group **collector1**. Set the sampling interval to 10 seconds.

```
[Device-telemetry] subscription A
[Device-telemetry-subscription-A] sensor-group test sample-interval 10
[Device-telemetry-subscription-A] destination-group collector1
[Device-telemetry-subscription-A] quit
```

Verifying the configuration

# Verify that the collector receives the device capability information of the interface module from the device at 10-second intervals. (Details not shown.)

gRPC collector-side application configuration example

Use a language (for example, C++) to develop a gRPC collector-side application on Linux to achieve the following goals:

- Collect device data by using Get, gNMI Capabilities, gNMI Get, or gNMI Subscribe operations in dial-in mode or by using dial-out mode.
- Send settings to the device by using gNMI Set or CLI operations in dial-in mode.

Prerequisites

1. Obtain proto definition files.
    - For dial-in mode, obtain the **grpc_service.proto** file and proto definition files for service modules.
    - For dial-out mode, obtain the **grpc_dialout.proto** file.
2. Obtain utility protoc from https://github.com/google/protobuf/releases.
3. Obtain the protobuf plug-in for C++ (**protobuf-cpp**) from https://github.com/google/protobuf/releases.

Generating the C++ code for the proto definition files

Dial-in mode

# Copy the required proto definition files to the current directory, for example, **grpc_service.proto** and **BufferMonitor.proto**.

```
$protoc --plugin=./grpc_cpp_plugin  --grpc_out=. --cpp_out=. *.proto
```

Dial-out mode

# Copy proto definition file **grpc_dialout.proto** to the current directory.

```
$ protoc --plugin=./grpc_cpp_plugin  --grpc_out=.  --cpp_out=. *.proto
```

Developing the collector-side application (dial-in mode)

In dial-in mode, the application needs to provide the code to be run on the gRPC client.

The C++ code generated from the proto definition files already encapsulates the service classes. For the gRPC client to initiate RPC requests, you only need to call the RPC method in the application.

The application performs the following operations:

- Log in to obtain the token_id.
- Prepare parameters for the RPC method, use the service classes generated from the proto definition files to call the RPC method, and resolve the returned result.
- Log out.

Service classes GrpcService and BufferMonitorService are used in this example.

To develop the collector-side application:

1. Create a GrpcServiceTest class.

   # In the class, use the GrpcService::Stub class generated from grpc_service.proto. Implement login and logout with the Login and Logout methods generated from grpc_service.proto.

```
class GrpcServiceTest
{
public:
   /* Constructor functions */
   GrpcServiceTest(std::shared_ptr<Channel> channel):
GrpcServiceStub(GrpcService::NewStub(channel))  {}

   /* Member functions */
   int Login(const std::string& username, const std::string& password);
   void Logout();
   void listen();
Status listen(const std::string& command);

   /* Member variable */
   std::string token;

private:
   std::unique_ptr<GrpcService::Stub> GrpcServiceStub;  // Use the
GrpcService::Stub class generated from grpc_service.proto.
};
```

2. Customize the Login method.

# Call the Login method of the GrpcService::Stub class to allow a user who provides the correct the username and password to log in.

```
int GrpcServiceTest::Login(const std::string& username, const std::string& password)
{
   LoginRequest request;   // Username and password.
   request.set_user_name(username);
   request.set_password(password);

LoginReply reply;
ClientContext context;

   // Call the Login method.
   Status status = GrpcServiceStub->Login(&context, request, &reply);
   if (status.ok())
   {
       std::cout << "login ok!" << std::endl;
      std::cout <<"token id is :" << reply.token_id() << std::endl;
       token = reply.token_id();  // The login succeeds. The token is obtained.
      return 0;
   }
   else{
       std::cout << status.error_code() << ": " << status.error_message()
                    << ". Login failed!" << std::endl;
      return -1;
   }
}
```

3. Initiate an RPC request to the device.

In this example, the application subscribes to interface packet drop events.

```
rpc SubscribePortQueDropEvent(PortQueDropEvent) returns
(grpc_service.SubscribeReply) {}
```

4. Create the BufMon_GrpcClient class to encapsulate the RPC method.

   # Use the BufferMonitorService::Stub class generated from BufferMonitor.proto to call the RPC method.

```
class BufMon_GrpcClient
{
public:
   BufMon_GrpcClient(std::shared_ptr<Channel> channel):
mStub(BufferMonitorService::NewStub(channel))
   {}

   std::string BufMon_Sub_AllEvent(std::string token);
   std::string BufMon_Sub_BoardEvent(std::string token);
   std::string BufMon_Sub_PortOverrunEvent(std::string token);
   std::string BufMon_Sub_PortDropEvent(std::string token);

   /* Get entries */
   std::string BufMon_Sub_GetStatistics(std::string token);
   std::string BufMon_Sub_GetGlobalCfg(std::string token);
   std::string BufMon_Sub_GetBoardCfg(std::string token);
   std::string BufMon_Sub_GetNodeQueCfg(std::string token);
   std::string BufMon_Sub_GetPortQueCfg(std::string token);

private:
   std::unique_ptr<BufferMonitorService::Stub> mStub; // Use the class generated from
BufferMonitor.proto.
};
```

5. Use std::string BufMon_Sub_PortDropEvent(std::string token) to implement interface packet drop event subscription.

```
std::string BufMon_GrpcClient::BufMon_Sub_PortDropEvent(std::string token)
{
   std::cout << "-------BufMon_Sub_PortDropEvent-------- " << std::endl;

   PortQueDropEvent stNodeEvent;
   PortQueDropEvent_PortQueDrop* pstParam = stNodeEvent.add_portquedrop();

   UINT uiIfIndex = 0;
   UINT uiQueIdx = 0;
   UINT uiAlarmType = 0;

   std::cout<<"Please input interface queue info : ifIndex queIdx alarmtype " <<
std::endl;
   cout<<"alarmtype : 1 for ingress; 2 for egress; 3 for port headroom"<<endl;

   std::cin>>uiIfIndex>>uiQueIdx>>uiAlarmType; // Set the subscription parameters
and interface index.
   pstParam->set_ifindex(uiIfIndex);
```

```
    pstParam->set_queindex(uiQueIdx);
    pstParam->set_alarmtype(uiAlarmType);

    ClientContext context;

    /* Token needs to be added to context */ // Set the token_id to be returned after
a successful login
    std::string key = "token_id";
    std::string value = token;
    context.AddMetadata(key, value);

    SubscribeReply reply;
    Status status = mStub->SubscribePortQueDropEvent(&context,stNodeEvent,&reply);
// Call the RPC method.

    return reply.result();
}
```

6. Use a loop to listen to event reports.

# Implement this method in the GrpcServiceTest class.

```
void GrpcServiceTest::listen()
{
    GetReportRequest reportRequest;
    ClientContext context;
    ReportEvent reportedEvent;

    /* Add the token to the request */
    reportRequest.set_token_id(token);

    std::unique_ptr< ClientReader< ReportEvent>>
reader(GrpcServiceStub->GetEventReport(&context, reportRequest)); // Use
GetEventReport (which is generated from grpc_service.proto) to obtain event
information.

    std::string streamName;
    std::string eventName;
    std::string jsonText;
    std::string token;

    JsonFormatTool jsonTool;

    std::cout << "Listen to server for Event" << std::endl;
    while(reader->Read(&reportedEvent) ) // Read the received event report.
    {
        streamName = reportedEvent.stream_name();
        eventName = reportedEvent.event_name();
        jsonText = reportedEvent.json_text();
        token = reportedEvent.token_id();

        std::cout << "/***********EVENT COME**************/" << std::endl;
        std::cout << "TOKEN: " << token << std::endl;
        std::cout << "StreamName: "<< streamName << std::endl;
        std::cout << "EventName: " << eventName << std::endl;
        std::cout << "JsonText without format: " << std::endl << jsonText << std::endl;
        std::cout << std::endl;
        std::cout << "JsonText Formated: " << jsonTool.formatJson(jsonText) <<
std::endl;
        std::cout << std::endl;
    }

    Status status = reader->Finish();
    std::cout << "Status Message:" << status.error_message() << "ERROR code :" <<
status.error_code();
}  // Login and RPC request finished.
```

7. To log out, call the Logout method.

```
void GrpcServiceTest:: Logout ()
{
    LogoutRequest request;
    request.set_token_id(token);
    LogoutReply reply;
    ClientContext context;
    Status status = mStub->Logout(&context, request, &reply);
std::cout << "Logout! :" << reply.result() << std::endl;
}
```

Developing the collector-side application (dial-out mode)

In dial-out mode, the application needs to provide the gRPC server code so the collector can receive and resolve data obtained from the device.

The application performs the following operations:

- Inherit the automatically generated GRPCDialout::Service class, overload the automatically generated RPC Dialout service, and resolve the fields.
- Register the RPC service with the specified listening port.

To develop the collector-side application in dial-out mode:

1. Inherit and overload RPC service Dialout.

# Create class DialoutTest and inherit GRPCDialout::Service.

```
class DialoutTest final : public GRPCDialout::Service { // Overload the automatically
generated abstract class.
    Status Dialout(ServerContext* context, ServerReader< DialoutMsg>* reader,
DialoutResponse* response) override; // Implement RPC method Dialout.
};
```

2. Register the DialoutTest service as a gRPC service and specify the listening port.

```
using grpc::Server;
using grpc::ServerBuilder;
std::string server_address("0.0.0.0:60057"); // Specify the address and port to listen
to.
DialoutTest dialout_test; // Define the object declared in step 1.
ServerBuilder builder;
```

```
builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());// Add
the listening port.
builder.RegisterService(&dialout_test); // Register the service.
std::unique_ptr<Server> server(builder.BuildAndStart()); // Start the service.
server->Wait();
```

3. Implement the Dialout method and data resolution.

```
Status DialoutTest::Dialout(ServerContext* context, ServerReader< DialoutMsg>*
reader, DialoutResponse* response)
{
       DialoutMsg msg;

       while( reader->Read(&msg))
       {
          const DeviceInfo &device_msg = msg.devicemsg();
          std::cout<< "Producer-Name: " << device_msg.producername() << std::endl;
          std::cout<< "Device-Name: " << device_msg.devicename() << std::endl;
          std::cout<< "Device-Model: " << device_msg.devicemodel() << std::endl;
          std::cout<<"Sensor-Path: " << msg.sensorpath()<<std::endl;
          std::cout<<"Json-Data: " << msg.jsondata()<<std::endl;
          std::cout<<std::endl;
       }
       response->set_response("test");

       return Status::OK;
}
```

4. After obtaining the DialoutMsg object (generated from the proto definition file) through the Read method, you can call the method to obtain the field values.