MDT Protocols

The document presents some of the key protocols/serializers enabling Model Driven Telemetry:

- GPB: telemetry optimized serializer
- gRPC: remote procedure calls framework
- gNMI: configuation and state managment protocol

Google Protocol Buffers (GPB)

Protocol buffers are a method of serializing data that can be transmitted over wire or be stored in files. Like JSON and XML, the Protobufs are language and platform-neutral. The Protobuf is optimized to be faster than JSON and XML and is making the transmitted data as small as possible. The definition of the data to be serialized is written in configuration files called ***proto files*** (.proto). These files will contain the configurations known as messages. The proto files can be compiled to generate the code in the user's programming language.

Protocol buffers use separation of concerns between context and data.

Consider a JSON example.

```
{
  first_name: "Arun",
  last_name: "Kurian"
}
```

In this example, the transmitted data has got an object literal with two properties, `first_name` and `last_name`, with values `Arun` and `Kurian`. This is highly readable, but this can take up more space. Here, every JSON message has to provide both of these pieces every single time. As our data grows, the transmission time will be increased significantly.

But in the case of Protobufs, things are different. We first define a message in a configuration file like this:

```
{
  string first_name = 1;
  string last_name = 2;
}
```

This configuration file contains the context information. The numbers are just identifiers of the fields. By using this configuration, we can send encoded data as

```
124Arun226Kurian
```

In the case of 124Arun, 1 stands for the field identifier, 2 for the data type (which is the string), and 4 is the length of the text. This is a bit more difficult to read than JSON for a human being; however, this will take very little space compared to JSON data.

In the context of telemetry the following .proto snippet define the message Telemetry:

```
syntax = "proto3";
option go_package = "telemetry_bis";

message Telemetry {
  oneof node_id {
    string node_id_str = 1;
  }
  oneof subscription {
    string subscription_id_str = 3;
  }
  string encoding_path = 6;
  uint64 collection_id = 8;
  uint64 collection_start_time = 9;
  uint64 msg_timestamp = 10;
  repeated TelemetryField data_gpbkv = 11;
  TelemetryGPBTable data_gpb = 12;
  uint64 collection_end_time = 13;
```

It is both valide for GPB-KV (Key Value) when key name is kept as a string and value is encoded as binary and for GPB (Compact) for which both key name and value are binary encoded.

The following examplifies TelemetryField data content for GPB-KV message:

```
node_id_str: “test-IOSXR"
subscription_id_str: ”if_rate"
encoding_path: "Cisco-IOS-XR-infra-statsd-
oper:infrastatistics/interfaces/interface/latest/data-rate"
collection_id: 3
collection_start_time: 1485793813366
msg_timestamp: 1485793813366
data_gpbkv {
timestamp: 1485793813374
fields {
name: "keys"
fields { name: "interface-name" string_value: "Null0" }
}
fields {
```

```
name: "content"
fields { name: "input-data-rate" 8: 0 }
fields { name: "input-packet-rate" 8: 0 }
fields { name: "output-data-rate" 8: 0 }
fields { name: "output-packet-rate" 8: 0 }
...
data_gpbkv {
timestamp: 1485793813389
fields {
name: "keys"
fields { name: "interface-name" string_value: "GigabitEthernet0/0/0/0" }
}
fields {
name: "content"
fields { name: "input-data-rate" 8: 8 }
fields { name: "input-packet-rate" 8: 1 }
fields { name: "output-data-rate" 8: 2 }
fields { name: "output-packet-rate" 8: 0 }
...
}
...
collection_end_time: 1485793813405
```

And the following gives an example for a TelemetryGPBTable data content in GPB compact message:

```
node_id_str: ”test-IOSXR"
subscription_id_str: ”if_rate"
encoding_path: "Cisco-IOS-XR-infra-
statsdoper:infrastatistics/interfaces/interface/latest/data-rate"
collection_id: 5
collection_start_time: 1485794640452
msg_timestamp: 1485794640452
data_gpb {
row {
timestamp: 1485794640459
keys: "\n\005Null0"
content:
"\220\003\000\230\003\000\240\003\000\250\0
03\000\260\003\000\270\003\000\300\003\000\
310\003\000\320\003\000\330\003\t\340\003\00
0\350\003\000\360\003\377\001"
}
row {
timestamp: 1485794640469
keys: "\n\026GigabitEthernet0/0/0/0"
content:
"\220\003\010\230\003\001\240\003\002\250\0
03\000\260\003\000\270\003\000\300\003\000\
310\003\000\320\003\300\204=\330\003\000\34
0\003\000\350\003\000\360\003\377\001"
}
collection_end_time: 1485794640480
```

gRPC

gRPC is an open-source Remote Procedure Call framework that is used for high-performance communication between services. It is an efficient way to connect services written in different languages with pluggable support for load balancing, tracing, health checking, and authentication. By default, gRPC uses protocol buffers for serializing structured data.

gRPC is based on the **HTTP/2 transport protocol** and it has support for Remote Procedure Calls (RPCs). These are operations that can be called remotely with a set of given parameters and they can return information back to the requestor. The following categories of RPCs exist:

- **Unary RPC** - the client sends a single request to the server and receives a single response back (e.g. polling information).
- **Server streaming RPC** - the client sends a request to get multiple streams of information from the server (e.g. model-driven telemetry); the client can wait for the completion marked in the metadata by the server.
- **Client streaming RPC** - the client sends multiple streams of information to the server; the server typically sends a single response back
- **Bidirectional streaming RPC** - streaming of multiple read-write streams between the client and the server.

In the context of network management, gRPC supports **SSL/TLS authentication**. It is also possible to add custom authentication methods by extending the implementation.

The client uses SSL/TLS authentication to authenticate to the server and to encrypt the messages. There is also an option to use certificates for authentication. The SSL credentials are attached to the gRPC channel.

You define gRPC services in ordinary proto files, with RPC method parameters and return types specified as protocol buffer messages:

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

In the above snippet the `HelloRequest` message definition specifies one field (name/value pair), the `name` data that you want to include in this type of message. The field has a name (here `name` 😃) and a type (`string`).
gRPC uses `protoc` with a special gRPC plugin to generate code from your proto file: you get generated gRPC client and server code, as well as the regular protocol buffer code for populating, serializing, and retrieving your message types.

---

gNMI

gNMI is a protocol built on top of gRPC for configuration manipulation and state retrieval of network devices. The advantage is it provides a single service for streaming telemetry and configuration management. It supports very efficient serialization and data access and offers an implemented alternative to NETCONF and RESTCONF.

The gNMI services are defined in a .proto file referenced at https://github.com/openconfig/gnmi/blob/master/proto/gnmi/gnmi.proto

The snippet below provides an explanation for the main RPCs:

```
service gNMI {
  // Capabilities allows the client to retrieve the set of capabilities
that
  // is supported by the target. This allows the target to validate the
  // service version that is implemented and retrieve the set of models
that
  // the target supports. The models can then be specified in subsequent
RPCs
  // to restrict the set of data that is utilized.
  // Reference: gNMI Specification Section 3.2
  rpc Capabilities(CapabilityRequest) returns (CapabilityResponse);
  // Retrieve a snapshot of data from the target. A Get RPC requests that
the
  // target snapshots a subset of the data tree as specified by the paths
  // included in the message and serializes this to be returned to the
  // client using the specified encoding.
  // Reference: gNMI Specification Section 3.3
  rpc Get(GetRequest) returns (GetResponse);
  // Set allows the client to modify the state of data on the target. The
  // paths to modified along with the new values that the client wishes
  // to set the value to.
  // Reference: gNMI Specification Section 3.4
  rpc Set(SetRequest) returns (SetResponse);
  // Subscribe allows a client to request the target to send it values
  // of particular paths within the data tree. These values may be
streamed
  // at a particular cadence (STREAM), sent one off on a long-lived
channel
  // (POLL), or sent as a one-off retrieval (ONCE).
  // Reference: gNMI Specification Section 3.5
  rpc Subscribe(stream SubscribeRequest) returns (stream
```

```
  SubscribeResponse);
  }
```

References

https://grpc.io/
https://developers.google.com/protocol-buffers
https://betterprogramming.pub/understanding-protocol-buffers-43c5bced0d47