gRPC Network Management Interface (gNMI)

Contributors:
Paul Borman, Marcus Hines, Carl Lebsack, Chris Morrow, Anees Shaikh, Rob Shakir
Date: November 10th, 2016
Version: 0.2.1

1 Introduction

This document defines an gRPC-based protocol for the modification and retrieval of configuration from a network element, as well as the control and generation of telemetry streams from a network element to a data collection system. The intention is that a single gRPC service definition can cover both configuration and telemetry - allowing a single implementation on the network element, as well as a single NMS element to interact with the device via telemetry and configuration RPCs.

All messages within the gRPC service definition are defined as protocol buffers (specifically proto3). gRPC service definitions are expected to be described using the relevant features of the protobuf IDL. In addition to this protocol description document, a reference protobuf definition is included in Appendix 4, and maintained in [REFERENCE-PROTO].

The service defined within this document is assumed to carry payloads that contain data instances of OpenConfig YANG schemas, but can be used for any data with the following characteristics:

1. structure can be represented by a tree structure where nodes can be uniquely identified by a path consisting of node names, or node names coupled with attributes;
2. values can be serialised into a scalar object,

Currently, values may be serialised to a scalar object through encoding as a JSON string, a byte-array, or a serialised protobuf object - although the definition of new serialisations is possible.

Throughout this specification the following terminology is used:

- *Telemetry* - refers to streaming data relating to underlying characteristics of the device - either operational state or configuration.
- *Configuration* - elements within the data schema which are read/write and can be manipulated by the client.
- *Target* - the device within the protocol which acts as the owner of the data that is being manipulated or reported on. Typically this will be a network device..
- *Client* - the device or system using the protocol described in this document to query/modify data on the target, or act as a collector for streamed data. Typically this will be a network management system.

2 Common Message Types and Encodings

2.1 Reusable Notification Message Format

When a target wishes to communicate data relating to the state of its internal database to an interested client, it does so via means of a common Notification message. Notification messages are reused in other higher-layer messages for various purposes. The exact use of the Notification message is described on a per-RPC basis.

The fields of the Notification message are as follows:

- `timestamp` - The time at which the data was collected by the device from the underlying source, or the time that the target generated the Notification message (in the case that the data does not reflect an underlying data source). This value is always represented according to the definition in 2.2.1.
- `prefix` - a prefix which is applied to all path fields (encoded as per 2.2.2) included in the `Notification` message. The paths expressed within the message are formed by the concatenation of `prefix + path`. The `prefix` always precedes the `path` elements. Further semantics of prefixes are described in 2.4.1.
- `alias` - a string providing an alias for the prefix specified within the notification message. The encoding of an alias, and the procedure for their creation is described in 2.4.2.

- `update` - a list of update messages that indicate changes in the underlying data of the target. Both modification and creation of data is expressed through the update message.
    - An `Update` message has two subfields:
        - `path` - a path encoded as per 2.2.2.
        - `value` - a value encoded as per 2.2.3.
    - The set of paths that are specified within the list of updates MUST be unique. In this context, the path is defined to be the fully resolved path (including the prefix). In the case that there is a duplicate path specified within an update, only the final update should be processed by the receiving entity.
- `delete` - a list of paths (encoded as per 2.2.2) that indicate the deletion of data nodes on the target.

The creator of a Notification message MUST include the `timestamp` field. All other fields are optional.

2.2 Common Data Types

2.2.1 Timestamps

Timestamp values MUST be represented as the number of nanoseconds since the Unix epoch (January 1st 1970 00:00:00 UTC). The value MUST be encoded as a signed 64-bit integer (`int64`)[1].

2.2.2 Paths

Paths are represented according to OpenConfig Path Conventions, a simplified form of XPATH. Rather than utilising a single string to represent the path - with the "/" character separating each element of the path, the path is represented by an ordered list of strings, starting at the root node, and ending at the most specific path element.

A path is represented by the `Path` message with the following fields:

- `element` -- a set of path elements, encoded as strings (see examples below).
- `origin` - field which MAY be used to disambiguate the path if necessary. For example, the origin may be used to indicate which organization defined the schema to which the path belongs.

Each `Path` element should correspond to a node in the data tree. For example, the path `/a/b/c/d` is encoded as:

```
path: <
```

[1] This matches the types that Go UnixNano and Java TimeUnit toNanos return, and hence is used rather than an unsigned integer.

```
  element: "a"
  element: "b"
  element: "c"
  element: "d"
>
```

Where attributes are to be specified, these are encoded alongside the node name within the path element, for example a node specified by `/a/e[key=k1]/f/g` would have the path encoded as:

```
path: <
  element: "a"
  element: "e[key=k1]"
  element: "f"
  element: "g"
>
```

The root node ('/') is indicated by encoding a single path element which is an empty string, as per the following example:

```
path: <
  element: ""
>
```

Paths (defined to be the concatenation of the `Prefix` and `Path` within the message) specified within a message MUST be absolute - no messages with relative paths should be generated.

2.2.3 Node Values

The value of a data node is encoded as a two-field message:

- `bytes` - an arbitrary series of bytes which indicates the value of the node referred to within the message context.
- `type` - a field indicating the type of data contained in the bytes field. Currently defined types are:

Name	Description	Value
JSON	A JSON encoded string as per 2.3.1.	0
Bytes	An arbitrary sequence of bytes as per 2.3.2.	1

Proto	A Protobuf encoded message, as per 2.3.3	2
ASCII	An ASCII encoded string representing text formatted according to a target-defined convention (described in Section 2.3.4).	3
JSON_IETF	A JSON encoded string as per 2.3.1 using JSON encoding compatible with draft-ietf-netmod-yang-json	4

2.3 Encoding Data in an Update Message

2.3.1 JSON and JSON_IETF

The JSON type indicates that the value included within the bytes field of the node value message is encoded as a JSON string. This format utilises the specification in RFC7159. Additional types (e.g., `JSON_IETF`) are utilised to indicate specific additional characteristics of the encoding of the JSON data (particularly where they relate to serialisation of YANG-modeled data).

For any JSON encoding:

- In the case that the data item at the specified path is a leaf node (i.e., has no children) the value of that leaf is encoded directly - i.e., the “bare” value is specified (i.e., a JSON object is not required, and a bare JSON value is included).
- Where the data item referred to has child nodes, the value field contains a serialised JSON entity (object or array) corresponding to the referenced item.

Using the following example data tree:

```
root +
     |
     +-- a +
           |
           +-- b[name=b1] +
                          |
                          +-- c +
                                |
                                +-- d (string)
                                +-- e (uint32)
```

The following serialisations would be used (note that the examples below follow the conventions for textproto, and Golang-style backticks are used for string literals that would otherwise require escaping):

For `/a/b[name=b1]/c/d`:

```
update: <
  path: <
    element: "a"
    element: "b[name=b1]"
    element: "c"
    element: "d"
  >
  value: <
    value: "AStringValue"
    type: JSON
  >
>
```

For `/a/b[name=b1]/c/e`:

```
update: <
  path: <
    element: "a"
    element: "b[name=b1]"
    element: "c"
    element: "e"
  >
  value: <
    Value: 10042    // decoded byte array
    type: JSON
  >
>
```

For `/a/b[name=b1]/c`:

```
update: <
  path: <
    element: "a"
    element: "b[name=b1]"
    element: "c"
  >
  value: <
    value: { "d": "AStringValue", "e": 10042 }
    type: JSON
  >
>
```

For `/a` :

```
update: <
  path: <
    element: "a"
  >
  value: <
    value: `{ "b": [
                      {
                        "name": "b1",
                        "c": {
                        "d": "AStringValue",
                          "e": 10042
                        }
                       }
                    ]
             }`
    type: JSON_IETF
  >
>
```

Note that all JSON values MUST be valid JSON. That is to say, whilst a value or object may be included in the message, the relevant quoting according to the JSON specification in RFC7159 must be used. This results in quoted string values, and unquoted number values.

`JSON_IETF` encoded data MUST conform with the rules for JSON serialisation described in RFC7951. Data specified with a type of JSON MUST be valid JSON, but no additional constraints are placed upon it. An implementation MUST NOT serialise data with mixed `JSON` and `JSON_IETF` encodings.

Both the client and target MUST support the JSON encoding as a minimum.

2.3.2 Bytes

The `BYTES` type indicates that the contents of the bytes field of the message contains a byte sequence whose semantics is opaque to the protocol.

2.3.3 Protobuf

The `PROTOBUF` type indicates that the contents of the bytes field of the message contains a serialised protobuf message. Note that in the case that the sender utilises this type, the receiver must understand the schema (and hence the type of protobuf message that is serialised) in order to decode the value. Such agreement is not guaranteed by the protocol and hence must be established out-of-band.

2.3.4 ASCII

The `ASCII` type indicates that the contents of the bytes field of the message contains system-formatted ASCII-encoded text. For configuration data, for example, this may consist of semi-structured CLI configuration data formatted according to the target platform. The gNMI protocol does not define the format of the text -- this must be established out-of-band.

2.4 Use of Data Schema Paths

2.4.1 Path Prefixes

In a number of messages, a prefix can be specified to reduce the lengths of path fields within the message. In this case, a prefix field is specified within a message - comprising of a valid path encoded according to Section 2.2.2. In the case that a prefix is specified, the absolute path is comprised of the concatenation of the list of path elements representing the prefix and the list of path elements in the `path` field.

For example, again considering the data tree shown in Section 2.3.1 if a `Notification` message updating values, a prefix could be used to refer to the `/a/b[name=b1]/c/d` and `/a/b[name=b1]/c/e` data nodes:

```
notification: <
    timestamp: (timestamp)      // timestamp as int64
    prefix: <
     element: "a"
      element: "b[name=b1]"
      element: "c"
    >
  update: <
    path: <
      element: "d"
    >
    value: <
      value: "AStringValue"
      type: JSON
    >
  >
  update: <
    path: <
      element: "e"
    >
    value: <
      value: 10042      // converted to int representation
```

```
      type: JSON
    >
  >
>
```

2.4.2 Path Aliases

In some cases, a client or target MAY desire to utilises aliases for a particular path - such that subsequent messages can be compressed by utilising the alias, rather than using a complete representation of the path. Doing so reduces total message length, by ensuring that redundant information can be removed.

Support for path aliases MAY be provided by a target. In a case where a target does not support aliases, the maximum message length SHOULD be considered, especially in terms of bandwidth utilisation, and the efficiency of message generation.

A path alias is encoded as a string. In order to avoid valid data paths clashing with aliases (e.g., `a` in the above example), an alias name MUST be prefixed with a # character.

The means by which an alias is created is defined on a per-RPC basis. In order to delete an alias, the alias name is sent with the path corresponding to the alias empty.

Aliases MUST be specified as a fully expanded path, and hence MUST NOT reference other aliases within their definition, such that a single alias lookup is sufficient to resolve the absolute path.

2.4.3 Interpretation of Paths Used in RPCs

When a client specifies a path within an RPC message which indicates a read, or retrieval of data, the path MUST be interpreted such that it refers to the node directly corresponding with the path **and** all its children. The path refers to the direct node and all descendent branches which originate from the node, recursively down to each leaf element. If specific nodes are expected to be excluded then an RPC MAY provide means to filter nodes, such as regular-expression based filtering, lists of excluded paths, or metadata-based filtering (based on annotations of the data schema being manipulated, should such annotations be available and understood by both client and target).

For example, consider the following data tree:

```
root +
     |
     +-- childA +
```

```
    |          |
    |          +-- leafA1
    |          +-- leafA2
    |          +-- childA3 --+
    |                        |
    |                        +-- leafA31
    |                        +-- leafA32
    |
    +-- childB +
               |
               +-- leafB1
               +-- leafB2
```

A path referring to “root” (which is represented by a Path consisting of a single element specifying an empty string) should result in the nodes `childA` and `childB` and all of their children (`leafA1`, `leafA2`, `leafB1`, `leafB2`, `childA3`, `leafA31` and `leafA32`) being considered by the relevant operation.

In the case that the RPC is modifying the state of data (i.e., a write operation), such recursion is not required - rather the modification operation should be considered to be targeted at the node within the schema that is specified by the path, and the value should be deserialized such that it modifies the content of any child nodes if required to do so.

2.5 Error handling

Where the client or target wishes to indicate an error, an `Error` message is generated. Errors MUST be represented by a canonical gRPC error code (Java, Golang, C++) . The entity generating the error MUST specify a free-text string which indicates the context of the error, allowing the receiving entity to generate log entries that allow a human operator to understand the exact error that occurred, and its context. Each RPC defines the meaning of the relevant canonical error codes within the context of the operation it performs.

The canonical error code that is chosen MUST consider the expected behavior of the client on receipt of the message. For example, error codes which indicate that a client may subsequently retry SHOULD only be used where retrying the RPC is expected to result in a different outcome.

A re-usable `Error` message MUST be used when sending errors in response to an RPC operation. This message has the following fields:

- `code` - an unsigned 32-bit integer value corresponding to the canonical gRPC error code.

- `message` - a human-readable string describing the error condition in more detail. This string is not expected to be machine-parsable, but rather provide contextual information which may be passed to upstream systems.
- `data` - an arbitrary sequence of bytes (encoded as `proto.Any`) which provides further contextual information relating to the error.

2.6 Schema Definition Models

The data tree supported by the target is expected to be defined by a set of schemas. The definition and format of these models is out of scope of this specification (YANG-modeled data is one example). In the case that such schema definitions are used, the client should be able to determine the models that are supported by the target, so that it can generate valid modifications to the data tree, and interpret the data returned by `Get` and `Subscribe` RPC calls.

Additionally, the client may wish to restrict the set of models that are utilised by the target so that it can validate the data returned to it against a specific set of data models. This is particularly relevant where the target may otherwise add new values to restricted value data elements (e.g., those representing an enumerated type), or augment new data elements into the data tree.

In order to allow the client to restrict the set of data models to be used when interacting with the target, the client MAY discover the set of models that are supported by the target using the `Capabilities` RPC described in Section 3.2. For subsequent `Get` and `Subscribe` RPCs, the client MAY specify the models to be used by the target. The set of models to use is expressed as a `ModelData` message, as specified in Section 2.6.1.

If the client specifies a set of models in a `Get` or `Subscribe` RPC, the target MUST NOT utilize data tree elements that are defined in schema modules outside the specified set. In addition, where there are data tree elements that have restricted value sets (e.g., enumerated types), and the set is extended by a module which is outside of the set, such values MUST NOT be used in data instances that are sent to the client. Where there are other elements of the schema that depend on the existence of such enumerated values, the target MUST NOT include such values in data instances sent to the client.

2.6.1 The ModelData message

The `ModelData` message describes a specific model that is supported by the target and used by the client. The fields of the `ModelData` message identify a data model registered in a model catalog, as described in [MODEL_CATALOG_DOC] (the schema of the catalog itself - expressed in YANG - is described in [MODEL_CATALOG_YANG]). Each model specified by a `ModelData` message may refer to a specific schema module, a bundle of modules, or an augmentation or deviation, as described by the catalog entry.

Each `ModelData` message contains the following fields:

- `name` - name of the model expressed as a string.
- `organization` - the organization publishing the model, expressed as a string.
- `version` - the supported (or requested) version of the model, expressed as a string which represents the semantic version of the catalog entry.

The combination of `name`, `organization`, and `version` uniquely identifies an entry in the model catalog.

3 Service Definition

A single gRPC service is defined - future revisions of this specification MAY result in additional services being introduced, and hence an implementation MUST NOT make assumptions that limit to a single service definition.

The service consists of the following RPCs:

- `Capabilities` - defined in Section 3.2 and used by the client and target as an initial handshake to exchange capability information
- `Get` - defined in Section 3.3, used to retrieve snapshots of the data on the target by the client.
- `Set` - defined in Section 3.4 and used by the client to modify the state of the target.
- `Subscribe` - defined in Section 3.5 and used to control subscriptions to data on the target by the client.

3.1 Session Security, Authentication and RPC Authorization

The session between the client and server MUST be encrypted using TLS - and a target or client MUST NOT fall back to unencrypted channels.

New connections are mutually authenticated -- each entity validates the x.509 certificate of the remote entity to ensure that the remote entity is both known, and authorized to connect to the local system.

If the target is expected to authenticate an RPC operation, the client MUST supply a username and password in the metadata of the RPC message (e.g., `SubscribeRequest`, `GetRequest` or `SetRequest`). If the client supplies username/password credentials, the target MUST authenticate the RPC per its local authentication functionality.

Authorization is also performed per-RPC by the server, through validating client-provided metadata. The client MAY include the appropriate AAA metadata, which MUST contain a username, and MAY include a password in the context of each RPC call it generates. If the

client includes both username and password, the target MUST authenticate and authorize the request. If the client only supplies the username, the target MUST authorize the RPC request.

A more detailed discussion of the requirements for authentication and encryption used for gNMI is in [GNMI-AUTH].

3.2 Capability Discovery

A client MAY discover the capabilities of the target using the `Capabilities` RPC. The `CapabilityRequest` message is sent by the client to interrogate the target. The target MUST reply with a `CapabilityResponse` message that includes its gNMI service version, the versioned data models it supports, and the supported data encodings. This information is used in subsequent RPC messages from the client to indicate the set of models that the client will use (for `Get`, `Subscribe` as described in Section 2.6) , and the encoding to be used for the data.

When the client does not specify the models it is using, the target SHOULD use all data schema modules that it supports when considering the data tree to be addressed. If the client does not specify the encoding in an RPC message, it MUST send JSON encoded values (the default encoding).

3.2.1 The CapabilityRequest message

The `CapabilityRequest` message is sent by the client to request capability information from the target. The `CapabilityRequest` message carries no additional fields.

3.2.2 The CapabilityResponse message

The `CapabilityResponse` message has the following fields:

- `supported_models` - a set of `ModelData` messages (as defined in Section 2.6.1) describing each of the models supported by the target
- `supported_encodings` - an enumeration field describing the data encodings supported by the target, as described in Section 2.3.
- `gNMI_version` - the semantic version of the gNMI service supported by the target, specified as a string. The version should be interpreted as per [OPENCONFIG-SEMVER].

3.3 Retrieving Snapshots of State Information

In some cases, a client may require a snapshot of the state that exists on the target. In such cases, a client desires some subtree of the data tree to be serialized by the target and

transmitted to it. It is expected that the values that are retrieved (whether writeable by the client or not) are collected immediately and provided to the client.

The `Get` RPC provides an interface by which a client can request a set of paths to be serialized and transmitted to it by the target. The client sends a `GetRequest` message to the target, specifying the data that is to be retrieved. The fields of the `GetRequest` message are described in Section 3.3.1.

Upon reception of a `GetRequest`, the target serializes the requested paths, and returns a `GetResponse` message. The target MUST reflect the values of the specified leaves at a particular collection time, which MAY be different for each path specified within the `GetRequest` message.

The target closes the channel established by the `Get` RPC following the transmission of the `GetResponse` message.

3.3.1 The GetRequest Message

The `GetRequest` message contains the following fields:

- `prefix` - a path (specified as per Section 2.2.2), and used as described in Section 2.4.1. The prefix is applied to all paths within the `GetRequest` message.
- `path` - a set of paths (expressed as per Section 2.2.2) for which the client is requesting a data snapshot from the target. The path specified MAY utilize wildcards. In the case that the path specified is not valid, the target MUST populate the `error` field of the `GetResponse` message indicating an error code of `InvalidArgument` and SHOULD provide information about the invalid path in the error message .
- `type` - the type of data that is requested from the target. The valid values for type are described below.
- `encoding` - the encoding that the target should utilise to serialise the subtree of the data tree requested. The type MUST be one of the encodings specified in Section 2.3. If the `Capabilities` RPC has been utilised, the client SHOULD use an encoding advertised as supported by the target. If the encoding is not specified, JSON MUST be used. If the target does not support the specified encoding, the target MUST populate the error field of the `GetResponse` message, specifying an error of `InvalidArgument`. The error message MUST indicate that the specified encoding is unsupported.
- `use_models` - a set of `ModelData` messages (defined in Section 2.6.1) indicating the schema definition modules that define the data elements that should be returned in response to the Get RPC call. The semantics of the `use_models` field are defined in Section 2.6..

Since the data tree stored by the target may consist of different types of data (e.g., values that are operational in nature, such as protocol statistics) - the client MAY specify that a subset of values in the tree are of interest. In order for such filtering to be implemented, the data

schema on the target MUST be annotated in a manner which specifies the type of data for individual leaves, or subtrees of the data tree.

The types of data currently defined are:

- `CONFIG` - specified to be data that the target considers to be read/write. If the data schema is described in YANG, this corresponds to the "config true" set of leaves on the target.
- `STATE` - specified to be the read-only data on the target. If the data schema is described in YANG, `STATE` data is the "config false" set of leaves on the target.
- `OPERATIONAL` - specified to be the read-only data on the target that is related to software processes operating on the device, or external interactions of the device.

If the `type` field is not specified, the target MUST return CONFIG, STATE and OPERATIONAL data fields in the tree resulting from the client's query..

3.3.2 The GetResponse message

The `GetResponse` message consists of:

- `notification` - a set of `Notification` messages, as defined in Section 2.1. The target MUST generate a `Notification` message for each path specified in the client's `GetRequest`, and hence MUST NOT collapse data from multiple paths into a single `Notification` within the response. The `timestamp` field of the `Notification` message MUST be set to the time at which the target's snapshot of the relevant path was taken.
- `error` -- an `Error` message encoded as per the specification in Section 2.5, used to indicate errors in the get request received by the target from the client.

3.3.3 Considerations for using Get

The `Get` RPC is intended for clients to retrieve relatively small sets of data as complete objects, for example a part of the configuration. Such requests are not expected to put a significant resource burden on the target. Since the target is expected to return the entire snapshot in the `GetResponse` message, `Get` is not well-suited for retrieving very large data sets, such as the full contents of the routing table, or the entire component inventory. For such operations, the `Subscribe` RPC is the recommended mechanism, e.g. using the `ONCE` mode as described in Section 3.5.

Another consideration for `Get` is that the timestamp returned is associated with entire set of data requested, although individual data items may have been sampled by the target at different times. If the client requires higher accuracy for individual data items, the `Subscribe` RPC is recommended to request a telemetry stream (see Section 3.5.2).

3.4 Modifying State

Modifications to the state of the target are made through the `Set` RPC. A client sends a `SetRequest` message to the target indicating the modifications it desires.

A target receiving a `SetRequest` message processes the operations specified within it - which are treated as a transaction (see Section 3.4.3). The server MUST process deleted paths (within the `delete` field of the `SetRequest`), followed by paths to be replaced (within the `replace` field), and finally updated paths (within the `update` field). The order of the replace and update fields MUST be treated as significant within a single `SetRequest` message. If a single path is specified multiple times for a single operation (i.e., within `update` or `replace`), then the state of the target MUST reflect the application of all of the operations in order, even if they overwrite each other.

In response to a `SetRequest`, the target MUST respond with a `SetResponse` message. For each operation specified in the `SetRequest` message, an `UpdateResult` message MUST be included in the response field of the `SetResponse`. The order in which the operations are applied MUST be maintained such that `UpdateResult` messages can be correlated to the `SetRequest` operations. In the case of a failure of an operation, the `error` field of the `UpdateResult` message MUST be populated with an `Error` message as per the specification in Section 3.4.7. In addition, the `error` field of the `SetResponse` message MUST be populated with an error message indicating the success or failure of the set of operations within the `SetRequest` message (again using the error handling behavior defined in Section 3.4.7).

3.4.1 The SetRequest Message

A `SetRequest` message consists of the following fields:

- `prefix` - specified as per Section 2.4.1. The prefix specified is applied to all paths defined within other fields of the message.
- `delete` - A set of paths, specified as per Section 2.2.2, which are to be removed from the data tree. A specification of the behavior of a delete is defined in Section 3.4.5.
- `replace` - A set of `Update` messages indicating elements of the data tree whose content is to be replaced.
- `update` - A set of `Update` messages indicating elements of the data tree whose content is to be updated.

The semantics of “updating” versus “replacing” content are defined in Section 3.4.4.

A re-usable `Update` message is utilised to indicate changes to paths where a new value is required. The `Update` message contains two fields:

- `path` - a path encoded as per Section 2.2.2 indicating the path of the element to be modified.
- `value` - a value encoded as per Section 2.2.3 indicating the value applied to the specified node. The semantics of how the node is updated is dependent upon the context of the update message, as specified in Section 3.4.4.

3.4.2 The SetResponse Message

A `SetResponse` consists of the following fields:

- `prefix` - specified as per Section 2.4.1. The prefix specified is applied to all paths defined within other fields of the message.
- `message` - an error message as specified in Section 2.5. The target SHOULD specify a `message` in the case that the update was successfully applied, in which case an error code of `OK (0)` MUST be specified. In cases where an update was not successfully applied, the contents of the error message MUST be specified as per Section 3.4.7.
- `response` - containing a list of responses, one per operation specified within the `SetRequest` message. Each response consists of an `UpdateResult` message with the following fields:
    - `timestamp` - a timestamp (encoded as per Section 2.2.1) at which the set request message was accepted by the system.
    - `path` - the path (encoded as per Section 2.2.2) specified within the `SetRequest`. In the case that a common prefix was not used within the `SetRequest`, the target MAY specify a `prefix` to reduce repetition of path elements within multiple `UpdateResult` messages in the `request` field.
    - `op` - the operation corresponding to the path. This value MUST be one of `DELETE`, `REPLACE`, or `UPDATE`.
    - `message` - an error message (as specified in Section 2.5). This field follows the same rules as the message field within the `SetResponse` message specified above.

3.4.3 Transactions

All changes to the state of the target that are included in an individual `SetRequest` message are considered part of a transaction. That is, either all modifications within the request are applied, or the target MUST rollback the state changes to reflect its state before any changes were applied. The state of the target MUST NOT appear to be changed until such time as all changes have been accepted successfully. Hence, telemetry update messages MUST NOT reflect a change in state until such time as the intended modifications have been accepted.

As per the specification in Section 3.4, within an individual transaction (`SetRequest`) the order of operations is `delete`, `replace`, `update`.

As the scope of a “transaction” is a single `SetRequest` message, a client desiring a set of changes to be applied together MUST ensure that they are encapsulated within a single `SetRequest` message.

3.4.4 Modes of Update: Replace versus Update

Changes to read-write values on the target are applied based on the `replace` and `update` fields of the `SetRequest` message.

For both replace and update operations, if the path specified does not exist, the target MUST create the data tree element and populate it with the data in the `Update` message, provided the path is valid according to the data tree schema. If invalid values are specified, the target MUST cease processing updates within the `SetRequest` method, return the data tree to the state prior to any changes, and return a `SetResponse` message indicating the error encountered.

For `replace` operations, the behavior regarding omitted data elements in the `Update` depends on whether they refer to non-default values (i.e., set by a previous `SetRequest`), or unmodified defaults. When the `replace` operation omits values that have been previously set, they MUST be treated as deleted from the data tree. Otherwise, omitted data elements MUST be created with their default values on the target.

For `update` operations, only the value of those data elements that are specified explicitly should be treated as changed.

3.4.5 Modifying Paths Identified by Attributes

The path convention defined in Section 2.2.2 allows nodes in the data tree to be identified by a unique set of node names (e.g., `/a/b/c/d`) or paths that consist of node names coupled with attributes (e.g., `/a/e[key=10]`). In the case where where a node name plus attribute name is required to uniquely identify an element (i.e., the path within the schema represents a list, map, or array), the following considerations apply:

- In the case that multiple attribute values are required to uniquely address an element - e.g., `/a/f[k1=10][k2=20]` - and a replace or update operation’s path specifies a subset of the attributes (e.g., `/a/f[k1=10]`) then this MUST be considered an error by the target system - and an error code of `InvalidArgument (3)` specified.
- Where the path specified refers to a node which itself represents the collection of objects (list, map, or array) a replace operation MUST remove all collection entries that are not supplied in the value provided in the `SetRequest`. An update operation MUST be considered to add new entries to the collection if they do not exist
- In the case that key values are specified both as attributes of a node, and as their own elements within the data tree, update or replace operations that modify instances of

the key in conflicting ways MUST be considered an error. The target MUST return an error code of `InvalidArgument (3)`.

For example, consider a tree corresponding to the examples above, as illustrated below.

```
root +
     |
     + a --+
           |
           +-- f[k1=10][k2=20] --+
           |                     |
           |                     +-- k1 = 10
           |                     +-- k2 = 20
           |
           +-- f[k1=10][k2=21] --+
                                 |
                                 +-- k1 = 10
                                 +-- k2 = 21
```

In this case, nodes k1 and k2 are standalone nodes within the schema, but also correspond to attribute values for the node "f". In this case, an update or replace message specifying a path of `/a/f[k1=10][k2=20]` setting the value of `k1` to 100 MUST be considered erroneous, and an error code of `InvalidArgument (3)` specified.

3.4.6 Deleting Configuration

Where a path is contained within the `delete` field of the `SetRequest` message, it should be removed from the target's data tree. In the case that the path specified is to an element that has children, these children MUST be recursively deleted. If a wildcard path is utilised, the wildcards MUST be expanded by the target, and the corresponding elements of the data tree deleted. Such wildcards MUST support paths specifying a subset of attributes required to identity entries within a collection (list, array, or map) of the data schema.

In the case that a path specifies an element within the data tree that does not exist, these deletes MUST be silently accepted.

3.4.7 Error Handling

When a client issues a `SetRequest`, and the target is unable to apply the specified changes, an error MUST be reported to the client. The error is specified in multiple places:

- Within a `SetResponse` message, the error field indicates the completion status of the entire transaction.

- With a `UpdateResult` message, where the error field indicates the completion status of the individual operation.

The target MUST specify the `message` field within a `SetResponse` message such that the overall status of the transaction is reflected. In the case that no error occurs, the target MUST complete this field specifying the `OK (0)` canonical error code.

In the case that any operation within the `SetRequest` message fails, then (as per Section 3.4.3), the target MUST NOT apply any of the specified changes, and MUST consider the transaction as failed. The target SHOULD set the `message` field of the `SetResponse` message to an error message with the code field set to `Aborted (10)`, and MUST set the `message` field of the `UpdateResult` corresponding to the failed operation to an `Error` message indicating failure. In the case that the processed operation is not the only operation within the `SetRequest` the target MUST set the `message` field of the `UpdateResult` messages for all other operations, setting the code field to `Aborted (10)`.

For the operation that the target is unable to process, the `message` field MUST be set to a specific error code indicating the reason for failure based on the following mappings to canonical gRPC error codes:

- When the client has specified metadata requiring authentication (see Section 3.1), and the authentication fails - `Unauthenticated (16)`.
- When the client does not have permission to modify the path specified by the operation - `PermissionDenied (7)`.
- When the operation specifies a path that cannot be parsed by the target - `InvalidArgument (3)`. In this case, the `message` field of the `Error` message specified SHOULD specify human-readable text indicating that the path could not be parsed.
- When the operation is an update or replace operation that corresponds to a path that is not valid - `NotFound (5)`. In this case the `message` field of the `Error` message specified SHOULD specify human-readable text indicating the path that was invalid.
- When the operation is an update or replace operation that includes an invalid value within the `Update` message specified - `InvalidArgument (3)`. This error SHOULD be used in cases where the payload specifies scalar values that do not correspond to the correct schema type, and in the case that multiple values are specified using a particular encoding (e.g., JSON) which cannot be decoded by the target.

3.5 Subscribing to Telemetry Updates

When a client wishes to receive updates relating to the state of data instances on a target, it creates a subscription via the `Subscribe` RPC. A subscription consists of one or more paths, with a specified subscription mode. The mode of each subscription determines the triggers for updates for data sent from the target to the client.

All requests for new subscriptions are encapsulated within a `SubscribeRequest` message - which itself has a mode which describes the longevity of the subscription. A client may create a subscription which has a dedicated stream to return one-off data (`ONCE`); a subscription that utilizes a stream to periodically request a set of data (`POLL`); or a long-lived subscription that streams data according to the triggers specified within the individual subscription's mode (`STREAM`).

The target generates messages according to the type of subscription that has been created, at the frequency requested by the client. The methods to create subscriptions are described in Section 3.5.1.

Subscriptions are created for a set of paths - which cannot be modified throughout the lifetime of the subscription. In order to cancel a subscription, the client closes the gRPC channel over which the `Subscribe` RPC was initiated, or terminates the entire gRPC session.

Subscriptions are fundamentally a set of independent update messages relating to the state of the data tree. That is, it is not possible for a client requesting a subscription to assume that the set of update messages received represent a snapshot of the data tree at a particular point in time. Subscriptions therefore allow a client to:

- Receive ongoing updates from a target which allow synchronization between the client and target for the state of elements within the data tree. In this case (i.e., a `STREAM` subscription), a client creating a subscription receives an initial set of updates, terminated by a message indicating that initial synchronisation has completed, and then receives subsequent updates indicating changes to the initial state of those elements.
- Receive a single view (polled, or one-off) for elements of the data tree on a per-data element basis according to the state that they are in at the time that the message is transmitted. This can be more resource efficient for both target and client than a Get request for large subtrees within the data tree. The target does not need to coalesce values into a single snapshot view, or create an in-memory representation of the subtree at the time of the request, and subsequently transmit this entire view to the client.

Based on the fact that subsequent update messages are considered to be independent, and to ensure that the efficiencies described above can be achieved, by default a target MUST NOT aggregate values within an update message.

In some cases, however, elements of the data tree may be known to change together, or need to be interpreted by the subscriber together. Such data MUST be explicitly marked in the schema as being eligible to be aggregated when being published. Additionally, the subscribing client MUST explicitly request aggregation of eligible schema elements for the

subscription - by means of the `allow_aggregation` flag within a `SubscriptionList` message. For elements covered by a subscription that are not explicitly marked within the schema as being eligible for aggregation the target MUST NOT coalesce these values, regardless of the value of the `allow_aggregation` flag.

When aggregation is not permitted by the client or the schema each update message MUST contain a (key, value) pair - where the key MUST be a path to a single leaf element within the data tree (encoded according to Section 2.2.2). The value MUST encode only the value of the leaf specified. In most cases, this will be a scalar value (i.e., a JSON value if a JSON encoding is utilised), but in some cases, where an individual leaf element within the schema represents an object, it MAY represent a set of values (i.e., a JSON or Protobuf object).

Where aggregation is permitted by both the client and schema, each update message MUST contain a key value pair, where the key MUST be the path to the element within the data tree which is explicitly marked as being eligible for aggregation. The value MUST be an object which encodes the children of the data tree element specified. For JSON, the value is therefore a JSON object, and for Protobuf is a series of binary-encoded Protobuf messages.

3.5.1 Managing Subscriptions

3.5.1.1 The SubscribeRequest Message

A `SubscribeRequest` message is sent by a client to request updates from the target for a specified set of paths.

The fields of the subscribe request are as follows:

- A group of fields, only one of which may be specified, which indicate the type of operation that the `SubscribeRequest` relates to. These are:
    - `subscribe` - a `SubscriptionList` message specifying a new set of paths that the client wishes to subscribe to.
    - `poll` - a `Poll` message used to specify (on an existing channel) that the client wishes to receive a polled update for the paths specified within the subscription. The semantics of the `Poll` message are described in Section 3.5.1.5.3.
    - `aliases` - used by a client to define (on an existing channel) a new path alias (as described in Section 2.4.2). The use of the aliases message is described in Section 3.5.1.6.

In order to create a new subscription (and its associated channel) a client MUST send a `SubscribeRequest` message, specifying the `subscribe` field. The `SubscriptionList` may create a one-off subscription, a poll-only subscription, or a streaming subscription.. In the case of ONCE subscriptions, the channel between client and target MUST be closed following the initial response generation.

Subscriptions are set once, and subsequently not modified by a client. If a client wishes to subscribe to additional paths from a target, it MUST do so by sending an additional `Subscribe` RPC call, specifying a new `SubscriptionList` message. In order to end an existing subscription, a client simply closes the gRPC channel that relates to that subscription. If a channel is initiated with a `SubscribeRequest` message that does not specify a `SubscriptionList` message with the `request` field, the target MUST consider this an error. If an additional `SubscribeRequest` message specifying a `SubscriptionList` is sent via an existing channel, the target MUST respond to this message with `SubscribeResponse` message indicating an error message, with a contained error message indicating an error code of `InvalidArgument (4)`; existing subscriptions on other gRPC channels MUST not be modified or terminated. .

If a client initiates a `Subscribe` RPC with a `SubscribeRequest` message which does not contain a `SubscriptionList` message, this is an error. A `SubscribeResponse` message with the contained `error` message indicating a error code of `InvalidArgument` MUST be sent. The error text SHOULD indicate that an out-of-order operation was requested on a non-existent subscription. The target MUST subsequently close the channel.

3.5.1.2 The SubscriptionList Message

A `SubscriptionList` message is used to indicate a set of paths for which common subscription behavior are required. The fields of the message are:

- `subscription` - a set of `Subscription` messages that indicate the set of paths associated with the subscription list.
- `mode` - the type of subscription that is being created. This may be `ONCE` (described in 3.5.1.5.1); `STREAM` (described in 3.5.1.5.2); or `POLL` (described in 3.5.1.5.3). The default value for the mode field is `STREAM`.
- `prefix`- a common prefix that is applied to all paths specified within the message as per the definition in Section 2.4.1. The default prefix is null.
- `use_aliases`- a boolean flag indicating whether the client accepts target aliases via the subscription channel. In the case that such aliases are accepted, the logic described in Section 2.4.2 is utilised. By default, path aliases created by the target are not supported.
- `qos` - a field describing the packet marking that is to be utilised for the responses to the subscription that is being created. This field has a single sub-value, `marking`, which indicates the DSCP value as a 32-bit unsigned integer. If the `qos` field is not specified, the device should export telemetry traffic using its default DSCP marking for management-plane traffic.
- `allow_aggregation` - a boolean value used by the client to allow schema elements that are marked as eligible for aggregation to be combined into single telemetry update messages. By default, aggregation MUST NOT be used.
- `use_models` - a `ModelData` message (as specified in Section 2.6.1) specifying the schema definition modules that the target should use when creating a subscription. When specified, the target MUST only consider data elements within the defined set of

schema models as defined in Section 2.6. When `use_models` is not specified, the target MUST consider all data elements that are defined in all schema modules that it supports.

A client generating a `SubscriptionList` message MUST include the `subscription` field - which MUST be a non-empty set of `Subscription` messages, all other fields are optional.

3.5.1.3 The Subscription Message

A `Subscription` message generically describes a set of data that is to be subscribed to by a client. It contains a `path`, specified as per the definition in Section 2.2.2.

There is no requirement for the path that is specified within the message to exist within the current data tree on the server. Whilst the path within the subscription SHOULD be a valid path within the set of schema modules that the target supports, subscribing to any syntactically valid path within such modules MUST be allowed. In the case that a particular path does not (yet) exist, the target MUST NOT close the channel, and instead should continue to monitor for the existence of the path, and transmit telemetry updates should it exist in the future. The target MAY send a `SubscribeResponse` message populating the error field with `NotFound (5)` to inform the client that the path does not exist at the time of subscription creation.

For `POLL` and `STREAM` subscriptions, a client may optionally specify additional parameters within the `Subscription` message. The semantics of these additional fields are described in the relevant section of this document.

3.5.1.4 The SubscribeResponse Message

A `SubscribeResponse` message is transmitted by a target to a client over an established channel created by the `Subscribe` RPC. The message contains the following fields:

- A set of fields referred to as the `response` fields, only one of which can be specified per `SubscribeResponse` message:
    - `update` - a Notification message providing an update value for a subscribe data entity as described in Section 3.5.2.3. The update field is also utilised when a target wishes to create an alias within a subscription, as described in Section 3.5.2.2.
    - `sync_response` - a boolean field indicating that a particular set of data values has been transmitted, used for `POLL` and `STREAM` subscriptions.
    - `error` - an `Error` message transmitted to indicate an error has occurred within a particular `Subscribe` RPC call.

3.5.1.5 Creating Subscriptions

3.5.1.5.1 ONCE Subscriptions

A subscription operating in the `ONCE` mode acts as a single request/response channel. The target creates the relevant update messages, transmits them, and subsequently closes the channel.

In order to create a one-off subscription, a client sends a `SubscribeRequest` message to the target. The `subscribe` field within this message specifies a `SubscriptionList` with the mode field set to `ONCE`. Updates corresponding to the subscription are generated as per the semantics described in Section 3.5.2.

Following the transmission of all updates which correspond to data items within the set of paths specified within the subscription list, a `SubscribeResponse` message with the `sync_response` field set to true MUST be transmitted, and the channel over which the `SubscribeRequest` was received MUST be closed.

3.5.1.5.2 STREAM Subscriptions

Stream subscriptions are long-lived subscriptions which continue to transmit updates relating to the set of paths that are covered within the subscription indefinitely.

A `STREAM` subscription is created by sending a `SubscribeRequest` message with the subscribe field containing a `SubscriptionList` message with the type specified as `STREAM`. Each entry within the `Subscription` message is specified with one of the following `modes`:

- **On Change** (`ON_CHANGE`) **-** when a subscription is defined to be "on change", data updates are only sent when the value of the data item changes. A heartbeat interval MAY be specified along with an on change subscription - in this case, the value of the data item(s) MUST be re-sent once per heartbeat interval regardless of whether the value has changed or not.
- **Sampled** (`SAMPLE`) **-** a subscription that is defined to be sampled MUST be specified along with a `sample_interval` encoded as an unsigned 64-bit integer representing nanoseconds. The value of the data item(s) is sent once per sample interval to the client. If the target is unable to support the desired `sample_interval` it MUST reject the subscription by returning a `SubscribeResponse` message with the error field set to an error message indicating the `InvalidArgument (3)` error code. If the `sample_interval` is set to 0, the target MUST create the subscription and send the data with the lowest interval possible for the target.
    - Optionally, the `suppress_redundant` field of the Subscription message may be set for a sampled subscription. In the case that it is set to true, the target SHOULD NOT generate a telemetry update message unless the value of the path being reported on has changed since the last update was generated.

Updates MUST only be generated for those individual leaf nodes in the subscription that have changed. That is to say that for a subscription to `/a/b` - where there are leaves `c` and `d` branching from the `b` node - if the value of `c` has changed, but `d` remains unchanged, an update for `d` MUST NOT be generated, whereas an update for 'c' MUST be generated.

- A `heartbeat_interval` MAY be specified to modify the behavior of `suppress_redundant` in a sampled subscription. In this case, the target MUST generate one telemetry update per heartbeat interval, regardless of whether the suppress redundant flag is set to true. This value is specified as an unsigned 64-bit integer in nanoseconds.

- **Target Defined** (`TARGET_DEFINED`) **-** when a client creates a subscription specifying the target defined mode, the target SHOULD determine the best type of subscription to be created on a per-leaf basis. That is to say, if the path specified within the message refers to some leaves which are event driven (e.g., the changing of state of an entity based on an external trigger) then an `ON_CHANGE` subscription may be created, whereas if other data represents counter values, a `SAMPLE` subscription may be created.

3.5.1.5.3 POLL Subscriptions

Polling subscriptions are used for on-demand retrieval of statistics via long-lived channels. A poll subscription relates to a certain set of subscribed paths, and is initiated by sending a `SubscribeRequest` message with encapsulated `SubscriptionList`. `Subscription` messages contained within the `SubscriptionList` indicate the set of paths that are of interest to the polling client.

To retrieve data from the target, a client sends a `SubscribeRequest` message to the target, containing a `poll` field, specified to be an empty `Poll` message. On reception of such a message, the target MUST generate updates for all the corresponding paths within the `SubscriptionList`. Updates MUST be generated according to Section 3.5.2.3.

3.5.1.6 Client-defined Aliases within a Subscription

When a client wishes to create an alias that a target should use for a path, the client should send a `SubscribeRequest` message specifying the `aliases` field. The `aliases` field consists of an `AliasList` message. An `AliasList` specifies a list of aliases, each of which consists of:

- `path` - the target path for the alias - encoded as per Section 2.2.2
- `alias` - the (client-defined) alias for the path, encoded as per Section 2.4.2.

Where a target is unable to support a client-defined alias it SHOULD respond with a `SubscribeResponse` message with the error field indicating an error of the following types:

- `InvalidArgument (3)` where the specified alias is not acceptable to the target.

- `AlreadyExists (6)` where the alias defined is a duplicate of an existing alias for the client.
- `ResourceExhausted (8)` where the target has insufficient memory or processing resources to support the alias.
- `Unknown (2)` in all other cases.

Thus, for a client to create an alias corresponding to the path `/a/b/c/d[id=10]/e` with the name `shortPath`, it sends a `SubscribeRequest` message with the following fields specified:

```
subscriberequest: <
  aliases: <
    alias: <
      path: <
        element: "a"
        element: "b"
        element: "c"
        element: "d[id=10]"
        element: "e"
      >
      alias: "#shortPath"
    >
  >
>
```

If the alias is acceptable to the target, subsequent updates are transmitted using the `#shortPath` alias in the same manner as described in Section 3.5.2.2.

3.5.2 Sending Telemetry Updates

3.5.2.1 Bundling of Telemetry Updates

Since multiple `Notification` messages can be included in the update field of a `SubscribeResponse` message, it is possible for a target to bundle messages such that fewer messages are sent to the client. The advantage of such bundling is clearly to reduce the number of bytes on the wire (caused by message overhead). Since `Notification` messages contain the timestamp at which an event occurred, or a sample was taken, such bundling does not affect the sample accuracy to the client. However, bundling does have a negative impact on the freshness of the data in the client - and on the client's ability to react to events on the target.

Since it is not possible for the target to infer whether its clients are sensitive to the latency introduced by bundling, if a target implements optimizations such that multiple `Notification` messages are bundled together, it MUST provide an ability to disable this functionality within the configuration of the gNMI service. Additionally, a target SHOULD

provide means by which the operator can control the maximum number of updates that are to be bundled into a single message, This configuration is expected to be implemented out-of-band to the gNMI protocol itself.

3.5.2.2 Target-defined Aliases within a Subscription

Where the `use_aliases` field of a `SubscriptionList` message has been set to true, a target MAY create aliases for paths within a subscription. A target-defined alias MUST be created separately from an update to the corresponding data item(s).

To create a target-defined alias, a `SubscribeResponse` message is generated with the `update` field set to a `Notification` message. The `Notification` message specifies the aliased path within the `prefix` field, and a non-null `alias` field, specified according to Section 2.4.2.

Thus, a target wishing to create an alias relating to the path `/a/b/c[id=10]` and subsequently update children of the `c[id=10]` entity must:

1. Generate a `SubscribeResponse` message and transmit it over the channel to the client:

```
subscriberesponse: <
  update: <
    timestamp: (timestamp)
    prefix: <
      element: "a"
      element: "b"
      element: "c[id=10]"
    >
    alias: "#42"
  >
>
```

2. Subsequently, this alias can be used to provide updates for the `child1` leaf corresponding to `/a/b/c[id=10]/child1`:

```
subscriberesponse: <
  update: <
    timestamp: (timestamp)
    prefix: <
      element: "#42"
    >
    update: <
      path: <
```

```
        element: "child1"
      >
      value: <
        value: 102                          // integer representation
        type: JSON_IETF
      >
    >
  >
>
```

3.5.2.3 Sending Telemetry Updates

When an update for a subscribed telemetry path is to be sent, a `SubscribeResponse` message is sent from the target to the client, on the channel associated with the subscription. The `update` field of the message contains a `Notification` message as per the description in Section 2.1. The `timestamp` field of the `Notification` message MUST be set to the time at which the value of the path that is being updated was collected.

Where a leaf node's value has changed, or a new node has been created, an `Update` message specifying the path and value for the updated data item MUST be appended to the `update` field of the message.

Where a node within the subscribed paths has been removed, the `delete` field of the `Notification` message MUST have the path of the node that has been removed appended to it.

When the target has transmitted the initial updates for all paths specified within the subscription, a `SubscribeResponse` message with the `sync_response` field set to true MUST be transmitted to the client to indicate that the initial transmission of updates has concluded. This provides an indication to the client that all of the existing data for the subscription has been sent at least once. For `STREAM` subscriptions, such messages are not required for subsequent updates. For `POLL` subscriptions, after each set of updates for individual poll request, a `SubscribeResponse` message with the `sync_response` field set to true MUST be generated.

4 Appendix: Current Protobuf Message and Service Specification

The latest Protobuf IDL gNMI specification will be found at https://github.com/openconfig/reference/.

5 Appendix: Current Outstanding Issues/Future Features

- Ability for the client to exclude paths from a subscription or get.

- "Dial out" for the target to register with an NMS and publish pre-configured subscriptions.

6 Copyright

```
Copyright 2016 Google Inc.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License
```

7 Revision History

- v0.2.1: November 10, 2016
  - Correct reference to `TEXT` vs. `ASCII` encoding type.
  - Ensure that encodings enumeration is numbered consistently.
  - Fix broken links.