YumaPro gRPC Manual

YANG-Based Unified Modular Automation Tools

YumaPro Google Remote Procedure Call [gRPC]

Version 21.10-2

Table of Contents

1 Preface

1.1 Legal Statements

1.2 Additional Resources

Other documentation includes:

YumaPro Quickstart Guide

YumaPro Installation Guide

YumaPro User Manual

YumaPro netconfd-pro Manual

YumaPro yangcli-pro Manual

YumaPro ypclient-pro Manual

YumaPro yangdiff-pro Manual

YumaPro yangdump-pro Manual

YumaPro Developer Manual

YumaPro API Quickstart Guide

YumaPro yp-system API Guide

YumaPro yp-show API Guide

YumaPro Yocto Linux Quickstart Guide

YumaPro yp-snmp Manual

YumaPro yp-gnmi Manual

To obtain additional support you may contact YumaWorks technical support department:

support@yumaworks.com

1.2.1 WEB Sites

- **YumaWorks**
  - https://www.yumaworks.com
  - Offers support, training, and consulting for YumaPro.
- **Netconf Central**
  - http://www.netconfcentral.org/
    - Free information on NETCONF and YANG, tutorials, on-line YANG module validation and documentation database
- Yang Central
  - http://www.yang-central.org

    - Free information and tutorials on YANG, free YANG tools for download
- **NETCONF Working Group Wiki Page**
    - http://trac.tools.ietf.org/wg/netconf/trac/wiki
    - Free information on NETCONF standardization activities and NETCONF implementations
- **NETCONF WG Status Page**
    - http://tools.ietf.org/wg/netconf/
    - IETF Internet draft status for NETCONF documents
- **libsmi Home Page**
    - http://www.ibr.cs.tu-bs.de/projects/libsmi/
    - Free tools such as smidump, to convert SMIv2 to YANG

1.2.2 Mailing Lists

- **NETCONF Working Group**
    - https://mailarchive.ietf.org/arch/browse/netconf/
    - Technical issues related to the NETCONF protocol are discussed on the NETCONF WG mailing list. Refer to the instructions on https://www.ietf.org/mailman/listinfo/netconf for joining the mailing list.
- **NETMOD Working Group**
    - https://datatracker.ietf.org/wg/netmod/documents/
    - Technical issues related to the YANG language and YANG data types are discussed on the NETMOD WG mailing list. Refer to the instructions on the WEB page for joining the mailing list.

1.3 Conventions Used in this Document

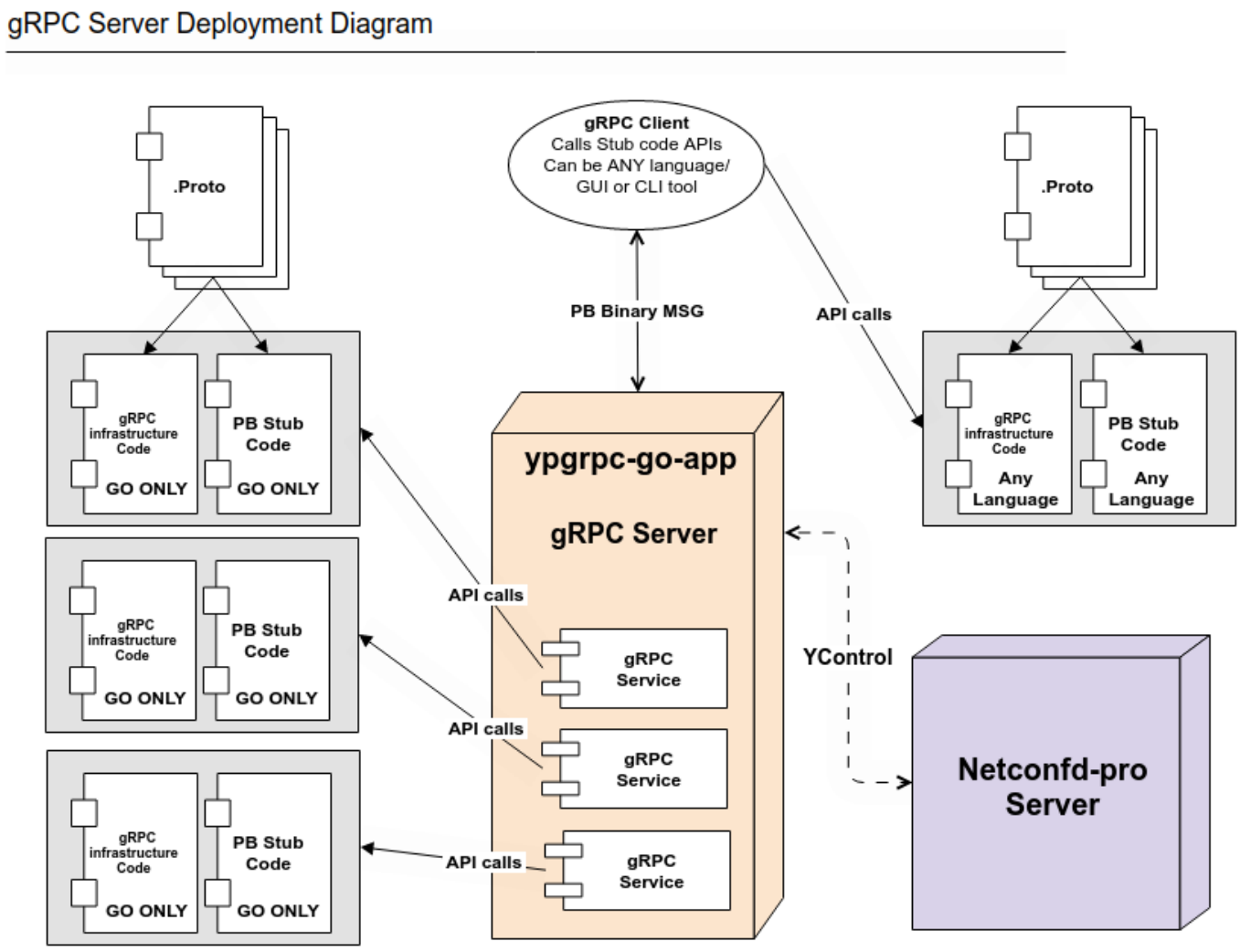
The following formatting conventions are used throughout this document:

Documentation Conventions

Convention	Description
--foo	CLI parameter foo
<f	XML element foo
f	**netconfd-pro** command or parameter
$FOO	Environment variable FOO
some text	Example command or PDU
some text	Plain text
Informational text	Useful or expanded information
Warning text	Warning information indicating possibly unexpected side-effects

2 YumaPro gRPC User Guide

This document describes the gRPC integration within the **netconfd-pro** server and **ypgrpc-go-app** application.



The above diagram illustrates deployment of the gRPC server, all its Services and messages handling and how the **netconfd-pro** server is integrated into this deployment.

2.1 Introduction

The gRPC server and **netconfd-pro** server integration is deployed with help of **ypgrpc-go-app** application that transfers information between integrated gRPC server and gRPC clients and **netconfd-pro** server. Also, this is an application that provides faster and easier platform to implement gRPC Services. It is similar to **db-api-app** where users create instrumentation for their Services and RPCs.

2.1.1 Features

The main YumaPro gRPC functionality and integration benefits:

- Developers provide .proto files
- Platform is **ypgrpc-go-app**, the application is maintained in **Golang** as well as gRPC server and server stub code
- The client gRPC code can be maintained in any language and can be used by any other tool to send a gRPC request for the data, it can be auto generated code, or some GUI tools, or CLI tools, etc
- Client sends gRPC requests directly to the **ypgrpc-go-app**
- Subsystem reports when gRPC stream starts and ends to the **netconfd-pro** server for monitoring information
- Stub code is generated using **protoc** tool
- Stub code is integrated into **ypgrpc-go-app** similar to **db-api-app**
    - All possible gRPC examples provided for faster and easier deployment, including:
        - Empty request and Empty response RPC
        - gRPC that represent single request and response
        - gRPC that represent single request and a streaming response
        - gRPC that represent a sequence of requests and a single responses
        - gRPC that represent a sequence of requests and responses with multiple different scenarios

The **netconfd-pro** server is a controller that provides more functionality to the gRPC client – server communication. The **netconfd-pro** server has the following interaction with gRPC server:

- **ypgrpc-go-app** registers its capabilities and all the information about gRPC Services
    - List of available Services
    - List of available RPCs
    - List of open streams and when they were started
    - Counters to keep track of open and closed streams
    - List of supported .proto files
    - Name, address and the port number of the gRPC server and when it was started
- <grpc-shutdown> RPC operation to shutdown the gRPC server

2.2 ypgrpc-go-app Overview

The **ypgrpc-go-app** application is YControl subsystem (similar to **db-api-app**) that communicates with the **netconfd-pro** server and also it is a gRPC server that communicates with RPC clients. The main role of the **ypgrpc-go-app** application is to host gRPC server and provide common place to implement and instrument gRPC services and also provide monitoring and control using **netconfd-pro** server.

2.2.1 ypgrpc-go-app Benefits

The **ypgrpc-go-app** application provides following benefits:

- Common place to implement and instrument .proto Services and RPCs
- Single gRPC Server to handle
- Subsystem reports to the **netconfd-pro** server with available capabilities
- Subsystem reports when subscriptions start and end to the **netconfd-pro** server for monitoring information
- Remote monitoring and control of gRPC server using **netconfd-pro** server
- Possibility to remotely shutdown the gRPC server using **netconfd-pro** server

2.2.2 ypgrpc-go-app Processing

The **ypgrpc-go-app** application is written in GO language and talks to the **netconfd-pro** server via socket and acts as a YControl subsystem (similar to **db-api-app**).

gRPC clients can be written in any languages that are supported for gRPC clients. The client part is out of the scope of this document and the current gRPC protocol integration does not include client part. The clients communicate to the gRPC server with help of the **ypgrpc-go-app** application and send gRPC request to the application.

The core of the **ypgrpc-go-app** is the gRPC server and integrated stub code from auto generated files from .proto files:

- Handle integrated stub code callbacks. Callbacks that are integrated from stub code that were generated from .proto files using **protoc** tool
- Register gRPC server for the protobuf message handling and gRPC Services callback invocation
- Run main Serve loop that handles all the client/server communication

The processing between gRCP client to the **netconfd-pro** server can be split into following components:

- **gRCP clients to the ypgrpc-go-app processing:** includes message parsing, gRPC Services callback invocation
- **The ypgrpc-go-app application to the netconfd-pro processing:** includes YControl messages exchange and stream information exchange when a new stream opens or closes
- **The netconfd-pro server internal processing**: includes subsystem registration, subsystem messages handling and parsing, gRPC monitoring information handling (gRPC server and streams status).

The **ypgrpc-go-app** implements multiple goroutines to manage the replies and the clients. All of this managers are goroutines. They run in parallel and asynchronously. The following goroutines are implemented in the **ypgrpc-go-app:**

- **Reply Manager goroutine.** This manager is responsible for any already parsed messages from the **netconfd-pro** server or gRPC client, it stores any not processed messages that are ready to be processed.

- **Message Manager goroutine.** This manager is responsible for storing any ready to be processed messages that are going to the **netconfd-pro** server and that are coming back from the server.

2.2.3 Startup Procedure

The **ypgrpc-go-app** application has the following startup steps for the gRPC server component:

- Initialize all the prerequisites and parse all the CLI parameters
- Open TCP socket to listen for clients requests
- Serve any incoming gRPC messages from gRPC clients and send **open-stream-event** or **close-stream-event** to the **netconfd-pro** server if needed with help of all the goroutine managers.

The **ypgrpc-go-app** acts as a YControl subsystem (similar to **db-api-app**), however, it does not terminate after one edit or get request. Instead it continuously listens to the **netconfd-pro** server and keeps the AF_LOCAL or TCP socket open to continue communication whenever it's needed. The communication is terminated only if the **ypgrpc-go-app** application is terminated, the **netconfd-pro** server terminates, or the **netconfd-pro** sends the request to terminate the **ypgrpc-go-app** application. All the message definitions described in the **yumaworks-yp-grpc.yang** YANG module.

The **ypgrpc-go-app** application has the following startup steps to initialize connection with the **netconfd-pro** server:

- Initialize all the prerequisites and parse all the CLI parameters
- Based on the **--proto** CLI parameter load all the .proto files and create capability structure for provided .proto files
- Open socket and send <ncx-connect> request to the server with
  - transport = netconf-aflocal
  - protocol = yp-grpc
- Register yp-grpc service
  - Send <register-request> to the server
  - Register **ypgrpc-go-app** subsystem and initialize all corresponding code in the **netconfd-pro** server to be ready to handle **ypgrpc-go-app** application requests
- Send **capability-ad-event** message to the **netconfd-pro** server to advertise all available Services, Methods and streams
- Keep listening socket until terminated

2.2.4 Configuration Parameter List

The following configuration parameters are used by **ypgrpc-go-app**. Refer to the CLI Reference for more details.

Ypgrpc-go-app CLI Parameters

parameter	description
--ca	Specifies the gRPC server CA certificate file
--cert	Specifies the gRPC server certificate file
--fileloc-fhs	Specifies whether the **ypgrpc-go-app** should use File system Hierarchy Standard (FHS) directory locations to create, store and use data and files
--insecure	Directs to skip TLS validation
--key	Specifies the gRPC server private key file
--log	Specifies the log file for the **ypgrpc-go-app** application
--log-console	Directs that log output will be be sent to STDOUT, after being sent to the log file and/or local syslog daemon
--log-level	Controls the verbosity level of messages printed to the log file or STDOUT, if no log file is specified
--port	Specifies the port value to use for gRPC server connections
--proto	Specifies the .proto file for the **ypgrpc-go-app** application to use
--protopath	Specifies the file search path for .proto files
--server-address	Specifies the **netconfd-pro** server IP address
--subsys-id	Specifies the subsystem identifier (gRPC Server ID) to use when registering with the **netconfd-pro** server

2.2.5 ypgrpc-go-app Source Files

This section describes the files that are contained in the **yumapro-grpc** package.

The following table lists the files that are included within the **netconf/src/ypgrpc** directory.

Directory	Description
cli	Handle the CLI parameters for **ypgrpc-go-app** application
credentials	Package credentials loads certificates and validates user credentials.
examples	Stub code example for .proto files (helloworld and example .protos)
log	Handle the Logging for **ypgrpc-go-app** application
message_handler	Auto-generated gostruct representation of the yumaworks-yp-grpc.yang file. Used for message handling
netconfd_connect	Handler for the **netconfd-pro** connection with **ypgrpc-go-app**
proto	.proto files handling, parsing, search and storing
utils	Generic utility functions
ycontrol	Utilities to handle the **netconfd-pro** YControl messages and connections

The **ypgrpc-go-app.go** is a main application that provides gRPC server functionality, connectivity to the **netconfd-pro** server and stub code gRPC Services callback handling.

2.3 Installation

The following sections describe the steps to install and test **ypgrpc-go-app** application.

2.3.1 Prerequisites

To install the Go programming language follow instructions here: https://golang.org/doc/install

Version go1.15 or higher is recommended. To verify the installation and to verify the version of the installed GO run the following:

```
mydir> go version
go version go1.16.5 linux/amd64
```

For **Protocol Buffer Compiler i**nstallation refer to:

https://grpc.io/docs/protoc-installation/

To verify the installation and to verify the version of the installed compiler run the following:

```
mydir> protoc --version  # Ensure compiler version is 3+
```

Go plugins for the protocol compiler:

Install the protocol compiler plugins for Go using the following commands:

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.26
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1
```

Update your PATH so that the **protoc** compiler can find the plugins:

```
$ export PATH="$PATH:$(go env GOPATH)/bin"
```

The gRPC client applications are out of the scope of this document and the current gRPC protocol integration does not include client part. There could be used any gRPC client application for verification. For more client applications refer to:

https://github.com/grpc-ecosystem/awesome-grpc#tools

To send the request to the gRPC server the following tool can be used as an example:

gRPC client cli Tool - generic gRPC command line client.

In order to install this tool download the binary and install it to **/usr/local** directory:

```
      > curl -L
https://github.com/vadimi/grpc-client-cli/releases/download/v1.10.0/grpc-client-
cli_darwin_x86_64.tar.gz | sudo tar -C /usr/local/bin -xz

      > GO111MODULE=on go get -u \
             github.com/vadimi/grpc-client-cli/cmd/grpc-client-cli@latest
```

To verify the installation and to verify the version of the installed client tool run the following:

```
      > grpc-client-cli --version
      grpc-client-cli version 1.10.0
```

You may need to update your **$GOPATH/bin** in order to run **grpc-client-cli** from you current directory, or run it as follows:

```
      > $HOME/go/bin/grpc-client-cli --version
      grpc-client-cli version 1.10.0
```

2.3.2 Binary Package Installation

If you do not have source code and want to install the YumaPro with a binary package then the application will be installed in the default **/usr/bin** location. If you'd like to use a different directory move the binary to your desired location.

You will have to create your workspace directory, setup **$GOBIN** and **$GOPATH** variables and install all dependencies. Create your workspace directory, **$HOME/go**. And set the **$GOPATH** environment variable. https://github.com/golang/go/wiki/SettingGOPATH

```
      mydir> mkdir -p ~/go
```

The **$GOPATH** can be any directory on your system. **$HOME/go** is the default **$GOPATH** on Unix-like systems since Go 1.8. Note that **$GOPATH** must not be the same path as your Go installation.

Edit your **~/.bash_profile** to add the following lines:

```
      export GOPATH=$HOME/go
      export GOBIN=$GOPATH/bin
```

Save and exit your editor. Then, source your **~/.bash_profile**:

```
mydir> source ~/.bash_profile
```

Using the 'go get' to install the following. Note that in this case **$GOBIN** and **$GOPATH** should be already setup:

```
mydir> GO111MODULE=off go get github.com/aws/aws-sdk-go/aws
mydir> GO111MODULE=off go get google.golang.org/grpc
mydir> GO111MODULE=off go get google.golang.org/grpc/codes
mydir> GO111MODULE=off go get google.golang.org/grpc/status
mydir> GO111MODULE=off go get google.golang.org/protobuf/types/known/emptypb
mydir> GO111MODULE=off go get golang.org/x/text/encoding/unicode
mydir> GO111MODULE=off go get golang.org/x/text/transform
mydir> GO111MODULE=off go get github.com/jessevdk/go-flags
mydir> GO111MODULE=off go get github.com/openconfig/goyang/pkg/yang
mydir> GO111MODULE=off go get github.com/openconfig/ygot/ygot
mydir> GO111MODULE=off go get github.com/openconfig/ygot/ytypes
mydir> GO111MODULE=off go get github.com/jhump/protoreflect/desc
mydir> GO111MODULE=off go get github.com/jhump/protoreflect/desc/protoparse
mydir> GO111MODULE=off go get github.com/clbanning/mxj
mydir> GO111MODULE=off go get github.com/golang/protobuf/proto
mydir> GO111MODULE=off go get github.com/davecgh/go-spew/spew
mydir> GO111MODULE=off go get github.com/golang/protobuf/protoc-gen-go/descriptor
```

If you get an error similar to:

```
go/src/github.com/openconfig/ygot/util/schema.go:65: s.Tag.Lookup undefined (type
reflect.StructTag has no field or method Lookup)
```

Upgrade to the latest go version. The version should be go1.15+

2.3.3 Binary Package ypgrpc-go-app Code Installation

The source code for **ypgrpc-go-app** application will be installed in **/usr/share/yumapro/src/ypgrpc/** directory for further modifications. However, there is no need to modify or run the application from that location if you want to test the application with out any modifications. The application has two .proto files implementations build in to illustrate its functionality. The .proto files are location at:

/usr/share/yumapro/src/ypgrpc/src/ypgrpc/examples/example/example.proto
/usr/share/yumapro/src/ypgrpc/src/ypgrpc/examples/helloworld/helloworld.proto

As well as theirs **pb.go** and **_grpc.pb.go** auto-generated files.

In order to use the application with modifications and new .proto files implementations the **ypgrpc-go-app** application can be modified and rebuild from **/usr/share/yumapro/src/ypgrpc/src/ypgrpc** directory or copied to the default $GOPATH location for modifications, for example:

```
mydir> cp -r /usr/share/yumapro/src/ypgrpc/src/ypgrpc $HOME/go/src/ypgrpc
```

After that the **ypgrpc-go-app** application can be modified, updated and run from **$HOME/go/src/ypgrpc** location.

- Compile and execute the **ypgrpc-go-app** code:

```
$HOME/go/src/ypgrpc> go run ypgrpc-go-app.go --log-level=debug \
          --fileloc-fhs \
          --insecure --proto=helloworld \
          --protopath=$HOME/protos
```

2.3.4 Source Code Installation

You will have to create your workspace directory, setup **$GOBIN** and **$GOPATH** variables and install all dependencies. Create your workspace directory, **$HOME/go**. And set the **$GOPATH** environment variable. https://github.com/golang/go/wiki/SettingGOPATH

```
mydir> mkdir -p ~/go
```

The **$GOPATH** can be any directory on your system. **$HOME/go** is the default **$GOPATH** on Unix-like systems since Go 1.8. Note that **$GOPATH** must not be the same path as your Go installation.

Edit your **~/.bash_profile** to add the following lines:

```
export GOPATH=$HOME/go
export GOBIN=$GOPATH/bin
```

Save and exit your editor. Then, source your **~/.bash_profile**:

```
mydir> source ~/.bash_profile
```

Using the 'go get' to install the following. Note that in this case **$GOBIN** and **$GOPATH** should be already setup:

```
mydir> GO111MODULE=off go get github.com/aws/aws-sdk-go/aws
mydir> GO111MODULE=off go get google.golang.org/grpc
mydir> GO111MODULE=off go get google.golang.org/grpc/codes
```

```
mydir> GO111MODULE=off go get google.golang.org/grpc/status
mydir> GO111MODULE=off go get google.golang.org/protobuf/types/known/emptypb
mydir> GO111MODULE=off go get golang.org/x/text/encoding/unicode
mydir> GO111MODULE=off go get golang.org/x/text/transform
mydir> GO111MODULE=off go get github.com/jessevdk/go-flags
mydir> GO111MODULE=off go get github.com/openconfig/goyang/pkg/yang
mydir> GO111MODULE=off go get github.com/openconfig/ygot/ygot
mydir> GO111MODULE=off go get github.com/openconfig/ygot/ytypes
mydir> GO111MODULE=off go get github.com/jhump/protoreflect/desc
mydir> GO111MODULE=off go get github.com/jhump/protoreflect/desc/protoparse
mydir> GO111MODULE=off go get github.com/clbanning/mxj
mydir> GO111MODULE=off go get github.com/golang/protobuf/proto
mydir> GO111MODULE=off go get github.com/davecgh/go-spew/spew
mydir> GO111MODULE=off go get github.com/golang/protobuf/protoc-gen-go/descriptor
```

OR

```
~/ypwork/netconf/src/ypgrpc$ make goget
```

If you have installed the YumaPro from source code then you need to build and install using WITH_GRPC=1 and WITH_YCONTROL=1 build variables. Build the **netconfd-pro** server with gRPC support:

```
make WITH_YCONTROL=1 WITH_GRPC=1
sudo make WITH_YCONTROL=1 WITH_GRPC=1 install
```

Plus your custom and optional GO_PATH=$CUSTOM_GOPATH GO_BIN=$CUSTOM_GOBIN flags if needed.

If you have created your custom workspace directory and would like the **ypgrpc-go-app** application to be installed in your custom location you will need to set the custom **$GOPATH** and **$GOBIN** Variables. Otherwise, this step is optional and during the installation **$HOME/go** GO workspace will be created and all **ypgrpc-go-app** application dependencies will be installed there. Follow these steps to setup custom workspace and Variables:

https://github.com/golang/go/wiki/SettingGOPATH

You will also need to provide Build Variables GO_PATH=$CUSTOM_GOPATH GO_BIN=$CUSTOM_GOBIN if you created your custom workspace and want the application to be installed there.

```
GO_BIN=<dirspec>: specify the $GOBIN variable dirspec to use when
  building YP-gRPC application. Default is $HOME/go/bin.
  Ignored if PACKAGE_BUILD=1 is also used.

GO_PATH=<dirspec>: specify the $GOPATH variable dirspec to use when
  building YP-gRPC application. Default is $HOME/go.
  Ignored if PACKAGE_BUILD=1 is also used.
```

In this case the **ypgrpc-go-app** will be installed into your custom **$GOBIN** location. By default the application is installed in the **/usr/bin/.**

The source code of the **ypgrpc-go-app** application will be installed to **$GO_PATH/src** (by default the **$GO_PATH** is **$HOME/go**).

2.3.5 Source Code ypgrpc-go-app Installation

The source code for **ypgrpc-go-app** application will be in **/netconf/src/ypgrpc** directory. However, there is no need to modify or run the application from that location if you want to test the application with out any modifications. The application has two .proto files implementations build in to illustrate its functionality. The .proto files are location at:

/netconf/src/ypgrpc/src/ypgrpc/examples/example/example.proto
/netconf/src/ypgrpc/src/ypgrpc/examples/helloworld/helloworld.proto

As well as theirs **pb.go** and **_grpc.pb.go** auto-generated files.

In order to use the application with modifications and new .proto files implementations the **ypgrpc-go-app** application should be copied to the default **$GOPATH** location and rebuild from that directory, for example:

```
mydir> cp -r /netconf/src/ypgrpc/src/ypgrpc/ $HOME/go/src/ypgrpc
```

After that the **ypgrpc-go-app** application can be modified, updated and run from **$HOME/go/src/ypgrpc** location.

- Compile and execute the **ypgrpc-go-app** code:

```
$HOME/go/src/ypgrpc> go run ypgrpc-go-app.go --log-level=debug \
          --fileloc-fhs \
          --insecure --proto=helloworld \
          --protopath=$HOME/protos
```

2.3.6 Generate the CA Certificates

Generate the client and server certificates if gRPC client uses TLS validation.

In order to skip TLC validation use **--insecure** CLI parameter.

If you have already installed the certificates required for TLS as described in the section “Configure TLS” of the YumaPro SDK Installation Guide make sure copies of the client and server keys and certs are in ~/certs and also the ca.crt is available.

```
-rw-rw-r--  1 john john  956 Aug  9 10:41 ca.crt
-rw-rw-r--  1 john john  969 Aug  2 11:00 client.crt
-rw-rw-r--  1 john john 1704 Aug  2 11:00 client.key
-rw-rw-r--  1 john john  964 Aug  2 11:01 server.crt
-rw-rw-r--  1 john john 1704 Aug  2 11:01 server.key
```

Then you can skip to the next section.

If you have not installed TLS certificates follow these steps:

```
     mydir> mkdir ~/certs
     mydir> cp /usr/share/yumapro/util/generate-keys.sh ~/certs
or
     mydir> cp netconfd/util/generate-keys.sh ~/certs
     mydir> cd ~/certs
     certs> ./generate-keys.sh
```

Note: **generate-keys.sh** script contains one line where it configures the server's Common Name and other parameters:

SUBJ="/C=/ST=/L=/O=/CN=your_target_name"

Change it to your target name. By default the value is "restconf". You may keep it for testing, but during the gRPC client requests make sure to specify this target name.

2.3.7 Running ypgrpc-go-app

Run the server with the setting **--with-grpc=true** flag as follows:

```
     mydir> sudo netconfd-pro --log-level=debug4 --with-grpc=true \
            -- fileloc-fhs=true
```

Start the **ypgrpc-go-app** application. Note that you have to provide your certificates to start the application:

```
     mydir> man ypgrpc-go-app
     mydir> ypgrpc-go-app --cert=~/certs/server.crt --ca=~/certs/ca.crt \
            --key=~/certs/server.key \
            --fileloc-fhs \
            --protopath=$HOME/protos \
            --proto=helloworld --proto=example
```

OR run it in “insecure” mode for test or verification:

```
     mydir> ypgrpc-go-app --log-level=debug --fileloc-fhs --insecure \
            --protopath=$HOME/protos \
            --proto=helloworld --proto=example
```

After this step the gRPC server starts to listen for any gRPC client requests and will handle all the request for the provided example.proto and helloworld.proto .proto files and will advertise gRPC capabilities and example.proto, helloworld.proto Services to the **netconfd-pro** server.

2.3.8 Closing ypgrpc-go-app

The **ypgrpc-go-app** can be shut down by typing Ctrl-C in the window that started the application.

If the **netconfd-pro** server is not running when **ypgrpc-go-app** is started the application will terminate with an error message stating that the **netconfd-pro** server is not running.

If the **netconfd-pro** server is shut down then **ypgrpc-go-app** will also shutdown.

The **netconfd-pro** server has <grpc-shutdown> NETCONF operation that can be triggered to shut down the **ypgrpc-go-app** application.

2.3.9 Proto Search Path

The **ypgrpc-go-app** uses configurable search paths to find .proto files that are needed during operation.

- If the **--protopath** parameter is specified with a path, that search path is tried, relative to the current working directory. If it is not found then the search terminates in failure. Sub-directories will be searched.

```
            --protopath=../../protos
```

- If the **--proto** is specified without **--protopath** path, then The **$HOME/protos** directory is checked by default. Sub-directories will be searched.

NOTE: In this manual examples the .proto files are stored in the $HOME/protos location.

3 ypgrpc-go-app Quick Start Guide

As in many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a gRPC server to handle client calls.

The **ypgrpc-go-app** subsystem provides an unified place where all the interfaces can be implemented and runs a gRPC server to handle client calls.

The first step when working with protocol buffers is to define the structure for the data you want to serialize in a .proto file: this is an ordinary text file with a .proto extension. Protocol buffer data is structured as messages, where each message is a small logical record of information containing a series of name-value pairs called fields.

Then you define gRPC services in ordinary .proto files, with RPC method parameters and return types specified as protocol buffer messages, refer to **$HOME/go/src/ypgrpc/examples/helloworld/helloworld.proto.** Here's a simple example:

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

  /* Sends another greeting */
  rpc SayHelloAgain (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;
}
```

3.1 ypgrpc-go-app Application Helloworld Example

This section gets you started with **ypgrpc-go-app** and gRPC server with a simple working example.

- Run the **netconfd-pro** server with the setting **--with-grpc=true** flag as follows:

```
mydir> sudo netconfd-pro --log-level=debug4 --with-grpc=true \
       --fileloc-fhs=true
```

The example code of **ypgrpc-go-app** after installation should be installed in **$HOME/go/src/ypgrpc** (refer to the installation steps in this manual) and example Service implementation is installed in **$HOME/go/src/ypgrpc/examples**. In order to run the example application:

- Change to the example directory:

```
> cd $HOME/go/src/ypgrpc
```

- Compile and execute the **ypgrpc-go-app** code:

```
ypgrpc> go run ypgrpc-go-app.go --log-level=debug --fileloc-fhs \
       --insecure --proto=helloworld --protopath=$HOME/protos
```

- Run the client application:

The gRPC client applications are out of the scope of this document and the current gRPC protocol integration does not include client part. There could be used any gRPC client application for verification. For more client applications refer to:

https://github.com/grpc-ecosystem/awesome-grpc#tools

In this example **grpc-client-cli** tool is used:

```
> grpc-client-cli --proto \
       $HOME/protos/helloworld.proto localhost:50830
? Choose a service: helloworld.Greeter
? Choose a method: SayHello
Message json (type ? to see defaults): {"name":"An example name"}
{
  "message": "Hello An example name"
}
```

After the request is sent the gRPC server will run the corresponding callback and reply to the client request with a message as defined in the .proto files. The **ypgrpc-go-app** application log may look as follows:

```
ypgrpc_server: Starting to serve

HelloRequest:{
  "name": "An example name"
}
```

Congratulations! You've just run a client-server application with gRPC.

3.1.1 Update ypgrpc-go-app Services

In this section you'll update the **ypgrpc-go-app** application with an extra server method.

The gRPC service is defined using following .proto file. The following .proto will generate both client and server stub code that have a **SayHello()** RPC method that takes a **HelloRequest** parameter from the client and returns a **HelloReply** from the server:

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

  /* Sends another greeting */
  rpc SayHelloAgain (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;
}
```

Open **$HOME/go/src/ypgrpc/examples/helloworld/helloworld.proto** and add a new **SayHelloOneMore**() method, with the same request and response types:

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

  /* Sends another greeting */
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}

  /* Sends another greeting */
  rpc SayHelloOneMore (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;
}
```

3.1.2 Regenerate gRPC Code

Before you can use the new service method, you need to recompile the updated .proto file.

While still in the examples directory, run the following command:

```
> cd $HOME/go/src/ypgrpc/examples

> protoc --go_out=. --go_opt=paths=source_relative \
       --go-grpc_out=. --go-grpc_opt=paths=source_relative \
       helloworld/helloworld.proto
```

This will regenerate the **helloworld/helloworld.pb.go** and helloworld/helloworld_grpc.pb.go files, which contain:

- Code for populating, serializing, and retrieving **HelloRequest** and **HelloReply** message types.
- Generated server stub code to integrate into **ypgrpc-go-app** application.

Server Code will be used by **ypgrpc-go-app** application that will register this Service and Methods and where the instrumentation will be done.

3.1.3 Update ypgrpc-go-app

After the regenerated server code is complete, now it can be implemented, called and intergated into the **ypgrpc-go-app** application.

Open **$HOME/go/src/ypgrpc/ypgrpc-go-app.go** and add the following function to it:

```
/* SayHelloOneMore implements helloworld.GreeterServer */
func (s *helloworldServer) SayHelloOneMore (ctx context.Context,
                                            in *helloworld.HelloRequest) (
    *helloworld.HelloReply,
     error) {

    log.Log_info("\n\nHelloRequest:")
    log.Log_dump_structure(in)

    return &helloworld.HelloReply{
        Message: "Say Hello OneMore" + in.GetName(),
    }, nil
}
```

3.1.4 Run Updated ypgrpc-go-app

Run the **netconfd-pro** server and the **ypgrpc-go-app** application like you did before.

- Run the **netconfd-pro** server:

```
mydir> sudo netconfd-pro --log-level=debug4 --with-grpc=true \
       --fileloc-fhs=true
```

- Change to the example directory:

```
> cd $HOME/go/src/ypgrpc
```

- Run updated **ypgrpc-go-app** application:

```
ypgrpc> go run ypgrpc-go-app.go --log-level=debug --fileloc-fhs \
       --insecure --proto=helloworld --protopath=$HOME/protos
```

- Run the client application:

The gRPC client applications are out of the scope of this document and the current gRPC protocol integration does not include client part. There could be used any gRPC client application for verification. For more client applications refer to:

https://github.com/grpc-ecosystem/awesome-grpc#tools

In this example **grpc-client-cli** tool is used:

```
> grpc-client-cli --proto \
       $HOME/protos/helloworld.proto localhost:50830
? Choose a service: helloworld.Greeter
? Choose a method: SayHelloOneMore
Message json (type ? to see defaults): {"name":" An example name"}
{
 "message": "Say Hello OneMore An example name"
}
```

After the request is sent the gRPC server will run the corresponding callback and reply to the client request with a message as defined in the .proto files. The **ypgrpc-go-app** application log may look as follows:

```
ypgrpc_server: Starting to serve

HelloRequest:{
  "name": " An example name"
}
```

4 ypgrpc-go-app and gRPC Services

This tutorial provides a basic Go programmer's introduction to working with **ypgrpc-go-app** and gRPC callbacks.

The first step is to define the gRPC Service and the method request and response types using protocol buffers. For the complete .proto file, see **$HOME/go/src/ypgrpc/examples/example/example.proto**

gRPC lets you define five kinds of service method, all of which are used in the ExampleService service:

- An **empty request** and empty response RPC

```
/* Empty request And Empty response RPC */
rpc EmptyCall(google.protobuf.Empty) returns (google.protobuf.Empty);
```

- A **simple RPC** where the client sends a request to the server using the stub and waits for a response to come back, just like a normal function call.

```
/* RPC that represent single request and response
 * The server returns the client payload as-is.
 */
rpc UnaryCall(SimpleRequest) returns (SimpleResponse);
```

- A **server-side streaming** RPC where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. As you can see in our example, you specify a server-side streaming method by placing the stream keyword before the response type.

```
/* RPC that represent single request and a streaming response
 * The server returns the payload with client desired type and sizes.
 */
rpc StreamingOutputCall(StreamingOutputCallRequest)
    returns (stream StreamingOutputCallResponse);
```

- A **client-side streaming** RPC where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them all and return its response. You specify a client-side streaming method by placing the stream keyword before the request type.

```
/* RPC that represent a sequence of requests and a sngle responses
 * The server returns the aggregated size of client payload as the result.
 */
rpc StreamingInputCall(stream StreamingInputCallRequest)
    returns (StreamingInputCallResponse);
```

- A **bidirectional streaming** RPC where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved. You specify this type of method by placing the stream keyword before both the request and the response.

```
/* RPC that represent a sequence of requests and responses
 * with each request served by the server immediately.
 * As one request could lead to multiple responses, this interface
 * demonstrates the idea of full duplexing.
 */
rpc FullDuplexCall(stream StreamingOutputCallRequest)
    returns (stream StreamingOutputCallResponse);

/* RPC that represent a sequence of requests and responses.
 * The server buffers all the client requests and then serves them in order.
 * A stream of responses are returned to the client when the server starts with
 * first request.
 */
rpc HalfDuplexCall(stream StreamingOutputCallRequest)
    returns (stream StreamingOutputCallResponse);
```

The .proto file also contains protocol buffer message type definitions for all the request and response types used in our service methods - for example, here's the **SimpleRequest** message type:

```
/* Unary request */
message SimpleRequest {
  EchoStatus response_status = 1;
  User user = 2;
}
```

4.1 ypgrpc-go-app Application

The **ypgrpc-go-app** subsystem provides an interface to add new gRPC Services and implement new Methods.

The **ypgrpc-go-app** program "**ypgrpc-go-app .go**" shows an example of how the gRPC interface can be used and how the Services and methods can be implemented.

Example **ypgrpc-go-app** Application:

```
/** helloworldServer is used to implement helloworld.GreeterServer */
type helloworldServer struct {
    helloworld.UnimplementedGreeterServer
}


/** exampleServer is used to implement example.ExampleServiceServer */
type exampleServer struct {
    example.UnimplementedExampleServiceServer
}

/** @} */


/*****************************************************************
*                                                                *
*                      F U N C T I O N S                         *
*                                                                *
*****************************************************************/


/* SayHello implements helloworld.GreeterServer */
func (s *helloworldServer) SayHello (ctx context.Context,
                                     in *helloworld.HelloRequest) (
    *helloworld.HelloReply,
     error) {

    log.Log_info("\n\nHelloRequest:")
    log.Log_dump_structure(in)

    return &helloworld.HelloReply{
        Message: "Hello " + in.GetName(),
    }, nil
}


/* SayHelloAgain implements helloworld.GreeterServer */
func (s *helloworldServer) SayHelloAgain (ctx context.Context,
                                          in *helloworld.HelloRequest) (
    *helloworld.HelloReply,
     error) {

    log.Log_info("\n\nHelloRequest:")
    log.Log_dump_structure(in)

    return &helloworld.HelloReply{
        Message: "Say Hello Again" + in.GetName(),
    }, nil
}
```

```
/* SayHelloOneMore implements helloworld.GreeterServer */
func (s *helloworldServer) SayHelloOneMore (ctx context.Context,
                                            in *helloworld.HelloRequest) (
    *helloworld.HelloReply,
     error) {

    log.Log_info("\n\nHelloRequest:")
    log.Log_dump_structure(in)

    return &helloworld.HelloReply{
        Message: "Say Hello OneMore" + in.GetName(),
    }, nil
}


/* Empty request And Empty response RPC */
func (exampleServer) EmptyCall (ctx context.Context,
                                in *emptypb.Empty) (
    *emptypb.Empty,
    error) {

    log.Log_info("\n\nEmptyRequest:")
    log.Log_dump_structure(in)

    /* Empty input and putput.
     * Run your Instrumentation here
     */
    return in, nil
}


/* RPC that represent single request and response
 * The server returns the client payload as-is.
 */
func (exampleServer) UnaryCall (ctx context.Context,
                                in *example.SimpleRequest) (
    *example.SimpleResponse,
    error) {

    log.Log_info("\n\nSimpleRequest:")
    log.Log_dump_structure(in)

    /* The Incoming Code must be 0 to signal that request is well formed  */
    if in.ResponseStatus != nil && in.ResponseStatus.Code != int32(codes.OK) {

        log.Log_info("\nReceived Error Code: %v",
            in.GetResponseStatus().GetCode())

        return nil, status.Error(codes.Code(in.ResponseStatus.Code), "error")
    }

    return &example.SimpleResponse{
        User: &example.User{
            Id:   in.GetUser().GetId(),
            Name: in.GetUser().GetName(),
        },
    }, nil
}


/* RPC that represent single request and a streaming response
 * The server returns the payload with client desired type and sizes.
 */
```

```
func (exampleServer) StreamingOutputCall (in *example.StreamingOutputCallRequest,
                    stream example.ExampleService_StreamingOutputCallServer) error {

    /* Example Request:

        { "response_parameters":[
            {"size":10,"interval_us":5},
            {"size":10,"interval_us":6}
         ],
         "user": {"id":13,"name":"Example-1"},
         "response_status":{"code":0,"message":"Example message 1"}
        }
    */
    log.Log_info("\n\nStreamingOutputCallRequest:")
    log.Log_dump_structure(in)

    rsp := &example.StreamingOutputCallResponse{
        User: &example.User{},
    }

    /* Starting/ Closing a Server Stream. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "StreamingOutputCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "StreamingOutputCall")

    count := int32(1)
    for _, param := range in.ResponseParameters {

        log.Log_info("\nInterval between responses: %v\n",
            param.GetIntervalUs())

        /* Wait as specified in the interval parameter */
        time.Sleep(time.Duration(param.GetIntervalUs()) * time.Second)

        if stream.Context().Err() != nil {
            /* Closing a Server Stream. Update monitoring information. */
            return stream.Context().Err()
        }

        buf := ""
        for i := 0; i < int(param.GetSize()); i++ {
            buf += in.GetUser().GetName()
        }

        count++
        rsp.User.Id = count
        rsp.User.Name = buf

        if err := stream.Send(rsp); err != nil {
            /* Closing a Server Stream. Update monitoring information. */
            return err
        }
    }

    log.Log_info("\nDone Streaming")

    return nil
}
```

```
/* RPC that represent a sequence of requests and a single responses
 * The server returns the aggregated size of client payload as the result.
 */
func (exampleServer) StreamingInputCall (stream
example.ExampleService_StreamingInputCallServer) error {

    log.Log_info("\n\nExampleService_StreamingInputCallServer:")
    log.Log_dump_structure(stream)

    /* Starting/ Closing a Client Stream. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "StreamingInputCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "StreamingInputCall")
    size := 0
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&example.StreamingInputCallResponse{
                AggregatedPayloadSize: int32(size),
            })
        }

        size += len(req.User.Name)

        if err != nil {
            return err
        }
    }
}


/* RPC that represent a sequence of requests and responses
 * with each request served by the server immediately.
 * As one request could lead to multiple responses, this interface
 * demonstrates the idea of full duplexing.
 */
func (exampleServer) FullDuplexCall (stream example.ExampleService_FullDuplexCallServer)
error {

    log.Log_info("\n\nExampleService_FullDuplexCallServer:")
    log.Log_dump_structure(stream)

    /* Starting/ Closing a Client/ Server Streams. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "FullDuplexCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "FullDuplexCall")
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return nil
        }

        if err != nil {
            return status.Error(codes.Internal, err.Error())
        }

        if req.ResponseStatus != nil && req.ResponseStatus.Code != int32(codes.OK) {
```

```
            return status.Error(codes.Code(req.ResponseStatus.Code), "error")

        }

        resp := &example.StreamingOutputCallResponse{User: &example.User{}}
        for _, param := range req.ResponseParameters {
            if stream.Context().Err() != nil {
                return stream.Context().Err()
            }

            buf := ""
            for i := 0; i < int(param.GetSize()); i++ {
                buf += req.GetUser().GetName()
            }

            resp.User.Name = buf

            if err := stream.Send(resp); err != nil {
                return err
            }
        }
    }
}


/* RPC that represent a sequence of requests and responses.
 * The server buffers all the client requests and then serves them in order.
 * A stream of responses are returned to the client when the server starts with
 * first request.
 */
func (exampleServer) HalfDuplexCall (stream example.ExampleService_HalfDuplexCallServer)
error {

    log.Log_info("\n\nExampleService_HalfDuplexCallServer:")
    log.Log_debug_dump(stream)

    /* Starting/ Closing a Client/ Server Streams. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "HalfDuplexCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "HalfDuplexCall")

    requests := []*example.StreamingOutputCallRequest{}
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            break
        }

        if err != nil {
            return status.Error(codes.Internal, err.Error())
        }

        requests = append(requests, req)
    }

    for _, req := range requests {
        resp := &example.StreamingOutputCallResponse{User: &example.User{}}

        for _, param := range req.ResponseParameters {
            if stream.Context().Err() != nil {
```

```
                return stream.Context().Err()
            }

            buf := ""
            for i := 0; i < int(param.GetSize()); i++ {
                buf += req.GetUser().GetName()
            }

            resp.User.Name = buf

            if err := stream.Send(resp); err != nil {
                return err
            }
        }
    }

    return nil
}


/**
* @brief MAIN IO server loop for the gRPC manager
*
*/
func main () {

    var res error = nil

    /* Connecton to netconfd-pro server:
     *   1) Parse CLI parameters:
     *       - Host
     *       - subsys-id
     *       - user
     *       - proto files
     *       - etc
     *   2) Open Socket and send NCX-Connect request
     *   3) Start to listen on the socket (select loop)
     *   4) Register YControl service (gRPC service)
     *   5) send <capability-ad-event> event
     *   7) Start to listen for any request <-> response
     *
     */

    /* Parse all the CLI parameters */
    res = cli.ParseCLIParameters()
    if res != nil {
        utils.Check_error(res)
    }

    /* set logging parameters */
    log.SetLevel()
    log.SetLogOut()

    /* Print all the CLi parameters provided */
    log.Log_dump_structure(cli.GetCliOptions())

    log.Log_info("\n\nStarting ypgrpc-go-app...")
    log.Log_info("\nCopyright (c) 2021, YumaWorks, Inc., " +
                 "All Rights Reserved.\n")

    /* Connect the the server */
    conn, res := netconfd_connect.Connect_netconfd()
    if res != nil {
```

```
        utils.Check_error(res)
    }

    /* Defer is used to ensure that a function call is performed
     * later in a program's execution, usually for purposes of cleanup.
     * defer is often used where e.g. ensure and finally would be used
     * in other languages.
     */
    defer conn.Close()

    /* Get the actual TCP host address to use for gRPC server,
     * Address CLI parameter + port number,
     * By default this will be:
     * 127.0.0.1:50830
     */
    addr := netconfd_connect.GetServerAddr()

    log.Log_info("\nStarting the gRPC server ...")
    log.Log_info("\nListening for client request on '%s'",
        addr)

    /* All initialiation is done
     * Start the gRPC server to listen for clients
     */
    lis, err := net.Listen("tcp", addr)
    if err != nil {
        log.Log_error("\nError: failed to Listen (%v)", err)
    }

    /* Start the gRPC server and Register all the gRPC Services */
    grpcServer := grpc.NewServer()

    /* Register All the Services here */
    helloworld.RegisterGreeterServer(grpcServer, &helloworldServer{})
    example.RegisterExampleServiceServer(grpcServer, &exampleServer{})


    log.Log_info("\nypgrpc_server: Starting to serve")
    if err := grpcServer.Serve(lis); err != nil {
        log.Log_error("\nError: failed to Serve (%v)", err)
    }

    result, res := ioutil.ReadAll(conn)
    utils.Check_error(res)

    log.Log_info("\n%s", string(result))

    log.CloseLogOut()
    os.Exit(0)

} /* main */
```

4.2 ypgrpc-go-app Interface Functions

The **ypgrpc-go-app** application is an YControl subsystem (similar to **db-api-app**) that has multiple API to communicate with the **netconfd-pro** server.

Setup

- **netconfd_connect.Connect_netconfd:** Initialize the YP-gRPC service with the YControl subsystem and advertise gRPC server capabilities to the **netconfd-pro**

Update monitoring information

- **netconfd_connect.Open_streams:** Sends subsystem event to advertise all the gRPC available and active capabilities during the registration time
- **netconfd_connect.Close_streams**: Sends subsystem event to advertise that the gRPC server or client stream was closed.

4.3 ypgrpc-go-app Implementing gRPC Service

The **ypgrpc-go-app** example gRPC server has a **exampleServer** structure type that implements the generated **ExampleService** interface:

```
type exampleServer struct {
     ...
}

func (exampleServer) EmptyCall (ctx context.Context,
                                in *emptypb.Empty) (
    *emptypb.Empty,
    error) {
     ...
}

func (exampleServer) UnaryCall (ctx context.Context,
                                in *example.SimpleRequest) (
    *example.SimpleResponse,
    error) {

     ...
}

func (exampleServer) StreamingOutputCall (in *example.StreamingOutputCallRequest,
                   stream example.ExampleService_StreamingOutputCallServer) error {
     ...
}

func (exampleServer) StreamingInputCall (stream
example.ExampleService_StreamingInputCallServer) error {
     ...
}
```

```
func (exampleServer) FullDuplexCall (stream example.ExampleService_FullDuplexCallServer)
error {
     ...
}

func (exampleServer) HalfDuplexCall (stream example.ExampleService_HalfDuplexCallServer)
error {
     ...
}
```

4.3.1 Empty RPC

The **exampleServer** implements all our service methods. Let's look at the **EmptyCall,** which just gets an empty request from the client and returns an empty response. This is the simplest example that illustrates empty request response mechanism.

```
/* Empty request And Empty response RPC */
func (exampleServer) EmptyCall (ctx context.Context,
                                in *emptypb.Empty) (
    *emptypb.Empty,
    error) {

    log.Log_info("\n\nEmptyRequest:")
    log.Log_dump_structure(in)

    /* Empty input and putput.
     * Run your Instrumentation here
     */
    return in, nil
}
```

4.3.2 Simple RPC

Now let's look at the simple **UnaryCall,** which just gets a **SimpleRequest** from the client and returns the corresponding **SimpleResponse** from the server.

```
/* RPC that represent single request and response
 * The server returns the client payload as-is.
 */
func (exampleServer) UnaryCall (ctx context.Context,
                                in *example.SimpleRequest) (
    *example.SimpleResponse,
    error) {

    log.Log_info("\n\nSimpleRequest:")
    log.Log_dump_structure(in)

    /* The Incoming Code must be 0 to signal that request is well formed  */
    if in.ResponseStatus != nil && in.ResponseStatus.Code != int32(codes.OK) {

        log.Log_info("\nReceived Error Code: %v",
            in.GetResponseStatus().GetCode())
```

```
        return nil, status.Error(codes.Code(in.ResponseStatus.Code), "error")
    }

    return &example.SimpleResponse{
        User: &example.User{
            Id:   in.GetUser().GetId(),
            Name: in.GetUser().GetName(),
        },
    }, nil
}
```

The method is passed a context object for the RPC and the client's **SimpleRequest** protocol buffer request. It returns a **SimpleResponse** protocol buffer object with the response information and an error. In the method we populate the **SimpleResponse** with the appropriate information, and then return it along with an nil error to tell gRPC that we've finished dealing with the RPC and that the **SimpleResponse** can be returned to the client.

4.3.3 Server-side Streaming RPC

Now let's look at one of our streaming RPCs. **StreamingOutputCall** is a server-side streaming RPC, so we need to send back multiple **StreamingOutputCallResponse** to our client.

```
/* RPC that represent single request and a streaming response
 * The server returns the payload with client desired type and sizes.
 */
func (exampleServer) StreamingOutputCall (in *example.StreamingOutputCallRequest,
                    stream example.ExampleService_StreamingOutputCallServer) error {

    /* Example Request:

        { "response_parameters":[
            {"size":10,"interval_us":5},
            {"size":10,"interval_us":6}
         ],
         "user": {"id":13,"name":"Example-1"},
         "response_status":{"code":0,"message":"Example message 1"}
        }
    */
    log.Log_info("\n\nStreamingOutputCallRequest:")
    log.Log_dump_structure(in)

    rsp := &example.StreamingOutputCallResponse{
        User: &example.User{},
    }

    /* Starting/ Closing a Server Stream. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "StreamingOutputCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "StreamingOutputCall")

    count := int32(1)
    for _, param := range in.ResponseParameters {

        log.Log_info("\nInterval between responses: %v\n",
            param.GetIntervalUs())
```

```
        /* Wait as specified in the interval parameter */
        time.Sleep(time.Duration(param.GetIntervalUs()) * time.Second)

        if stream.Context().Err() != nil {
            /* Closing a Server Stream. Update monitoring information. */
            return stream.Context().Err()
        }

        buf := ""
        for i := 0; i < int(param.GetSize()); i++ {
            buf += in.GetUser().GetName()
        }

        count++
        rsp.User.Id = count
        rsp.User.Name = buf

        if err := stream.Send(rsp); err != nil {
            /* Closing a Server Stream. Update monitoring information. */
            return err
        }
    }

    log.Log_info("\nDone Streaming")

    return nil
}
```

As you can see, instead of getting simple request and response objects in our method parameters, this time we get a request object and a special **StreamingOutputCallRequest** object to write our responses. The server returns the aggregated size of client payload as the result.

In the method, we populate as many **StreamingOutputCallResponse** objects as we need to return, writing them to the **ExampleService_StreamingOutputCallServer** using its Send() method. Finally, as in our simple RPC, we return a **nil** error to tell gRPC that we've finished writing responses. Should any error happen in this call, we return a **non-nil** error; the gRPC layer will translate it into an appropriate RPC status to be sent on the wire.

4.3.4 Client-side Streaming RPC

Now let's look at another example: the **client-side streaming** method **RecordRoute**, where we get a stream of **StreamingInputCallRequest** from the client and return a single **StreamingInputCallResponse** with extra information.

As you can see, this time the method doesn't have a request parameter at all. Instead, it gets a **ExampleService_StreamingInputCallServer** stream, which the server can use to both read and write messages - it can receive client messages using its **Recv()** method and return its single response using its **SendAndClose()** method.

```
/* RPC that represent a sequence of requests and a single responses
 * The server returns the aggregated size of client payload as the result.
 */
func (exampleServer) StreamingInputCall (stream
example.ExampleService_StreamingInputCallServer) error {

    log.Log_info("\n\nExampleService_StreamingInputCallServer:")
```

```
    log.Log_dump_structure(stream)

    /* Starting/ Closing a Client Stream. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "StreamingInputCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "StreamingInputCall")
    size := 0
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&example.StreamingInputCallResponse{
                AggregatedPayloadSize: int32(size),
            })
        }

        size += len(req.User.Name)

        if err != nil {
            return err
        }
    }
}
```

In the method body we use the **ExampleService_StreamingInputCallServer**'s **Recv()** method to repeatedly read in our client's requests to a request object (in this case a **StreamingInputCallResponse**) until there are no more messages: the server needs to check the error returned from **Recv()** after each call. If this is **nil**, the stream is still good and it can continue reading; if it's **io.EOF** the message stream has ended and the server can return its response. If it has any other value, we return the error "as is" so that it'll be translated to an RPC status by the gRPC layer.

4.3.5 Bidirectional Streaming RPC

Finally, let's look at the bidirectional streaming RPC **FullDuplexCall()**. As one request could lead to multiple responses, this interface demonstrates the idea of **full duplexing**.

```
/* RPC that represent a sequence of requests and responses
 * with each request served by the server immediately.
 * As one request could lead to multiple responses, this interface
 * demonstrates the idea of full duplexing.
 */
func (exampleServer) FullDuplexCall (stream example.ExampleService_FullDuplexCallServer)
error {

    log.Log_info("\n\nExampleService_FullDuplexCallServer:")
    log.Log_dump_structure(stream)

    /* Starting/ Closing a Client/ Server Streams. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "FullDuplexCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
```

```
                                        "FullDuplexCall")
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return nil
        }

        if err != nil {
            return status.Error(codes.Internal, err.Error())
        }

        if req.ResponseStatus != nil && req.ResponseStatus.Code != int32(codes.OK) {
            return status.Error(codes.Code(req.ResponseStatus.Code), "error")

        }

        resp := &example.StreamingOutputCallResponse{User: &example.User{}}
        for _, param := range req.ResponseParameters {
            if stream.Context().Err() != nil {
                return stream.Context().Err()
            }

            buf := ""
            for i := 0; i < int(param.GetSize()); i++ {
                buf += req.GetUser().GetName()
            }

            resp.User.Name = buf

            if err := stream.Send(resp); err != nil {
                return err
            }
        }
    }
}
```

This time we get a **ExampleService_FullDuplexCallServer** stream that, as in our client-side streaming example, can be used to read and write messages. However, this time we return values via our method's stream while the client is still writing messages to their message stream.

The syntax for reading and writing here is very similar to our client-streaming method, except the server uses the stream's **Send()** method rather than **SendAndClose()** because it's writing multiple responses. Although each side will always get the other's messages in the order they were written, both the client and server can read and write in any order — the streams operate completely independently.

The following example is also bidirectional streaming and the RPC **HalfDuplexCall()**. However, now the server buffers all the client requests and then serves them in order. A stream of responses are returned to the client when the server starts with first request. This interface demonstrates the idea of **half duplexing**.

```
/* RPC that represent a sequence of requests and responses.
 * The server buffers all the client requests and then serves them in order.
 * A stream of responses are returned to the client when the server starts with
 * first request.
 */
func (exampleServer) HalfDuplexCall (stream example.ExampleService_HalfDuplexCallServer)
error {
```

```
    log.Log_info("\n\nExampleService_HalfDuplexCallServer:")
    log.Log_debug_dump(stream)

    /* Starting/ Closing a Client/ Server Streams. Update monitoring information. */
    netconfd_connect.Open_streams("example",
                                  "ExampleService",
                                  "HalfDuplexCall")
    defer netconfd_connect.Close_streams("example",
                                         "ExampleService",
                                         "HalfDuplexCall")

    requests := []*example.StreamingOutputCallRequest{}
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            break
        }

        if err != nil {
            return status.Error(codes.Internal, err.Error())
        }

        requests = append(requests, req)
    }

    for _, req := range requests {
        resp := &example.StreamingOutputCallResponse{User: &example.User{}}

        for _, param := range req.ResponseParameters {
            if stream.Context().Err() != nil {
                return stream.Context().Err()
            }

            buf := ""
            for i := 0; i < int(param.GetSize()); i++ {
                buf += req.GetUser().GetName()
            }

            resp.User.Name = buf

            if err := stream.Send(resp); err != nil {
                return err
            }
        }
    }

    return nil
}
```

4.3.6 Starting gRPC Server

Once we've implemented all our methods, we also need to start up a gRPC server so that clients can actually use our service. The following snippet shows how we do this for our RouteGuide service in the **ypgrpc-go-app** application:

```
    /* All initialiation is done
     * Start the gRPC server to listen for clients
```

```
 */
lis, err := net.Listen("tcp", addr)
if err != nil {
    log.Log_error("\nError: failed to Listen (%v)", err)
}

/* Start the gRPC server and Register all the gRPC Services */
grpcServer := grpc.NewServer()

/* Register All the Services here */
helloworld.RegisterGreeterServer(grpcServer, &helloworldServer{})
example.RegisterExampleServiceServer(grpcServer, &exampleServer{})

log.Log_info("\nypgrpc_server: Starting to serve")
if err := grpcServer.Serve(lis); err != nil {
    log.Log_error("\nError: failed to Serve (%v)", err)
}
```

gRPC server implementation consist of the following steps:

- Specify the port we want to use to listen for client requests using based on the –**port** CLI parameter (default is 50830)
- Create an instance of the gRPC server using **grpc.NewServer(...)**
- Register our service implementation with the gRPC server.
- Call **Serve()** on the server with our port details

5 gRPC State Monitoring

The following YANG tree diagram defines gRPC Monitoring Information that can be retrieved from the **netconfd-pro** server.

```
module: yumaworks-grpc-mon
  +--ro grpc-state
     +--ro statistics
     |  +--ro active-server-streams?   yang:zero-based-counter32
     |  +--ro active-client-streams?   yang:zero-based-counter32
     |  +--ro total-active-streams?    yang:zero-based-counter32
     |  +--ro total-closed-streams?    yang:zero-based-counter32
     +--ro server* [name]
     |  +--ro name                     string
     |  +--ro address                  inet:host
     |  +--ro port?                    inet:port-number
     |  +--ro start-time?              yang:date-and-time
     |  +--ro proto*                   string
     |  +--ro active-server-streams?   yang:zero-based-counter32
     |  +--ro active-client-streams?   yang:zero-based-counter32
     |  +--ro closed-streams?          yang:zero-based-counter32
     |  +--ro services
     |     +--ro service* [name]
     |        +--ro name      string
     |        +--ro method* [name]
     |           +--ro name                string
     |           +--ro client-streaming?   boolean
     |           +--ro server-streaming?   boolean
     +--ro server-streams!
     |  +--ro stream* [name]
     |     +--ro name             string
     |     +--ro creation-time?   yang:date-and-time
     |     +--ro location         inet:uri
     +--ro client-streams!
        +--ro stream* [name]
           +--ro name             string
           +--ro creation-time?   yang:date-and-time
           +--ro location         inet:uri

  rpcs:
    +---x grpc-shutdown
```

5.1 gRPC Monitoring Example

Run the server with the setting **--with-grpc=true** flag as follows:

```
mydir> sudo netconfd-pro --log-level=debug4 --with-grpc=true \
       -- fileloc-fhs=true
```

Start the **ypgrpc-go-app** application in “insecure” mode for this scenario:

```
mydir> ypgrpc-go-app --log-level=debug --fileloc-fhs --insecure \
       --protopath=$HOME/protos \
       --proto=helloworld --proto=example
```

Now, when the **ypgrpc-go-app** application is running and serving the gRPC server there is an option to check the gRPC server capabilities by sending <get> request to the **netconfd-pro** server as follows, for this example we will use **yangcli-pro** to send the request:

```
yangcli-pro> sget /grpc-state
```

OR, as an another example, RESTCONF request:

```
curl http://restconf-dev/restconf/data/grpc-state \
       -H "Accept:application/yang-data+json"
```

The request above may produce with the following (JSON output in this case):

```
{
    "yumaworks-grpc-mon:grpc-state": {
        "statistics": {
            "active-server-streams": 0,
            "active-client-streams": 0,
            "total-active-streams": 0,
            "total-closed-streams": 0
        },
        "server": [
            {
                "name": "example-grpc",
                "address": "192.168.0.216",
                "port": 50830,
                "start-time": "2021-10-21T00:27:00Z",
                "proto": [
                    "example"
                ],
                "active-server-streams": 0,
                "active-client-streams": 0,
                "closed-streams": 0,
                "services": {
                    "service": [
                        {
                            "name": "example.ExampleService",
                            "method": [
                                {
                                    "name": "EmptyCall",
                                    "client-streaming": false,
                                    "server-streaming": false
```

```
                                },
                                {
                                    "name": "FullDuplexCall",
                                    "client-streaming": true,
                                    "server-streaming": true
                                },
                                {
                                    "name": "HalfDuplexCall",
                                    "client-streaming": true,
                                    "server-streaming": true
                                },
                                {
                                    "name": "StreamingInputCall",
                                    "client-streaming": true,
                                    "server-streaming": false
                                },
                                {
                                    "name": "StreamingOutputCall",
                                    "client-streaming": false,
                                    "server-streaming": true
                                },
                                {
                                    "name": "UnaryCall",
                                    "client-streaming": false,
                                    "server-streaming": false
                                }
                            ]
                        }
                    ]
                }
            }
        ]
    }
}
```

5.2 <grpc-shutdown> Operation

The <grpc-shutdown> operation is used to shut down the **ypgrpc-go-app** application.

By default, only the 'superuser' account is allowed to invoke this operation.

If permission is granted, then the **netconfd-pro** server will send request to shutdown the **ypgrpc-go-app** application and its gRPC server. All gRPC streams will be dropped, during the **ypgrpc-go-app** application shutdown.

<grpc-shutdown> operation

Min parameters:	0
Max parameters:	0
Return type:	none
YANG file:	yumaworks-grpc-mon.yang
Capabilities needed:	none

Mandatory Parameters:

- none

Optional Parameters:

- none

Returns:

- none; **ypgrpc-go-app** will be shutdown upon success

Possible Operation Errors:

- access denied

Example Request:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc message-id="2"
  xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <grpc-shutdown xmlns="http://yumaworks.com/ns/yumaworks-grpc-mon"/>
</rpc>
```

Example Reply:

```
[no reply will be sent; ypgrpc-go-app will be shutdown instead.]
```

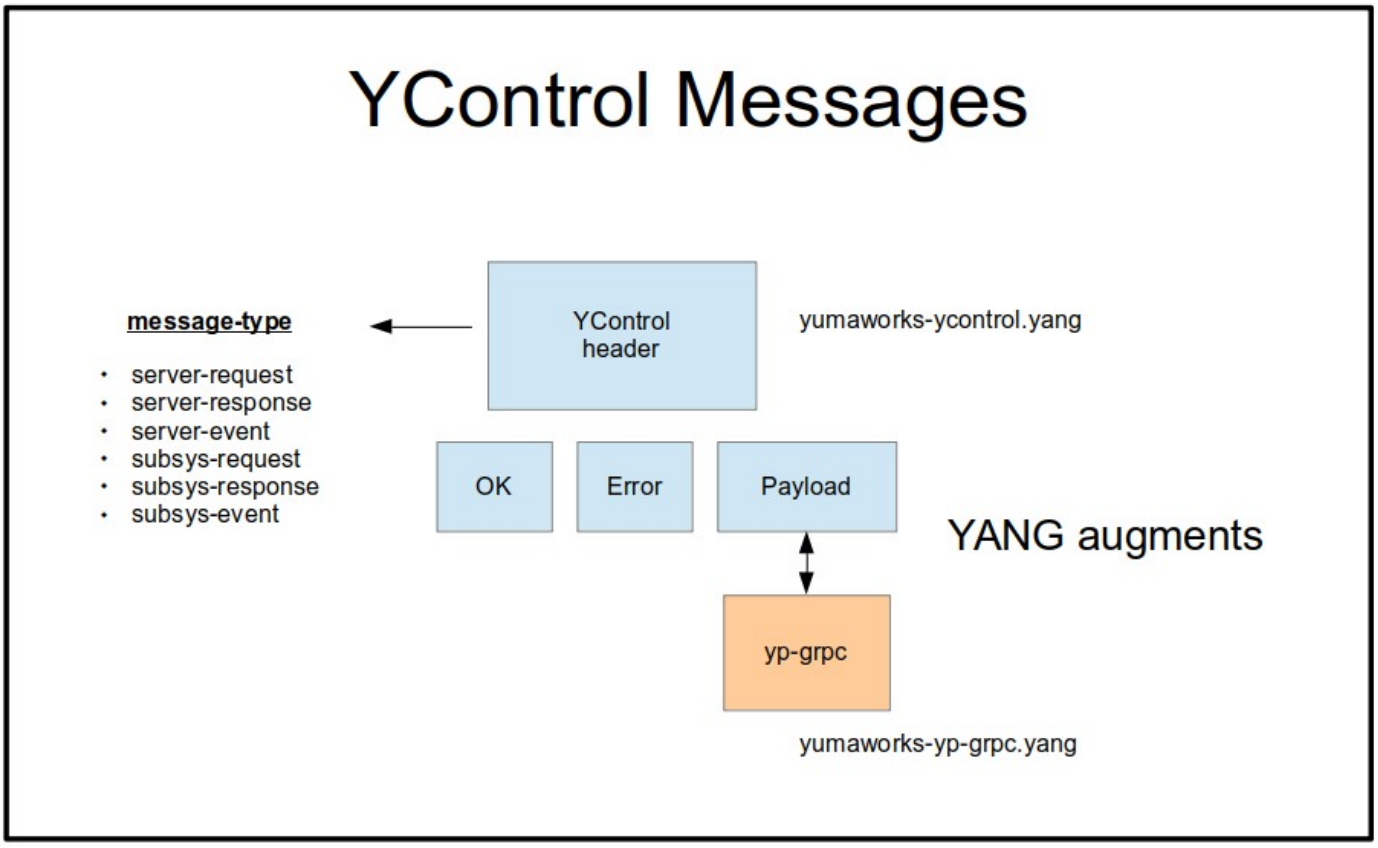
6 YP-gRPC Subsystem Messages

The **ypgrpc-go-app** application uses several messages to interact with the netconfd-pro server.

6.1 Message Format

These messages are defined in the **yumaworks-yp-grpc** YANG module. The **ypgrpc-go-app** payload is defined as a YANG container that augments the YControl “message-payload” container.

The following diagram illustrates the YControl messages overview:



6.1.1 YControl Integration

The **ypgrpc-go-app** YControl subsystem service and the **netconfd-pro** have the following messages interaction:

- **capability-ad-event**: **ypgrpc-go-app** sends subsystem event to advertise all the gRPC available and active capabilities during the registration time
- **open-stream-event**: **ypgrpc-go-app** sends subsystem event to advertise a new gRPC server or client stream(s)
- **close-stream-event**: **ypgrpc-go-app** sends subsystem event to advertise that the gRPC server or client stream was closed.

6.1.2 Registration Message Flow

During the startup phase the server will initialize the **yp-grpc** subsystem callback functions and handlers (similar way as for **db-api** module does).

The connection with the server is getting started with <ncx-connect> message that adds the YControl subsystem with the “example-grpc” subsystem ID to the server (**agt_connect** module).

YControl protocol connection parameters:

- transport: **netconf-aflocal**
- protocol: **yp-grpc**
- <port-num> not used

Additional parameters:

- subsys-id: **example-grpc**

The Registration message flow looks as follows:

- **Ypgrpc-go-app to Server:** *register-request:*
  The **yp-grpc** service registers callbacks supported by the subsystem.
- **Server to ypgrpc-go-app:** *ok:*
  The server responds to the register request with an <ok> or an <error> message
- **Ypgrpc-go-app to Server:** *capability-ad-event:*
  Sends subsystem event to advertise all the gRPC available and active capabilities during the registration time

The module “yumaworks-yp-grpc.yang” defines all the messages

6.1.3 Yumaworks-yp-grpc YANG Module

The YP-gRPC subsystem service messages are defined in the **yumaworks-yp-grpc.yang** module.

```
module yumaworks-yp-grpc {
  yang-version 1.1;
  namespace "http://yumaworks.com/ns/yumaworks-yp-grpc";
  prefix "ypgrpc";

  import yumaworks-ycontrol { prefix yctl; }
  import ietf-yang-types { prefix yang; }
  import ietf-inet-types { prefix inet; }

  organization "YumaWorks, Inc.";

  contact
    "Support <support@yumaworks.com>";

  description
    "YumaPro gRPC Application message definitions.

     Copyright (c) 2014 - 2021 YumaWorks, Inc. All rights reserved.

     Redistribution and use in source and binary forms, with or
     without modification, is permitted pursuant to, and subject
     to the license terms contained in, the BSD 3-Clause License
     http://opensource.org/licenses/BSD-3-Clause";

  revision 2021-08-23 {
    description
      "Initial version";
  }


  augment "/yctl:ycontrol/yctl:message-payload/yctl:payload/yctl:payload" {
    container yp-grpc {

      choice message-type {
        mandatory true;

        case register-request-case {
          leaf register-request {
            type empty;
            description
              "Message type: subsys-request;
               Purpose: register the YP-gRPC subsystem
               Expected Response Message: ok or error";
          }
        }

        container capability-ad-event {
          description
            "Subsystem event to advertise all the gRPC
             available and active capabilities during
             the registration time.
             Type: subsys-event
             Expected Response Message: none";
```

```
        leaf name {
          mandatory true;
          type yang:yang-identifier;
          description
            "Name of the gRCP server.";
        }

        leaf address {
          type inet:host;
          mandatory true;
          description
            "IP Address or host name for the gRPC server.
             The value returned is implementation specific
             (e.g., hostname, IPv4 address, IPv6 address)";
        }

        leaf port {
          type inet:port-number;
          description
            "TCP port number for the gRPC server.
             If not present then the default port for
             the protocol will be used.";
        }

        leaf-list proto {
          type string;
          description
            "The list of proto files that gRPC server supports.";
        }

        list service {
          key name;
          description
            "List of gRPC Services supported by the gRPC server and
             related information.";

          leaf name {
            type string;
            description
              "Name of the gRCP Service associated with this list entry.";
          }

          list method {
            key name;
            description
              "The list of methods supported by the gRPC server and
               related information.";

            leaf name {
              type string;
              description
                "Name of the Service Method associated with this list entry.";
            }
            leaf client-streaming {
              type boolean;
              description
                "True if the client supports streaming for this method
                 FALSE otherwise.";
            }
            leaf server-streaming {
              type boolean;
              description
                "True if the server supports streaming for this method.
```

```
                FALSE otherwise.";
          }
        }
      }
    }

    container open-stream-event {
      description
        "Subsystem event to advertise a new gRPC
         server or client stream(s).
         Type: subsys-event
         Expected Response Message: none";

      list server-stream {
        key name;
        description
          "Contains a list of open server gRPC streams.";

        leaf name {
          type string;
          description
            "Name of a gRCP server stream.";
        }

        leaf location {
          type inet:uri;
          mandatory true;
          description
            "Contains a URL that represents the RPC that uses
             this client stream.";
        }
      }

      list client-stream {
        key name;
        description
          "Contains a list of open client gRPC streams.";

        leaf name {
          type string;
          description
            "Name of a gRCP client stream.";
        }

        leaf location {
          type inet:uri;
          mandatory true;
          description
            "Contains a URL that represents the RPC that uses
             this server stream.";
        }
      }
    }

    container close-stream-event {
      description
        "Subsystem event to advertise that the gRPC server or
         client stream was closed.
         Type: subsys-event
         Expected Response Message: none";

      leaf-list server-stream {
        type string;
```

```
            description
              "The list of closed server gRPC streams.";
          }

          leaf-list client-stream {
            type string;
            description
              "The list of closed client gRPC streams.";
          }
        }
      }
    }
  }
}
```

7 CLI Reference

The **ypgrpc-go-app** program uses command line interface (CLI) parameters to control program behavior.

The following sections document all the CLI parameters relevant to this program, in alphabetical order.

7.1 --ca

The **--ca** parameter specifies the gRPC server CA certificate file. The path to the CA certificate should be absolute.

--ca parameter

Syntax	**filespec**
Default:	none
Min Allowed:	1
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --ca=/tmp/ca.crt

7.2 --cert

The **--cert** parameter specifies the gRPC server certificate file. The path to the certificate should be absolute.

--cert parameter

Syntax	**filespec**
Default:	none
Min Allowed:	1
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --cert=/tmp/cert.crt

7.3 --fileloc-fhs

The **--fileloc-fhs** parameter specifies whether the **ypgrpc-go-app** should use Filesystem Hierarchy Standard (FHS) directory locations to create, store and use data and files. May need to run as root. If false then the server will use $HOME/.yumapro and other file locations to store application data and to access the **netconfd-pro** files.

--fileloc-fhs parameter

Syntax	**boolean**
Default:	false
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --fileloc-fhs=true

7.4 --insecure

The **--insecure** parameter directs to skip TLS validation and allowing to run the program with out **--ca** and **--cert** parameters.

--insecure parameter

Syntax	**boolean**
Default:	false
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --insecure=true

7.5 --key

The **--key** parameter specifies the gRPC server private key file. The path to the key should be absolute.

--key parameter

Syntax	**filespec**
Default:	none
Min Allowed:	1
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --key=/tmp/cert.crt

7.6 --log

The **--log** parameter specifies the file path of the application log. Filespec for the log file to use instead of STDOUT. Leave out to use STDOUT for log messages.

--log parameter

Syntax	**filespec**
Default:	none
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --log=/var/log/app-log.log

7.7 --log-console

The --l**og-console** parameter directs that log output will be be sent to STDOUT, after being sent to the log file and/or local syslog daemon. (This assumes that **--log** parameter is present).

--log-console parameter

Syntax	**boolean**
Default:	none
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --log-console=true

7.8 --log-level

The **--log-level** parameter controls the verbosity level of messages printed to the log file or STDOUT, if no log file is specified.

The log levels are incremental, meaning that each higher level includes everything from the previous level, plus additional messages.

There are 4 settings that can be used:

- **error**: Error messages are printed, indicating problems that require attention.
- **warning**: Warning messages are printed, indicating problems that may require attention.
- **info**: Informational messages are printed, that indicate program status changes.
- **debug**: Debugging messages are printed that indicate program activity.

--log-level parameter

Syntax	**enumeration**: **error** **warning** **info** **debug**
Default:	info
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --log=/var/log/app-log.log \ --log-level=debug

7.9 --port

The **--port** parameter specifies the gRPC server binding. This is the address that the gRPC client will use top contact the gRPC server. By default the address is the local host and default port is 50830.

--port parameter

Syntax	**inet:port-number**
Default:	:50830
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --port=50051

7.10 --proto

The **--proto** parameter specifies the .proto files for the **ypgrpc-go-app** application to use. There can be multiple .proto files specified.

--proto parameter

Syntax	**string**
Default:	none
Min Allowed:	0
Max Allowed:	N
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --proto=example \ --proto=helloworld

7.11 --protopath

The **--protopath** parameter specifies the .proto file search path to use while searching for .proto files.

-- protopath parameter

Syntax	**string**
Default:	$HOME/protos
Min Allowed:	0
Max Allowed:	N
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --protopath=/tmp/proto \ --protopath=$HOME/protos

7.12 --server-address

The --**server-address** parameter specifies the **netconfd-pro** server IP address. The default is '127.0.0.1' if no value is specified.

--server-address parameter

Syntax	**inet:ip-address**
Default:	127.0.0.1
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --server-address=10.10.0.11

7.13 --subsys-id

The --**subsys-id** parameter specifies the subsystem identifier to use when registering with the **netconfd-pro** server. The default is 'example-grpc' if no value is specified.

--subsys-id parameter

Syntax	**string**
Default:	example-grpc
Min Allowed:	0
Max Allowed:	1
Supported by:	ypgrpc-go-app
Example:	ypgrpc-go-app --subsys-id=subsys1