QORTEX™ DTC 2.4 for Q-Track API Reference

Q-Track

QUANERGY
See beyond.

Notices

Contact

Quanergy Solutions, Inc.
128 Baytech Drive
San Jose, CA 95134
https://quanergy.com/

- For purchases made directly from Quanergy: contact sales@quanergy.com
- For purchases from a third party such as value-added reseller/system integrator: contact them for support

Follow Us!

https://www.linkedin.com/company/quanergy/

https://twitter.com/quanergy/

https://www.facebook.com/quanergy/

https://www.youtube.com/@QuanergySolutions/

Revision History

Version	Date	What Changed	Change Location
A	09/15/24	Initial release. Split the QORTEX DTC API out from the QORTEX DTC User Guide.	

Text Styles

In this guide certain fonts are applied to provide a visual means to interpret text. See *Table 1. Text Styles and Meanings*.

Table 1. Text Styles and Meanings

Font Style	Meaning
Blue underline	Hyperlink that opens a file outside of this document. Typically, this is a web page.
Italic underline	Hyperlink that moves you to a location within the document. Typically, this is a section, table, or figure.
Bold	General term for emphasis. Typically, this introduces a definition or description.
Bold	An item in a Graphical User Interface (GUI). This includes page, window, or panel labels, fields where you enter or select information, or checkboxes and buttons you click.
Bold colors	Examples of colors used in GUIs.
Italic	Identifies book or section titles. Provides emphasis for terms or ideas. Also identifies options for selecting from a menu, entering in a field, or replacing in a command string.
`Code italic`	Identifies a variable, where the intent is you provide a literal value in place of the variable. Used in paragraphs and code strings.
`Code`	Examples of commands you enter in a field or terminal, or responses from the system.
`Code on gray`	Examples of commands you enter into a command line interface (CLI) terminal or responses from the system.
Text in box	Notes and Tips. Useful related information. Format used to call attention to the content or to describe side-bar content.
Text in box	Cautions and Warnings. Read these and comply with the information provided. Comply with Cautions or Warnings to prevent equipment damage or injury to humans and other living things.

Contents

Figures

Tables

1. Getting Started

QORTEX DTC™ 2.4 for Q-Track is the software portion of a LiDAR-based solution that provides three-dimensional perception and volumetric sensing. This system detects, tracks, and classifies (DTC) person and vehicle objects to:

- Accurately locates objects to be tracked uniquely in real time.
- Customize Event zones with flexible commands.

With data intelligence and centimeter-level accuracy, QORTEX DTC provides a cost-effective solution enabling Internet of Things (IoT) applications in different sectors, such as security and smart cities and spaces with interests in monitoring traffic and crowd flow and perimeter intrusion.

QORTEX DTC has both a browser-based graphical user interface (GUI) and an application programmer interface (API). This guide documents the QORTEX DTC API.

QORTEX DTC 2.4 Improvements

QORTEX DTC 2.4 added and improved the following features:

- Support for Q-Track sensors: Q-Track LR, Q-Track HD, Q-Track Dome

Unique Features

Q-Track is a solution that includes both Quanergy LiDAR Q-Track sensors and QORTEX DTC software.

This solution accomplishes its demanding goals through the following features:

- **Extended Detection Range in Real-Time.** Q-Track sensors can continuously track static and moving objects accurately and in real time within a 20-meter, 40-meter, or 70-meter radius depending on the Q-Track sensor selected.
- **Classification of Shape Data.** The sensor produces surveillance data the host computer collects, records, visualizes, analyzes, classifies, and outputs to an Object List. This list provides a basis for further action outside of QORTEX DTC, such as the notification of external alarm systems via LAN, TCP, UDP, and HTTP GET.
- **Persistence in Tracking.** The commercial-grade, static LiDAR visualization system senses and displays the movement of objects over time, persisting even through blockages and crowd gaps. This gives users the ability to track and record the historical movements of potential threats.
- **Resilience in Suboptimal Conditions.** The sensor is designed for use indoors and outdoors and in bright or dark conditions, with no infrared signature needed. It also

withstands extreme weather, from bitter cold to baking sun, with complete ingress protection from mist, rain, snow, and dust.

- **Fault Tolerance, with Automatic Reconnect.** Designed for persistence, the QORTEX DTC server recovers persistently, since it runs as a daemon service. If a sensor disconnects from a server, the server sends a notification of the event, continues to run with the remaining connected sensors, and auto-reconnects with the sensor without reboot. The QORTEX DTC client also automatically resumes in Monitor mode after restart, with 3D visualization restored.
- **Configurable.** The Q-Track LR sensor can be configured through the web server to adjust the frame rate from 5 to 20 hertz, with single or multiple (up to 3) returns. We strongly recommend that you set the sensor to single return mode when using it with QORTEX DTC. The field of view is adjustable when less than a full 360° and is preferred.

  Configure sensors using the Sensor Settings Management application (webserver) built into the sensor. See *Managing the Sensor* in your sensor-specific user guide. Request the sensor user guide from support@quanergy.com.

  The Q-Track HD and Q-Track Dome sensors do not have a web server and do not support multiple returns. Use Q-View to configure them.
- **Leverage Q-View Calibration.** Q-View™ is the Quanergy sensor discovery and management toolkit, which calibrates multiple sensors into the same space and outputs the resulting `transform_alignment.xml` file. QORTEX DTC takes the output file and aligns the overlapping vision of multiple sensors into an enriched Multi-LiDAR Fusion view.
- **Enhances Legacy Systems.** QORTEX DTC reduces or eliminates false positive threats and dramatically boosts the surveillance effectiveness of the Video Management Systems (VMS) by measuring and providing exact 3D coordinates of people.

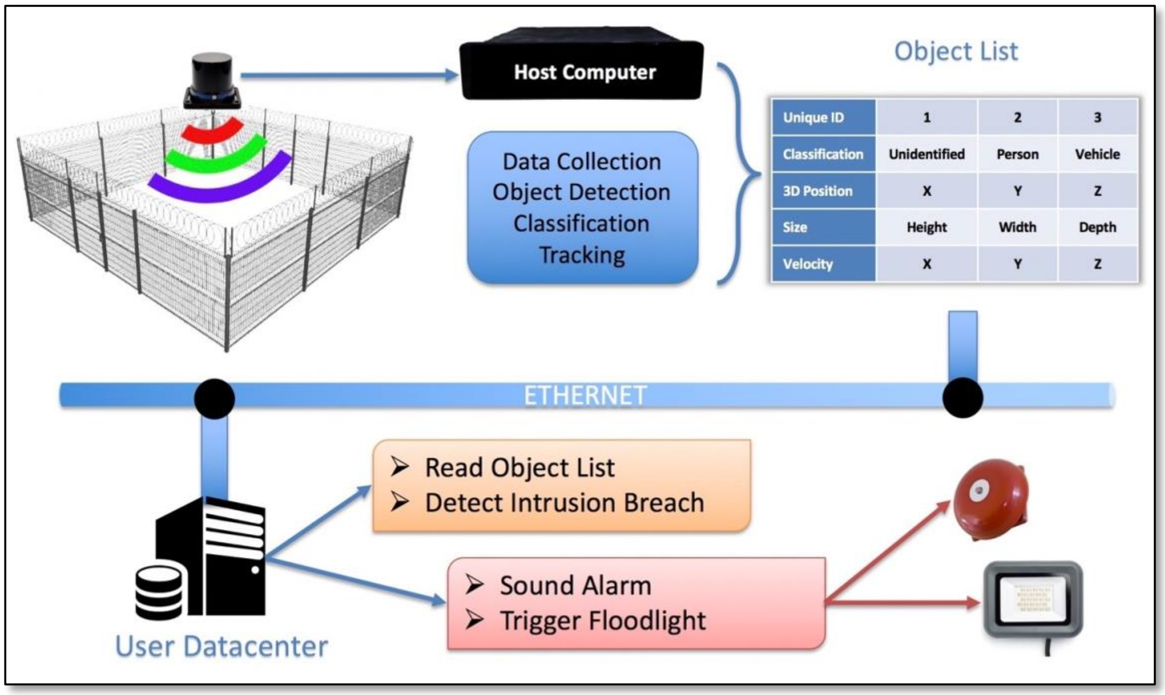
System Architecture

The QORTEX DTC for Q-Track system architecture deploys in a simple distributed scenario that includes the QORTEX DTC server and a QORTEX DTC client or third-party application to consume the server output.

- The server software interfaces with statically installed LiDAR sensors and produces output to your own network infrastructure, including a real-time list of tracked objects accessed through the QORTEX API. See QORTEX API (page 14). Configuration and control of the server software are achieved through the client interface and/or the gRPC API. See *API for Remote Procedure Calls* (page 23). The gRPC API can be used to develop a customized client-user interface.
- The client software enables control and creation of a location (area of interest), with visualization of the corresponding server output point cloud, detected objects, tracks, classification, and zones. Any number of potential third-party host infrastructure

applications may subscribe to published data stream outputs for surveillance or visualization. See *QORTEX API* (page 14).

The QORTEX DTC solution blocks of functionality are shown in *Figure 1. QORTEX DTC System Architecture Overview*. LiDAR sensors that integrate into an existing deployment might be able to leverage that existing power and communication infrastructure, if compatible.



Unique ID	1	2	3
Classification	Unidentified	Person	Vehicle
3D Position	X	Y	Z
Size	Height	Width	Depth
Velocity	X	Y	Z

Figure 1. QORTEX DTC System Architecture Overview

System Components

To use Quanergy QORTEX DTC for monitoring objects in a location requires:

- Physical site installation of the sensors. See the sensor User Guides.
- Computers meeting defined requirements for software installation of QORTEX DTC server, and QORTEX DTC client. See *Qortex Q-Track DTC User Guide*.
- Software configuration:
  - **License Manager**. See *QORTEX Q-Track DTC User Guide*.
  - **Sensor Settings Management** application (web server). See the sensor user guides. Request the sensor user guide from support@quanergy.com.
  - **Q-View**. See the *Q-View User Guide*. Download from https://downloads.quanergy.com/.
  - **QORTEX DTC Server.** See the *QORTEX Q-Track DTC User Guide*.
  - **QORTEX DTC Client.** See the *QORTEX Q-Track DTC User Guide*.
- All the sensors, PTZ cameras, and QORTEX DTC running on a single Ethernet network.

Supported Quanergy Sensors

- Q-Track sensor family: Q-Track LR, Q-Track HD, Q-Track Dome

Object and Capacity Specifications

Table 2. QORTEX DTC Server Specifications lists the scope and range specifications for QORTEX server.

Table 2. QORTEX DTC Server Specifications

Parameter	Specification
Object Information	Provides an object list with 3D direction and position, velocity, and classification in Protobuf, JSON, NDJSON, or XML formats
Classification Types	Human, Vehicle, Unknown Additional SubVehicle licensing adds: TWOWHEELER_VEHICLE, PASSENGER_VEHICLE, COMMERCIAL_VEHICLE
Continuous Tracking Range	Q-Track LR sensors: 140 meters (70-meter radius) Q-Track HD sensors: 80 meters (40-meter radius) Q-Track Dome sensors: 40 meters (20-meter radius)
Number of Simultaneous Objects	Depends on the QORTEX DTC server and sensors.
Video Management System (VMS) Compatibility	Most major VMSs such as Milestone XProtect®, and Genetec™ Security Center, Hanwha Wave, Network Optix, Mirasys, Flir Latitude, Salient, Digifort, Surveill, and many more.
Cameras	PTZ: no hard limit, up to 5 cameras tested

Documentation

Table 3. Related Documents lists the latest versions of the relevant user documents and how to access them.

Table 3. Related Documents

Topic area	Guide	Access
QORTEX-specific guides	QORTEX *DTC 2.4 User Guide* QORTEX *DTC 2.4 Quick Start Card* QORTEX *Q-Track DTC User Guide*	Download from https://downloads.quanergy.com/
Sensor management guides	*Q-View User Guide* *Q-View Quick Start Card*	Download from https://downloads.quanergy.com/

Topic area	Guide	Access
Sensor user guides for each LiDAR sensor model	*User Guide* *Quick Start Card*	Request from support@quanergy.com
Sensor datasheets	*Datasheets*	Download documents from https://quanergy.com/downloads/

2. QORTEX API

The QORTEX DTC 2.4 server publishes the object/trackable list, zone list, and state list as streams of data output from its processing action in a serialized format on the Ethernet network.

The QORTEX DTC 2.4 client consumes and visualizes these streams of published data, but any number of potential third-party host infrastructure applications can subscribe to them for surveillance or visualization purposes.

Allowing TCP ports

> **Note:** TCP port 17177 is reserved for gRPC communication between client and server. Other port numbers (17161, 17168, 17175, 17176) can be defined in the `settings.xml` file. See *QORTEX Q-Track DTC User Guide.*
>
> **Note:** If security is enabled, user authentication is required to enable connection between the QORTEX DTC server and QORTEX DTC client. See *Cybersecurity* (page 92).

Outputs to Consume

Adjust the `settings.xml` file `<Publisher>` sections to determine which outputs are published, and in what format. See *Figure 2. Configuration of Generated settings.xml File Publisher Parameters*. The output data is available in three different formats:

- **Protobuf** — delivers in binary packages; recommended for most efficient use of bandwidth.
- **JSON** — delivers in human-readable serial format.
- **XML** — delivers in human-readable serial format.
- **NDJSON** — delivers in human-readable serial format.

If the output format is changed, restart the server. See *QORTEX Q-Track DTC User Guide.*

```
<TCPPublishers>
<!-- TCPPublishers section lists all publishing APIs, each publisher section has minimum:
Fixed Name, this is a string referenced in code to identify specific API, it could be:
        QORTEX_ZONE_LIST, . . .
Publishing Format 'json', 'xml', 'protoful' or 'none': Optional parameter. Default is 'none'
Publishing TCP Port: Optional parameter. Default is hard-coded value
-->
<Publisher>
    <Name>SENSOR_HEALTH_STATE</Name>
    <Format>none</Format>
    <Port>17168</Port>
    <AddDataSize>false</AddDataSize>
    <NetworkByteOrder>false</NetworkByteOrder>
</Publisher>
<!-- Data intended for QORTEX DTC Client -->
<Publisher>
    <Name>CONFIGURATION_CLIENT_DATA</Name>
    <Port>17175</Port>
</Publisher>
<Publisher>
    <Name>MONITOR_CLIENT_DATA </Name>
    <Port>17176</Port>
</Publisher>
</TCPPublishers>
```

Figure 2. Configuration of Generated `settings.xml` File Publisher Parameters

Object Lists

QORTEX DTC 2.4 outputs objects in a Trackable List and has the following characteristics:

Purpose: An existing system may subscribe to this data stream to gain detailed information and long-range movements of people and/or vehicles in the area.

Format: Default is none, which disables publication. To enable publication, specify protobuf (preferred), json, or xml.

Frequency: The `<frequency>` parameter in the `<TrackerPipeline>` section of the `settings.xml` file determines the maximum rate, in Hz, at which the object list is published.

> **For a single LiDAR area**: The output object list rate cannot exceed the rate at which the LiDAR is spinning, but the `<frequency>` parameter can reduce the rate at which the object list is published.
>
> **For a multi-LiDAR area**: The multi-LiDAR pipeline performs several mergers and calculations before publishing the result. If this time-intensive step cannot be done at the rate described by the `<frequency>` parameter, a log message is published to state that the multi-LiDAR pipeline cannot keep up.

> **Note:** For the multi-LiDAR pipeline, the rate at which the object list is published does not necessarily match the timestamps of objects in those messages.

Trackable List (Port 17161)

The Trackable List is compatible with QORTEX DTC 2.0. See *Figure 3. Protobuf Output Format for QORTEX DTC 2.x Trackable List.* This trackable list is distinguished by these characteristics.

Classification:

- `PERSON [mobile, movable], (trackableClassFlag 322)`
- `VEHICLE [mobile, movable], (trackableClassFlag 194)`
- `UNKNOWN, [0]`

The following additional classification options require `SubVehicle` license.

- `TWOWHEELER [mobile, movable], (trackableClassFlag 1218)`
- `PASSENGER_VEHICLE [mobile, movable], (trackableClassFlag 2242)`
- `COMMERCIAL_VEHICLE [mobile, movable], (trackableClassFlag 4290)`

Location: On TCP port 17161

Publishes: Compound data including timestamp, state, and decorators for each detected merged trackable, including:

- Unique track ID (64-bit length).
- Timestamp epoch time in milliseconds.
- Classification (person [mobile, movable], vehicle [mobile, movable], twowheeler [vehicle, mobile, movable], passenger [vehicle, mobile, movable], commercial [vehicle, mobile, movable], unknown)
- 3D position (X, Y, Z) in meters.
- Size in each axis (height, width, depth) in meters.
- Velocity (X, Y, Z) in meters per second.
- Centroid (X, Y, Z geometric center of the 3D shape) in meters.
- Acceleration (X, Y, Z) in meters per second squared.
- Float heading within a range of –π to π in radians.
- Float heading rate: rate of change of the heading, applicable only for vehicles, otherwise value is 0.

```
Message QTrackableArray {
    uint64 timestamp = 1;       //Epoch time in milliseconds
    Repeated QTrackable trackable = 2;
}
// Trackable Classification (PERSON, TWOWHEELER_VEHICLE, PASSENGER_VEHICLE, COMMERCIAL_VEHICLE, VEHICLE and UNKNOWN is available and PERSON, TWOWHEELER_VEHICLE, PASSENGER_VEHICLE, COMMERCIAL_VEHICLE, VEHICLE are classified as MOBILE and MOVABLE as well)
// Note that TWOWHEELER_VEHICLE, PASSENGER_VEHICLE, COMMERCIAL_VEHICLE are only populated when a valid subclassification license add-on is present
Message TrackableClass {
    enum TrackableClassFlags {
        UNKNOWN             0x00;
        IMMOVABLE           0x01;
        MOVABLE             0x02;
        GROUND              0x03;
```

```
       WALL                   0x04;
       CLUTTER                0x10;
       STATIONARY             0x20;
       MOBILE                 0x40;
       VEHICLE                0x80;
       PERSON                 0x100;
       BICYCLE                0x200;
       TWOWHEELER_VEHICLE     0x400; //Requires SubVehicle license
       PASSENGER_VEHICLE      0x800; //Requires SubVehicle license
       COMMERCIAL_VEHICLE     0x1000;//Requires SubVehicle license
    }
    //Bitwise-OR of TrackableClassFlags.
    uint32 trackableClassFlags = 1; // Bit-wise or of the TrackableClassFlags
}
Message QTrackable {
    uint64 id = 1;                     // Unique ID
    uint64 timestamp = 2;              // Epoch time in milliseconds
    TrackableClass trackable_class = 3;
    Vector3 position = 4;       // Position (x, y, z) unit: meter
    Vector3 size = 5;           // Size in each axis (width, depth, height)
    Vector3 velocity = 6;       // Velocity (x, y, z) unit: meter/second
    Vector3 centroid = 7;       // unit: meter
    Vector3 acceleration = 8;   // Acceleration (x, y, z) unit: meter/second
    float heading = 9;          // unit: radians
                                // range -u to u
    float heading_rate = 10;    // Rate of change of the heading
                                // This is applicable only for vehicles, otherwise value is 0
}
```

Figure 3. Protobuf Output Format for QORTEX DTC 2.x Trackable List

Object Movement Filter parameters

Object Movement Filter parameters are in the `settings.xml` file.

```
<OutputFilter>
<useMovementThresholdFilter>false</useMovementThresholdFilter>
<!-- Enable or disable movement threshold filter. This can be used to filter out
occasionally moving static objects (e.g. grass, foliage) which cause false objects.
-->
<minMovementThreshold>0.3</minMovementThreshold>
<!-- Trackables whose movement is less than this threshold (in meters) are filtered
out.
Movement is measured by the distance of trackable's current position from its
initial, i.e., first detected position. -->
</OutputFilter>
```

Zone List

Event zones are created and configured. See *QORTEX Q-Track DTC User Guide*. See *Figure 4. Protobuf Output Format for QORTEX DTC Zone List*. It has these characteristics:

Available: On TCP port 17172.

Level 0: Publishes zones with a list of the objects inside.

Level 1: Publish zones based on changing state of trackables inside [The Q-Track violation zone has a "Tigger state" field which specify one of the current state for trackables: "Enter", "Exit", "Appear", "Disappear"]

Level 2: Publish zones based on changing state of trackables and event configuration of zone. The format of this level is same as Level 1

Publishes at a regular frequency, typically 10 Hz:

- Unique IDs (match the object/trackable list IDs) of trackables inside a zone. Unique IDs are 128 bit length.
- Timestamp.
- String name.
- Polygon 2D shape and coordinates.
- Specific data to detect if an object track enters or exits a defined zone.
- Count the objects entered by type. This includes sub-classifications when licensed.
- Optionally defined data related to alerting the host environment.

Purpose: An existing system may wish to subscribe to this data stream to gain detailed information on people breaching pre-defined Event zones, including their long-range movements, and to determine the handling of potential threats, (use a third-party system to turn on a visual PTZ camera or alarm, for example).

Format: Default is none, which disables publication. To enable publication, the user specifies `protobuf` (preferred), `json`, or `xml`.

Frequency: Published per frame to maintain the heartbeat for the server to sensor connection. The zone list is repeated in each message as long as the zone includes a trackable.

```
message QZoneArray {
    Header header = 1;
    repeated QZone zones = 2;
)
message Header {
    uint64 timestamp = 1;  //Epoch time in milliseconds
    string frame_id = 2;
    uint32 sequence = 3;   // A counter that increases with each publish,
                          // regardless if any listener is listening
}
message QZone (
    string uuid = 1;            // string representation of a 128-bit Universally Unique ID
    uint64 timestamp = 2;
    string name = 3;
    Polygon2D shape = 4;        // supported for polygon/rectangle zones only
    uint32 object_count = 7;
    repeated int64 object_ids = 5;
    double z_min = 8;
    double z_max = 9;

// Zone Classification
    enum ZoneClass
    {
        FENCE = 0;
        EXCLUSION_ZONE = 1;
    }
    ZoneClass zone_class = 6;   // object list only for FENCE

    repeated ObjectCountByType object_counts = 10;
}

message ObjectCountByType {
  string object_class = 1;
  uint32 object_count = 2;
  repeated int64 object_ids = 3;
}
```

Figure 4. Protobuf Output Format for QORTEX DTC Zone List

State List

The State List is distinguished by these characteristics:

Purpose: Administrators may subscribe to this data stream to monitor and resolve a Missing sensor issue quickly, or to check a Connected sensor health.

Publishes: During data collection or recording, reports a specific sensor current:

- **state of connection:**
    - **Connected**. The sensor is connected and sending point cloud information to the QORTEX DTC server. If the sensor loses its connection, it is automatically re-connected to QORTEX DTC.
    - **Missing**. The sensor point cloud is missing, which could indicate a sensor malfunction or other network congestion issue.
- **state of health:**

For the Q-Track sensor, health status is available only for Rev D5 and up, or Rev D4 with a patch updating Base firmware to 7.03.

- Temperature
- Frame rate
- Triggered error states

Refer to the *Troubleshooting Issues* section in the specific sensor User Guide, which is available upon request from support@quanergy.com.

Format: User selects Protobuf (preferred), JSON, or XML.

QORTEX DTC 2.x State List is available on TCP port 17168. See *Figure 5. Protobuf Output Format for QORTEX DTC 2.x State List*. Sensor Health outputs information at 0.2 to 0.05 Hz, or every 5 to 20 seconds. Generally, the more sensors are polled, the slower the publishing rate becomes. It is useful to maintain different frameIDs to identify which SNMP data reflects which sensor update. The output uses the frameID in the settings file, which may be different for each sensor. Whenever any sensor SNMP data is ready, all sensors' status is output, along with the particular sensor frameID (whose SNMP data is ready) under the **frameID** field.

```
message SensorState {
   enum ConnectionStatus {
      CONNECTED = 0;
      DISCONNECTED = 1;
}
   enum SensorMaskingStatus {
    UNKNOWN = 0;
    NOT_DETECTED = 1;
    DETECTED = 2;
}
   enum SensorType {
      UNSPECIFIED = 0;
      LIDAR_PHYSICAL_Q-Track = 1;
      LIDAR_PHYSICAL_S3 = 2;
      CAMERA_PTZ = 3;
}
   message SNMP_Q-Track {
      double temperature = 1;
      double frame_rate = 2;
      // double hfov = 3;
      int32 nmea_status = 4;
      int32 error_code = 5;
}
   message SNMP_S3 {
      double temperature = 1;
      double frame_rate = 2;
      double min_range = 3;
```

```
        double max_range = 4;
        int32 start_angle = 5;
        int32 stop_angle = 6;
}
    message ONVIF_PTZ {
        int32 state =1;
        string message =2;
        string json_cam_model_data =3;
}
    one of SNMPValue {
        SNMP_Q-Track snmp_Q-Track = 1;
        SNMP_S3 snmp_s3 = 2;
        ONVIG_PTZ onvif_ptz = 8;
}
    ConnectionStatus connection = 3;
    SensorType sensor_type = 4;
    string sensor_name = 5;
    string sensor_ip = 6;
    string sensor_model = 7;
    SensorMaskingStatus masking_status = 9;
}
message SensorStateList {
    uint64 timestamp = 1;
    uint64 sequence = 2;
    string sensor_id = 3;
    repeated SensorState sensor_state_list = 4;
}
```

Figure 5. Protobuf Output Format for QORTEX DTC 2.x State List

Publication-Confirmation Utility

The netcat utility has a similar function as the Listener in bypassing the client to verify that the server is publishing serial streams. However, it only works for human readable formats (JSON and XML), not for binary (Protobuf). For Windows 10 or Windows 11, use PuTTY.

Use the netcat utility as follows:

1. Open an Ubuntu terminal window.
2. To monitor the TCP port where serial data is being published, edit this `netcat` (`nc`) command to include the IP address of the server host computer and the desired port number, then execute it. See *QORTEX Q-Track DTC User Guide.*

```
    $ nc x.x.x.x 171xx
```

   If serial data is published on that port, it prints directly to the terminal window.

3. After data starts flowing (and assuming the zones and objects are active), the Ctrl+C command can be executed to terminate the process.

API Help Command

The API help command is available. In an Ubuntu terminal window, execute the following to return a comprehensive list of arguments and parameters. See *Figure 6. API Help Commands*.

```
$ cd /opt/quanergy/qortex-server
$ ./Qortex-server --help
Qortex Server Application:
    -h [ --help]                       Display this help message
    -s [ --settings-file] arg          Object tracking settings file
    -o [ --csv-file] arg               Output a CSV file with tracking results
    --output-trackable-points          Output CSVs representing point data for
                                       each trackage (only effective when
                                       --cvs-file is used)
--record-path arg                      Path to create qlog directory for
                                       recording. When specified, all data is
                                       recorded unless --record-manually or
                                       --record-events is specified
                                       (incompatible with --replay-path)
--record-manually                      Recording starts/stops when spacebar is
                                       pressed (requires -record-path,
                                       incompatible with -record-events)
-C [ --compression-level] arg (=3)     If recording, compression level
                                       (0-9) to use, where 0 is no compression,
                                       9 is max compression
--replay-path arg                      Path to qlog file(s) or directory of qlog
                                       file(s) (incompatible with -record-path)
-t [ --start-time] arg (=0)            If replaying, start at this time into
                                       replay (seconds)
-r [ --replay-factor] arg (=1)         If replaying, multiply playback speed by
                                       this factor
-p [ --start-paused]                   If replaying, start paused
--license arg                          For detail information about license
                                       options use -license 'usage'
--loglevel arg                         Set log level for console output
-v [ --version]                        Display version information
```

Figure 6. API Help Commands

3. API for Remote Procedure Calls

Third-party client software can make use of open-source remote procedure calls (https://grpc.io/) through the QORTEX DTC 2.4 server API commands.

> **Note:** If security is enabled, user authentication is required to enable connection between the QORTEX DTC server and QORTEX DTC client. Any application reading the QORTEX API TCP ports must authenticate with the QORTEX Server and decrypt the API data. See *Cybersecurity* (page 92) and *QORTEX Q-Track DTC User Guide.*

Creating a python gRPC Client

In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier to create distributed applications and services. As in many RPC systems, gRPC is based on 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. On the client side, the client has a stub (referred to as a client in some languages) that provides the same methods as the server.

Creating a python gRPC client includes two steps:

1. Generate client code using the protocol buffer compiler.
2. Use the client code with gRPC API to make an API call.

Generate Client Code Using the Protocol Buffer Compiler

The protocol buffer compiler creates a `.py` file for each `.proto` file input. The names of the output files are computed by taking the name of the `.proto` file. Basically, the compiler generates two class files for each proto file provided. This class file must be created only once. If there are any changes/update in the proto file, then the class files must be created again. The command to generate class files from proto file is:

```
python -m grpc_tools.protoc -I <path to proto file>
--python_out= <path to create pb2.py file>
--grpc_python_out= <path to create pb2_grpc.py file> <proto file>
```

For example:

```
python -m grpc_tools.protoc -I ../../
--python_out=.
--grpc_python_out=. ../../Qortex_service.proto
```

The above command generates two files in the folder where the command is executed:

- `Qortex_service_pb2.py` - contains message classes

- `Qortex_service_pb2_grpc.py` - contains server and client classes

The following example command refers to multiple `.proto` files, which are available upon request from support@quanergy.com:

```
python -m grpc_tools.protoc -I ../../
--python_out=.
--grpc_python_out=.
Qortex_geometric_types.proto sensor_state_list.proto qortex_server_modes.proto
qtrackable.proto zone_3d.proto qortex_service.proto server_message.proto
zone_requests.proto
```

Use the Client Code with gRPC API to Make an API Call

Once the class files (client code) are generated, a channel and stub need to be created to send an API request along with metadata.

A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub.

To call service methods, first create a stub. Create a stub by instantiating the `Qortex ServiceStub` class of the `qortex_service_pb2_grpc` module, generated from our `.proto` file:

```
channel = grpc.insecure_channel("192.168.0.1:17177")
# Creates an insecure Channel to a server with port 17177
stub = qortex_service_pb2_grpc.QortexServiceStub(channel)
# Creating stub. Client sends a request to the server using the stub
```

Metadata is information about a particular RPC call (such as client ID) in the form of a list of key-value pairs, where the keys are strings and the values are typically strings. Metadata lets the client provide information associated with the call to the server and vice versa. QORTEX DTC uses client ID as metadata. For example:

```
client_id = "82ef205c-ed14-4013-89cf-85e11b6827b1"
# An example client ID. String representation of a 128-bit Universally unique ID
metadata = (("client_id", client_id),)
# Metadata to be transmitted to the service-side of the RPC
```

The QORTEX DTC server distinguishes API calls from multiple clients with the help of metadata. Each client has unique client ID, resulting in unique metadata, so API calls originating from client should have metadata associated with it.

Issuing API Commands

Each API command has its own permission for interacting with the client.

- **Configure Mode permission APIs**, the client must call `SwitchToConfigMode` API first, or else a `Permission denied` error occurs.
- **Monitor Mode permission APIs**, the client can call them with or without calling the `SwitchToConfigMode` API.

When software was installed on the server host computer, the default folder `/home/quanergy/quanergy/Qortex` was assigned permission level `777`.

SwitchToConfigMode API

Set Client in configuration mode. Required for multiple configuration actions.

Send and response: `SwitchToConfigMode`

```
    rpc SwitchToConfigMode (stream Empty) returns (stream Empty) {}
```

General Enumerations

gRPC uses Google Protocol Buffer (Protobuf 3) as its default parameter type.

- Some messages are used for general requests, such as `Empty`, `StatusResult`, and `RequestResult`.
- An `Empty` message means there are no other specific parameters (similar to void type in C++).
- The `StatusResult` and `RequestResult` are used to get a running status or to indicate if a request was successful.

Definitions of the messages are:

```
message Empty {}
message StatusResult {
 enum Status {
 Running = 0;
 Stopped = 1;
 }
 Status current_status = 1;
}

message RequestResult {
 bool result = 1;
}
```

Some enumeration types of messages are also frequently used by those APIs, including:

```
enum PubSubTopic {
 UNKNOWN  = 0;          /** reserved, to avoid use default enum type value */
 CONNECTION_STATUS = 1; /** Streaming type, used only for C++ client, to notify
connection is established/interrupted */ (see note 1)
 SERVER_MESSAGE = 2;    /** General type message; like server error, replay status
etc */
 SERVER_STATE = 3;      /** Will be published when play status is changed via
start/stop/pause/resume in live/playback position */
 SENSOR_STATE = 4;      /** SensorState will be published when fatal error in snmp
field or sensor changed to disconnected */
 ZONE_STATE = 5;        /** Will be published when zone is created/updated/deleted
*/
}
enum PlaybackCommand {
 GET_QLOG_LIST = 0;
 SPEED_UP = 1;
 SLOW_DOWN = 2;
 PAUSE = 3;
 RESUME = 4;
 SEEK = 5;
 STEP = 6;
 REPLAY = 7;
}
enum ServerMode {
 LIVE   = 0;
 PLAYBACK  = 1;
}

(Note 1) CONNECTION_STATUS is only used for a C++ client. The client will only
receive meaningful messages when the server side publishes them. Otherwise, blank
messages are received, and in C++ those blank messages are ignored. The expected
client behavior (depending on the C++ implementation) is: (1) The client sets the
topic of interest and sends a request to the server. (2) The client gets the
expected response with a corresponding data field that is contained in one of the
response fields.
```

PubSub API

`PubSubTopic` defines types of the topics that the API supports for Pub/Sub.

`PubSub` is acronym for Publish-Subscribe. This API is used by client to Subscribe to server, once subscribed, the state change of server gets published to all the clients that are subscribed. State changes such as starting a location, stopping a location, sensor spun up, sensor disconnected, recording started, recording stopped, add/edit/removing a zone, playback started, playback completed, sensor status (temperature, error code, ...) license state, and so forth. shall get published.

These state changes are published from server in below message categories.

```
CONNECTION_STATUS = 1;
    /** Streaming type, only used for notifying connection has established/interrupted */
SERVER_MESSAGE    = 2;
    /** General type message; like server error, replay status etc */
SERVER_STATE      = 3;
    /** Will be published when play status is changed, like start/stop/pause/resume live/playback position */
SENSOR_STATE      = 4;
    /** SensorState will be published when there's fatal error in snmp field or sensor changed to disconnected */
ZONE_STATE        = 5;
    /** Will be published when zone is created/updated/deleted */
JSON_PUSH         = 6;
    /** Will be used for "pushing" notification from server in JSON format */
```

1. `Server_message` shall get published to monitor client as well, where as `server_state`, `sensor_state` and `zone_state` gets published to config client only.
2. Below is the `PubSub` API call to subscribe to server with a topic. The below code can be put inside a function after assigning a topic, similarly for different topic, different function can be created. Threading or multiprocessing can be used to call all the functions with different topics

```
def stream():
yield qtrack_service_pb2.PubSubRequest(topic= <'SERVER_MESSAGE' or
'SERVER_STATE' or 'SENSOR_STATE' or 'ZONE_STATE' >)
# Provide any one
input_stream = stub.PubSub(stream(), metadata=metadata)
while 1:
for i in input_stream:
print i.WhichOneof("response")
print ("\n PubSub Output is \n %s" % i)
print i.ClearField("response")
```

Send and response: `PubSub`

```
rpc PubSub(stream PubSubRequest) returns (stream PubSubResponse) {}
```

Sample

```
// Topics
message PubSubResponse {
 PubSubTopic topic = 1;
 oneof response {
 ServerMessage server_message = 2;
 ServerState server_state  = 3;
 Empty connection_status  = 4;
 SensorState sensor_state  = 5;
 ZoneState zone_state   = 6;
 }}
```

Playback API

`Playback` lists numbers of commands supported on the server side for playing back recorded QLogs.

Send and response: `Playback`

```
rpc Playback(PlaybackRequest) returns (PlaybackResponse) {}
```

ServerMode API

`ServerMode` indicates which mode the server is running.

Sample Python Code

Python code for using a General Enumerations API sample.

```
def getserverstate(metadata, stub):
    """
    API call to get server state
    :param stub: stub object
    :param metadata: metadata to be sent
    :return: the server current state info such as server mode, version, is
        capturing, is paused, etc. .
    """
    state_req = qtrack_service_pb2.Empty()   # Creating an empty request
    state_resp = stub.GetServerState(state_req, metadata=metadata)
                                                            # Sending the request to server via
                                                            stub and getting response
    print "Server state is :\n", state_resp
    return state_resp

def getzone(metadata, stub, zonetype):
    """
    API call to get zone data
    :param stub: stub object
    :param metadata: metadata to be sent
    :param zonetype: type of zone (EVENT or EXCLUSION)
    :return: zone data such as zone name, UUID, class, vertices, etc. . of all the zones
        (Event/Exclusion)
    """
    getzone_req = zone_requests_pb2.ZonesRequest(zone_class=zonetype)
        # Creating a request with zone_class=<required zone class>
    getzone_resp = stub.GetZones(getzone_req, metadata=metadata)
        # Sending the request to server via stub and getting response
    print "The zone info is :\n", getzone_resp
    return getzone_resp

if __name__ == '__main__':
    channel = grpc.insecure_channel("192.168.0.1:17177")
        # Creates an insecure Channel to a server
    stub = qtrack_service_pb2_grpc.QortexServerStub(channel)
        # Creating stub. Client sends a request to the server using the stub and waits for a responseto come
back, just like a normal function call
    client_id = "82ef205c-ed14-4013-89cf-85e11b6827b"     # An example client ID
    metadata = (("client_id", client_id),)                # Optional metadata to
        be transmitted to the service-side. Here metadata with client ID is a must
```

Figure 7. Sample Python Code: General Enumerations API

Calling Service Methods

There are several ways to call a service method.

Simple RPCs

A simple RPC is a unary RPC where the client sends a single request to the server and gets a single response back, just like a normal function call. For example:

```
liveModReq = qtrack_service_pb2.StartPlayRequest(mode="LIVE",)
    # Construct start Play request with mode value as LIVE
liveModRespon = stub.StartPlay(liveModReq,metadata=metadata)
    # send the request to server and wait for response
print "The start play result response is: ", liveModResponse.result
    # print the bool response result value
```

A call to the unary `StartPlay` API is nearly as straightforward as calling a local method. The RPC call waits for the server to respond, then either returns a response (a boolean value) or raise an exception.

List of QORTEX DTC Server Simple/Unary RPCs

QORTEX DTC server has 26 simple/unary RPCs, grouped into the following modules:

`Config`	`Track`
`Settings`	`Status`
`Play/Playback`	

Send and response: `config` module RPCs

```
rpc GetZones(ZonesRequest) returns (QZone3DArray) {} (see note 1)
rpc AddZone (AddZoneRequest) returns (RequestResult) {}
rpc EditZone (QZone3D) returns (RequestResult) {}
rpc RemoveZone (RemoveZoneRequest) returns (RequestResult) {} (see note 2)
rpc SetZonesEnabled (EnableZonesRequest) returns (RequestResult) {}
rpc GetZonesEnabled (ZonesRequest) returns (ZonesEnableStatus) {}
rpc ChangeEventZoneViolationAction (ChangeViolationActionRequest) returns
(RequestResult) {}
rpc ChangeEventZoneViolationTrigger (ChangeViolationTriggerRequest) returns
(RequestResult) {}
rpc GetEventZoneViolationActions(GetEventZoneViolationRequest) returns
(GetEventZoneViolationResponse) {}
rpc SetEventZoneViolationRecordingEnabled (ViolationRecodingStatus) returns
(RequestResult) {}
rpc GetEventZoneViolationRecordingEnabled (Empty) returns (ViolationRecodingStatus)
{}
rpc GetEventZoneViolationHandlerEnabled (Empty) returns (ViolationHandlerStatus) {}
rpc GetSettingsTemplateList(Empty) returns (GetSettingsTemplateResponse) {}

(Note 1) The request parameter for this RPC is:
ZonesRequest::QZoneClass::EVENT
ZonesRequest::QZoneClass::EXCLUSION
The message response for this RPC is:
message ZonesRequest {
 QZoneClass zone_class = 1;
}
(Note 2) The message response for this RPC is:
message RemoveZoneRequest {
 string uuid = 1;
}
```

Send and response: `settings` module

```
rpc GetSettings(GetSettingsRequest) returns (SettingsSectionResponse) {}
rpc SetSettings(SettingsSectionResponse) returns (RequestResult){}
rpc Settings(SettingsRequest) returns (SettingsResponse) {}
// config/recording module
rpc SetRecordingParams(RecordingParams) returns(Empty) {}
rpc GetRecordingParams(Empty) returns(RecordingParams) {}
rpc SetRecordingStatus(RecordingStatus) returns(Empty) {}
rpc GetRecordingStatus(Empty) returns(RecordingStatus) {} (See note 1)

(Note 1) The response for this RPC is the general usage message type, and the
result will reflect whether the request was successful:
message RequestResult {
 bool result = 1;
}
```

Send and response: `play/playback` module

```
//Stop capturing data in LIVE/PLAYBACK modes
    rpc StopPlay(Empty) returns(RequestResult) {}
//Start location in LIVE/PLAYBACK mode
    rpc StartPlay(StartPlayRequest) returns(RequestResult) {}
    rpc Playback(PlaybackRequest) returns (PlaybackResponse) {}
```

Send and response: `track` module

```
rpc ResetTrack(Empty) returns(RequestResult) {}
```

Send and response: `status` module

```
rpc GetServerState(Empty) returns (ServerState) {}
rpc GetSensorState(GetSensorStateRequest) returns (SensorStateList) {}
```

Bidirectional Streaming RPCs

A bidirectional streaming RPC occurs 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 read a message then write a message, or some other combination of reads and writes.

The order of messages in each stream is preserved. Specify this type of method by placing the stream keyword before both the request and the response.

Streaming sample

```
def stream():
 yield qtrack_service_pb2.Empty() # Empty Stream of data from client
input_stream = stub.SwitchToConfigMode(stream(), metadata=metadata,
timeout=400000000)
# Send Empty stream of data from client and receive empty stream of data from server
while 1:
 print('switched to Config Mode.. \n {}'.format(next(input_stream)))
# indefinite receiving of empty data from server
```

List of QORTEX DTC Server Bidirectional RPCs

QORTEX DTC server has four bidirectional streaming RPCs, grouped by module:

```
Auth
Streaming
```

Send and response: bidirectional `auth` API

```
rpc SwitchToConfigMode (stream Empty) returns (stream Empty) {}
```

Send and response: bidirectional `streaming` API

```
rpc GetFile(stream FileChunk) returns (stream FileChunk) {} (see Note 1)
rpc PutFile(stream FileChunk) returns (stream FileChunkIndex) {}
rpc PubSub(stream PubSubRequest) returns (stream PubSubResponse) {}

(Note 1) The request parameter for this RPC needs to set
FileChunk::FileInfo::FileType as RECORDING_QLOG_FILE.
The message definition for file transmission is:
enum FileType {
 CALIBRATION_FILE  = 0; /* can only upload from client to server */
 QGUARD_SETTINGS_FILE = 1; /* can only upload from client to server */
 LOG_FILE = 2; /* can only download from server to client */
 RECORDING_QLOG_FILE = 3; /* can only download from server to client */
 GENERAL_FILE   = 4; /* can download/upload from/to server to/from client*/
}

message FileChunk {
 message FileInfo {
 FileType file_type = 1;
 string file_name = 2;
 uint64 file_size = 3;
 uint32 chunk_count = 4;
 }
 message RawData {
 uint32 chunk_index = 1;
 bytes data   = 2;
 }
 oneof Chunk {
 FileInfo file_info = 1;
 RawData raw_data = 2;
 FileChunkIndex chunk_index = 3;
 }
}

message FileChunkIndex {
 uint32 chunk_index = 1;
}
```

Authorization Module—SwitchToConfigMode API

`SwitchToConfigMode` API is an Authorization Module that puts the client into configuration mode for the requesting client, so that only one client can configure settings. There can be only one client in configuration (config) mode, so any request coming from another client for configuration is rejected.

`SwitchToConfigMode` API is activated by the request/response stream. The client initiates an empty request and the server responds with an empty request.

Sample

```
def stream():
 yield qtrack_service_pb2.Empty()
# Empty Stream of data from client
 input_stream = stub.SwitchToConfigMode(stream(), metadata=metadata,
 timeout=400000000)
# Send and receive Empty stream of data
 while 1:
 print('switched to Config Mode.. \n {}'.format(next(input_stream)))
# indefinite receiving of empty data from server
```

Certain APIs work only after the server verifies that the client is in configuration mode. If the server checks and discovers that the client is not in configuration mode, those APIs instigate an `UNAUTHENTICATED` error. Therefore, when calling those APIs, the `SwitchToConfigMode` API stream should be running in parallel, so that the above code would run in a separate python thread.

Terminating the `SwitchToConfigMode` API call switches the client into monitor mode. Terminating is done by stopping the python thread that started.

APIs That Do Not Require SwitchToConfigMode Module

Below is the list of APIs that **do not** require the `SwitchToConfigMode` API stream to run in parallel. The APIs listed below work in **Monitor** mode, while the remaining APIs need the client to be in config mode. These APIs are grouped into the following modules:

```
Zone
Status
Streaming
```

Send and response: `zone` module

```
rpc GetZones(ZonesRequest) returns (QZone3DArray) {}
rpc GetEventZoneViolationActions(GetEventZoneViolationRequest) returns
(GetEventZoneViolationResponse) {}
rpc GetEventZoneViolationRecordingEnabled (Empty) returns (ViolationRecodingStatus)
{}
rpc GetEventZoneViolationHandlerEnabled (Empty) returns (ViolationHandlerStatus) {}
```

Send and response: `status` module

```
rpc GetServerState(Empty) returns (ServerState) {}
rpc GetSensorState(GetSensorStateRequest) returns (SensorStateList) {}
```

Send and response: `streaming` module

```
rpc GetFile(stream FileChunk) returns (stream FileChunk) {}
rpc PubSub(stream PubSubRequest) returns (stream PubSubResponse) {}
```

Visualization Module

The Visualization module includes the APIs for `StartPlay`, `StopPlay`, `ResetTrack`, `SetRecordingParams`, `GetRecordingParam`, `SetRecordingStatus`, `GetRecordingStatus`, and `Playback`. These APIs enable the visualization of real-time (Live) or recorded (Playback) point cloud data. Most of these are simple RPCs. Sample Python code for using a Visualization API.

```
def startplay(metadata, stub):
        """
        API call to start a location in live mode
        :param stub: stub object
        :param metadata: metadata to be sent
        : return: Bool value
        """
        print("starting live mode")
        liveModeReq =
        qtrack_service_pb2.StartPlayRequest(mode="LIVE")
            # Constructing start play request with
ServerMode enum value
        liveModeResponse =
        stub.StartPlay(liveModeReq,metadata=metadata)
        print "The start play result response is: ",
            liveModeResponse.result
        return liveModResponse.result
if __name__ ** '__main__':
        channel =
        grpc.insecure_channel("192.168.0.1:17177")
            # Creates an insecure Channel to a server
        stub =
        qtrack_service_pb2_grpc.QortexServiceStub(channel)
            # Creating stub. Client sends a request to the server using the
            # stub and waits for a response to come back, just like a normal
            # function call
        client_id = "82ef285c-ed14-4813-89cf-85e11b6827b1"
            # An example client ID
```

```
enum ServerMode {
      LIVE=0;
      PLAYBACK=1;
}
```

```
/********************
 * StartPlay
 */
message
StartPlayRequest {
      SeverMode mode=1;
      string
qlog_name=2;
}
```

```
    metadata = (("client_id", client_id),)
        # Optional metadata to be transmitted to the service-side of the
        # RPC. Here metadata to be transmitted to the service_side of the
        # RPC.
    startplay(metadata,stub)
```

Figure 8. Sample Python Code: Visualization API

Each API is explained below, with an example of the call and response. These are:

`GetRecordingParam`	`SetRecordingParam`
`GetRecordingStatus`	`SetRecordingStatus`
`Playback`	`StartPlay`
`ResetTrack`	`StopPlay`
`Set_Qlog`	

GetRecordingParam API

Fetches the recording parameters set in the server by using the `SetRecordingParams` API. It returns the record folder filepath and compression level set.

Send and response: `GetRecordingParams`

```
rpc GetRecordingParams(Empty) returns(RecordingParams) {}
```

Sample

```
getRecParm = qtrack_service_pb2.Empty()
    # Construct an empty request
getRecParmResp = stub.GetRecordingParams(getRecParm,metadata=metadata)
    # Send the Empty request to GetRecordingParams API and get response
print getRecParmResp
    # print recording folder name and compression level
```

GetRecordingStatus API

Fetches the recording status (whether a recording is started/in-progress or stopped/completed).

Send and response: `GetRecordingStatus`

```
rpc GetRecordingStatus(Empty) returns(RecordingStatus) {}
```

Sample

```
recsts_req = qtrack_service_pb2.Empty()
    # Construct an empty request
recSts_resp = stub.GetRecordingStatus(recsts_req,metadata=metadata)
    # Send the Empty request to GetRecordingStatus API and get response
print "Recording status is (0=STOP,1=START): ", recSts_resp.status
    # print the response value 0 (STOP) or 1 (START)
```

Playback API

Used to play a recorded dataset. This API can pause, resume, or replay a dataset, step a duration, seek to a duration, and speed up or slow down a playing dataset. Playing a recorded dataset is a two-step process. First, the client must send a request to the server to get all available datasets from the `datasets` folder with `Playback` API having `cmd=GET_QLOG_LIST`. Second, switch to playback mode to play the dataset with `StartPlay` API with `mode=Playback` and `qlog_name = <required qlog name>` fetched from first step.

Send and response: `Playback`

```
rpc Playback(PlaybackRequest) returns (PlaybackResponse) {}
```

Sample

Step 1

```
playbck_req = qtrack_service_pb2.PlaybackRequest(cmd= "GET_QLOG_LIST")
    # construct request to get all available qlog/datasets from datasets folder
playback_resp = stub.Playback(playbck_req, metadata=metadata)
    # Send request to Playback API in server, get qlog/dataset list as response
print playback_resp
```

Step 2

```
playbackmodereq = qtrack_service_pb2.StartPlayRequest(mode= "PLAYBACK",
qlog_name=<qlog_name>)
    # Construct request to switch to playback mode with mode=PLAYBACK and
    qlog_name=<any one qlog name from step1
playbackmodresp = stub.StartPlay(playbackmodereq, metadata=metadata)
    # Send the request to StartPlay API in server and get bool response
return playbackmodresp.result
```

PAUSE, RESUME or REPLAY

To PAUSE, RESUME or REPLAY a dataset `Playback` API with a different `cmd` value, see the following example. The request should contain the required command and the qlog/dataset name. The constructed request must be sent via stub to `Playback` API in the server.

```
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="PAUSE",qlog_name=<qlog_name>)
    # Request to pause a running dataset
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="RESUME",qlog_name=<qlog_name>)
    # Request to resume a paused dataset
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="REPLAY",qlog_name=<qlog_name>)
    # Request to replay a dataset
```

STEP or SEEK

To STEP or SEEK, the `Playback` API must import `google.protobuf.timestamp_pb2`. The protobuf timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. For example, 15.23 seconds is represented in Google timestamp as `Timestamp(seconds=15, nanos=230000000)`. STEP allows step dataset play for a constant value, and it can be performed only when the dataset is paused. SEEK allows the dataset to jump to the required time, and it can be performed in a paused or running state.

```
playbck_req = qtrack_service_pb2.PlaybackRequest(cmd= "STEP",
qlog_name=<qlog_name>,timestamp=Timestamp(seconds=0, nanos=100000000))
   # Request to step 100 ms
playbck_req = qtrack_service_pb2.PlaybackRequest(cmd= "SEEK",
qlog_name=<qlog_name>,timestamp=Timestamp(seconds=15, nanos=230000000))
   # Request to seek/jump to 15.23 seconds (eg values)
```

Change Speed

To change the speed of playback, a running playback/recorded dataset can be sped up or slowed down. The speed of the dataset increments is twice the previous value (2x, 4x, 8x, 16x, 32x, 64x ,128x) until the max value of 128x for every call of `SPEED_UP` is reached. Similarly, `SLOW_DOWN` reduces the speed at twice its earlier speed up to a maximum value of 0.001x for every call of `SLOW_DOWN`. For example:

```
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="SPEED_UP",qlog_name=<qlog_name>)
   # Request to speed up a dataset
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="SLOW_DOWN",qlog_name=<qlog_name>)
   # Request to slow down a dataset
```

ResetTrack API

Resets the trackables on the server side. The server restarts the tracking from scratch.

Send and response: `ResetTrack`

```
rpc ResetTrack(Empty) returns(RequestResult) {}
```

Sample

```
resetReq = qtrack_service_pb2.Empty()
   # Construct an empty request
resetResp = stub.ResetTrack(resetReq, metadata=metadata)
   # Send the Empty request to ResetTrack API in server and get response
print "The reset track response is: ",resetResp.result
   # print bool result response
```

Set_Qlog API

Used to add the datasets folder path to the QORTEX DTC server configuration so the server can read/write dataset in `settings.xml` file.

```
playbck_req =
qtrack_service_pb2.PlaybackRequest(cmd="SET_QLOG",qlog_name=<qlog_name>)
    # Request to change the path to datasets folder path.
```

SetRecordingParam API

Sets recording parameters such as Recording folder name and compression level of recording data. The folder name provided is created inside the `datasets` folder located in `/home/quanergy/quanergy/qortex`. All recordings are stored inside the created folder. If no folder name is given, the recordings are created inside the datasets folder. The API does not accept multiple folder level names, such as `Recordings/Zone1` or `/home/user/Recordings/`.

For the compression level, any integer value from `0` to `9` is accepted (`0` = no compression; `9` = high compression). A location has to be running/playing in live or in playback mode in order to call this API. If location is not running/playing in live or playback, this error occurs: `Server-side pipeline is not running. Please start play first`. Recording cannot be started when a playback is running.

Send and response: `SetRecordingParams`

```
rpc SetRecordingParams(RecordingParams) returns(Empty) {}
```

Sample

```
setRecParm = qortex_server_modes_pb2.RecordingParams(record_path="Recordings",
compression_level=<0 to 9>)
    # Construct RecordingParams request with folder name and compression level
SetRecParmResp = stub.SetRecordingParams(setRecParm,metadata=metadata)
    # Send request to SetRecordingParams API in server,get EMPTY/BLANK response
 print "Recording params settings response is EMPTY", SetRecParmResp
```

SetRecordingStatus API

Used to **START** or **STOP** a recording. The status parameter value should be either **START** or **STOP**. The recording starts only when a location is running in live mode, or else this error occurs: `Server-side pipeline is not running. Please start play first.`

Send and response: `SetRecordingStatus`

```
rpc SetRecordingStatus(RecordingStatus) returns(Empty) {}
```

Sample

```
setrecsts_req = qortex_server_modes_pb2.RecordingStatus(status= "START or STOP")
    # Construct RecordingStatus with status value as START or STOP
recresp = stub.SetRecordingStatus(setrecsts_req, metadata=metadata)
```

```
    # Send request to SetRecordingStatus API in server,get EMPTY/BLANK response
print "Set(Start/Stop) Recording status response is EMPTY ", recresp
```

StartPlay API

Starts sensor connection and specifies **LIVE** or **PLAYBACK** mode in the request parameter. If **PLAYBACK** is specified, this API must also specify the additional `qlog` name parameter. If the mode is **LIVE**, the API starts a location in **Live** mode.

Send and response: `StartPlay`

```
rpc StartPlay(StartPlayRequest) returns(RequestResult) {}
```

Sample

```
liveModReq = qtrack_service_pb2.StartPlayRequest(mode= "LIVE")
    # Construct startPlayrequest with mode value as LIVE
liveModResponse = stub.StartPlay(liveModReq,metadata=metadata)
    # send the request to StartPlay API in server and get response.
print "The start play result response is: ", liveModResponse.result
    # print the bool response result value
```

StopPlay API

Stops the sensor connection/live mode location. This API is also used to stop the playback data set.

Send and response: `StopPlay`

```
rpc StopPlay(Empty) returns(RequestResult) {}
```

Sample

```
stopLoc = qtrack_service_pb2.Empty()
    # Construct an empty request
stop_loc = stub.StopPlay(stopLoc, metadata=metadata)
    # Send the Empty request to StopPlay API in server and get response
print "The stop play result response is: ", stop_loc.result
    # print bool result response
```

Configuration Module

Configuration Module APIs allow you to configure settings in the QORTEX DTC server. Most of these are simple RPCs. These APIs enable control of location, settings, zones, counter lines, and client.

Sample Python code for using a Configuration API. See *Figure 9. Sample Python Code: Configuration API* and *Figure 10. Sample Python Code: Add Zone.*

```
def add_zone(stub, metatdata, name, UUID, zMin, zMax, ZoneClass, Enabled,
X_vertices_list, Y_vertices_list, vertices_list, sensor_list, zone_info):
"""
    API call to add a zone in settings.xml file
    :param stub: stub object; :param metadata: metadate to be sent
    :param name: Name of the zone to be created in string
    :param UUID: Unique ID of the zone // string representation of a 128-bit Universally Unique ID
    :param zMIN: Z axis minimum height
    :param zMax: Z axis maximum height
    :param ZoneClass: Type of Zone (EVENT or EXCLUSION or INCLUSION)
    :param Enables: Enabled value in bool, true/false. true to view Exclusion zones.
    :param X_vertices_list: list of x axis vertices of the zone
    :param Y_vertices_list: list of y axis vertices of the zone
    :return: bool True or False
"""
vertices_list = []
try:
    zone_info = qtrack_service_pb2.zone—3d__pb2.QZone(name=str(name),
type="POLYGON") for x, y in zip(X_vertices_list, Y_vertices_list):
vertices_list.append(
qtrack_service_PB2.zone__3d__pb2.qortex__geometric__types__pb2.Vector2
(x=float(x), y=float(y)))
    Zone_vertices =
qtrack_service_PB2.zone__3d__pb2.qortex__geometric_types__pb2.Polygon2D
(vertices=vertices_list)
    zone_vert = qtrack_service_PB2.zone__3d__pb2.QPolygonGeometry
(vertices=Zone_vertices)
    zone_params = qtrack_service_PB2.zone__3d__pb2.QZone3D(uuid=str(UUID),
zone=zone_info, z_min=float(zMin), z_max=float(zMax), zone_class=str(ZoneCass),
enabled=str_to_bool(Enables), polygon_3d=zone_vert)
    zone_req = zone_requests_pb2.AddZoneRequest(zone=zone_params)
    req_resp = stub.AddZone(zone_req, metadata=metadata)
        # Sending add zone request to server
    print "Add Zone response is: ", req_resp.result
except Exception as err:
    print err
If __name__ == '__main__':
channel = grpc.insecure_channel("192.168.0.1:17177")
    # Creates an insecure Channel to a server
stub = qtrack_service_pb2_grpc.QortexServiceStub(channel)
    # Creating stub. Client sends a request to the server using the stub and waits for a response to
    # come back, just like a normal function call
client_id = "82ef205c-ed14-4013-89cf-85e11b6827b1")
    # An example client ID
metadata = (("client_id", client_id),)
    # Optional metadata to be transmited to the service-side of the RPC. Here metadata with
    # client ID is a must
add_zone(stub, metadata, name="parking", UUID=”61de205c-ed14-4013-89cf-
75e11b6827b2", zMin="0", zMax="4", ZoneClass="EVENT", Enabled=True,
X_vertices_list=[2.1,3.2,5.4], Y_vertices_list=[4.1,2.1,9.3])
```

Figure 9. Sample Python Code: Configuration API

```
def add_zone(stub, metadata):
    Y_vertices_list = [2.6, -1.9, 2.4]
    X_vertices_list = [6.3, 5.3, -9]
    vertices_list = []
    sensor_list = ["10.1.22.160"]
    zone_info = qtrack_service_pb2.zone__3d__pb2.QZone(name="Zone_12",
       type="POLYGON")
    for x, y in zip(X_vertices_list, Y_vertices_list):
        vertices_list.append(qortex_geometric_types_pb2.Vector2(x=float(x),
          y=float(y)))
    Zone_vertices = qortex_geometric_types_pb2.Polygon2D(vertices=vertices_list)
       # creating a request with vertices list for polygon2
    zone_vert =
qtrack_service_pb2.zone__3d__pb2.QPolygonGeometry(vertices=Zone_vertices)
    # creating a request with vertices list for QPolygonGeometry
    zone_params = qtrack_service_pb2.zone__3d__pb2.QZone3D(uuid="9a748505-d8b8-
4119-bbd7-89d95568513b", zone=zone_info,
    z_min=-3.0, z_max=4.0,zone_class="EXCLUSION", enabled=True,
polygon_3d=zone_vert,exclusion_sensors=sensor_list)
    zone_req = zone_requests_pb2.AddZoneRequest(zone=zone_params)
       # Constructing AddZone request
    req_resp = stub.AddZone(zone_req, metadata=metadata)
    print  (req_resp)
```

Figure 10. Sample Python Code: Add Zone

Counter Line APIs

Use the Counter Line to count trackable objects that cross a specified line within an area. The line can have multiple points. The minimum number of points to create the line is two. Trackables are counted as they cross the line in either a FORWARD or BACKWARD direction. Forward or Backward are typically relative to entering or exiting a zone or area. The data collected is the number of and timestamp for trackables crossing the line.

The command contains:

- ID: message ID, in this case is AddCounterLine
- Params: see below

The params contain:

- Name: CounterLine name assigned on the client
- Points: the points representing counter line segment(s)
- UserDirection: the direction of trackable given by user
- Point0: the starting point of the user given direction
- Point1: the end point of the user given direction
- segmentIndex: the segment index of counter line where the user direction is pointing at
- direction: the crossing type of trackable object when it goes according to the given direction
- enabled: indicate whether this counterline is enabled (true) or not (false)

CounterLine Settings in settings.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<Settings>
    <CounterLines>
        <CounterLine>
            <name>0</name>
            <Points>
              <Point>
                <x>0</x>
                <y>10</y>
              </Point>
              <Point>
                <x>0</x>
                <y>0</y>
              </Point>
              <Point>
                <x>10</x>
                <y>0</y>
              </Point>
              <Point>
                <x>10</x>
                <y>10</y>
              </Point>
            </Points>
            <UserDirection>
              <Point0>
                <x>-2</x>
                <y>5</y>
              </Point0>
              <Point1>
```

```
                <x>-1</x>
                <y>5></y>
              </Point1>
              <direction>FORWARD</direction>
              <segmentIndex>1</segmentIndex>
            </UserDirection>
        </CounterLine>
    </CounterLines>
</Settings>
```

Counter Line Object List Message (Port 17163)

```
message QCounterLineObjectArray {
  QCounterLineObjectsHeader header = 1;     // header
  repeated QCounterLineObject object = 2;   // group of published objects
}

message QCounterLineObjectsHeader {
  string messageType = 1;   // message type, "DETECTION"
  uint64 numObjects = 2;    // number of objects
  uint64 sequence = 3;      // message sequence id
  uint64 timestamp = 4;     // Epoch time in milliseconds
  string version = 5;       // application protobuf version, "1.0.0"
}

message QCounterLineObject {
  uint64 id = 1;                  // Unique ID
  uint64 timestamp = 2;           // Epoch time in milliseconds
  Vector3 position = 3;           // Position (x, y, z) unit: meter

  float speed = 4;                // Scalar speed
  string objectClass = 5;         // QTrack object class
  string direction = 6;           // trackable direction crossing the line, one of:
["FORWARD", "BACKWARD"]
  string lineName = 7;            // CounterLine name
  uint64 duration = 8;            // object duration
}
```

Counter Line Detection Message

```
{
  "header": {
    "messageType": "DETECTION",
    "numObjects": "1",
    "sequence": "1",
    "timestamp": "1539029356950793000",
    "version": "1.0.0"
  },
  "object": [
    {
      "id": "1",
      "timestamp": "1539029251157398000",
      "position": {
       "x": 0.1,
       "y": 0.1,
       "z": 0.1
       },
      "speed": 0.0656,
      "objectClass": "HUMAN",
      "direction": "FORWARD",
      "lineName": "door1",
      "duration": ""
    }
  ]
}
```

CounterLine Protobuf

```
syntax = "proto3";
package quanergy.qortex.protobuf;
import "qortex_geometric_types.proto";
message QCounterLineObjectArray {
  QCounterLineObjectsHeader header = 1;     // header
  repeated QCounterLineObject object = 2;   // group of published objects
}

message QCounterLineObjectsHeader {
  string messageType = 1;   // message type, "DETECTION"
  uint64 numObjects = 2;    // number of objects
  uint64 sequence = 3;      // message sequence id
  uint64 timestamp = 4;     // Epoch time in milliseconds
  string version = 5;       // application protobuf version, "1.0.0"
}

message QCounterLineObject {
  uint64 id = 1;                  // Unique ID
  uint64 timestamp = 2;           // Epoch time in milliseconds
  Vector3 position = 3;           // Position (x, y, z) unit: meter

  float speed = 4;                // Scalar speed
  string objectClass = 5;         // QTrack object class
  string direction = 6;           // trackable direction crossing the line, one of:
```

```
                                   // ["FORWARD", "BACKWARD"]
  string lineName = 7;             // CounterLine name
  uint64 duration = 8;             // object duration
}
```

Counter Line TCP Publisher Output

```
{
 "header": {
  "timestamp": "1596710280552",
  "frameId": "quanergy",
  "sequence": 3
 },
 "zones": [
  {
   "uuid": "92d38325-7315-4312-bfc4-9d3a9bd26e08",
   "timestamp": "1596710280552",
   "name": "EVENT ZONE 0",
   "shape": {
    "vertices": [
     {
      "x": 4.98169136,
      "y": -16.4128914
     },
     {
      "x": 4.75,
      "y": -7.11669922
     },
     {
      "x": 8.85139465,
      "y": -6.75
     },
     {
      "x": 10.75,
      "y": -15.6122742
     }
    ]
   },
   "objectCount": 2,
   "objectIds": [
    "45",
    "46"
   ],
   "zMin": 0,
   "zMax": 4,
   "zoneClass": "FENCE",
   "objectCounts": [
    {
      "objectClass": "person",
      "objectCount": 1,
      "objectIds": ["45"]
    },
    {
      "objectClass": "unknown",
      "objectCount": 1,
      "objectIds": ["46"]
    }
   ]
  }
 ]
}
```

Counter Line TCP Publisher when objectCounts are Zero

```
{
 "header": {
  "timestamp": "1596710283705",
  "frameId": "quanergy",
  "sequence": 19
 },
 "zones": [
  {
   "uuid": "92d38325-7315-4312-bfc4-9d3a9bd26e08",
   "timestamp": "1596710283705",
   "name": "EVENT ZONE 0",
   "shape": {
    "vertices": [
     {
      "x": 4.98169136,
      "y": -16.4128914
     },
     {
      "x": 4.75,
      "y": -7.11669922
     },
     {
      "x": 8.85139465,
      "y": -6.75
     },
     {
      "x": 10.75,
      "y": -15.6122742
     }
    ]
   },
   "objectCount": 0,
   "objectIds": [],
   "zMin": 0,
   "zMax": 4,
   "zoneClass": "FENCE",
   "objectCounts": []
  }
 ]
}
```

Counter Line TCP Publisher Output in NDJSON

```
U{"header":{"messageType":"DETECTION","numObjects":"1","sequence":"422","timestamp"
:"1568309378714","version":"1.0.0"},"object":[{"id":"189762","timestamp":"156830937
8714","position":{"x":1.68940258,"y":-1.36005783,"z":-
0.221662849},"speed":1.5546937,"objectClass":"person","direction":"FORWARD","lineNa
me":"Counter Line 0","duration":"0"}]}
```

AddCounterLine command message

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "AddCounterLine",
    "params": {
      "name": "CounterLine 1",
      "Points": [
        {
          "x": 0,
          "y": 10
        },
        {
          "x": 0,
          "y": 0
        },
        {
          "x": 10,
          "y": 0
        },
        {
          "x": 10,
          "y": 10
        }
      ],
      "UserDirection": {
        "Point0": {
          "x": -2,
          "y": 5
        },
        "Point1": {
          "x": -1,
          "y": 5
        },
        "segmentIndex": 0,
        "direction": "FORWARD"
      },
      "enabled": true
    }
  }
}
```

AddCounterLine response message

```
{
  "reply":"OK"
}
```

CounterLine state message

Server publishes this message to all connected clients to indicate that counter lines state on the server has changed. The message includes:

- id: State
- mode: indicate the type of update, with three possible values: AddCounterLine, EditCounterLine, or RemoveCounterLine. The example below with AddCounterLine indicate that a counter line has been added.
- params: content of the change requested

Client that does not initiate the update needs to apply the update accordingly.

```
{
  "group":"COUNTERLINE",
  "command":{
    "id":"State",
    "mode":"AddCounterLine",
    "params":{
      "name":"Counter Line 0",
      "Points":[
        {
          "x":"-15.75",
          "y":"6.04919"
        },
        {
          "x":"-9.87442",
          "y":"5.5"
        }
      ],
      "UserDirection":{
        "Point0":{
          "x":"-12.8809",
          "y":"5.04015"
        },
        "Point1":{
          "x":"-13.1799",
          "y":"1.84115"
        },
        "segmentIndex":"0",
        "direction":"FORWARD"
      },
      "enabled":"true"
    }
  }
}
```

EditCounterLine command message

EditCounterLine command message contains details about an existing CounterLine to be updated. See AddCounterLine for more details.

Note: newName field is optional and when it's present then CounterLine name will be updated using the newName.

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "EditCounterLine",
    "params": {
      "name": "CounterLine 1",
      "newName": "CounterLine new name",
      "Points": [
        {
          "x": 0,
          "y": 10
        },
        {
          "x": 0,
          "y": 0
        },
        {
          "x": 10,
          "y": 0
        },
        {
          "x": 10,
          "y": 10
        }
      ],
      "UserDirection": {
        "Point0": {
          "x": -2,
          "y": 5
        },
        "Point1": {
          "x": -1,
          "y": 5
        },
        "segmentIndex": 0,
        "direction": "BACKWARD"
      },
      "enabled": false
    }
  }
}
```

EditCounterline response message

```
{
  "reply":"OK"
}
```

GetCounterLines command message

GetCounterLines command message can be used to request all existing Counter Lines from server. Params would be empty for this message.

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "GetCounterLines",
    "params": {}
  }
}
```

GetCounterLines response message

```
{
  "reply":"OK",
  "result":[
    {
      "name":"Counterline 1",
      "Points":[
        {
          "x":"0",
          "y":"10"
        },
        {
          "x":"0",
          "y":"0"
        },
        {
          "x":"10",
          "y":"0"
        },
        {
          "x":"10",
          "y":"10"
        }
      ],
      "UserDirection":{
        "Point0":{
          "x":"-2",
          "y":"5"
        },
        "Point1":{
          "x":"-1",
          "y":"5"
        },
        "segmentIndex":"0",
        "direction":"FORWARD"
      },
      "enabled":"true"
    },
    {
      "name":"Counterline 2",
      "Points":[
        {
          "x":"0",
          "y":"-2"
        },
        {
          "x":"0",
          "y":"-7"
        },
        {
          "x":"10",
          "y":"-7"
        }
      ],
      "UserDirection":{
        "Point0":{
          "x":"-2",
          "y":"-5"
        },
        "Point1":{
          "x":"-1",
          "y":"-5"
        },
        "segmentIndex":"0",
        "direction":"FORWARD"
      },
      "enabled":"false"
    }
  ]
}
```

GetCounterLinesEnabled command message

GetCounterLinesEnabled command message can be used to request the status of (all) CounterLines from server. Params would be empty for this message.

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "GetCounterLinesEnabled",
    "params": {}
  }
}
```

GetCounterLinesEnabled response message

GetCounterLinesEnabled response message contains counterLinesEnabled field which indicate the current CounterLines status, whether they are enabled (true) or disabled (false).

```
{
  "reply":"OK",
  "counterLinesEnabled":"true"
}
```

RemoveCounterLine command message

RemoveCounterLine command message contains info about an existing CounterLine to be removed.

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "RemoveCounterLine",
    "params": {
      "name": "CounterLine 1"
    }
  }
}
```

RemoveCounterLine response message

```
{
  "reply":"OK"
}
```

SetCounterLinesEnabled command message

SetCounterLinesEnabled command message can be used to set the status of (all) CounterLines on server, whether they are enabled (true) or disabled (false).

```
{
  "group": "COUNTERLINE",
  "command": {
    "id": "SetCounterLinesEnabled",
    "params": {
      "enabled": true
    }
  }
}
```

SetCounterLinesEnabled response message

```
{
  "reply":"OK"
}
```

PTZ Camera Configuration APIs

Here is a list of PTZ Camera Configuration APIs, followed by an explanation of each one and an example of the call and response.

PTZ Camera API Requirements

- Requires QORTEX DTC 2.4 or later.
- Requires `config` class.

Note: PTZ Cameras APIs do not work with Playback mode.

AddCamera API

Add a PTZ camera to the QORTEX DTC network. Provide identifying and usage parameters. Select a field of view shape: `Polygon`, `Sector`, or `Rectangle`.

Sample

```
{"group": "PTZ",
"command":
   {"id": "AddCamera","params":
   {"PTZCamera":
   {"FieldOfView":
      {"shape":                              #Choose one of polygon, sector, or rectangle
      "polygon", "Polygon": [[0,0], [0,1], [1,1], [1,0]]},
      "sector","Sector": [0,1.57,32]},
      "rectangle","Rectangle": [1.5708,1,1,64,64]},
   "counter_clockwise_rotation":0,
   "custom":"{\"pan_gain\":1, \"tilt_gain\":1}",
   "ip": "2.3.4.111",
   "name": "test-camera-007",
   "password": "123",
   "pose":
      {"position": [1,2,3],
      "ypr": [0.314159,0.75,0.33]},
   "type": "OnVif",
   "camera_model": "Axis",
   "video_url": "",
   "onvif_activation_url": "",
   "user_name": "user",
   "control": {"home_pan":0,"home_tilt":0,"home_zoom":0}
}}}}
```

EditCamera API

Change configuration of an existing PTZ camera to the QORTEX DTC network. Provide identifying and usage parameters. Select a field of view shape: `Polygon`, `Sector`, or `Rectangle`. Option, `new_ip`, to change the PTZ camera IP address.

Sample

```
{"group": "PTZ",
"command":
    {"id": "EditCamera","params":
    {"PTZCamera":
    {"FieldOfView":
       {"shape":                              #Choose one of polygon, sector, or rectangle
       "polygon", "Polygon": [[0,0], [0,1], [1,1], [1,0]]},
       "sector","Sector": [0,1.57,32]},
       "rectangle","Rectangle": [1.5708,1,1,64,64]},
    "counter_clockwise_rotation":0,
    "custom":"{\"pan_gain\":1, \"tilt_gain\":1}",
    "ip": "2.3.4.111",
    "new_ip": "2.3.4.112",                    #Edit option to change IP address
    "name": "test-camera-007",
    "password": "123",
    "pose":
       {"position": [1,2,3],
       "ypr": [0.314159,0.75,0.33]},
    "type": "OnVif",
    "camera_model": "Axis",
    "video_url": "",
    "onvif_activation_url": "",
    "user_name": "user",
    "control": {"home_pan":0,"home_tilt":0,"home_zoom":0}
}}}}
```

GetCameras API

Returns an array of PTZ cameras and details: name, FOV, position, orientation, home location, IP address, camera name and credentials, model, and URLs.

Sample request and response

```
{"group": "PTZ",
"command":
   {"id": "GetCameras","params”: {}
}}

{
  "reply": "OK",
  "result": [
    {
   {"PTZCamera":
   {"FieldOfView":
      {"shape":
      "rectangle","Rectangle": [0,0,0,8,8]},
   "counter_clockwise_rotation":0,
   "custom":"{\"pan_gain\":1, \"tilt_gain\":1}",
   "ip": "10.10.10.1",
   "name": "name1",
   "password": "123",
   "pose": {"position": [1,2,3],
      "ypr": [-0,0,-0]},
   "type": "OnVif",
   "camera_model": "Axis",
   "video_url": "",
   "onvif_activation_url": "",
   "user_name": "user",
   "control": {"home_pan":0,"home_tilt":0,"home_zoom":0}
}}},
    {
   {"PTZCamera":
   {"FieldOfView":
      {"shape":
      "rectangle","Rectangle": [0,0,0,8,8]},
   "counter_clockwise_rotation":0,
   "custom":"{\"pan_gain\":1, \"tilt_gain\":1}",
   "ip": "10.10.10.2",
   "name": "name2",
   "password": "123",
   "pose":
      {"position": [1,2,3],
      "ypr": [-0,0,-0]},
   "type": "OnVif",
   "camera_model": "Axis",
   "video_url": "",
   "onvif_activation_url": "",
   "user_name": "user",
   "control": {"home_pan":0,"home_tilt":0,"home_zoom":0}
}}}]}
```

GetCameraModels API

Returns entire JSON with PTZ camera model data to the client.

PTZ camera model data stored in JSON file at the same location as `qortex_client.ini`. PTZ camera model for IP address stored in `qortex_client.ini`. This is added as part of PTZ camera activation. The PTZ camera model is inserted to the model field in QORTEX DTC client after the PTZ camera is activated. The model data populates corresponding fields for PTZ camera editing. For example, streaming and activation URL.

Sample

```
{
    "group": "PTZ",
    "command":
    {"id": "GetCameras","params”: {}
}}
```

GetCameraTiming API

To set the number and frequency of switching between tracked camera objects.

EditCameraTiming message PTZTimeSwitching : true

```
{
   "group":"RULES",
   "command":{
      "id":"EditCameraTiming",
      "params":{
         "PTZTimeSwitching":"true",
         "timeSwitching":{
            "interval":"1",
            "bufferSize":"32"
      }
   }
}
```

EditCameraTiming message with PTZTimeSwitching : false

```
{
   "group":"RULES",
    "command":{
        "id":"EditCameraTiming",
        "params":{
            "PTZTimeSwitching":"false"
        }
    }
}
```

GetCameraTiming message

```
{
    "group":"RULES",
    "command":{
        "id":"GetCameraTiming",
        "params":{
        }
    }
}
```

GetCameraTiming PTZTimeSwitching : true

```
{
    "reply":"OK",
    "result":{
        "PTZTimeSwitching":"true",
        "timeSwitching":{
            "interval":"1",
            "bufferSize":"32"
 }    }    }
```

GetCameraTiming PTZTimeSwitching: false

```
{
    "reply":"OK",
    "result":{"PTZTimeSwitching":"false"}
}
```

PTZ Camera Calibration

AddCameraCalibration API

Updates PTZ camera position information. Returns validation command accepted. Provide changed `position` and/or `ypr` values.

Sample request and response

```
{
  "group": "PTZ",
  "command": {
    "id": "AddCameraCalibration",
    "params": {
      "ip": "2.3.4.55",
      "pose": {
        "position": [1, 2, 3],                        # Change values
        "ypr": [0.11, 0.22, 0.33]
}}}}

{
  "reply":"OK"
}
```

RemoveCamera API

Removes PTZ camera from sensor network. PTZ camera specified by IP address.

```
{
    "group": "PTZ",
    "command":
        {"id": "RemoveCamera",
        "params":
            {"ip": "1.2.3.222"}
}}
```

PTZ Camera Move Camera

SetCameraFollow API

Sets camera to follow a specific trackable manually. Trackable object ID defined in QORTEX DTC server side.

Sample request

```
{
 "group":"PTZ",
 "command":{
 "id":"SetCameraFollowObject",
 "params":{
   "ip":"10.1.22.28",
   "id":1035,                         # Trackable object ID from server
  "on":1                              # Set manual tracking on.
}}}
```

GetCameraFollowObject API

Returns the trackable object ID that the PTZ camera is following manually.

Sample request and response

```
{
 "group":"PTZ",
 "command":{
 "id":"GetCameraFollowObject",
 "params":{
   "ip":"10.1.22.28"
}}}
{
 "reply":"OK",
 "result":{
 "track_id":1035                                    # ID of object being tracked
}}
```

StopFollowingAll API

Stops all the PTZ cameras from following all the QORTEX DTC derived trackable objects.

Sample request

```
{
 "group":"PTZ",
 "command":{
 "id":"StopFollowingAll",
 "params":{ }
}}
```

SetHomePos API

For the specified PTZ camera, set the current Pan/Tilt/Zoom values as the default home location. The PTZ camera returns to the home location when it is idle after movement stops. PTZ camera specified by IP address.

Sample request

```
{
 "group": "PTZ",
 "command": {
   "id": "SetHomePos",
   "params": {
     "ip": "1.2.3.222"
}}}
```

GotoHomePos API

Directs the specified PTZ camera to return immediately to the set default home Pan/Tilt/Zoom location. PTZ camera specified by IP address.

Sample request

```
{
  "group": "PTZ",
  "command": {
    "id": "GotoHomePos",
    "params": {
      "ip": "1.2.3.222"
}}}
```

PTZ Camera Control Enabled

SetPTZ Control API

Enables PTZ camera automatic control. PTZ camera automatically follows trackable objects according to defined rules.

Sample request

```
{
 "group":"PTZ",
 "command":{
   "id":"SetPTZControlEnabled",
   "params":{
   "enabled":1                                    # Value 1 enables automatic tracking.
}}}
```

GetPTZ Control API

Returns the PTZ camera control enabled status.

Sample request and response

```
{
 "group":"PTZ",
 "command":{
   "id":"GetPTZControlEnabled",
   "params":{ }
}}

{
  "reply":"OK",
  "result":{
    "enabled":1              # Enabled status 1: PTZ camera automatically tracking objects.
}}
```

SendJsonText API

This API is used to send a JSON formatted string message from QORTEX DTC client to QORTEX DTC server. This API is used to Add/Remove/Get/enable/disable or move/calibrate PTZ camera settings and rules with help of JSON formatted message.

To configure PTZ camera and rules, see *QORTEX Q-Track DTC User Guide.*

An example implementation is shown below for adding a PTZ camera, similar approach can be used to edit, remove, get, disable, or enable camera or rules and move or calibrate camera.

```
jsonmsg_addCamera = {
"group": "PTZ",
"command": {
"id": "AddCamera",
"params": {
"PTZCamera": {
"FieldOfView": {
"shape": "polygon",
"Polygon": [
               [
0,
0
 ],
                [
0,
1
 ],
                [
1,
1
 ],
                [
1,
0
 ]
           ]
        },
"counter_clockwise_rotation": 0,
"ip": "2.3.4.111",
"name": "test-camera-007",
"pan_gain": 0.11,
"password": "123",
"pose": {
"position": [
1,
2,
3
 ],
"ypr": [
0.314159,
0.75,
0.33
```

```
 ]
          },
"tilt_gain": 0.22,
"type": "OnVif",
"camera_model": "Axis",
"user_name": "user",
"zoom_gain": 0.33,
"control": {
"home_pan": 0,
"home_tilt": 0,
"home_zoom": 0
 }
                }
            }
        }
    }
   ### Converting json request to string format
jsontext = json.dumps(jsonmsg_addCamera)
   ### construct the API call request with the json string text
jsonreq = qtrack_service_pb2.server__message__pb2.JsonMessage(text=jsontext)
jsonresp = stub.SendJsonText(jsonreq, metadata=metadata)
   ## Sending the json request to server and waiting for response
print jsonresp
```

Rules APIs

Here is a list of Rules APIs, followed by an explanation of each one and an example of the call and response.

```
AddRule command msg                    GetRules command msg
      AddRule response msg                   GetRules response msg
EditRule command msg                   RemoveRule command msg
      EditRule response msg                  RemoveRule response msg
```

Rules API Requirements

- Requires QORTEX DTC 2.4 or later.
- Requires `config` class.

Each Rule command message contains:

- **ID**: message ID, for example AddRule
- **Params**: The params contain:
  - **Type**: The Rule type. Options: `Recording`, `PTZ`, or `NetworkAction`

- **Order**: The rule order and priority assigned on the client
- **Name**: The rule name assigned on the client
- **Zone ID**: The UUID for the Event Zone
- **Track type**: The Violation Classification type. Options: `person`, `vehicle`, `person & vehicle`, or *`none`*
- **Track start**: The condition to start tracking. For rule type PTZ, option: `Enter`
- **Track stop**: The condition to stop tracking. For rule type PTZ, options: `Exit`, `Disappear`, `NewObject`
- **Enabled flag**: Always true for AddRule. EditRule can have `true` (to enable Rule) or `false` (to disable Rule).
- **Polymorphic**: An optional object that can be provided for certain types. For NetworkAction type, this contains `entry_urls` and `exit_urls` fields.

AddRule API

Adds a rule of type PTZ, recording, or network action rule.

Sample request and response

```
{
  "group": "RULES",
  "command": {
    "id": "AddRule",
    "params": {
      "type": "NetworkAction",
      "order": 1,
      "name": "Zone Rule 1",
      "zone_id": "7feb24af-fc38-44de-bc38-04defc3804de",
      "track_type": 8,
      "track_start": "None",
      "track_stop": "None",
      "enabled": true,
      "Polymorphic": {
          #For rule type: network action
        "entry_urls": {
          "url1": "http://url1/entry",
          "url2": "http://url2/entry",
          "url3": "http://url3/entry"
        },
        "exit_urls": {
          "url1": "http://url1/entry",
          "url2": "http://url2/entry",
          "url3": "http://url3/entry"
}}}}}
 // No Polymorphic elements
{
  "group": "RULES",
  "command": {
    "id": "AddRule",
    "params": {
      "type": "Recording",
      "order": 1,
      "name": "Zone Rule 1",
      "zone_id": "7feb24af-fc38-44de-bc38-04defc3804de",
      "track_type": 8,
      "track_start": "Enter",
      "track_stop": "Exit",
      "enabled": true
}}}

{
  "reply":"OK"
}
```

EditRule API

Edits existing rules. All Rule fields except `zone_id` and type can be edited. To change a `zone_id`, send RemoveRule for the old type, then issue an AddRule for the new type.

To disable a Rule, set EditRule > `enabled` to `false`.

To enable a disabled rule, set EditRule > `enabled` to `true`.

Sample request and response

```
{
  "group": "RULES",
  "command": {
    "id": "EditRule",
    "params": {
      "type": "Recording",
      "order": 1,
      "name": "Zone Rule 1",
      "zone_id": "7feb24af-fc38-44de-bc38-04defc3804de",
      "track_type": 8,
      "track_start": "Enter",
      "track_stop": "Exit",
      "enabled": true
}}}

{
  "reply":"OK"
}
```

GetRule API

Returns all existing Rules from server.

Sample request and response

```
{
  "group": "RULES",
  "command": {
    "id": "GetRules",
    "params": {}
}}

{
  "reply":"OK",
  "result":[
    {
      "type":"Recording",
      "order":"0",
      "name":"Zone Rule 0",
      "zone_id":"273100ff-f33b-466b-bba5-5489008dbff0",
      "track_type":"644",
      "track_start":"Enter",
      "track_stop":"Exit",
      "enabled":"true",
      "Polymorphic":{
        "urls":{"url":"https://www.google.com"}
      }
    },
    {
      "type":"PtzCamera",
      "order":"1",
      "name":"Zone Rule 1",
      "zone_id":"3b1c7599-bdd4-4d2a-9bd0-b4f996024a0e",
      "track_type":"388",
      "track_start":"Enter",
      "track_stop":"Exit",
      "enabled":"true"
    },
    {
      "type":"NetworkAction",
      "order":"2",
      "name":"Zone Rule 2",
      "zone_id":"861810e6-6d28-4e07-b0e8-d73093f9d0a4",
      "track_type":"644",
      "track_start":"Enter",
      "track_stop":"Disappear",
      "enabled":"true",
      "Polymorphic":{
        "entry_urls":{
          "url1": "http://url1/entry",
          "url2": "http://url2/entry",
          "url3": "http://url3/entry"
        },
        "exit_urls":{
          "url1": "http://url1/entry",
          "url2": "http://url2/entry",
          "url3": "http://url3/entry"
}}}]}
```

RemoveRule API

RemoveRule command message contains details about an existing Rule to be removed.

```
{
  "group": "RULES",
  "command": {
    "id": "RemoveRule",
    "params": {
      "type": "NetworkAction",
      "name": "Zone Rule 1",
      "zone_id": "7feb24af-fc38-44de-bc38-04defc3804de"
    }
 } }
```

Event Zone Rule Type: Object Stitching

When an object disappears then reappears within a zone and object stitching is enabled, Qortex DTC assigns the re-found object the same object ID it was previously assigned.

Object stitching identifies objects that have temporarily disappear then reappeared. Object stitching then updates the newly found object ID to match its original object ID.

Object Stitching parameters

```
<stitching_distance_threshold>10.0</stitching_distance_threshold>
<!--This parameter is the distance threshold in meters. Any new appeared id with a
distance < threshold from the disappeared object is considered for stitching. -->

<stitching_dead_time_threshold>10</stitching_dead_time_threshold>
<!--This parameter is the time threshold in seconds to remember the disappeared
objects, after this time, we throw away the disappeared object and never consider
for stitching.-->
```

```
<stitching_add_dead_objects_to_trackable>false</stitching_add_dead_objects_to_track
able>
<!--This boolean if set to true, adds disappeared objects in the trackable_map.-->

<stitching_classification_type_check>644</stitching_classification_type_check>
<!--This parameter is a bitset which specifies which classes are considered for
stitching. Currently supported:
{any(value: 0), human(value: 644), vehicle(value: 388), human&vehicle(value: 900),
unknown(value: 1)}-->
```

EditObjectStitchingParams Message

```
{
   "group":"RULES",
   "command":{
      "id":"EditObjectStitchingParam",
      "params":{
         "stitching_distance_threshold":"7.0",
         "stitching_dead_time_threshold":"100",
         "stitching_add_dead_objects_to_trackable":"false",
         "stitching_classification_type_check":"0"
      }
   }
}
```

GetObjectStitchingParams Message

```
{
   "group":"RULES",
   "command":{
      "id":"GetObjectStitchingParam",
      "params":{
      }
   }
}
```

GetObjectSticthingParams Result

```
{
   "reply":"OK",
   "result":{
      "stitching_distance_threshold":"7.0",
      "stitching_dead_time_threshold":"100",
      "stitching_add_dead_objects_to_trackable":"false",
      "stitching_classification_type_check":"0"
      }
   }
```

```
}
```

Security APIs

gRPC Push notification server requests to the client.

Security State change notification

Security State Push Notification to specific or all clients

```
  // Push notification to one or all clients for "the system is secured" or
"credentials change" or both
  std::stringstream ss;
  ss << "{
      "\"group\" : \"Security\","
      "\"command\" :  {"
        "\"id\" : \"State\","
        "\"params\": {"
          "\"secured\" : ";
  ss << (secured ? "\"v2\"" : "\"none\"");
  if (credentialsChange) {
    ss << ", "
          "\"pwd_change\" : \"true\"";
  }
  ss << "}"
      "}"
    "}";

"secured" state can be now either "v2" or "none" which means was secure for version
2 or not secured at all
"pwd_change" state set to "true" for the case when the password was changed for the
user by the admin
```

Security User Account Expiring notification

Security Account Expiring Push Notification to specific client

```
// Push notification to one client for "the account is expiring in given time
[hours]"
std::stringstream ss;
ss << "{"
    "\"group\" : \"Security\","
    "\"command\" :  {"
      "\"id\" : \"Expiring\","
      "\"params\": {"
```

```
         "\"hours\" : ";
ss << hours;
ss << "}"
    "}"
  "}";
```

Security User Account Expired notification

Security Account Expired Push Notification to specific client

```
// Push notification to one client for "the account is expired"
"{"
  "\"group\" : \"Security\","
  "\"command\" :  {"
    "\"id\" : \"Expired\""
  "}"
"}";
```

Settings Configuration APIs

Here is a list of Zone Configuration APIs, followed by an explanation of each one and an example of the call and response.

```
AddSettings                     GetSimplifiedSettings
GetSettings                     SetSettings
GetSettingsTemplateList         Settings
```

AddSettings API

Used to create a single-sensor or multi-sensor location. This API call deletes the existing settings file in the locations folder and creates a new settings file with the help of the provided template. The `AddSettings` API is a bidirectional stream API. This API has two request modes. One is for creating a single-sensor location (it has information for only one sensor/lidar). The other is for creating a multi-sensor location (it has multiple sensors/lidars) from the Transformation xml file (calibration file) or from the `settings.ini` file (qguard file - legacy `settings.ini` file), which has information for multiple sensors/lidars.

Paired with template selection, calibration file, or single sensor name and IP address, it creates settings file on server.

1. **To create a single-sensor location**, the API call needs the sensor IP address, sensor name (user-given name), and the settings template name (`general_tracking_settings.xml` or `person_tracking_settings.xml` or `vehicle_tracking_settings.xml`).

2. **Before creating a multi-sensor location**, the `calibration.xml` / `transform.xml` file or `settings.ini` should be present in the locations folder. To upload a `settings.ini` or `transformation.xml` file to locations folder, use the `PutFile` API call.

3. **To create a multi-sensor location**, the API call needs the template name, world lidar name, list of lidars, calibration file available, bool value, and qguard settings available bool value. When creating a multi-sensor location from the `transform.xml` or `calibration.xml` file, list of lidars that needs to be picked from the file can be provided, if the location is created from `qguard.settings.ini` file, the list of lidars has no effect, since all lidars in the `qguard.settings.ini` file are taken into account when creating the `settings.xml` file.

4. **Once uploaded to the location folder**, the transform/calibration xml file changes its name to `qortex.calib.temp.xml` and the `qguard.settings.ini` file changes its name to `qortex.qgsettings.temp.ini`. When these newly renamed files are available in the locations folder, the `AddSettings` API can create a location, or else it produces an error.

5. **The AddSettings API call is made** as a stream API due to multi-sensor location creation functionality. For multi-sensor location creation, a `qguard.settings.ini` file or calibration file should be available, so the `AddSettings` API call initiates the stream from client with location information and client status as `SENT_LOCATION_INFO` first, then makes a `PutFile` API call to upload the relevant `qguard.ini` or calibration file to the server, then makes an `AddSettings` API call with location info and client status while `SENT_ALL_DATA` is performed. Once this three-step process is done, a multi-sensor location `settings.xml` file gets created in the locations folder.

Send and response: `AddSettings`

```
rpc AddSettings(stream AddSettingsRequest) returns (stream AddSettingsResponse) {}
```

Sample: Required options

`qguard settings ini file calibration/ transform xml file`	`calibration_file_available = False calibration_file_available = True`	`qguard_settings_available = True qguard_settings_available = False`

Sample: For creating single-sensor location.

```
def addSinglelocation_request(template, sensorName, sensorIP):
 single_sensor_loc =
 qtrack_service_pb2.AddSettingsRequest.SingleSensorLocationInfo(
  settings_template_name=template, sensor_name=sensorName,
  sensor_ip=sensorIP)
   # Create a request to add single-sensor location
 yield qtrack_service_pb2.AddSettingsRequest(
  single_sensor_loc=single_sensor_loc,
  client_status= 'SENT_LOCATION_INFO')
   # Send the single-sensor location data with client status as SENT_LOCATION_INFO'
 yield qtrack_service_pb2.AddSettingsRequest(
  single_sensor_loc=single_sensor_loc, client_status='SENT_ALL_DATA')
   # Send the single-sensor location data with client status as SENT_ALL_DATA'
def addSinglelocation(template, sensorName, sensorIP, metadata, stub):
 addsingle = stub.AddSettings(addSinglelocation_request(
  template, sensorName,sensorIP), metadata=metadata, timeout=5)
  for r in addsingle:
   print("Create Single Location out is " + str(r))
```

Sample: For creating a multi-sensor location.

```
def addMultiplelocation_request(template, worldlidar,
 list_of_lidars, calibration_file, qguard_settings):
  li = tuple(list_of_lidars.split(","))
  multi_sensor_loc =
  qtrack_service_pb2.AddSettingsRequest.MultiSensorLocationInfo(
   settings_template_name=template, world_lidar_name=worldlidar,
   lidars=li, calibration_file_available=calibration_file,
   qguard_settings_available=qguard_settings)
 yield qtrack_service_pb2.AddSettingsRequest (
  multi_sensor_loc=multi_sensor_loc,
  client_status=ENT_LOCATION_INFO')
   # Send the single-sensor location data with client status as SENT_LOCATION_INFO'
```

Sample: For multi-sensor location, After `client_status` as `SENT_LOCATION_INFO`, call putfile API to upload the qguard or calibration file to server and then send `client_status` as `SENT_ALL_DATA`.

```
yield qtrack_service_pb2.AddSettingsRequest(
 multi_sensor_loc=multi_sensor_loc,
 client_status='SENT_ALL_DATA')
   #send the single-sensor location data with client status as SENT_ALL_DATA'
def addMultiplelocation(template, worldlidar,list_of_lidars,
 calibration_file,qguard_settings, metadata, stub):
  addmulti = stub.AddSettings
(addMultiplelocation_request
(template, worldlidar, list_of_lidars, calibration_file, qguard_settings),
metadata=metadata, timeout=5)
   for r in addmulti:
 print("Create Multi Location out is " + str(r))
```

Sample: `AddSettings` API call and response in protobuf format is:

```
rpc AddSettings (stream AddSettingsRequest) returns
(streamAddSettingsResponse) {}
message AddSettingsRequest {
 message MultiSensorLocationInfo {
 string settings_template_name = 1;
 string world_lidar_name   = 2;
 repeated string lidars   = 3;
 bool calibration_file_available = 4; /* The calibration file in xml
format */
 bool qguard_settings_available = 5; /* The calibration file in ini format
*/
 }
 message SingleSensorLocationInfo {
 string settings_template_name = 1;
 string sensor_name   = 2;
 string sensor_ip = 3;
 }

 enum ClientStatus {
 STARTED   = 0;
 SENT_LOCATION_INFO = 1;
 PROCESSING   = 2;
 SENT_ALL_DATA  = 3;
 FINISHED   = 4;
 CLIENT_FAILED  = 5;
 }

 oneof LocationInfo {
 MultiSensorLocationInfo multi_sensor_loc = 1;
 SingleSensorLocationInfo single_sensor_loc = 2;
 }
 ClientStatus client_status  = 3;
}
```

GetSettings API

Used to get/fetch the location settings or publisher settings from `settings.xml` file. This API call has two sections. One section fetches location data such as the sensor name, sensor IP, and transformation information from the `settings.xml` file. The other section fetches the publisher information such as port number, port name, publishing format, `adddatasize` bool value, and `networkbyteorder` bool value from the `settings.xml` file.

Add type in `gRPC GetSettings` when it calls `SENSOR_LOCATION` section

Use `<maxBinDistance>` parameter to just be the max range from any sensor that they care about.

Send and response: `GetSettings`

```
rpc GetSettings(GetSettingsRequest) returns (GetSettingsResponse) {}
```

Sample

```
getSetReq =
qtrack_service_pb2.GetSettingsRequest(section=<PUBLISHER or SENSOR_LOCATION>)
```

```
getSetResp = stub.GetSettings(getSetReq, metadata=metadata)
print "The Get Settings response is: \n", getSetResp
```

Sample: `GetSettings` API call and response in protobuf format:

```
rpc GetSettings (GetSettingsRequest) returns (GetSettingsResponse) {}
message GetSettingsResponse {
 // SENSOR_LOCATION parameter
 message Sensor {
 string name  = 1;
 string ip  = 2;
 Vector3 position = 3;

 message Transform {
  string from_name = 1;
  string to_name = 2;
  oneof Orientation {
   Quaternion quaternion = 3;
   EulerYPR euler_ypr = 4;
  }
 }
 Transform transform = 5;
 string sensor_type = 6;
 }

 message SensorLocation {
 repeated Sensor sensor = 1; // oneof doesn't support repeated field
 }

 SettingsSection section = 1;
 oneof SettingsResponse {
 SensorLocation sensor_location = 2;
 }
}
 // Publisher parameter
  message Publisher {
  string name              = 1;
  OutputFormat format      = 2;
  uint32 port              = 3;
  bool   add_data_size     = 4;
  bool   network_byte_order = 5;
  }
  message Publishers {
    repeated Publisher publisher = 1;
  }

  SettingsSection section = 1;
  oneof SettingsResponse {
    SensorLocation sensor_location = 2;
    Publishers publishers = 3;
  }
}
```

GetSettingsTemplateList API

Used to fetch the template names for a `settings.xml` file. Available in configuration mode. Three templates can be used to create a `settings.xml` file. This API fetches the template names: `general_tracking_settings.xml`, `person_tracking_settings.xml`, and `vehicle_tracking_settings.xml`.

Send and response: `GetSettingsTemplateList`

```
rpc GetSettingsTemplateList(Empty) returns (GetSettingsTemplateResponse) {}
```

Sample

```
templReq = qtrack_service_pb2.Empty()
templResp = stub.GetSettingsTemplateList(templReq, metadata=metadata)
print "Templates are: ", templResp.settings_template
```

GetSimplifiedSettings API

`GetSimplifiedSettings` message is sent by client to request the content of simplified settings JSON file.

GetSimplifiedSettings command

```
{
  "group": "SIMPLIFIED_SETTINGS",
  "command": {
    "id": "GetSimplifiedSettings",
    "params": {}
  }
}
```

Sample response

GetSimplifiedSettings response message (Successful)

```
{
  "reply": "OK",
  "result": {
    "name": "SimplifiedSettings",
    "parameters": [
      {
        "name": "maxBinDistance",
        "value": {
          "path": "BackgroundFilter.maxBinDistance",
          "display_name": "Max Scanned Range",
          "description": "The maximum distance to which the sensor distinguishes
between the background (static points) and the foreground (moving points)."
        }
      },
      {
        "name": "useNanAsUnobstructed",
        "value": {
          "path": "BackgroundFilter.useNanAsUnobstructed",
          "display_name": "Open Space",
          "description": "Select True when it is possible that there is open space
beyond the maximum range of the lidar (e.g. in an open outdoor space) and False
when the environment constrains the maximum range of the lidar (e.g. indoors)."
        }
      },
      {
        "name": "cellSize",
        "value": {
          "path": "ClusterPipeline.CCClusterer.cellSize",
          "display_name": "Min Distance Between Objects",
          "description": "Detected objects below this distance will be merged
together."
        }
      },
      {
        "name": "minNumClusterPoints",
        "value": {
          "path": "ClusterPipeline.ClusterFilter.minNumClusterPoints",
          "display_name": "Object Detection Sensitivity",
          "description": "Sets the minimum number of points for a cluster to be
considered an object. A low value can increase sensitivity which can increase false
positives while a high value can increase the likelihood of missing objects."
        }
      },
      {
        "name": "minSize",
        "value": {
          "path": "OutputFilter.minSize",
          "display_name": "Minimum Object Size",
          "description": "Objects whose largest dimension is smaller than this are
filtered out."
        }
      },
      {
        "name": "maxSize",
        "value": {
          "path": "OutputFilter.maxSize",
          "display_name": "Maximum Object Size",
          "description": "Objects whose largest dimension is larger than this are
filtered out."
        }
      },
      {
        "name": "maxArea",
        "value": {
          "path": "OutputFilter.maxArea",
          "display_name": "Maximum Object Area",
          "description": "Objects whose area (length * width) is larger than this
are filtered out."
        }
      },
      {
        "name": "minSpeed",
        "value": {
          "path": "OutputFilter.minSpeed",
          "display_name": "Minimum Object Speed",
          "description": "Objects whose speed (m/s) is lower than this are filtered
out."
        }
      },
      {
        "name": "maxSpeed",
        "value": {
          "path": "OutputFilter.maxSpeed",
          "display_name": "Maximum Object Speed",
          "description": "Objects whose speed (m/s) is higher than this are
filtered out."
        }
      }
    ]
  }
}
```

SetSettings API

Used to set the publisher settings in the `settings.xml` file. This API is used to set the QORTEX DTC object list, zone list, sensor state and QORTEX DTC state publishing port number, format value, data size bool value, and network byte order bool value. The API response is a bool value.

`SetSettings` allows values to be set when the when the location is not running. To set values, stop the location and run `SetSettings` API. The `SetSettings` API does not allow multiple channels to be set on the same Port, this means the Port numbers must be unique.

Send and response: `SetSettings`

```
rpc SetSettings(SettingsSectionResponse) returns (RequestResult){}
```

Sample

```
def sub_setsettings():
   yield qtrack_service_pb2.SettingsSectionResponse.Publisher(name="Qortex
1_TRACKABLE_LIST", port=17161, format= "XML", add_data_size =
True, network_byte_order = True)
   yield qtrack_service_pb2.SettingsSectionResponse.Publisher(name= "Qortex
1_QORTEX_ZONE_LIST", port=17172, format= "JSON", add_data_size =
True, network_byte_order = True)
   yield qtrack_service_pb2.SettingsSectionResponse.Publisher(name=
"QTRACK_TRACK_LIST", port=17161, format= "XML", add_data_size =
True, network_byte_order = False)
   yield qtrack_service_pb2.SettingsSectionResponse.Publisher(name=
"SENSOR_HEALTH_STATE", port=17168, format= "PROTOBUF", add_data_size =
False, network_byte_order = False)
   yield qtrack_service_pb2.SettingsSectionResponse.Publisher(name= "Qortex
1_STATE", port=17178, format="NONE", add_data_size = True, network_byte_order
= True)

setsetting_req =
qtrack_service_pb2.SettingsSectionResponse.Publishers
(publisher=sub_setsettings())
requ =
qtrack_service_pb2.SettingsSectionResponse
(section="PUBLISHER",publishers=setsetting_req)
resp = stub.SetSettings(requ, metadata=metadata)
print resp.result
```

Settings API

Used to set or get the values of various parameters in the `settings.xml` file. The API has `command = SET_COMMAND` or `GET_COMMAND` and `format = XML_FORMAT` or `JSON_FORMAT`. if the format is xml, the stream of data should be in xml format. If the format is JSON, the stream of data should be in JSON format. The stream data should be formatted according to the `settings.xml` tag hierarchy.

Send and response: `Settings`

```
rpc Settings(SettingsRequest) returns (SettingsResponse) {}
```

Sample

```
setreq = qtrack_service_pb2.SettingsRequest(
 command=SET_COMMAND or GET_COMMAND>, stream=stream,
 format=<XML_FORMAT or JSON_FORMAT>)
```

```
settingresp = stub.Settings(setreq, metadata=metadata)
print "Settings response is: \n", settingresp
```

State Module

The state module includes two simple unary RPCs. State gRPC API commands enable access to a report of the status of the server, zone, or trigger-based alarm. This module has the following APIs.

GetLicenseInitState	GetRulesEnabled
GetLicenseState	GetSensorState
GetPTZEnabled	GetServerState

Sample Python code for using a State API.

```
def getserverstate(metadata, stub):
    """
    API call to get server state
    :param stub: stub object
    :param metadata: metadata to be sent
    :return: the server current state info such as server mode, version, in
        capturing, is paused, etc. .
    """
    state_req = qtrack_service_pb2.Empty()   # Creating an empty request
    state_resp = stub.GetServerState(state_req, metadata=metadata)
        # Sending the request to server via stub and getting response
    print "Sever state is :\n", state_resp
    return state_resp

def getzone metadata, stub, zonetype):
    """
    API call to get server state
    :param stub: stub object
    :param zonetype: type of zone (EVENT or EXCLUSION)
    :return: zone data such as zone name, UUID, class, vertices, etc. .. of all the zone
        (Event/Exclusion)
    """


    getzone_req = zone_requests_pb2.ZonesRequest(zone_class=zonetype)
        # Creating a request with zone_class=<required zone class>
    getzone_resp = stub.GetZones(getzone_req, metadata=metadata)
        # Sending the request to server via stub and getting response
    print "The zone info is ", getzone_resp
    return getzone_resp
```

```
if __name__ == '__main__':

    channel = grpc.insecure_channel("192.168.0.1”17177")
        # Creates an insecure Channel to a server
    stub = qtrack_service_pb2_grpc.QortexServiceStuf(channel)
        # Creating stub. Client sends a request to the server using the stub and waits for a response
        # to come back, just like a normal function call
    client_id = "82ef205c_ed14-4013-89cf-85e11b6827b1"
        # An example client ID
    metadata = (("client_id", client_id),)
        # Optional metadata to be transmitted to the service_side. Here metadata with client ID is a
        # must
```

Figure 11. Sample Python Code: State API

Each API in the status module is explained below, followed by an example of the call and response.

GetLicenseInitState, GetLicenseState API

```
rpc GetLicenseInitState(GetLicenseInitStateRequest) returns (RequestResult) {}
rpc GetLicenseState(GetLicenseStateRequest) returns (RequestResult) {}
message GetLicenseInitStateRequest {
string app_name = 1;
string app_version = 2;
int32 app_license_product_id = 3;
}
message GetLicenseStateRequest {
int32 sensor_count = 1;
string license_type = 2;
}
```

Suggest adding the number sensors per license as a custom parameter set, for example, version and tier. The implementation must allow FAEs to input the number of sensor licenses in the customer parameter field when the license is created. User defined field entry would no longer be needed.

GetPTZEnabled API

Get PTZ enabled (enabled by license). Works in either configuration or monitor mode. Requires QORTEX DTC 2.4 or later.

This API is used to get PTZ camera enabled value. The enabled value is `True` or `False`. This value is retrieved from the License of the Qortex-Server. If the `ptz_Count >0` this API returns `True`, else returns `False`. To check the License verify the command below.

```
./Qortex-Server --license info
```

Sample

```
req = qtrack_service_pb2.Empty()
resp = stub.GetPTZEnabled (req, metadata=metadata)
print "The Get PTZEnabled 'result' value is : ", resp.result
print "The Get PTZEnabled 'enabled' value is : ", resp.enable
```

Send and response

```
rpc GetPTZEnabled (Empty) returns (EnableResult) {}
```

GetRulesEnabled API

Get rules enabled (enabled by license). Works in either configuration or monitor mode. Requires QORTEX DTC 2.4 or later.

This API is used to get the rules enabled value true or false. When the Add-Ons in license is having values as `Rules` then this API respond `True`, if the `Add-Ons` value is empty in license, then this API shall respond `False`. Location has to be running in order to get the True or False response. To check the license `Add-Ons` type below command.

```
./Qortex-Server --license info
```

Sample

```
req = qtrack_service_pb2.Empty()
resp = stub.GetRulesEnabled (req, metadata=metadata)
print "The Get rule status 'result' value is : ", resp.result
print "The Get rule status 'enabled' value is : ", resp.enable
```

Send and response

```
rpc GetRulesEnabled (Empty) returns (EnableResult) {}
```

GetSensorState API

Used to get the health status such as sensor temperature, frame rate, NMEA status, and error codes, as well as sensor model, name, IP address, PTZ camera state, and connection status (`connected` or `disconnected`). The API accepts a list of sensor IP addresses and responds back with its status.

Send and response

```
rpc GetSensorState(GetSensorStateRequest) returns (GetSensorStateResponse) {}
```

Sample

```
SensorReq = qtrack_service_pb2.GetSensorStateRequest(
 sensor_ip_list=<list of sensor ip>)
 sensorResp = stub.GetSensorState(SensorReq, metadata=metadata)
print "Get sensor state response is: ", sensorResp
```

GetServerState API

Used to get the current state of the server. An empty request from the `GetServerState` API to the server produces a response stating the **server** mode (`Live` or `Playback`), settings file path, version, config client port number, monitor client port number, `is_paused` bool value (data set indicating whether it is paused or not), `is_capturing` bool value (whether a location is running live or not), and a qlog name (whether a data set is running).

Send and response

```
rpc GetServerState(Empty) returns(ServerState) {}
```

Sample

```
req=qtrack_service_pb2.Empty()
resp=stub.GetServerState(req, metadata=metadata)
print "Status request is :\n", resp
```

Streaming Module

The streaming module includes bidirectional read-write RPCs. See *Bidirectional Streaming RPCs* (page 31). These APIs send a request to the server to download or upload files with a specialized type. Each API in the streaming module is explained below, followed by an example of the call and response. The APIs for this module are:

```
GetFile
PutFile
putfile_request
```

GetFile API

Used to Get/Download a file from the server to the client. The `GetFile` API call works in monitor mode as well, so the `SwitchToConfigMode` API call does not need to run in parallel. In `GetFile` API call it is possible to download only `GENERAL_FILE` and `LOG_FILE` (`qortex.server.log`). When the API call has `FileType= "LOG_FILE"`, the server searches for the `qortex.server.log` file in the `/home/quanergy/quanergy/qortex` folder location, and if the file is found, it sends it to the client, or it produces a file-not-found error. The construction of an API call for `GetFile` is very similar to the `PutFile` API call.

1. In `GetFile` API call, Construct first level of request with file type and file name (file info) and send to server, the server responds back with total number of chunks client shall receive.

```
file_info = qtrack_service_pb2.FileChunk.FileInfo(file_type=filetype,
file_name=Getfilename)
return qtrack_service_pb2.FileChunk(file_info=file_info)
```

2. After the chunk count is received, Construct the second level of request with file info and chunk index and send to server and get the raw data as response from server.

3. After the 1st chunk of raw data is received, send next request with file info and incremented chunk index, repeat this until all the chunk index are sent and its corresponding raw data is received.

Send and response

```
rpc GetFile(stream FileChunk) returns (stream FileChunk) {}
```

If the `GetFile` points to a directory of files, such as with the user logs, a userLogs.zip file is returned.

Complete sample python code for `GetFile` API call

```
def getFileChunk(Getfile):
    get_filetype = getfiletype(Getfile)
        ## getting file type from file name (refer PutFile API function for its def
    file_info = qtrack_service_pb2.FileChunk.FileInfo(file_type=get_filetype,
file_name=Getfile)
        ## First level of request
    return qtrack_service_pb2.FileChunk(file_info=file_info)
def getFileChunk_count(Getfile):
     yield getFileChunk(Getfile)
def getfile_request(chunk_count, Getfile):
     yield getFileChunk(Getfile)
        ## Second level request sending file info
    for i in range(chunk_count):
         yield qtrack_service_pb2.FileChunk(chunk_index=
(qtrack_service_pb2.FileChunkIndex(chunk_index=i)))
            # Seconds level of request sending chunk index
def getFile(Getfile,metadata, stub):
    ##### Main Function #####
     chunk_count = 0
     items = stub.GetFile(getFileChunk_count(Getfile), metadata=metadata)
        # Send first level of request to server and get total chunk of file data, calling getFileChunk_count
function
    for item in items:
        chunk_count_str = str(item)
        try:
            chunk_count_str = chunk_count_str.split("\n")[4]
            chunk_count = int(filter(str.isdigit, chunk_count_str))
                # Extracting chunk count from the received response
        except:
            pass
        print "The total chunk count to be received is : {} \n
\n".format(chunk_count)
    getfileout = stub.GetFile(getfile_request(chunk_count, Getfile),
metadata=metadata, timeout=50000)
        ## Seconds level of request with chunk count(chunk index) and file info
    for resp, i in zip(getfileout, range(-1,chunk_count+2)):
        ## Getting raw data response
    print resp
        # Printing response
    text = str(resp).split("\n")[1].strip()
        # separating raw data snd chunk index and printing chunk index
    if i == 0 or i == -1 or "data" in text or "chunk_count" in text:
        continue
    else:
        searchText = "chunk_index: " + str(i)
    if searchText in text:
        print ('correct chunk index received : {}\n \n'.format(text))
    else:
        print "inconsitent chunk index"
```

PutFile API

Used to put a file or upload a file from the client host to the server host. `PutFile` API needs the client to be in config mode, so the `SwitchToConfigMode` API should be running in parallel when calling a `PutFile` API stream. `PutFile` API uploads only general file (txt), `qguard.settings.ini` file, and the `calibration.xml` file to a default `/home/quanergy/quanergy/qortex` filepath. This folder location cannot be changed.

Send and response

```
rpc PutFile(stream FileChunk) returns (stream FileChunkIndex) {}
```

Sample

```
CALIBRATION_FILE = 0;
  /* can only be uploaded from client to server; for location setup purpose */
GUARD_SETTINGS_FILE = 1;
  /* can only be uploaded from client to server; for location setup purpose */
LOG_FILE = 2;
  /* can only be downloaded from server to client */
RECORDING_QLOG_FILE = 3;
  /* can only be downloaded from server to client */
GENERAL_FILE = 4;
  /* can be downloaded/uploaded from/to server to/from client, such as text file, deb package, zip file, etc.. */
```

putfile_request API

Uploads QORTEX DTC `settings.ini` file to locations folders and changes the file name to `qortex.settings.temp.ini`, then uploads `calibration.xml` file to the location folder and changes the file name to `qortex.calib.temp.xml`. A target file name cannot be set for calibration file and `qguard.settings.ini` file. Uploading a file with `PutFile` API requires the following process:

1. **The client side checks** if the file exits and can be opened; if so, get the file name and total chunks to send; total chunks = file_size/chunk size for each message (by default 16 kb, but it can be 32 kb or 64 kb). Split the file into these smaller chunks. Now the split file has n number of chunks for an n chunk index. For example, a 128 kb file is split into smaller chunks with each chunk size at 16 kb, so there are 8 chunks of data, so chunk index is 8.
2. **Construct first level of request** with

```
FileChunk.FileInfo(
file_type=
   <CALIBRATION_FILE | QGUARD_SETTINGS_FILE | LOG_FILE |
   RECORDING_QLOG_FILE |GENERAL_FILE>,
file_name=<TargetFileName>,
chunk_count=<number of chunks>
)
```

   Then send this first level of request to server.

```
inside putfile_request() function
 file_info = qtrack_service_pb2.FileChunk.FileInfo(
 file_type='CALIBRATION_FILE', chunk_count=8)
 yield qtrack_service_pb2.FileChunk(file_info=file_info)
```

3. **Construct a second level of request** with the split data/chunk data along with its chunk index and send them to the server.

```
inside putfile_request() function
dataList contains the split data as list format
for data, i in zip(dataList, range(8)):
 print "The value of i is ", i
 print "starting chunk {} .......\n".format(i)
 raw_data = qtrack_service_pb2.FileChunk.RawData(
  data=data, chunk_index=i)
 yield qtrack_service_pb2.FileChunk(raw_data=raw_data)
 print "finished chunk {} .......\n".format(i)
```

4. **The server sends a response** with the expected chunk index that the client is going to send (should start with 0). Client then reads 16 KB binary data from the file and puts them into the request together with the chunk index to send. Meanwhile, the client checks if all chunks have been sent (expected_chunk_index >= total chunk index). If so, the client closes the connection. (This step may repeat multiple times depending on the total size to upload.)

```
items = stub.PutFile(putfile_request(dataList, filePath), metadata=metadata)
 for item in items:
 print "the response is {} \n".format(item)
```

Sample PutFile and putfile_request code. If the code is not formatted properly, check if there may be missing tab or multiple spaces/tab.

```
def defaultfiletype(fileExt):
   if fileExt == "xml":
      filetype = "CALIBRATION_FILE"      # 0
   elif fileExt == "ini":
      filetype = "QGUARD_SETTINGS_FILE" # 1
   elif fileExt == "log":
      filetype = "LOG_FILE"             # 2
   elif fileExt == ".q*":
      filetype = "RECORDING_QLOG_FILE"  # 3
   else:
      filetype = "GENERAL_FILE"         #  4
   return filetype
def getfiletype(filename):
   try:
      filenameExt = filename.rsplit(".", 1)[1]
      get_file_type = defaultfiletype(filenameExt)
      return get_file_type
   except Exception as err:
   print err
def read_in_chunks(file_object, chunk_size=1024):
   # first function call from main
   """Default chunk size: 1k."""
   while True:
      dataOut = file_object.read(chunk_size)
   if not dataOut:
      break
   yield dataOut
def putfiletype(file_Path):
   # function call from putfile_request function
   try:
      TargetFileName = file_Path.rsplit("/", 1)[1]
         # from the file path, getting the file name
   filenameExt = (file_Path.rsplit("/", 1)[1]).split(".")[1]
      # from file path getting file extension eg .txt file or . ini file or . xml
file, etc..
   put_file_type = defaultfiletype(filenameExt)
      # Getting the file type (Calibration_file or Log_File etc..) from the file
extension
   return put_file_type, TargetFileName
      # return the file type and file name to caller function
   except Exception as err:
   print err
def putfile_request(dataList,filePath):
   # Second function call from main
   numOfChunk = len(dataList)
      # From dataList list get total number chunks
   put_file_type, TargetFileName = putfiletype(filePath)
```

```
        # With putfiletype function, extract the file type and file name from the
filepath provided
    file_info = qtrack_service_pb2.FileChunk.FileInfo (file_type=put_file_type,
file_name=TargetFileName, chunk_count=numOfChunk)
        # constructing first level of request which has fileinfo, filename and total
chunk count
    yield qtrack_service_pb2.FileChunk(file_info=file_info)
        # Send the fileinfo request to server
    for data, i in zip(dataList, range(numOfChunk)):
        # from the dataList and total chunk count, iterate each data chunk and its
chunk index and send them to server one by one
    print "The value of i is ", i
    print "starting chunk {} .......\n".format(i)
        raw_data =
qtrack_service_pb2.FileChunk.RawData(data=data,chunk_index=i)
            # constructing request to send chunked data and its index to server
    yield qtrack_service_pb2.FileChunk(raw_data=raw_data)
        # sending chunked data and data index to server as raw data
    print "finished chunk {} .......\n".format(i)

def putFile(filePath, metadata, stub):
    ######## Main function #########
    dataList = []
        # Create empty list to append the split file content/data
    data = open(filePath)
        # Open the file to read the content
    for piece in read_in_chunks(data, 16000):
        # call read_in_chunks functions which splits the file content in to 16kb
    dataList.append(piece)
        # Add the split file content in dataList list
    items = stub.PutFile(putfile_request(dataList, filePath), metadata=metadata)
        # send the constructed request to server and get chunk index as response
    for item in items:
    print "the response is {} \n".format(item)
```

Zone Configuration APIs

Here is a list of Zone Configuration APIs, followed by an explanation of each one and an example of the call and response.

```
AddZone                     ZoneViolation
EditZone                      ChangeEventZoneViolationAction
ObjectStitching               ChangeEventZoneViolationTrigger
GetZones                      GetEventZoneViolationActions
```

```
GetZonesEnabled                 GetEventZoneViolationHandlerEnabled
RemoveZone                      GetEventZoneViolationRecordingEnabled
SetZonesEnabled                 SetEventZoneViolationRecordingEnabled
```

AddZone API

Used to add zone information into a settings file. The Event, Execution, or Inclusion zone information such as zone name, unique id, dimensions, height, type, and enabled parameters are sent via this API to the server and are written to the `settings.xml` file. Creating a zone requires at least three vertices (3 X and Y values). Every zone has a unique UUID, a UUID is 128-bit Universally Unique ID value such as `a1ebb897-ed68-4af9-8f62-cac86512db2b`. Each zone of a certain class should have a unique UUID value. Duplicating UUID values between different `Zone_Class` is allowed (the UUID value of an Event zone can be used as the UUID value for an Exclusion zone), but that is not recommended. When the `Zone_Class` parameter is used to reference a zone that is either an Event zone, Exclusion zone, or Inclusion zone, its values are `EVENT`, `EXCLUSION`, or `INCLUSION`. `Zmin` and `Zmax` represent the minimum/start and maximum/stop of zone height values. The enabled value represents that the zone is visible in the UI client. If false, the zone is not visible in the QORTEX DTC client UI. When the `Zone_Class` parameter is selected as Exclusion zone, then an extra parameter is added to select an exclusion zone sensor.

Send and response: `AddZone`

```
rpc AddZone (AddZoneRequest) returns (RequestResult) {}
```

Sample

```
vertices_list=[]
    # Creating empty list to store zone vertices / dimensions
zone_info =
 zone_3d_pb2.QZone(name=str(name), type= "POLYGON")
    # creating request with zone name and zone type
 for x, y in zip(X_vertices_list, Y_vertices_list):
    # iterating loop of x and y values of zone vertices, a minimum of 3 x and y values are required
 vertices_list.append(qortex_geometric_types_pb2.Vector2(x=float(x),
 y=float(y)))
    # Adding vertices (x and y) to vector and creating request
Zone_vertices =
 qortex_geometric_types_pb2.Polygon2D(vertices=vertices_list)
    # creating a request with vertices list for polygon2
zone_vert =
 zone_3d_pb2.QPolygonGeometry(vertices=Zone_vertices)
    # creating a request with vertices list for QPolygonGeometry
zone_params =
 zone_3d_pb2.QZone3D(uuid=str(UUID), zone=zone_info,
 z_min=float(zMin), z_max=float(zMax), zone_class=str(ZoneClass),
 enabled=str_to_bool(Enabled), polygon_3d=zone_vert,
    # constructing complete zone request with all information such as name, uuid, vertices, type, etc.
zone_req =
 zone_requests_pb2.AddZoneRequest(zone=zone_params)
    # Constructing AddZone request
req_resp = stub.AddZone(zone_req, metadata=metadata)
    # sending addzone request to server and getting bool result as response
print "Add Zone response is: ", req_resp.result
```

EditZone API

Used to edit an already existing or added zone. The `EditZone` API identifies a zone with the help of UUID, so, to edit an existing zone in a `settings.xml` file, the UUID of that zone should be provided. The edit zone request construction is the same as the `AddZone` request construction shown above, but it also calls the `EditZone` API when sending a request to the server. It is not possible to edit the UUID value of a zone, and a new UUID value represents a new zone. Using a UUID that does not exist in a settings xml file produces an error for the `EditZone` API.

Send and response: `EditZone`

```
rpc EditZone (QZone3D) returns (RequestResult) {}
```

Sample

```
req_resp =
 stub.EditZone(zone_params, metadata=metadata)
    # Edit Zone API request with zone_params request construction similar to add Zone API request
construction
print "Edit Zone response is: ", req_resp.result
    # Sending Edit Zone API request to server and getting bool response
```

GetZones API

Used to retrieve all the available zone information from the `settings.xml` file. It is used to retrieve all Event, Exclusion, or Inclusion zones.

```
getzone_req =
 zone_requests_pb2.ZonesRequest(zone_class=<zone_class>)
    # Construct zonerequest with zone class as either 'EVENT' or 'EXCLUSION' or ‘INCLUSION’
getzone_resp =
 stub.GetZones(getzone_req, metadata=metadata)
    # Send the request to server and get zone information as response
print(getzone_resp)
```

GetZonesEnabled API

Used to get the value of enabled property from the `settings.xml` file. If the enabled value is true under the zone in the settings xml file, then the API response is true. If the enabled value is false under the zone in the settings xml file, then the API response value is false.

Send and response: `GetZonesEnabled`

```
rpc GetZonesEnabled (ZonesRequest) returns (ZonesEnableStatus) {}
```

Sample

```
getzoneEnbReq =
 zone_requests_pb2.ZonesRequest(zone_class=zone_class)
getZoneEnbResp =
 stub.GetZonesEnabled(getzoneEnbReq, metadata=metadata)
print "Get Zone Enabled value is: ", getZoneEnbResp.enable
```

RemoveZone API

Used to remove a zone with its associated information from the `settings.xml` file. The remove zone request identifies the zone with its UUID value and the `Zone_Class` value of the zone and removes it.

Send and response: `RemoveZoneRequest`

```
rpc RemoveZone (RemoveZoneRequest) returns (RequestResult) {}
```

Sample

```
zone_info=
 zone_requests_pb2.RemoveZoneRequest(zone_class=ZoneClass, uuid=UUID)
    # Construct a remove zone request with zone_class and UUID
RemoveZoneresp =
 stub.RemoveZone(zone_info, metadata=metadata)
    # send the request to server and get bool response
print "Remove Zone response is: ", RemoveZoneresp.result
```

SetZoneEnabled API

Used to enable (make visible) an EVENT/EXCLUSION zone in QORTEX DTC client UI. If the enabled value is true, the zone is visible in the QORTEX DTC client UI. If the enabled value is false, the zone is not visible in the QORTEX DTC client.

Send and response: `SetZoneEnabled`

```
rpc SetZoneEnabled (EnableZonesRequest) returns (RequestResult) {}
```

Sample

```
setzoneEnbReq =
 zone_requests_pb2.EnableZonesRequest(zone_class=zone_class,
 enable=eval(bool_enable))
    # construct a request with zone class and enable value
setZoneEnbResp =
 stub.SetZonesEnabled(setzoneEnbReq, metadata=metadata)
    # send the request to server and get bool response
```

```
print "Set Zone Enabled response is: ", setZoneEnbResp.result
```

ZoneViolation APIs

ZoneViolations listed as deprecated apply to older versions of Qortex DTC.

GetEventZoneViolationRecordingEnabled API

Used to get the bool value of the Event zone violation recording enabled parameter from settings xml file.

Send and response: `GetEventZoneViolationRecordingEnabled`

```
rpc GetEventZoneViolationRecordingEnabled (Empty) returns (ViolationRecodingStatus)
{}
```

Sample

```
req = qtrack_service_pb2.Empty()
resp = stub.GetEventZoneViolationRecordingEnabled(
 req, metadata=metadata)
print "The Get Event zone violation recording result value is : ",
 resp.result
print "The Get Event zone violation recording enabled value is : ",
 resp.enable
```

SetEventZoneViolationRecordingEnabled API

Used to set the Event zone recording enabled value true or false. If the Event zone violation recording Enabled value is True, then if a violation occurs, the recording starts. But if its value is False, then if a violation occurs, the recording will not start.

Send and response: `SetEventZoneViolationRecordingEnabled`

```
rpc SetEventZoneViolationRecordingEnabled (ViolationRecodingStatus) returns
(RequestResult) {}
```

Sample

```
requ = zone_requests_pb2.ViolationRecodingStatus(enable=<Bool value>)
resp = stub.SetEventZoneViolationRecordingEnabled(
 requ, metadata=metadata)
print "The set Event zone violation recording enabled response is: ",
 resp.result
```

4. Cybersecurity

QORTEX DTC supports user authentication and data encryption. This is an optional mode for the QORTEX DTC server. By default, **Security** mode is `off`. Enable it through the QORTEX DTC client in **Configuration** mode.

- **HTTP Digest** authentication protocol—Internet access is not required. Certificate of Authority (CA) is not required. Authentication setting applies for the duration of the QORTEX DTC client session.
- **AES-256** encryption—The object list, zone list and sensor health APIs are encrypted.

The QORTEX DTC cybersecurity API enables the entire data flow to be controlled and secured upon request. The flow is controlled with events initiated by the user and delivered through signal-slot connections.

When Security mode is enabled, user authentication is required for the following actions:

- Connecting the QORTEX DTC client to the QORTEX DTC server
- Connecting any third-party client to the QORTEX DTC server
- Sending and receiving gRPC API calls.
- Reading QORTEX DTC API (such as zone list and sensor health). Anytime an application is reading the QORTEX API TCP ports. The Quanergy TCP listener includes HTTP digest and AES-256 support. See *QORTEX Q-Track DTC User Guide.*

To enable the QORTEX DTC server **Security** feature, install QORTEX DTC server and Debian, and activate or refresh the Quanergy license.

- Every user can secure or unsecure the entire QORTEX DTC client to server system to enable or disable the secured data flow.
    - First user logging in starts the secured encrypted data flow with the new security token.
    - Last user logging out stops the secured encrypted data flow and drops the security token.
    - The QORTEX DTC data session is secured in between new security token created and dropped.

QORTEX HTTP Server

The QORTEX HTTP server provides a command line interface for all QORTEX DTC clients connecting to the QORTEX DTC server. The URL is `https://<IP_Address>:8080/v1/<login>`.

HTTP Digest for Authentication

Hiding credentials. The HTTP authentication *digest* principle enables you to hide all or part of the access credentials. Most commonly, the username is sent by plain text but the password with the client uses calculated MD5 (or other) hash against which the server. The server compares its own hash calculated using the same formula. See Wikipedia article, Digest access authentication.

Two-phased request. HTTP authentication is a two-phased version of HTTP GET request. Briefly, a first phase triggers the server to a nonce (number once value) to enforce the password hash. In the second phase the client sends the password hash with the HTTP GET. The server checks and confirms both client and server hashes match, then the server replies with a standard `HTTP 200 OK` response and completes internal actions to accommodate the client secure session.

QORTEX implementation. In QORTEX DTC, we use custom HTTP header for additional bi-directional communication. The client always sends `Q-Auth` with the command and parameters delimited by (`:`) colon. The server sends the reply as `Q-*`. Where `*` can be multiple variables filled with data depending on the command and its succession.

Application-Level Commands

The following describes the QORTEX DTC custom client headers and server replies.

Every command is sent though its HTTP header. One command per request. Authentication can be with either License ID/ password or username/ password. Recommended preference is to use username authentication and use the license/ID as a backup.

GetSecured Command

Sending the `Q-Auth` header with the `GetSecured` request does not require authentication. The HTTP response sets the `Q-Secured` header to either `true` or `false`. A `true` value means QORTEX DTX is now secured. This means, authenticate the client with `GetToken`.

The `GetSecured` request also sends `Q-UserPresent` header. The HTTP response is either `true` or `false`. A `true` value means the username is set up in the system. The License ID/ License Password is always available, but it is intended for administrative operations.

GetToken Command

Sending the `Q-Auth` header with the `GetToken` request requires authentication. The HTTP response is either `get HTTP 401K unauthenticated`, which means authentication failed due to wrong credentials or the `Q-Token` header is set to a 64-character hex token. This token is then unhexed to AES 256 key. QORTEX DTC Python and C++ TCP Listeners are used for the actual data decryption. Due to listener traits, make sure the of the corresponding port, using `<TCPPublisher><AddDataSize>true</AddDataSize></TCPPublisher>`. The QORTEX DTC client does not need the port configuration for ports in use.

SetSecured Command

Sending the `Q-Auth` header with the `SetSecured:true` or `SetSecured:false` requires authentication. Depending on `true` or `false` after `SetSecured:` the server decides whether to secure the system for `true` or unsecure it for `false`. The HTTP response is `Q-SetSecured` header set to either `ok` or `failed`.

SetCredentials Command

Sending the `Q-Auth` header with the `SetCredentials:username:encryptedPassword`, for encrypted password, See *HTTP Password Encryption Formula* (page 100). The QORTEX DTC client enforces username complexity now of 4 or more characters including both letters and digits. The user password complexity is of 6 or more characters and both capital and lowercase letters plus digits required. The HTTP response for `Q-CredentialsSet` can be `true` or `false`. A `true` value indicates the request was successful. This HTTP response includes a security token in `Q-Token` header, as well. See *QORTEX Q-Track DTC User Guide*.

Authentication Example

The `QAuthenticator` object is used via the slot for `QNetworkAccessManager::authenticationRequired` signal with `QORTEX:`

QORTEX example authentication

```
/// Basic semantics for passing the credentials [client side code callback]
QNetworkAccessManager nam;
    connect(&nam_, &QNetworkAccessManager::authenticationRequired,
        this, &HttpClient::authRequired);

void HttpClient::authRequired(QNetworkReply* reply, QAuthenticator* authenticator)
{
    Q_UNUSED(reply)
    qDebug() << "Authentication required: " << authenticator->realm();
        // we get realm@quanergy

    authenticator->setUser(user_);
        // Set the user name [or License ID for configuration]
    authenticator->setPassword(pwd_);
        // Set the user password [or License password for configuration]
}
```

Client Authentication Python Example

```
import sys
import requests
import os
import binascii
import socket
import struct
import uuid
import hashlib
import json
from requests.auth import HTTPDigestAuth
from Crypto.Cipher import AES
from Crypto.Util import Counter

def https_authentication(server_ip, username, password):
    #Get authentication url
    url = 'http://' + server_ip + ':8080/v1/login'
    r = requests.get(url)
    print "url", url
    print 'r', r
    #Get another request with authentication header added
    r = requests.get(url, auth=HTTPDigestAuth(username, password),
headers={'Q-Auth': 'GetToken'})
    #Error proofing if url is down or if authentication failed
    if r.status_code != 200:
        print("Could not connect to url, check ip address")
        return False, False
     else:
        if "Authorized" in r.text:
            qtoken = r.headers['Q-Token']
            return r, qtoken
        else:
            print("Could not authenticate, check username or password")
            return False, False
def get_secured(server_ip):
    #Get authentication url
    url = 'http://' + server_ip + ':8080/v1/login'
    r = requests.get(url)
    #Get another request with authentication header added
    r = requests.get(url, headers={'Q-Auth': 'GetSecured'})
    secure_flag = r.headers['Q-Secured']
    return r, secure_flag
def read_tcp(server_ip, port, packet_size):
    # Get data stream
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server_ip, port))
    msg = s.recv(packet_size)
    return msg
def get_msg_size(msg, uuid_obj):
    """
     - Msg and uuid_obj is extracted from the tcp socket
     - The next 8 bytes after the uuid_obj is the msg size represented as
bytes
```

```
     - Split msg with uuid, get next 8 bytes, unpack back to an int
     - For the decryption module IV = md5hash(bytes(msg_size^2))
     - Return the Square the msg_size in byte form
     """
    #Split byte string from uuid and rest of the msg
    split = msg.split(uuid_obj.bytes)
    #Get the first 8 bytes of the second chunk of msg
    msg_size_byte = split[1][0:8]
    ciphertext = split[1][8:]
    #Decode the byte string into an integer to square before converting back into
bytes
    msg_size = struct.unpack("Q", msg_size_byte)
    msg_size_square = msg_size[0]**2
    msg_size_square_bytes = struct.pack("Q", msg_size_square)
    return msg_size_square_bytes, ciphertext, msg_size_byte
def int_of_string(s):
    return int(binascii.hexlify(s), 16)
def decrypt_message(key, msg_size_square, ciphertext):
    """
     - AES CTR requires a key and counter, here counter and IV can be used
interchangeably
     - Key = Qtoken in its 32byte binary for
     - IV = md5hash(byte(msg_size_square)) which is 16 bytes
     - counter needs a ctr object
     - Counter.new(nbits, other params), nbits is the number of bits.
128bits = 16bytes
     """
    #Unhex qtoken to byte form, and return byte form of md5hash for IV
    key = key.decode('hex')
    iv = hashlib.md5(msg_size_square)
    iv_hash = iv.digest()
    #Initialize counter with the int_of_string of the iv, then decrypt
    ctr = Counter.new(128, initial_value=int_of_string(iv_hash))
    aes = AES.new(key, AES.MODE_CTR, counter=ctr)
    decrypted_msg = aes.decrypt(ciphertext)
    return iv, int_of_string(iv_hash),decrypted_msg
def encrypt_message(key, msg_size_square, ciphertext, uuid_obj,
msg_size_byte):
    """
     - AES CTR requires a key and and counter, here counter and IV can be
used interchangeably
     - Key = Qtoken in its 32byte binary for
     - IV = md5hash(byte(msg_size_square)) which is 16 bytes
     - counter needs a ctr object
     - Counter.new(nbits, other params), nbits is the number of bits.
128bits = 16bytes
```

```
    """
    #Unhex qtoken to byte form, and return byte form of md5hash for IV
    key = key.decode('hex')
    iv = hashlib.md5(msg_size_square)
    iv_hash = iv.digest()
    #Initialize counter with the int_of_string of the iv, then decrypt
    ctr = Counter.new(128, initial_value=int_of_string(iv_hash))
    aes = AES.new(key, AES.MODE_CTR, counter=ctr)
    encrypted_msg = aes.encrypt(ciphertext)
    #Add in the uuid and byte size headers to the front of the encrypted string
    encrypted_msg = uuid_obj.bytes + msg_size_byte + encrypted_msg
    return iv, int_of_string(iv_hash),encrypted_msg
def main():
    """
    - Get qortex to enable sercure mode
    - Connect to https authentication server with username and password
    - If authentication is successful, a QToken will be returned as a
header
    - For decryption, we need the key and the msg_size
    - The key is the QToken returned from https authentication, will be a
hex represenation of a string
    - The msg_size will be part of the data stream returned from TCP
    - Secured TCP data stream format: [separator_uuid: 16
bytes][orig.size: 8 bytes][encrypted data]
    - TCP data stream will be an encrypted byte string
    - The uuid will be 3debc6c9-08cc-7a02-9af0-1a082c39f4d2 in byte form
    - The msg size will be the next 8 bytes after the uuid
    - After obtaining both the Qtoken msg size we are ready for AES CTR
decryption
    """
    username = 'user1'
    password = 'Letmein433'
    port = 17161
    server_ip = '127.0.0.1'
    packet_size = sys.maxint / 10000000000
    uuid_str = '3debc6c9-08cc-7a02-9af0-1a082c39f4d2'
    uuid_obj = uuid.UUID('{' + uuid_str +'}')
    # while True:
    # Ping Qortex HTTPS server with credentials to receive Q-Token
    r_authentication, qtoken = https_authentication(server_ip, username,
password)
    r_secure, secure_flag =  get_secured(server_ip)
    #Get Data Stream from TCP
    msg = read_tcp(server_ip, port, packet_size)
    #Find msg size
    msg_size_square, ciphertext, msg_size_byte = get_msg_size(msg, uuid_obj)
    #AES CTR Decryption
    iv, int_of_string, decrypted_msg = decrypt_message(qtoken,
msg_size_square, ciphertext)
    # AES CTR Encryption
    iv, int_of_string, encrypted_msg = encrypt_message(qtoken,
msg_size_square, decrypted_msg, uuid_obj, msg_size_byte)
    #Test
    print msg
    print(msg, type(msg), len(msg))
    print("\n")
    # print("Secure status:", secure_flag)
    print("UUID bytes (Hex/Bytes):", uuid_str, uuid_obj.bytes)
    print("Msg Square Bytes:", msg_size_square)
    print("Qtoken (Hex/Bytes):", qtoken, qtoken.decode('hex'))
    print("IV (Hex/Bytes)", iv.hexdigest(), iv.digest())
    print("Int of string", int_of_string)
    print decrypted_msg
    # test = decrypted_msg[4:]
    # print test
    # print decrypted_msg[1]
    # json.loads(test)
    # print("Reencrypted msg", encrypted_msg, type(encrypted_msg),
len(encrypted_msg))
    # print("Org msg == Reencrypted Msg?:", msg == encrypted_msg)
if __name__ == "__main__":
    main()
```

Client Q-Auth HTTP Request Header Example

Set the request with custom `Q-Auth` header with QORTEX:

QORTEX example requests

```
/// Basic semantics for all requests
QNetworkAccessManager nam;
QNetworkReply* request(QString command) {
   QNetworkRequest req("http://127.0.0.1/v1/login");
   req.setRawHeader("Q-Auth", command);
   return nam.get(req);
 }
request("GetSecured");
   // Request if system is secured [does not need credentials]
request("SetSecured",
   // Request to set system secured [requires either user or license credentials]
   true);
      // {true/ false} where "true" is to set the secure mode and "false" to
unsecure
```

```
request("SetCredentials"
    // Request to set new user credentials [requires either user or license
credentials]
        we only have one user and we replace its credentials]
    "New_UserName_:New_Password_AES_128_CBC_Hash");
        // The encryption is described below
request("GetToken");
    // Request user security token [requires user credentials]
    // Will get reply as [256 bit key as hex string, will be used with AES 256 CTR
encryption]
```

Brief Server HTTP Reply Header Example

Set the reply with a few `key : value` pairs.

QORTEX example response handling

```
/// Basic spec for passing the server replies back to the client
void HttpClient::requestDone()
{
    QString errMsg = QStringLiteral("Unknown reason");
    if (reply_ && reply_->isFinished()) {
        QByteArray qbData = reply_->readAll();
        reply_->deleteLater();
        qDebug() << "HTTP reply body:" << qbData;
        QString token = reply_->rawHeader("Q-Token");
        QString secured = reply_->rawHeader("Q-Secured");
        QString setSecured = reply_->rawHeader("Q-SetSecured");
        QString credSet = reply_->rawHeader("Q-CredentialsSet");
        if (!token.isEmpty())
            emit securityTokenUpdate(uuid_token_ = token);
        if (!secured.isEmpty())
            emit securedState(is_secured_ = secured == "true");
        if (!setSecured.isEmpty())
            emit setSecuredSuccessful(setSecured == "ok");
        if (!credSet.isEmpty())
            emit credentialsSet(credSet == "true");
        auto httpResponseCode = reply_->attribute(
QNetworkRequest::HttpStatusCodeAttribute );
        if (!httpResponseCode.isValid() || httpResponseCode.toInt() != 200) {
            QString reason = reply_->attribute(
QNetworkRequest::HttpReasonPhraseAttribute ).toString();
            emit lastRequestFailed(reason);
        }
        return;
    }
    emit securityTokenUpdate(uuid_token_ = QString());
    qWarning() << "HTTP Digest failed due to" << errMsg;
}
```

HTTP Password Encryption Formula

The expectation is that encryption is primarily managed by the application library. QORTEX DTC handles, the HTTP digest MD5 combined hash and has only one explicit encryption in the client code for this part of functionality: Encrypting the new password for `Q-Auth:SetCredentials` request header.

For example, an unsecured `Q-Auth` request header value might be: `quser:433LakeSide`, and encrypted version would be: `quser:19eb639bb525204842e7e7daeb3fbc69`.

The hash formula is:

New password encryption formula

```
const auto& userNameHex = crypto::MD5Hash(user);
const auto& pwdHex = crypto::MD5Hash(pwd);
crypto::EncryptHelper<crypto::AES_128_CBC_Cipher> dh(userNameHex, pwdHex);
const auto& newPwdCrypted = dh.encode(newPwd.toStdString());
```

The encryption with MD5 is always 128-bit value and while not reversible to the source it is same for same source. AES CBC 128 bit accepts both `key` and `input vector` as 128-bit hashes while processing `newPwd`. This ensures the return is one possible decryption for the same value. This is really hard to break as there is no old password in plain text sent via HTTP. The QORTEX DTC server only confirms the old password by comparing the expression hash with the QORTEX DTC client HTTP digest result hash.

HTTP User Security Options

Use these as part of `SetCredentials`.

```
DeleteCredentials(string admin_hash, string username) -> {success/failed}
EnableUser(string admin_hash, string username, bool on) -> {success/failed}
EditGlobalSettings(admin_hash,
{JSON for multiple variables}
)
EditUserSettings(admin_hash, user_name,
{JSON for user role maybe more}
)
ListUsers(admin_hash)
```

QORTEX DTC TCP Listener with Security Enabled

The QORTEX DTC TCP listener uses the `getToken` API for decrypting secured messages. All other decryption handling is through the QORTEX DTC client. Qortex DTC uses AES 256 and AES 128 encryption.

- AES 256 CTR—for encrypting TCP data and decrypting per the TCP listener's example.
- AES 128 CBC—for encrypting new passwords sent from the client.

TCP Listener Requirements

To use the Qortex DTC TCP listener, install the following:

- CMake 2.8.1 or later
- Boost 1.66 or later

Sample TCP Listener Actions

All the examples will use 0.0.0.0 as sample host IP as sample port number.

To start the listener for detected objects, run the following command for Qortex-Server:

```
 $ ./qortex_listener -h 0.0.0.0 -p 17161 -t qtrack
```

To start the listener for defined zone list, run the following command for Qortex-Server:

```
 $ ./qortex_listener -h 0.0.0.0 -p 17172 -t qzone
```

To start the listener for current sensor status, run the following command for Qortex-Server:

```
 $ ./qortex_listener -h 0.0.0.0 -p 17178 -t qstate
```

To start the listener for point cloud, run the following command for Qortex-Server:

```
 $ ./qortex_listener -h 0.0.0.0 -p 17173 -t pointcloud
```

To start the listener for current sensor state list, run the following command for Qortex-Server:

```
$ ./qortex_listener -h 0.0.0.0 -p 17168 -t qsensorstate
```

To start the listener for detected objects with authentication using username (user) and password (pwd), run the following command for Qortex-Server:

```
$ ./qortex_listener -h 0.0.0.0 -p 17161 -t qtrack -u user -w pwd
```

Sample HTTP TCP Listener Encryption

```
/***************************************************************
 **                                                          **
 **  Copyright(C) 2021 Quanergy Solutions. All Rights Reserved.**
 **  Contact: http://www.quanergy.com                        **
 **                                                          **
 ***************************************************************/
#include <string>
// The HttpClient class below provides a way to run HTTP requests
// against Qortex Server to obtain a security token
class HttpClient
```

```
{
public:
   /** Construction */
   HttpClient();
   /** Return the instance of this class */
   static HttpClient* instance();
  /**
   * @brief login with the following info
   * @param user user name
   * @param pwd password
   */
   void logIn(const std::string& user, const std::string& pwd);
  /**
   * @brief setServerIp sets HTTP Server IP which expected to be same as gRPC
   *       and TCP publisher's and we also assume the port will be 8080 or
   *       "HTTP Alternative" standard port which we control ourselves.
   * @param ip IP address
   * @param port Port number
   */
   void setServerIp(const std::string& ip, int port = 8080);
   bool isSecured() const { return is_secured_; }
  /**
   * @brief securityToken
   * @return the previously retrieved security token
   */
   const std::string& securityToken() const { return uuid_token_; }
protected:
  /**
   * @brief getToken initialize HTTP Digest Authentication for obtaining
   *       an authentication token
   * @param user user name
   * @param pwd password
   */
   void getToken(const std::string& user, const std::string& pwd);
  /**
   * @brief httpGetRequest call GET request with the given credentials
   * @param user user name
   * @param pwd password
   * @param auth given Q-Auth command
   */
   void httpGetRequest(const std::string& user, const std::string& pwd,
                       const std::string& auth);
   bool is_secured_{ false };
   std::string url_;
   std::string user_;
   std::string pwd_;
   std::string uuid_token_;
   static HttpClient* instance_;
private:
};
```

Sample TCP Listener API Python

```
/*************************************************************
 **                                                         **
 **  Copyright(C) 2017 Quanergy Solutions. All Rights Reserved.**
 **  Contact: http://www.quanergy.com                       **
 **                                                         **
 *************************************************************/
#ifndef QGUARD_TCP_LISTENER_H_
#define QGUARD_TCP_LISTENER_H_
#include <vector>
#include <boost/asio.hpp>
#include <crypto_helper.h>
class QortexTcpListener
{
  static constexpr std::size_t HEADER_LENGTH = sizeof(std::uint32_t);
  static constexpr std::size_t UUID_LENGTH = 16;
  static constexpr std::size_t RAW_DATA_BUFFER_LENGTH = 1024 * 1024;
  static constexpr std::size_t MAX_EXPECTED_DATA_LENGTH = 10 * 1024 * 1024;
public:
  enum MessageType
  {
    QOBJECT = 0,
    QZONE = 1,
    QSTATE = 2,
    QTRACKABLE = 3,
    POINTCLOUD = 4,
    QSENSORSTATE = 5
  };
  using ByteVector = std::vector<std::uint8_t>;
public:
 /**
   * @brief constructor
   * @param[in] boost::asio::io_service object as a reference
   *           boost::asio::ip::tcp::resolver
   *           message_type -- is enum of type QOBJECT, QZONE, QSTATE, QTRACKABLE,
POINTCLOUD
   */
  QortexTcpListener(boost::asio::io_service& io_service,
                    boost::asio::ip::tcp::resolver::iterator
endpoint_iterator,
                    MessageType message_type,
                    const std::string& user,
                    const std::string& pwd);
  /**
   * @brief close closes the tcp connection
   */
  void close();
private:
  /**
   * @brief connect connects to tcp endpoint specified by endpoint_iterator.
   *        calls the readHeader handler if connection was established
   * @param[in] endpoint_iterator tcp endpoint to connect to
   */
  void connect(boost::asio::ip::tcp::resolver::iterator endpoint_iterator);
  /**
   * @brief readHeader reads the first 4 (HEADER_LENGTH) bytes of the recieved
   *        message to obtain the message body length. Calls readBody handler
   *        if body length is non-zero.
   */
  void readHeader();
  /**
   * @brief readBody reads the 4th+(body_length_) bytes of the received message,
   *        tries to deserialize the message to a QObjectArray protobuf object
   *        and prints out the objects. Otherwise, prints out the received bytes
   *        as plain text (helpful in case of json/xml formats)
   */
  void readBody();
  /** @brief readData and do decryption if necessary */
  void tryReadEncrypted(const std::vector<uint8_t>& new_data);
  /** @brief do an async read operation */
  void doRead();
  /**
   * @brief byteArrayToInt converts a four-byte char array into an int32.
   * @param[in] source the four-byte char byte-array
   * @return a 32bit integer represented by the byte-array
   */
  std::size_t byteArrayToInt(std::array<char, HEADER_LENGTH> source );
  /** @brief find UUID to set isEncrypted flag */
  int findUuid(const std::vector<uint8_t>& data, bool& isEncrypted) const;
  /** @brief request authentication token */
  bool requestToken(std::string& token);
  /** @brief reset decryptor with the given token */
  void resetDecryptor(const std::string& token);
  void readHeader(const std::vector<uint8_t>& decoded);
  void readBody(const std::vector<uint8_t>& decoded);
  void parseProtobuf(const void* databuf);
```

```
private:
  boost::asio::io_service& io_service_;
  boost::asio::ip::tcp::socket socket_;
  std::size_t body_length_ = 0;
  std::vector<uint8_t> data_;
  ByteVector tcp_buffer_;
  ByteVector slack_;
  /**
   * @brief  message_type is an enum of type QOBJECT, QZONE, QSTATE, QTRACKABLE,
POINTCLOUD
   */
  MessageType message_type_;
  /**
   * @brief Incoming data stream order of data size is cached for referencing in
the class.
   */
  bool reversed_byte_order_ = {true};
  std::string user_;
  std::string password_;
  boost::asio::ip::tcp::tcp::resolver::iterator endpoint_iterator_;
  std::shared_ptr<
    quanergy::crypto::TcpDecryptHelper<quanergy::crypto::AES_256_CTR_Cipher,
ByteVector>
  > decryptor_;
  std::array<std::uint8_t, UUID_LENGTH> uuid_buf_sep_;
};
#endif // #QGUARD_TCP_LISTENER_H_
```

Sample TCP Listener API C++

HTTP Authentication and getToken

```
//--------------------------------------------------------------------------------
// :: HTTP Authentication and get Token (uuid_token_) [see qortex_http_client.cpp]
//--------------------------------------------------------------------------------

void HttpClient::logIn(const std::string& user, const std::string& pwd)
{
  getToken(user, pwd);
}
//--------------------------------------------------------------------------------

void HttpClient::getToken(const std::string& user, const std::string& pwd)
{
  uuid_token_.clear();
  if (user.empty() || pwd.empty() || url_.empty()) {
    std::cerr << "Cannot authenticate due to empty username/password/URL\n";
    return;
  }
  // Auth is GetToken
  httpGetRequest(user, pwd, "GetToken");
}
//--------------------------------------------------------------------------------
// Send GET request with Auth "GetToken"
void HttpClient::httpGetRequest(const std::string &user, const std::string &pwd,
                                const std::string &auth)
{
  if (user.empty() || pwd.empty() || url_.empty()) {
    std::cerr << "Cannot authenticate due to empty username/password/URL\n";
    return;
  }
  static const int timeout_msec = 2000;
  user_ = user;
  pwd_ = pwd;
  int result = -1;
  std::string reason;
  boost::asio::io_context io_context;
  try
  {
    // Create client.
    auto client = std::make_shared<simple_http::get_client>(
      io_context,
      [&result, &reason, this](simple_http::empty_body_request& req,
                               simple_http::string_body_response& resp)
      {
        result = resp.result_int();
        if (resp.result_int() != 200)
        {
          reason = std::string(resp.reason());
          std::cerr << "Failed response " << resp << "\n";
        } else {
          std::cout << "Success response " << resp << "\n";
          uuid_token_ = std::string(resp.base()["Q-Token"]);
          is_secured_ = true;
        }
      });
    client->setAuthorization(user_, pwd_);
    // Optionally set a fail action
    client->setFailHandler(
      [&result, &reason](const simple_http::empty_body_request& req,
                         const simple_http::string_body_response& resp,
                         simple_http::fail_reason fr,
                         boost::string_view message)
      {
        result = 0;
        reason = std::string(message);
      });
    client->get(simple_http::url(url_));
    // Run this HTTP conversation up to the time limit specified
    io_context.run_for(std::chrono::milliseconds(timeout_msec));
    if (result == 200)
    {
      std::cout << "Request completed, response: " << result << "\n";
    }
    else // Other response code
    {
      // Although there are other successful responses that are not 200,
      // this is likely a failure, log as such.
      std::cerr << "Request failed, response: " << result << "\n";
      if (result == -1)
      {
        std::cerr << "Can't complete request: Connection timed out.\n";
      }
      // Log other error message [as TCP or another protocol failed]
      else if (result == 0)
      {
        std::cerr << "Can't complete request: " << reason << "\n";
      }
    }
  }
  catch(const std::exception& e)
  {
    std::cerr << "Can't complete authentication request: " << e.what() << '\n';
  }
}
```

TCP Listener requestToken

```
//--------------------------------------------------------------------------------
// :: TCP Listener request Token and decrypt message [see qortex_tcp_listener.cpp]
//--------------------------------------------------------------------------------
bool QortexTcpListener::requestToken(std::string& token)
{
  std::cout << "Getting Token ..." << std::endl;
  auto* httpClient = HttpClient::instance();
  const auto& ip = endpoint_iterator_->endpoint().address().to_string();
  httpClient->setServerIp(ip);
  httpClient->logIn(user_, password_);
  token = httpClient->securityToken();
  return !token.empty();
}
//--------------------------------------------------------------------------------
// Create decryptor_ which is a crypto class that can decrypt message
void QortexTcpListener::resetDecryptor(const std::string& tokenHexStr)
{
  if (!tokenHexStr.empty())
  {
    std::array<std::uint8_t, 32> key_arr;
    std::vector<std::uint8_t> token =
quanergy::util::unhex<std::uint8_t>(tokenHexStr);
    std::copy(token.begin(), token.end(), key_arr.data());
    // Setup decryptor
    decryptor_ = std::make_shared<
      quanergy::crypto::TcpDecryptHelper<quanergy::crypto::AES_256_CTR_Cipher,
ByteVector>>
      ( key_arr, uuid_buf_sep_ );
    std::cout << "decryptor created" << std::endl;
  }
  else
  {
    decryptor_.reset();
  }
}
//--------------------------------------------------------------------------------
// Reading encrypted data and call the handle: tryReadEncrypted(encrypted_data)
void QortexTcpListener::doRead()
{
  socket_.async_read_some(
    boost::asio::buffer(tcp_buffer_),
    [this](boost::system::error_code ec, std::size_t bytes_transferred) {
      if (!ec)
      {
        tcp_buffer_.resize(bytes_transferred);
        // Signal that data has arrived
        ByteVector data = std::move(tcp_buffer_);
        tcp_buffer_.resize(RAW_DATA_BUFFER_LENGTH);
        tryReadEncrypted(data);
        // Read more data
        doRead();
      }
      else if (ec != boost::asio::error::operation_aborted)
      {
        socket_.close();
      }
    });
}
//------------------------------------------------------------------------------
// Find UUID to see if buffer is encrypted
int QortexTcpListener::findUuid(const std::vector<uint8_t>& data, bool&
isEncrypted) const
{
  auto it = std::search(data.cbegin(), data.cend(),
                        uuid_buf_sep_.begin(), uuid_buf_sep_.end());
  if (it != data.cend()) {
    isEncrypted = true;
    // std::cout << "The uuid " << uuid_buf_sep_str_ << " found at offset "
    //            << it - slack_.begin() << std::endl;
    return it - data.cbegin();
  }
  isEncrypted = false;
  return -1;
}
//------------------------------------------------------------------------------
void QortexTcpListener::tryReadEncrypted(const std::vector<uint8_t>& new_data)
{
  try {
    bool encrypted = false;
    slack_.insert(slack_.end(), new_data.begin(), new_data.end());
    auto start_of_uuid = findUuid(slack_, encrypted);
    if (encrypted && slack_.size() > start_of_uuid + UUID_LENGTH + sizeof(size_t))
{
      ByteVector decoded_data;
      const std::uint8_t* encr_ptr = slack_.data() + start_of_uuid + UUID_LENGTH;
      const size_t& count = reinterpret_cast<const size_t&>(*(encr_ptr));
    // Check if the decryption possible to avoid the crash
      if (count > 0 && count <= slack_.size() - UUID_LENGTH) {
     // here we pass the address of [size][encrypted_bytes] portion of data
        decoded_data = decryptor_->decode_with_md5_counter_iv(encr_ptr);
        if (!decoded_data.empty()) {
      // Cut the processed data out. The original size is maybe a bit less than
      // encrypted but the algorithm rely on next UUID to be found anyway
          int old_size = start_of_uuid + UUID_LENGTH + sizeof(size_t) +
decoded_data.size();
          int new_size = slack_.size() - old_size;
          ByteVector clone = std::move(slack_);
          if (new_size > 0) {
            slack_.resize(new_size);
            std::copy(clone.cbegin() + old_size, clone.cend(), slack_.begin());
          }
      // Dispatch for further processing
```

```
           readHeader(decoded_data);
         }
       }
     }
     else {
       readHeader(slack_);
       slack_.clear();
     }
   }
   catch (std::exception& exc) {
     std::cerr << "QortexTcpListener::readData: " << exc.what() << std::endl;
     slack_.clear();
   }
   catch (...) {
     std::cerr << "QortexTcpListener::readData (unknown exception)" << std::endl;
     slack_.clear();
   }
   if (slack_.size() > MAX_EXPECTED_DATA_LENGTH) {
     std::cerr << "Could not find encryption separator for too long" << std::endl;
     slack_.clear();
   }
}
//*******************************************************************************
```

5. Template Parameter Settings

QORTEX DTC 2.4 offers three templates for optimizing the configuration of the software to track person, vehicle, or general (mixture). See *QORTEX Q-Track DTC User Guide*. Beyond these default optimizations, the settings can be fine-tuned for particular needs by changing the parameter values through the gRPC API (below) or through the client user interface. *Table 4. Sensor Settings Parameters for StaticCloudPipeline*, *Table 5. Sensor Settings Parameters for ClusterPipeline*, *Table 6. Sensor Settings Parameters for TrackerPipeline*, *Table 7. Sensor Settings Parameters for OutputFilter*, and *Table 8. Sensor Settings Parameters for MultiLidarPipeline* list the parameter values.

Changes via gRPC API

One way to change the parameter values is by editing the xml file through Configuration gRPC API commands from the terminal. The following is an example of using the gRPC API when updating parameter settings values.

1. Specify the `<ClusterPipeline><CCClusterer><cellSize>` parameter settings value and the data format (`json` or `xml`). For example:

   For xml:

```
    stream = "<Settings>
    <ClusterPipeline>
    <CCClusterer>
    <cellSize>0.6</cellSize>
    </CCClusterer>
    </ClusterPipeline>
    </Settings>"
```

   For json:

```
stream = "{"Settings": {"ClusterPipeline": {"CCClusterer": {"cellSize": 0.6}}}}"
```

2. Use proto format for the settings API

```
// The new Settings Getter/Setter API
enum SettingsCommand {
 SettingsCommand_UNSPECIFIED = 0;
 SET_COMMAND = 1;
 GET_COMMAND = 2;
}
enum SettingsFormat {
 SettingsFormat_UNSPECIFIED = 0;
 XML_FORMAT = 1;
 JSON_FORMAT = 2;
}
```

```
message SettingsRequest {
 SettingsCommand command = 1;
 bytes stream   = 2;
 SettingsFormat format = 3;
}

Note (for message SettingRequest):
command = SET_COMMAND or GET_COMMAND
stream = json or xml data
format = XML_FORMAT or JSON_FORMAT (use appropriate format because wrong format
matching, such as stream = xml and format = json, may not set or get data.
```

Changes via Client User Interface

Another way to change the parameter values is by editing fields in the **Sensor** tab.

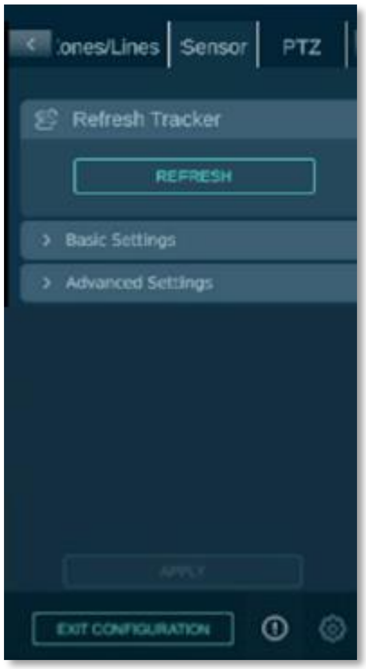


Figure 12. Settings: Sensor Tab

An example portion of the parameter settings is shown with default values in the client user interface and the xml file.

qtrack_custer_algorithm_settings_general.xml

```
<ClussterPipeline>
   <CCClusterer>
<!-- Size of cells (cells are square). -->
      <cellSize>
         <value>0.40<value>
         <min>0.1</min>
```

```
          <max>1.0</max>
          <type>double</type>
       </cellSize>
    </CCClusterer>
    <ClusterFilter>
<!-- Clusters with fewer than this many points are discarded. -->
       <minNumClusterPoints>
          <value>10<value>
          <min>1</min>
          <max>20</max>
          <type>int</type>
       </minNumClusterPoints>
<!-- Clusters whose width and length are smaller than this are discarded. -->
       <minSize>
          <value>0.2<value>
          <min>0.1</min>
          <max>2.0</max>
          <type>double</type>
       </minSize>
<!-- Clusters are discarded if either its width or length exceeds this. -->
       <maxSize>
          <value>100.0<value>
          <min>1.0</min>
          <max>100.0</max>
          <type>double</type>
       </maxSize>
</ClusterFilter>
<ClusterSplitter>
<!-- Switch to enable/disable cluster splitter. -->
<enable>
```

Figure 13. Settings: User Interface (top) and Template File (bottom) examples

Parameter Settings Values

The following tables list the parameters available for a variety of pipelines and filters, and a description of each. Each table includes recommended settings for each parameter with regard to General, Person, or Vehicle sensing (the three columns on the far left of each table), as well as the minimum and maximum value the parameter accepts. The parameter types include double (accepts decimal values) or int (accepts only non-decimal values). Unlabeled parameter types accept only true or false values.

StaticCloudPipeline parameters

`StaticCloudPipeline` filters background and Exclusion zone points and then passes filtered foreground points to next pipeline. *Table 4. Sensor Settings Parameters for StaticCloudPipeline* describes each `BackgroundFilter` parameter and its recommended settings.

Table 4. Sensor Settings Parameters for StaticCloudPipeline

`Section, Parameter`	**Description**	**General**	**Person**	**Vehicle**
`StaticCloudPipeline` `BackgroundFilter` `maxBinDistance`	Maximum distance for voxelizing in a radial direction in meters.	value: 100 min: 50 max: 200 type: int	value: 50 min: 50 max: 200 type: int	value: 10 min: 50 max: 200 type: int
`StaticCloudPipeline` `BackgroundFilter` `distanceBinLength`	The bin size in the radial direction.	value: 1.0 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.5 min: 0.1 max: 2.0 type: double
`StaticCloudPipeline` `BackgroundFilter` `timeUntilBackground`	Maximum amount of time (in seconds) for an occupied location to go from considered as foreground to considered as background. If an object enters an unoccupied location and stays there, it fades into the background in this amount of time (less time if location was recently occupied).	value: 150 min: 10 max: 10000 type: int	value: 150 min: 10 max: 10000 type: int	value: 500 min: 10 max: 10000 type: int
`StaticCloudPipeline` `BackgroundFilter` `timeUntilForeground`	Maximum amount of time (in seconds) a location remains unoccupied for an object entering it to be considered foreground. If an object remains in an unoccupied space for this amount of time, the object is guaranteed to be considered foreground (may require less time if previous occupations were short).	value: 300 min: 10 max: 10000 type: int	value: 300 min: 10 max: 10000 type: int	value: 8000 min: 10 max: 10000 type: int
`StaticCloudPipeline` `BackgroundFilter` `ForegroundLock`	Makes a location a permanent free space location once observed as free space for a certain consecutive amount of time	value: false option: true option: false	value: false option: true option: false	value: false option: true option: false

Section, Parameter	Description	General	Person	Vehicle
`StaticCloudPipeline` `BackgroundFilter` `timeUntilAlwaysForeground`	Time (in seconds) a location must be observed as free space to be considered free space permanently. An object moving into a free space location is considered to be in the foreground.	value: 10 min: 5 max: 50 type: int	value: 10 min: 5 max: 50 type: int	value: 15 min: 5 max: 50 type: int
`StaticCloudPipeline` `BackgroundFilter` `useNanAsUnobstructed`	If true, NaN points are ignored and not used. If false, NaN indicates free space (to infinity) at that point (e.g., if NaN meets maximum range and nothing was detected). Use Exclusion Zones to adjust for false foregrounds, if needed.	value: false option: true option: false	value: false option: true option: false	value: true option: true option: false
`StaticCloudPipeline` `BackgroundFilter` `updateEveryNthFrame`	Sets frequency at which the background filter updates. Value of 1: updates occur every frame. Value of 2: updates occur every 2nd frame, etc. Changing this value affects times in other settings. For example, if value is 10, time moves ten times slower in the background filter, since it counts frames to determine time.	value: 5 min: 1 max: 10 type: int	value: 5 min: 1 max: 10 type: int	value: 5 min: 1 max: 10 type: int

ClusterPipeline parameters

`ClusterPipeline` groups filtered for group points into clusters, using a different algorithm. *Table 5. Sensor Settings Parameters for ClusterPipeline* describes each `CCClusterer`, `ClusterFilter`, and `ClusterSplitter` parameter and its recommended settings.

Table 5. Sensor Settings Parameters for ClusterPipeline

Section, Parameter	Description	General	Person	Vehicle
`ClusterPipeline` `CCClusterer` `cellSize`	Size of cells. These cells are square.	value: 0.40 min: 0.1 max:1.0 type: double	value: 0.15 min: 0.1 max:1.0 type: double	value: 0.5 min: 0.1 max:1.0 type: double
`ClusterPipeline` `ClusterFilter` `minNumClusterPoints`	Clusters with fewer than this number of points are discarded.	value: 10 min: 1 max: 20 type: int	value: 10 min: 1 max: 20 type: int	value: 8 min: 1 max: 20 type: int

`ClusterPipeline ClusterFilter minSize`	Clusters whose area (width * length) is smaller than this are discarded.	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.3 min: 0.1 max: 2.0 type: double
`ClusterPipeline ClusterFilter maxSize`	Clusters whose area (width * length) is larger than this are discarded.	value: 100.0 min: 1.0 max: 100.0 type: double	value: 5.0 min: 1.0 max: 100.0 type: double	value: 100.0 min: 1.0 max: 100.0 type: double
`ClusterPipeline ClusterSplitter enable`	Enables or disables cluster splitter.	value: false option: true option: false	value: false option: true option: false	value: false option: true option: false
`ClusterPipeline ClusterSplitter onlySplitPersonClass`	Only splits person classes. Setting this to true disables all current splitting, as cluster class labels are unknown.	value: false option: true option: false	value: false option: true option: false	value: true option: true option: false
`ClusterPipeline ClusterSplitter onlySplitMultiLidar`	Only splits multi-lidar clusters. Setting this to true disables splitting of single-lidar clusters.	value: true option: true option: false	value: true option: true option: false	value: true option: true option: false
`ClusterPipeline ClusterSplitter minStandardDeviation`	Minimum standard deviation threshold above which the object is split. This value is the standard deviation of cluster points (in meters).	value: 0.15 min: 0.01 max: 5.0 type: double	value: 0.15 min: 0.01 max: 5.0 type: double	value: 0.15 min: 0.01 max: 5.0 type: double
`ClusterPipeline ClusterSplitter threshStandardDeviation`	Threshold standard deviation threshold each split object should satisfy. This value is the standard deviation of cluster points (in meters).	value: 0.45 min: 0.01 max: 5.0 type: double	value: 0.45 min: 0.01 max: 5.0 type: double	value: 0.45 min: 0.01 max: 5.0 type: double
`ClusterPipeline ClusterSplitter maxStandardDeviation`	Maximum standard deviation threshold below which the object is split. This value is the standard deviation of cluster points (in meters).	value: 1.00 min: 0.01 max: 5.0 type: double	value: 1.00 min: 0.01 max: 5.0 type: double	value: 1.00 min: 0.01 max: 5.0 type: double
`ClusterPipeline ClusterSplitter minNumPoints`	Minimum number of points an object contains required for split.	value: 5 min: 1 max: 100 type: int	value: 5 min: 1 max: 100 type: int	value: 5 min: 1 max: 100 type: int

TrackerPipeline parameters

`TrackerPipeline` associates clusters with existing trackables. It then updates existing trackables or generates new trackables. *Table 6. Sensor Settings Parameters for TrackerPipeline* describes each `DataAssociation`, `Tracker`, `TrackableFilter`, `TrackableDimensionsUpdater`, `TrackableProximityFinder`, and `TrackableFilter` parameter and its recommended settings.

Table 6. Sensor Settings Parameters for TrackerPipeline

Section, Parameter	Description	General	Person	Vehicle
`TrackerPipeline` `frequency`	Frequency (in Hz) at which the tracker pipeline outputs trackables.	value: 10 min: 1 max:20 type: int	value: 10 min: 1 max:20 type: int	value: 10 min: 1 max:20 type: int
`TrackerPipeline` `DataAssociation` `enableKdTreeLookup`	Enables k-d tree lookup to speed data association when there are high numbers of clusters and trackables.	value: true option: true option: false	value: true option: true option: false	value: true option: true option: false
`TrackerPipeline` `DataAssociation` `maxAssociationRadius`	Clusters and trackables must be closer than this Euclidean distance (as measured from their centroid positions, in meters) to be associated.	value: 4.00 min: 0.0 max: 10.0 type: double	value: 1.00 min: 0.0 max: 10.0 type: double	value: 5.00 min: 0.0 max: 10.0 type: double
`TrackerPipeline` `DataAssociation` `maxAssociationDistance`	Clusters and trackables must be closer than this distance to be associated. Note that overlapping clusters and trackables have a negative distance.	value: 1.5 min: 0.0 max: 5.0 type: double	value: 0.3 min: 0.0 max: 5.0 type: double	value: 2.0 min: 0.0 max: 5.0 type: double
`TrackerPipeline` `DataAssociation` `maxMergeClusterAssociationDistance`	Clusters and trackables must be closer than this distance to have an additional cluster-trackable association after the first one. Only applies to complex data association. Overlapping clusters and trackables have a negative distance.	value: 1.0 min: 0.0 max: 5.0 type: double	value: 0.0 min: 0.0 max: 5.0 type: double	value: 2.0 min: 0.0 max: 5.0 type: double

Section, Parameter	Description	General	Person	Vehicle
`TrackerPipeline DataAssociation numWorkerThreads`	The number of worker threads on which to run classification. In environments with many objects, this value can be increased to improve the rate at which objects get classified if the server has enough available cores.	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int
`TrackerPipeline DataAssociation defaultMeasurementVarianceHeading`	Default heading variance for measurements.	value: 0.07 min: 0.007 max: 0.07 type: double	value: 0.07 min: 0.007 max: 0.07 type: double	value: 0.07 min: 0.007 max: 0.07 type: double
`TrackerPipeline Tracker numWorkerThreads`	The number of worker threads on which to run classification. In environments with many objects, this number can be increased to improve the rate at which objects get classified if the server has enough available cores.	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int
`TrackerPipeline Tracker initVariancePosX`	Initial position variance of a new trackable in the X-direction.	value: 0.1 min: 0.1 max: 0.2 type: double	value: 0.1 min: 0.1 max: 0.2 type: double	value: 0.2 min: 0.1 max: 0.2 type: double
`TrackerPipeline Tracker initVariancePosY`	Initial position variance of a new trackable in the Y-direction.	value: 0.1 min: 0.1 max: 0.2 type: double	value: 0.1 min: 0.1 max: 0.2 type: double	value: 0.2 min: 0.1 max: 0.2 type: double
`TrackerPipeline Tracker initVariancePosZ`	Initial position variance of a new trackable in the Z-direction.	value: 0.1 min: 0.1 max: 0.2 type: double	value: 0.1 min: 0.1 max: 0.2	value: 0.2 min: 0.1 max: 0.2

Section, Parameter	Description	General	Person	Vehicle
			type: double	type: double
`TrackerPipeline Tracker initVarianceVelX`	Initial velocity variance of a new trackable in the X-direction.	value: 10 min: 10 max: 400 type: int	value: 10 min: 10 max: 400 type: int	value: 400 min: 10 max: 400 type: int
`TrackerPipeline Tracker initVarianceVelY`	Initial velocity variance of a new trackable in the Y-direction.	value: 10 min: 10 max: 400 type: int	value: 10 min: 10 max: 400 type: int	value: 400 min: 10 max: 400 type: int
`TrackerPipeline Tracker initVarianceVelZ`	Initial velocity variance of a new trackable in the Z-direction.	value: 10 min: 10 max: 10 type: int	value: 10 min: 10 max: 10 type: int	value: 10 min: 10 max: 400 type: int
`TrackerPipeline Tracker initVarianceAccX`	Initial acceleration variance of a new trackable in the X-direction.	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int
`TrackerPipeline Tracker initVarianceAccY`	Initial acceleration variance of a new trackable in the Y-direction.	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int
`TrackerPipeline Tracker initVarianceAccZ`	Initial acceleration variance of a new trackable in the Z-direction.	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int	value: 1 min: 1 max: 1 type: int
`TrackerPipeline Tracker initVarianceHeading`	Initial heading variance of a new trackable.	value: 2.46 min: 2.46 max: 2.46 type: double	value: 2.46 min: 2.46 max: 2.46 type: double	value: 2.46 min: 2.46 max: 2.46 type: double
`TrackerPipeline Tracker initVarianceHeadingRate`	Initial heading variance rate of a new trackable.	value: 0.01 min: 0.01	value: 0.01	value: 0.01

Section, Parameter	Description	General	Person	Vehicle
		max: 0.01 type: double	min: 0.01 max: 0.01 type: double	min: 0.01 max: 0.01 type: double
`TrackerPipeline Tracker processNoiseVariancePosX`	Process noise variance of the position in the X-direction.	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.04 min: 0.01 max: 0.04 type: double
`TrackerPipeline Tracker processNoiseVariancePosY`	Process noise variance of the position in the Y-direction.	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.04 min: 0.01 max: 0.04 type: double
`TrackerPipeline Tracker processNoiseVariancePosZ`	Process noise variance of the position in the Z-direction.	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.01 min: 0.01 max: 0.04 type: double	value: 0.01 min: 0.01 max: 0.04 type: double
`TrackerPipeline Tracker processNoiseVarianceVelX`	Process noise variance of the velocity in the X-direction.	value: 0.05 min: 0.05 max: 0.1 type: double	value: 0.05 min: 0.05 max: 0.1 type: double	value: 0.1 min: 0.05 max: 0.1 type: double
`TrackerPipeline Tracker processNoiseVarianceVelY`	Process noise variance of the velocity in the Y-direction.	value: 0.05 min: 0.05 max: 0.1 type: double	value: 0.05 min: 0.05 max: 0.1	value: 0.1 min: 0.05 max: 0.1

Section, Parameter	Description	General	Person	Vehicle
			type: double	type: double
`TrackerPipeline Tracker processNoiseVarianceVelZ`	Process noise variance of the velocity in the Z-direction.	value: 0.1 min: 0.1 max: 0.1 type: double	value: 0.1 min: 0.1 max: 0.1 type: double	value: 0.1 min: 0.1 max: 0.1 type: double
`TrackerPipeline Tracker processNoiseVarianceAccX`	Process noise variance of the acceleration in the X-direction.	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double
`TrackerPipeline Tracker processNoiseVarianceAccY`	Process noise variance of the acceleration in the Y-direction.	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double
`TrackerPipeline Tracker processNoiseVarianceAccZ`	Process noise variance of the acceleration in the Z-direction.	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double	value: 1e–6 min: 1e–6 max:1e–6 type: double
`TrackerPipeline Tracker processNoiseVarianceHeading`	Process noise variance of the heading.	value: 1e–3 min: 1e–3 max:1e–3 type: double	value: 1e–3 min: 1e–3 max:1e–3 type: double	value: 1e–3 min: 1e–3 max:1e–3 type: double

Section, Parameter	Description	General	Person	Vehicle
`TrackerPipeline TrackableFilter minSize`	Trackables whose largest dimension is smaller than this are filtered out.	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.3 min: 0.1 max: 2.0 type: double
`TrackerPipeline TrackableFilter maxSize`	Trackables whose largest dimension is larger than this are filtered out.	value: 100.0 min: 1.0 max: 100.0 type: double	value: 5.0 min: 1.0 max: 100.0 type: double	value: 100.0 min: 1.0 max: 100.0 type: double
`TrackerPipeline TrackableFilter maxArea`	Trackables whose area (length * width) is larger than this are filtered out.	value: 100.0 min: 1.0 max: 1000.0 type: double	value: 25.0 min: 1.0 max: 1000.0 type: double	value: 100.0 min: 1.0 max: 1000.0 type: double
`TrackerPipeline TrackableFilter maxSpeed`	Trackables whose speed (m/s) is higher than this are filtered out.	value: 50.0 min: 2.0 max: 100.0 type: double	value: 20.0 min: 2.0 max: 100.0 type: double	value: 50.0 min: 2.0 max: 100.0 type: double
`TrackerPipeline TrackableFilter maxPositionStdDev`	Trackables whose position standard deviation along any axis is larger than this are filtered out.	value: 1.0 min: 1.0 max: 5.0 type: double	value: 1.0 min: 1.0 max: 5.0 type: double	value: 2.0 min: 1.0 max: 5.0 type: double
`TrackerPipeline TrackableDimensions Updater sensitivity`	When receiving a new dimension measurement of a trackable, adjusts the old dimensions by this fraction of the measurement only.	value: 0.1 min: 0.1 max: 0.5 type: double	value: 0.1 min: 0.1 max: 0.5 type: double	value: 0.3 min: 0.1 max: 0.5 type: double

Section, Parameter	Description	General	Person	Vehicle
`TrackerPipeline TrackableProximityFinder enable`	Enables or disables proximity finder. This module helps when working with vehicles but is not beneficial when tracking people. Additionally, it has a high computational cost. As such, some users can disable it for people tracking.	value: true option: true option: false	value: false option: true option: false	value: true option: true option: false
`TrackerPipeline TrackableProximityFinder maxDistance`	The maximum distance between two trackables for them to be considered in proximity of each other (i.e., to be considered for merging).	value: 0.5 min: 0.5 max: 5.0 type: double	value: 0.5 min: 0.5 max: 5.0 type: double	value: 2.0 min: 0.5 max: 5.0 type: double
`TrackerPipeline TrackableProximityFinder frameInterval`	Only process every nth frame of data, where n = frameInterval.	value: 10 min: 1 max: 10 type: int	value: 10 min: 1 max: 10 type: int	value: 3 min: 1 max: 10 type: int
`TrackerPipeline TrackableSplitter enable`	Enables or disables trackable splitter.	value: false option: true option: false	value: false option: true option: false	value: false option: true option: false
`TrackerPipeline TrackableSplitter onlySplitPersonClass`	Splits person classes only. Setting this to true for TrackableSplitter splits the current pedestrian trackable only.	value: true option: true option: false	value: false option: true option: false	value: true option: true option: false
`TrackerPipeline TrackableSplitter onlySplitMultiLidar`	Splits multi-lidar trackables only. Setting this to true for TrackableSplitter disables splitting of single-lidar trackables.	value: true option: true option: false	value: true option: true option: false	value: true option: true option: false
`TrackerPipeline TrackableSplitter minStandardDeviation`	Minimum standard deviation threshold above which an object is split. This is the standard deviation of cluster points (in meters).	value: 0.15 min: 0.01 max: 5.0	value: 0.45 min: 0.01	value: 0.15 min: 0.01

Section, Parameter	Description	General	Person	Vehicle
		type: double	max: 5.0 type: double	max: 5.0 type: double
`TrackerPipeline TrackableSplitter threshStandardDeviation`	Threshold standard deviation threshold each split object should satisfy. This is the standard deviation of cluster points (in meters).	value: 0.45 min: 0.01 max: 5.0 type: double	value: 0.45 min: 0.01 max: 5.0 type: double	value: 0.45 min: 0.01 max: 5.0 type: double
`TrackerPipeline TrackableSplitter maxStandardDeviation`	Maximum standard deviation threshold below which an object is split. This is the standard deviation of cluster points (in meters).	value: 1.00 min: 0.01 max: 5.0 type: double	value: 1.00 min: 0.01 max: 5.0 type: double	value: 1.00 min: 0.01 max: 5.0 type: double
`TrackerPipeline TrackableSplitter minNumPoints`	Minimum number of points in an object required for split.	value: 5 min: 1 max: 100 type: int	value: 5 min: 1 max: 100 type: int	value: 5 min: 1 max: 100 type: int

OutputFilter parameters

`OutputFilter` filters trackables according to a defined rule. _Table 7. Sensor Settings Parameters for OutputFilter_ describes each `OutputFilter` parameter and its recommended settings.

Table 7. Sensor Settings Parameters for OutputFilter

Section, Parameter	Description	General	Person	Vehicle
`OutputFilter mustBeClassified`	Trackables classified as “unknown” are filtered out.	value: false option: true option: false	value: false option: true option: false	value: false option: true option: false
`OutputFilter minMeasCount`	Trackables not measured at least this many times are filtered out.	value: 2 min: 1 max: 5 type: int	value: 2 min: 1 max: 5 type: int	value: 2 min: 1 max: 5 type: int

Section, Parameter	Description	General	Person	Vehicle
`OutputFilter` `minClusterPoints`	Trackables whose latest measurement did not contain at least this number of points are filtered out.	value: 2 min: 1 max: 20 type: int	value: 2 min: 1 max: 20 type: int	value: 2 min: 1 max: 20 type: int
`OutputFilter` `minSize`	Trackables whose largest dimension is smaller than this are filtered out.	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: double
`OutputFilter` `maxSize`	Trackables whose largest dimension is larger than this are filtered out.	value: 100.0 min: 1.0 max: 100.0 type: double	value: 100.0 min: 1.0 max: 100.0 type: double	value: 100.0 min: 1.0 max: 100.0 type: double
`OutputFilter` `maxArea`	Trackables whose area (length * width) is larger than this are filtered out.	value: 100.0 min: 1.0 max: 1000.0 type: double	value: 25.0 min: 1.0 max: 1000.0 type: double	value: 100.0 min: 1.0 max: 1000.0 type: double
`OutputFilter` `minSpeed`	Trackables whose speed (m/s) is lower than this are filtered out.	value: 0.0 min: 0.0 max: 5.0 type: double	value: 0.0 min: 0.0 max: 5.0 type: double	value: 0.0 min: 0.0 max: 5.0 type: double
`OutputFilter` `maxSpeed`	Trackables whose speed (m/s) is higher than this are filtered out.	value: 50.0 min: 2.0 max: 100.0 type: double	value: 20.0 min: 2.0 max: 100.0 type: double	value: 50.0 min: 2.0 max: 100.0 type: double
`OutputFilter` `maxPositionStdDev`	Trackables whose position standard deviation along any axis is larger than this are filtered out.	value: 2.0 min: 1.0 max: 5.0 type: double	value: 2.0 min: 1.0 max: 5.0 type: double	value: 2.0 min: 1.0 max: 5.0 type: double

MultiLidarPipeline parameters

`MultiLidarPipeline` is used when multiple sensors are operating. Single sensor data goes through previous pipelines separately, then it is combined into a multi-lidar pipeline that fuses the cluster points from the single-lidar pipelines. `MultiLidarPipeling` also recalculates clustering and tracking. *Table 8. Sensor Settings Parameters for MultiLidarPipeline* describes `MultiLidarPipeling` and other parameters, and their recommended settings.

Table 8. Sensor Settings Parameters for MultiLidarPipeline

Section, Parameter	Description	General	Person	Vehicle
`MultiLidarPipeline` `numWorkerThreads`	The number of worker threads on which to run classification. In environments with many objects, this value can be increased to improve the rate at which objects get classified, provided that the server has enough available cores.	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int	value: 2 min: 1 max: 10 type: int
`MultiLidarPipeline` `useSingleLidarTracking`	When enabled, uses Kalman filter models for single-lidar pipelines to predict a fused point cloud. If disabled, skips fused point cloud processing and uses single-lidar clusters and global Kalman filter models directly. Disabling helps process a high number of trackables at the cost of smaller bounding boxes for people, while generating some jitter in visualization.	value: true option: true option: false	value: true option: true option: false	value: true option: true option: false
`MultiLidarPipeline` `useRegulator`	Regulates data flow when useSingleLidarTracking is true. May slow down object list publishing rate when handling high number of trackables, but reduces jitter in Visualizer. Enabling has no effect when useSingleLidarTracking is true.	value: true option: true option: false	value: true option: true option: false	value: true option: true option: false
`MultiLidarPipeline` `TrackableFilter`	Trackables coming from the single-lidar pipeline pass through these filters before their points are used to construct the multi-lidar point cloud.			

Section, Parameter	Description	General	Person	Vehicle
`MultiLidarPipeline TrackableFilter minMeasCount`	Trackables not measured at least this number of times are filtered out.	value: 2 min: 1 max: 5 type: int	value: 2 min: 1 max: 5 type: int	value: 5 min: 1 max: 5 type: int
`MultiLidarPipeline TrackableFilter minClusterPoints`	Trackables whose latest measurement does not contain at least this number of points are filtered out.	value: 2 min: 1 max: 20 type: int	value: 2 min: 1 max: 20 type: int	value: 5 min: 1 max: 20 type: int
`MultiLidarPipeline TrackableFilter minSize`	Trackables whose largest dimension is smaller than this are filtered out.	value: 0.2 min: 0.1 max: 2.0 type: double	value: 0.2 min: 0.1 max: 2.0 type: int	value: 0.3 min: 0.1 max: 2.0 type: int
`MultiLidarPipeline TrackableFilter maxArea`	Trackables whose area (length * width) is larger than this are filtered out.	value: 100.0 min: 1.0 max: 1000.0 type: double	value: 25.0 min: 1.0 max: 1000.0 type: double	value: 100.0 min: 1.0 max: 1000.0 type: double
`MultiLidarPipeline TrackableFilter maxSpeed`	Trackables whose speed (m/s) is higher than this are filtered out.	value: 50.0 min: 2.0 max: 100.0 type: double	value: 20.0 min: 2.0 max: 100.0 type: double	value: 50.0 min: 2.0 max: 100.0 type: double
`MultiLidarPipeline TrackableFilter maxPositionStdDev`	Trackables whose position standard deviation value along any axis is larger than this are filtered out.	value: 2.0 min: 1.0 max: 5.0 type: double	value: 2.0 min: 1.0 max: 5.0 type: double	value: 4.0 min: 1.0 max: 5.0 type: double

Anti-Masking parameters

Set error percent high to detect blockage and lifted percent a little lower. The default 80 for error and 50 for lifted should be fine. The idea is to make the two values not too close to one another (>= 10% difference is recommended) to remove the possibility of bouncing.

```
<MaskingDetector>
    <enabled>false</enabled>
    <!-- true to enable, false otherwise -->
    <maskingErrorPercent>80</maskingErrorPercent>
    <!-- The percentage of points with no detected return must be greater than
    this threshold to consider masking (of the sensor) is detected.
    Valid range : 1.0 to 100 % -->

    <maskingLiftedPercent>50</maskingLiftedPercent>
    <!-- The percentage of points with no detected return must be less than or
    fall below this threshold to consider masking (of the sensor) is no longer
    detected.
    Valid range : 1.0 to 100 % -->
</MaskingDetector>
```

Quanergy Solutions, Inc

128 Baytech Drive
San Jose, CA 95134
https://quanergy.com

For purchases made directly from Quanergy: contact sales@quanergy.com.

For purchases made from a third party such as a value-added reseller or system integrator: contact them for support.