Virtual RSU (VZMODE) - Instructions for using API

Author: Julie Vogelman

Version 0.42

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Document History & Document Approval

Version	Date	Change	Verizon POC
0.1	May 2020		VZMODE
0.2	June 1, 2020	Information for non-vehicle type clients; Protobuf serialization of messages published into the system	
0.3	June 3, 2020	corrected typo in Appendix A comment	
0.4	June 4, 2020	Added new “fused data” category for coding of object IDs	
0.5	June 5, 2020	Protobuf serialization should specify time in UTC	
0.6	June 7, 2020	Current MQTT Topic Version is 2	
0.7	June 8, 2020	Appendix B: example code for vehicle	
0.8	June 16, 2020	For VRU publishing its own PSM there’s no reason it should need to prefix it with a 1. Keep it simple.	
0.9	June 26, 2020	Clarifications on “Security”	
0.10	June 30, 2020	Clarification on JSON support for messages	
0.11	July 8, 2020	New section on MQTT General Topics (Static message topics and Regional topics)	
0.12	July 10, 2020	Clarification that STATIC_ADD is used for updates and not just adds	
0.13	July 12, 2020	Clarification on format support for Regional Messages	
0.14	July 14, 2020	Static messages now become part of the Regional topic message structure	
0.15	July 22, 2020	“XML” and “JSON” message types should be “XER” and “JER”	
0.16	July 30, 2020	Adding some comments to StaticAddMsg and clarification on publish path for static messages	
0.17	August 3, 2020	Introduction of “AutoPublishMsg”	
0.18	August 11, 2020	New message field in StaticAddMsg and AutoPublishAddMsg: “msgType”; modification of “id” field to string from int	
0.19	August 14, 2020	Fix to StaticDeleteMsg and AutoPublishAddMsg: “id” type is a string, not an int	
0.20	August 21, 2020	Some fields in RoutedMsg and AutoPublishAddMsg changed from int32 type to uint32 type	
0.21	August 23, 2020	RoutedMsg Protobuf updated to include clientIdsOpt field	
0.22	August 26, 2020	Static messages and AutoPublish messages now have corresponding responses to acknowledge receipt, start, and stop	
0.23	November 18, 2020	Clarifications to the meaning of the “Version” subtopic.	
0.24	January 21, 2020	AutoPublishDeleteMsg	
0.25	January 27, 2020	Additional statement in “Overview”	
0.26	May 13, 2021	REGIONAL topics now use 8 character geohash rather than 7	
0.27	May 17, 2021	Fixed mistake in example REGIONAL/STAT topics	
0.28	May 27, 2021	Version 3 of REGIONAL/DYN format includes MsgFormat subtopic	
0.29	June 3, 2021	REGIONAL/SAMP messages added	
0.30	June 4, 2021	Fixed internal links to use bookmarks instead	
0.31	June 23, 2021	AutoPublish Protobuf now includes LineCrossing	
0.32	June 28, 2021	AutoPublishAddMsg Protobuf - new field entityTypes; new statement about J2735 version	

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

0.33	July 1, 2021	AutoPublish Protobuf updates - description field, list of ids returned in AutoPublishDeleteAllMsg
0.34	July 30, 2021	New fields in Static Message Protobuf
0.35	August 6, 2021	AutopublishAddMsg.LineCrossing.Direction field uses value 0 to mean ‘Undefined’
0.36	August 8, 2021	AutopublishAddMsg.endTime no longer optional
0.37	November 10, 2021	new OutboundMsg Protobuf for OUT and REGIONAL topics; also example code added for SW Entity types
0.38	December 3, 2021	references to VZCV2X/2 have been replaced with references to VZCV2X/3, as well as REGIONAL/STAT/2 replaced with REGIONAL/STAT/3
0.39	December 14, 2021	fixed typo in LineCrossing.Direction Protobuf spec
0.40	December 30, 2021	topic under “Pub/Sub for Infrastructure clients” mistakenly listed “OUT” for direction instead of “IN”
0.41	December 31, 2021	added “INTENT” and “EVA” messages to supported types
0.42	January 12, 2022	Link to ProtocolBuffers online guide added

Name	Title	Company	Date of Approval

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Table of Contents

Document History & Document Approval 2
Table of Contents 4

Overview 6
Client Registration Service 6
MQTT 6

Security 7

J2735 Version 7

Register 8
Register - Vehicle and VRU Clients 8
Register - Infrastructure Clients 9
Register - Software-based Clients 11

Pub/Sub (MQTT) 12
Topics Specific to Individual Clients 12
Sub 13
Pub 13
Publishing Repeated Messages 14
Persistent Client Entity ID 15
Topics General to All Clients 16
Regional topics 16
Dynamic data over Regional Topics 16
Message Content 17
Static Messages 17
How are Static Messages Added to the System? 19
Persistent Client Entity ID 19
Pub/Sub for Vehicles/VRUs 20
Pub 20
Sub 20
Pub/Sub for Infrastructure Clients 20
Pub/Sub for Software-Based Clients 21
Pub 21
Sub 21
Decoding BSMs/PSMs 22

Appendix A: routed_msg.proto 23

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Appendix B: static_msg.proto 26

Appendix C: Example code for vehicles 28

Appendix D: Example code for Software Entity Types 34

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Overview

VZMODE is a system for routing messages between clients. The goal is to not overload clients with extraneous messages, so thus to target messages to them. The messages that flow through the system are generally *geography-based*: typically they have some central geoposition associated with them.

VZMODE has the notion of multiple types of clients:

- Entities on the road - vehicles and VRUs
- Infrastructure
- Software-based

Being a client in the system means:

- receiving a unique ID
- being able to publish messages over MQTT
- being able to subscribe to messages over MQTT

Vehicle clients receive messages in their own personal geofence generally, but can also subscribe to topics that are global for all users. Software clients generally subscribe to global topics.

Client Registration Service

The Client should initially register with the Client Registration Service in the MEC in order to:

- get a unique ID (referred to as “Entity ID” in this document)
- supply information about itself
- supply information about the messages it’s interested in receiving over MQTT topics, if applicable

The Client Registration Service is a REST interface over HTTP. The client will issue a REST POST. If the client’s request is valid, it will return a unique ID for that client to use for pub/sub over MQTT.

In the future, there will be a public cloud-based service that a client will connect to initially, which will authenticate it and direct it to the local MEC Client Registration Service. This will allow for a mobile client, such as a vehicle or VRU, to move between MEC instances as the client moves.

MQTT

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

In general messages from the client to VZMODE and from VZMODE to the client will be over MQTT. Each client will have its own unique set of topics, and there will be general topics as well.

Security

We have an option which can be turned on for a given deployment which provides both authentication and MQTT topic authorization to clients using mTLS and assigned tokens. More details to come….

J2735 Version

As of this writing, the Virtual RSU has been made compatible with the 2016 version of the J2735 standard, and not yet with the 2020 version. The changes between the two documents appear to be minimal, only adding some additional fields that are optional with the exception of one field (TimeMark) in the Signal Phase and Timing (SPaT) message, which has different numbering. What implications does being on the 2016 standard have? Since the Virtual RSU does transcode messages for clients who need special encodings, then if a new field is used it will not be propagated.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Register

The client will issue a REST POST message to the Client Registration Service.

The swagger documentation is located at: http://vzmode.dltdemo.io/

Register - Vehicle and VRU Clients

In the message, the vehicle or VRU Client specifies:

- information about itself: see definitions of “Entity Type”, “Entity Subtype”, and “Vendor ID” in *Pub/Sub* section below.
- information about the messages it wants to receive (on its client-specific topics)
    - which messages (currently supported are: BSM, PSM, SPaT, RSA, TIM, EVA, INTENT)
    - which sources of those messages (for example: messages from cameras, from ACME vehicles, from all vehicles, from all sources, etc.)
    - whether to receive messages based on a default radius around the vehicle, or some specific radius that is less than the default
    - for high rate messages (like BSM/PSM), whether or not to receive sampled data rather than all data
    - whether to use the default serialization, UPER (recommended for most clients), or a different serialization of ASN

The messages that are received will arrive over the client’s specific subscribe topics (see *Pub/Sub* section below).

Here is an example request to receive the currently available messages, from all sources, with default parameters:

```
{
        "ClientInformation":
        {
                "EntityType": "VEH",
                "EntitySubtype": "PSGR",
                "VendorID": "ACME"
        },
        "BSM":
        {
                "MsgFormat": "UPER"
        },
        "PSM":
        {
                "MsgFormat": "UPER"
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
        },
        "SPAT":
        {
                "MsgFormat": "UPER"
        },
        "RSA":
        {
                "MsgFormat": "UPER"
        },
        "TIM":
        {
                "MsgFormat": "UPER"
        }

}
```


The response contains the unique “Entity ID” assigned to the client, which the client will use to know which MQTT topic(s) to publish/subscribe to.

Note that the interface also includes “Update” and “Delete” messages. The client should ideally deregister using the “DELETE” message.

Register - Infrastructure Clients

In our system a Client is any entity that has its own unique topics in MQTT, either for publishing, subscribing, or both. Some examples of Infrastructure clients:

- Traffic controller (or Traffic Controller proxy)
- Street-based camera

An AI-based camera is an example of a client that might just *publish* messages. A Traffic Controller could theoretically publish messages as well as receive messages. In cases in which the software on the infrastructure can’t be modified itself, a software proxy can be set up in between to do so.
In the REST POST message to the Client Registration Service, the Client specifies:

- information about itself: see definitions of “Entity Type”, “Entity Subtype”, and “Vendor ID” in *Pub/Sub* section below
- its position: see “StaticPosition” key in Swagger documentation
- if applicable, the types of messages it wishes to subscribe to (which will fall within its geofence)

If the Client has registered to subscribe to messages, those will arrive over the client’s specific subscribe topics (see *Pub/Sub* section below).

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Here is an example request for an A/I-based camera, which is only publishing messages:

```
{
        “ClientInformation”:
        {
                "EntityType": "CAM",
                "EntitySubtype": "NA",
                "VendorID": "ACME"
        },
        “StaticPosition”:
        {
                “Latitude”: “369571987”,
                “Longitude”: “-1220254553”
        }
}
```

Here is an example request for an RSU, which publishes PC5 messages that it receives and subscribes to receive all currently available messages, with default parameters, that are within its geofence:

```
{
        "ClientInformation":
        {
                "EntityType": "RSU",
                "EntitySubtype": "NA",
                "VendorID": "ACME"
        },
        “StaticPosition”:
        {
                “Latitude”: “369571995”,
                “Longitude”: “-1220254518”
        },
        "BSM":
        {
                "MsgFormat": "UPER"
        },
        "PSM":
        {
                "MsgFormat": "UPER"
        },
        "SPAT":
        {
                "MsgFormat": "UPER"
        },
        "RSA":
        {
                "MsgFormat": "UPER"
        },
        "TIM":
        {
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
                "MsgFormat": "UPER"
        }

}
```

The response contains the unique “Entity ID” assigned to the client, which the client will use to know which MQTT topic(s) to publish/subscribe to.

Note that the interface also includes “Update” and “Delete” messages. The client should either deregister using the “DELETE” message at shutdown, or should persist its ID between restarts of its application so that it can re-use the original ID after restart rather than re-registering.

Register - Software-based Clients

Finally, there can be clients that don’t represent infrastructure or vehicles/VRUs but are merely software-based: if some third party needs to publish a message related to a road hazard, for example.

In the REST POST message to the Client Registration Service, the Client specifies:

- information about itself: see definitions of “Entity Type”, “Entity Subtype”, and “Vendor ID” in *Pub/Sub* section below

Here is an example request:

```
{
        “ClientInformation”:
        {
                "EntityType": "SW",
                "EntitySubtype": "NA",
                "VendorID": "ACME"
        }
}
```

The response contains the unique “Entity ID” assigned to the client, which the client will use to know which MQTT topic(s) to publish to.

Note that the interface also includes “Update” and “Delete” messages. The client should either deregister using the “DELETE” message at shutdown, or should persist its ID between restarts of its application so that it can re-use the original ID after restart rather than re-registering.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Pub/Sub (MQTT)

We will have 2 forms of MQTT topics:

- Topics that are specific to individual clients
- Topics that are general for all clients

Topics Specific to Individual Clients

These are for messages published by clients and messages received by clients which are tailored to them.

The topic structure is defined as follows:

`VZCV2X/<Version>/<Direction>/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/<Msg_Type>`

Where

ROOT (VZCV2X)	Is the root topic for all incoming and outgoing messages. This will allow different applications to be served by the same broker. Another root might be defined for inter-module communication.
Version	Single integer value. **Current version described by this document is 3.** (In version 3 outbound messages are serialized into OutboundMsg Protobuf (see Appendix A))
Direction	IN/OUT: IN: for messages coming from the entity OUT: for messages being sent to the entity
EntityType	The source generating these messages. VEH (vehicle) VRU CAM (infrastructure camera) TRAFLT (traffic light) RSU OBSEN (on-board sensor) SW (doesn't represent any physical entity) Others TBD
EntitySubtype	VEH • PSGR • TRUCK • EV VRU • PED

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

	• BIC: regular not motorized bicycle • MOTO: for motorized bicycle • SCOOT OBSEN • CAM • LIDAR any: • NA
VendorID	Indicates the Company providing this service. Verizon is VZ Manufacturer: If vehicle manufacturer Type is known (ACME|GM|TOYOTA|...) NA: for vehicles of unknown origin.
EntityID	Unique identifier for client assigned by Client Registration Service. For vehicles and VRUs, this is what should go in their self-published BSMs and PSMs as "TemporaryID".
MsgFormat	UPER XER (i.e. XML) JER (i.e. JSON) OER BER DER
MsgType	Message type (PSM, BSM, RSA, SPAT, TIM, etc)

Sub

Any clients that subscribe to messages should do that over topics that have "Direction" set to "OUT". The "Entity_type", "Entity_subtype", and "Vendor_ID" should match that which they passed to the Client Registration Service. The "Entity_ID" should be the unique ID they received from the Client Registration Service.

The message content, according to version 3 of the topic, is now an OutboundMsg (see Appendix A) which encapsulates the actual message data.

Pub

Any clients that publish messages should do that over topics that have the "Direction" subtopic set to "IN". The "Entity_type", "Entity_subtype", and "Vendor_ID" should match that which they passed to the Client Registration Service. The "Entity_ID" should be the unique ID they received from the Client Registration Service.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

All messages published into the system that are to be routed to other clients in the system should be serialized into the “RoutedMsg” Protobuf message, included in Appendix A of this document.

Publishing Repeated Messages

If a client wants to set a message to automatically be published repeatedly, an additional interface is offered for that:

VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/**AUTOPUB_ADD**
VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/**AUTOPUB_DELETE**
VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/**AUTOPUB_DELETE_ALL**

The message should be serialized into the Protobuf structures specified below, contained within Appendix A, and published over QoS 1. In the Protobuf spec, you can see that different criteria can determine how the message is delivered. Options include:

- to everyone within a georadius at a specified periodicity
- to everyone crossing a line segment defined by two geopositions

<MsgType> subtopic	Content	Ack <MsgType> subtopic	Ack Content
AUTOPUB_ADD	AutoPublishAddMsg Protobuf	AUTOPUB_ADD_ACK	AutoPublishAddAck Protobuf
AUTOPUB_DELETE	AutoPublishDeleteMsg Protobuf	AUTOPUB_DELETE_ACK	AutoPublishDeleteAck Protobuf
AUTOPUB_DELETE_ALL	no content	AUTOPUB_DELETE_ALL_ACK	AutoPublishDeleteAllAck Protobuf

Each message will have a corresponding acknowledgment shown in the table above, which will come over OUT topics corresponding to the IN topics. Note that the AUTOPUB_ADD message can be used either to add a new message or update an existing message.

If you look at the Protobuf specification, you'll notice the "id" field is specified by the publisher in this case. This puts the onus on the publisher to establish IDs that are unique unto itself for that message type (doesn't need to be unique system-wide of course).

In addition, the client can expect to receive the following messages when a message has been started or stopped. These will come regardless of whether the message was *scheduled* for start/stop or whether it was started or stopped (deleted) immediately.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

<MsgType> subtopic	Content
AUTOPUB_STARTED	AutoPublishStarted Protobuf
AUTOPUB_STOPPED	AutoPublishStopped Protobuf

Persistent Client Entity ID

For clients publishing to this interface, it is important that we avoid a situation in which they add messages to the system, their software crashes and restarts, and the original messages are still lingering in the system listed under a separate entity ID. For that reason, it is recommended that they just register with the Client Registration Service once to obtain a unique ID and store that ID somewhere persistent for the next time that their software restarts. It may also make sense for them to store their messages/IDs in persistent storage as well to avoid re-adding those, or otherwise they can always clear their messages using “AUTOPUB_DELETE_ALL” and re-add.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Topics General to All Clients

Regional topics

There are going to be certain back end services that are essentially interested in subscribing to *all* of our data; however, it may be that they can't process all of the data within a single service and may need to break that service into microservices - it only makes sense for them to do this regionally (i.e. each microservice subscribing to a different region).

The data is broken into topics that are divided by GEOHASH (a unique alphanumeric-encoded binary value for each rectangular region on Earth). These messages are only published to by VZMODE itself.

Dynamic data over Regional Topics

This is the data that is regularly flowing through VZMODE, from IN topics routed to OUT topics. It is "dynamic" in its nature. VZMODE publishes it to these Regional topics as well (UPER and JER only).

```
REGIONAL/DYN/<Version>/<GEOHASHID>/<EntityType>/<EntitySubtype>/<VendorID>/<
/MsgFormat>/<MsgType>
```

Where:

ROOT (REGIONAL)	Root topic
DataType	One of: STAT (see section below on Static Messages) DYN (dynamic messages) SAMP (for high rate messages like BSMs, a sampled set of messages)
Version	Single integer value, specifies topic version for this particular "DataType". Single integer value. **Current version described by this document is 4.** In version 4 messages are serialized into OutboundMsg Protobuf (see Appendix A)
GEOHASHID	A series of 8 subtopics, each one representing one alphanumeric character of the geohash. Dividing the geohash into subtopics enables being able to subscribe to different levels of granularity, by using wildcards in the subscription as needed.
MsgFormat	Currently includes "UPER" and "JER" (JSON)
All other fields	Please see section on "Topics Specific to Individual Clients" for these definitions

For high rate messages (currently just PSM and BSM), we offer a set of Regional topics which represent just a sampled set of messages (for subscribers who don't want to receive every message):

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

REGIONAL/SAMP/<Version>/<GEOHASHID>/<EntityType>/<EntitySubtype>/<VendorID>/</MsgFormat>/<MsgType>

For each vehicle, the messages flow at approximately 2 meters between consecutive positions, or for slow/non-moving vehicles every 2 seconds. For every pedestrian (PSM), the messages flow at approximately 1 meter between consecutive positions, or for slow/non-moving pedestrians every 2 seconds.

Message Content

The content of the Regional messages is serialized into an OutboundMsg (see Appendix A), as part of version 4.

Static Messages

Some of the J2735 messages represent information that rarely changes:

- MAP - road geometry
- Certain RSA (Roadside Alert) messages - road hazards
- Certain TIM (Traveler Information) messages - general information
- possibly others

We will make these messages available to clients to grab as needed, from a single set of topics.

MQTT has this idea of a "RETAIN" message. Each MQTT topic can have 1, and what it means is that anytime a client subscribes to that topic it receives that retained message. Static messages live on topics that have RETAIN messages, so clients can access them anytime.
We will have the following topics which can be subscribed to by clients (they are published to by VZMODE itself):

- **REGIONAL/STAT/<Version>/<GEOHASHID>/<EntityType(source)>/<EntitySubtype(source)>/<VendorID(source)>/<MsgFormat>/<MsgType>/<ID>** - this one contains a single message (just the message itself, no need for Protobuf). We will publish for each supported MsgFormat.
- **REGIONAL/STAT/<Version>/LIST** - this one contains a list of each message along with its geoposition so that the client can optionally subscribe to individual messages as-needed. This will be formatted into the Protobuf structure *StaticMsgList* in Appendix B.

Current version is 3.

Please reference above section for definitions of subtopics.

So, clients have some options for subscribing:

1. Subscribe to "REGIONAL/STAT/<Version>/+/+/+/+/+/+/+/+/+/+/+/<MsgFormat>/+/+". Upon subscription, client will receive all Retained messages. If updates occur they will receive them. If deletions occur,

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

they will receive notice that a 0 byte message (i.e. deletion) was published to that topic. If new messages occur, they will automatically receive them.

2. Subscribe to messages as-needed depending on current location of client. Subscribe to "REGIONAL/STAT/<Version>/LIST" and get the list of IDs that it references (which are used to format the topics to subscribe to). Each one is listed with a geoposition. Remain subscribed to the LIST topic in case there is an update (new message or deleted message).
3. Subscribe to the individual messages regionally, for example: “REGIONAL/STAT/<Version>/9/b/0/2/+/+/+/+/+/+/+/<MsgFormat>/+/+". Upon subscription, client will receive all Retained messages within this region. If updates occur they will receive them. If deletions occur, they will receive notice that a 0 byte message (i.e. deletion) was published to that topic. If new messages occur, they will automatically receive them.

The publishing of the messages will be over QoS 1, and the subscriptions to them should also be over QoS 1.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

How are Static Messages Added to the System?

Publishers will generally be "SW clients" of VZMODE. We can use the existing VZCV2X topic structure but with Msg_Type set to "STATIC_ADD" (for adds and updates both) and "STATIC_DELETE"/"STATIC_DELETE_ALL":

VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/STATIC_ADD
VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/STATIC_DELETE
VZCV2X/<Version>/IN/<Entity_Type>/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/STATIC_DELETE_ALL

The message should be serialized into the Protobuf structures specified below, contained within Appendix B, and published over QoS 1:

<MsgType> subtopic	**Content**	**Ack <MsgType> subtopic**	**Ack Content**
STATIC_ADD	StaticAddMsg Protobuf	STATIC_ADD_ACK	StaticAddAck Protobuf
STATIC_DELETE	StaticDeleteMsg Protobuf	STATIC_DELETE_ACK	StaticDeleteAck Protobuf
STATIC_DELETE_ALL	no content	STATIC_DELETE_ALL_ACK	StaticDeleteAllAck Protobuf

Each message will have a corresponding acknowledgment shown in the table above, which will come over OUT topics corresponding to the IN topics.

If you look at the Protobuf specification, you'll notice the "id" field is specified by the publisher in this case. This puts the onus on the publisher to establish IDs that are unique unto itself for that message type (doesn't need to be unique system-wide of course).

In addition, the client can expect to receive the following messages for each <Msg_Format> when a message has been started or stopped. These will come regardless of whether the message was *scheduled* for start/stop or whether it was started or stopped (deleted) immediately.

<MsgType> subtopic	**Content**
STATIC_STARTED	StaticStarted Protobuf
STATIC_STOPPED	StaticStopped Protobuf

Persistent Client Entity ID

For clients publishing to this interface, it is important that we avoid a situation in which they add messages to

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

the system, their software crashes and restarts, and the original messages are still lingering in the system listed under a separate entity ID. For that reason, it is recommended that they just register with the Client Registration Service once to obtain a unique ID and store that ID somewhere persistent for the next time that their software restarts. It may also make sense for them to store their messages/IDs in persistent storage as well to avoid re-adding those, or otherwise they can always clear their messages using "STATIC_DELETE_ALL" and re-add.

Pub/Sub for Vehicles/VRUs

Pub

A vehicle client should publish its BSM to:

```
VZCV2X/3/IN/VEH/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/UPER/BSM
```

Ideally, this should be every 100ms. (Client will be removed from the system if enough time (TBD) passes with no published BSM.)

The BSM should be serialized into the "RoutedMsg" Protobuf message, included in Appendix A of this document.

A VRU client will do the same, except with Message Type *PSM*.

Sub

Client will receive messages that are specific to it on topics with this structure:

```
VZCV2X/3/OUT/VEH/<Entity_subtype>/<Vendor_ID>/<Entity_ID>/<Msg_Format>/<Msg_Type>
```

The message being received is being serialized into the "OutboundMsg" Protobuf messag (see Appendix A). The messages will be geofenced around the client, and will also be according to any other parameters specified as part of the Client Registration request.

The client can also subscribe to Static Messages. See here for details.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Pub/Sub for Infrastructure Clients

An Infrastructure client can publish, subscribe, or both.

Unlike the vehicle or VRU client, it does not need to publish its position regularly, as it is assumed to not be moving, and it should have supplied it as part of the initial Client Registration.

As an example, an AI-based camera publishing BSMs should serialize them into the “RoutedMsg” Protobuf message and should publish them to:

`VZCV2X/3/IN/CAM/NA/<Vendor_ID>/<Entity_ID>/UPER/BSM`

An Infrastructure Client can optionally subscribe to messages that are within its georadius (VZCV2X topics) and/or any messages listed in the “Topics General to All Clients” section.

Pub/Sub for Software-Based Clients

Pub

A software-based client might publish messages that are static (rarely changing) or messages that are dynamic.
For messages that are static, please see the How are Static Messages Added to the System? section for guidance on how to publish those.
An additional interface exists for messages that are not changing but it’s still desired to send them within a geofence of the vehicle. Please see the section on Publishing Repeated Messages.

Sub

A software-based client can optionally subscribe to any messages listed in the “Topics General to All Clients” section.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Decoding BSMs/PSMs

If the Client subscribes to BSM and PSM messages, how will it distinguish the source of the BSM or PSM? Ultimately, we will implement a “data fusion” capability to be able to deliver a single set of data, correlated from the various sources. Until then, a BSM delivered by camera A should be distinguishable from a BSM delivered by camera B, which should be distinguishable by a vehicle which published its own position, which should be distinguishable from a vehicle with a dashboard camera that published a position of a vehicle that it captured. For this reason, the 32 bit Temporary IDs in the BSM and PSM will follow the following numbering scheme:

- The MOST Significant 4 bits will be used to identify data sources (up to 16 categories). They will be used as follows

0	Vehicle and VRU: For Vehicles and VRUs sending their status and receiving messages from the network - The next 12 bits will be unused - The final 16 bits will be the entity’s assigned ID
1	TBD
2	Camera - The next 12 bits will be used to identify the Camera (up to 4K cameras) - The final 16 bits will be used to identify objects
3	On board sensors: Objects detected by the car sensors - The next 16 bits will be the vehicle’s assigned ID - The final 12 bits will be used to identify objects
4	RSU - The next 12 bits will be used to identify the RSU (up to 4K RSUs) - The final 16 bits will be used to identify objects
5-14	TBD
15	Fused data - The next 12 bits will be used to identify the ID of the “SW” Entity type that fused the data - The final 16 bits will be used to identify objects

Note that a BSM and PSM may have the same Temporary ID. The Message Type distinguishes one from the other, so there should be no confusion if they do.

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Appendix A: routed_msg.proto

Please see here for how to encode/decode Protocol Buffers (Protobuf).

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

package routedmsgpb;

message Position {
  int32 latitude = 1; // in 1/10th microdegrees [-900000000,900000001]
  int32 longitude = 2; // in 1/10th microdegrees [-1799999999,1800000001]
}

message ClientIdentity {
  uint32 clientId = 1;
  string entityType = 2;
}


message RoutedMsg {
  bytes msgBytes = 1;
  google.protobuf.Timestamp time = 2; // in UTC time

  Position position = 3;

  // optional...only applies to some types of messages
  uint32 customRadiusOpt = 4;
  bool clientCanOverrideRadiusOpt = 5; // only applies if customRadiusOpt is set

  repeated ClientIdentity clientIdsOpt = 6;  // if specified, these are the
                                                potential clients to receive the
                                                message, rather than those within a
                                                geofence

}

message AutoPublishAddMsg {
    bytes msgBytes = 1;
    string msgType = 2; // such as "TIM", "RSA", etc
    string id = 3; // must just be unique for the publishing client for the given
                   // msgType
    string description = 11;
    repeated string entityTypes = 12; // optional, to publish to certain
                                      // EntityTypes and not others; if
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
                                        // empty it implies send to everyone


    // the next 4 fields pertain to messages that should be delivered to everyone
    // within a radius at some periodicity
    Position positionOpt = 4;
    uint32 customRadiusOpt = 5; // optional;
    bool clientCanOverrideRadiusOpt = 6; // only applies if customRadiusOpt is set
    uint32 frequencySecOpt = 7;


    google.protobuf.Timestamp startTime = 8; // just set to current time if it's
                                             // not in the future
    google.protobuf.Timestamp endTime = 9; // this should be set within 24 hours of
                                           // startTime

    // if LineCrossing is defined then deliver messages to anyone crossing that
    // line
    LineCrossing lineCrossingOpt = 10;

 }

 message LineCrossing {
   Position endpoint1 = 1;
   Position endpoint2 = 2;
   string roadIdentifier = 3;
   enum Direction {
     Undefined = 0;
     NB = 1;
     EB = 2;
     SB = 3;
     WB = 4;
   }
   Direction direction = 4;
 }


// this is the message a client should publish to remove an Autopublished Message
// that it added
message AutoPublishDeleteMsg {
  string id = 1;
  string msgType = 2; // such as "TIM", "RSA", etc

}

// acknowledgment for the AutoPublishAddMsg
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
message AutoPublishAddAck {
 string msgType = 1; // such as "TIM", "RSA", etc
 string id = 2;
 string msgDescription = 5; // matches 'description' from original AutoPublishAddMsg
 bool success = 3;
 string failMsg = 4; // only set in case of failure
}

// acknowledgment for the AutoPublishDeleteMsg
message AutoPublishDeleteAck {
 string msgType = 1; // such as "TIM", "RSA", etc
 string id = 2;
 string msgDescription = 5; // matches 'description' from original AutoPublishAddMsg
 bool success = 3;
 string failMsg = 4; // only set in case of failure
}

// acknowledgment for the request to delete all messages for the client (which has no
associated Protobuf)
message AutoPublishDeleteAllAck {
 bool success = 1;
 string failMsg = 2; // only set in case of failure
 repeated string ids = 3;
}

// response to client that a message started (applies both to messages that were
scheduled and to messages that were started immediately)
message AutoPublishStarted {
 string msgType = 1; // such as "TIM", "RSA", etc
 string id = 2;
 string msgDescription = 3; // matches 'description' from original AutoPublishAddMsg
}

// response to client that a message stopped (applies both to messages that expired
and to messages that were stopped immediately)
message AutoPublishStopped {
 string msgType = 1; // such as "TIM", "RSA", etc
 string id = 2;
 string msgDescription = 3; // matches 'description' from original AutoPublishAddMsg
}


 message OutboundMsg {
  ClientIdentity clientIdentity = 1; // identifies the publisher
  google.protobuf.Timestamp time = 2; // in UTC time
  bytes msgBytes = 3;
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Appendix B: static_msg.proto

Please see here for how to encode/decode Protocol Buffers (Protobuf).

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
package staticmsgpb;
message Position {
  int32 latitude = 1; // in 1/10th microdegrees
  int32 longitude = 2; // in 1/10th microdegrees
}

// this is the message a client should publish to add a new Static Message
// into the system
message StaticAddMsg {
  bytes msgBytes = 1; // the actual content of the message
  string msgType = 2; // such as "TIM", "RSA", etc
  string id = 3; // must just be unique for the publishing client for this msgType
  string description = 7;  // can be empty

  google.protobuf.Timestamp startTime = 4; // just set to current time if it's not
                                           // in the future
  google.protobuf.Timestamp endTime = 5; //
  Position position = 6; // approximate central geoposition
}


// this is the message a client should publish to remove a Static Message
// that it added
// the reason this is a Protobuf message at all (given that it has one field)
// is in case we need to expand it in the future, this gives us flexibility
message StaticDeleteMsg {
  string id = 1;
}

// this is the list of static messages that a subscribing client has access to
message StaticMsgList {
  repeated StaticMsg messages = 1;
}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
message StaticMsg {
  string msgType = 1; // such as "TIM", "RSA", etc
  string id = 2; // must just be unique for the publishing client for the given
msgType

  Position position = 3;
}

// acknowledgment for the StaticAddMsg
message StaticAddAck {
  string msgType = 1; // such as "TIM", "RSA", etc
  string id = 2;
  string msgDescription = 5; // matches 'description' from original StaticAddMsg
  bool success = 3;
  string failMsg = 4; // only set in case of failure
}

// acknowledgment for the StaticDeleteMsg
message StaticDeleteAck {
  string msgType = 1; // such as "TIM", "RSA", etc
  string id = 2;
  string msgDescription = 5; // matches 'description' from original StaticAddMsg
  bool success = 3;
  string failMsg = 4; // only set in case of failure
}

// acknowledgment for the request to delete all messages for the client (which has no
associated Protobuf)
message StaticDeleteAllAck {
  bool success = 1;
  string failMsg = 2; // only set in case of failure
  repeated string ids = 3;
}

// response to client that a message started (applies both to messages that were
scheduled and to messages that were started immediately)
message StaticStarted {
  string msgType = 1; // such as "TIM", "RSA", etc
  string id = 2;
  string msgDescription = 3; // matches 'description' from original StaticAddMsg
}

// response to client that a message stopped (applies both to messages that expired
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
and to messages that were stopped immediately)
message StaticStopped {
  string msgType = 1; // such as "TIM", "RSA", etc
  string id = 2;
  string msgDescription = 3; // matches 'description' from original StaticAddMsg
}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Appendix C: Example code for vehicles

```
package main

import (
   "bytes"
   "crypto/tls"
   "crypto/x509"
   "encoding/json"
   "fmt"
   "io/ioutil"
   "net/http"
   "os"
   "os/signal"
   "strconv"
   "syscall"
   "time"

   pahomqtt "github.com/eclipse/paho.mqtt.golang"

   "github.com/golang/protobuf/proto"
   "github.com/golang/protobuf/ptypes/timestamp"

   // this is my copy of the auto-generated code that the Protocol Buffer compiler created for Go
using RoutedMsg
   "github.com/VerizonVirtualRSU/protobuf/routedmsgpb"
)

func main() {

   serverHost := os.Args[1]
   registrationServerPort, _ := strconv.Atoi(os.Args[2])
   mqttBrokerPort, _ := strconv.Atoi(os.Args[3])

   var tlsConfig *tls.Config
   //tlsConfig = generateTLSConfig(serverCACert)

   clientIdentity := ClientIdentity{
       EntityType:    "VEH",
       EntitySubtype: "PSGR",
       Vendor:        "ACMD",
   }

   // register with the Client Registration interface and get a unique ID back
   register(&clientIdentity, serverHost, registrationServerPort, tlsConfig)
   fmt.Printf("got client ID %d back from Client Registration Service\n", clientIdentity.ID)

   mqttConnection := MQTTConnection{
       brokerHost: serverHost,
       brokerPort: mqttBrokerPort,
       username:   "your-username",
       password:   "your-password",
       tlsConfig:  tlsConfig,
   }

   // subscribe to topics
   mqttSubscribe(mqttConnection, clientIdentity)
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    // set up timed function to publish BSM once per second
    startPublishingBSM(mqttConnection, clientIdentity)

    // prevent application from closing until SIGTERM is issued
    signalChannel := make(chan os.Signal, 1)
    signal.Notify(signalChannel, syscall.SIGINT, syscall.SIGTERM)
    <-signalChannel

    // deregister with the Client Registration interface prior to shutdown
    deregister(serverHost, registrationServerPort, tlsConfig, clientIdentity)

    fmt.Println("Shutting down")
}

type ClientIdentity struct {
    EntityType    string
    EntitySubtype string
    Vendor        string
    ID            int
}

type RegistrationResponse struct {
    // unique ID assigned to client
    ID int
}

// register with the Client Registration Server and get a unique ID back
// This is a REST interface (POST request)
func register(clientIdentity *ClientIdentity, host string, port int, tlsConfig *tls.Config) {

    registrationRequestJson := fmt.Sprintf(`
    {
        "ClientInformation":
        {
            "EntityType": "%s",
            "EntitySubtype": "%s",
            "VendorID": "%s"
        },

        "RSA":
        {
            "MsgFormat": "UPER"
        },
        "SPAT":
        {
            "MsgFormat": "UPER"
        }
    }
    `, clientIdentity.EntityType, clientIdentity.EntitySubtype, clientIdentity.Vendor)

    // send the POST request to Client Registration Service
    var restBaseUrl string
    if tlsConfig == nil {
        restBaseUrl = fmt.Sprintf("http://%s:%d", host, port)
    } else {
        restBaseUrl = fmt.Sprintf("https://%s:%d", host, port)
    }

    req, err := http.NewRequest("POST", restBaseUrl+"/registration",
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
bytes.NewBufferString(registrationRequestJson))
    if err != nil {
        panic(err.Error())
    }
    req.Header.Set("Content-Type", "application/json")

    restClient := &http.Client{}
    if tlsConfig != nil {
        restClient = &http.Client{
            Transport: &http.Transport{
                TLSClientConfig: tlsConfig,
            },
        }
    }
    resp, err := restClient.Do(req)
    if err != nil {
        panic(err.Error())
    }
    if resp.StatusCode < 200 || resp.StatusCode >= 300 {
        panic(fmt.Sprintf("bad status code in response from POST to /registration: %s",
resp.Status))
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(fmt.Sprintf(fmt.Sprintf("difficulty reading in response from POST to /registration,
err:%s", err)))
    }

    var response RegistrationResponse
    if err := json.Unmarshal(body, &response); err != nil {
        panic(fmt.Sprintf("couldn't unmarshal json response from POST to /registration, err:%s",
err))
    }

    clientIdentity.ID = response.ID
}

func deregister(host string, port int, tlsConfig *tls.Config, clientIdentity ClientIdentity) {
    // send the DELETE request to Client Registration Service
    var restBaseUrl string
    if tlsConfig == nil {
        restBaseUrl = fmt.Sprintf("http://%s:%d", host, port)
    } else {
        restBaseUrl = fmt.Sprintf("https://%s:%d", host, port)
    }

    restUrl := fmt.Sprintf("%s/registration/%s/%d", restBaseUrl, clientIdentity.EntityType,
clientIdentity.ID)
    req, err := http.NewRequest("DELETE", restUrl, bytes.NewBufferString(""))
    if err != nil {
        panic(err.Error())
    }
    restClient := &http.Client{}
    if tlsConfig != nil {
        restClient = &http.Client{
            Transport: &http.Transport{
                TLSClientConfig: tlsConfig,
            },
        }
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    }
    resp, err := restClient.Do(req)
    if err != nil {
        panic(err.Error())
    }
    if resp.StatusCode < 200 || resp.StatusCode >= 300 {
        panic(fmt.Sprintf("bad status code in response from DELETE: status code = %s", resp.Status))
    }

}

func generateTLSConfig(serverCACert string) *tls.Config {
    caCert, err := ioutil.ReadFile(serverCACert)
    if err != nil {
        panic(err)
    }
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)

    return &tls.Config{
        RootCAs: caCertPool,
    }
}

type MQTTConnection struct {
    brokerHost string
    brokerPort int
    username   string
    password   string
    tlsConfig  *tls.Config
}

func mqttSubscribe(mqttConnection MQTTConnection, clientIdentity ClientIdentity) {
    // We registered to receive RSA and SPAT messages as part of our request to the Client
Registration Server
    // Now subscribe to those topics
    var topic string
    topic = fmt.Sprintf("VZCV2X/+/OUT/%s/%s/%s/%d/UPER/RSA", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)
    err := mqttSubscribeToTopic(topic, 0, mqttConnection)
    if err != nil {
        panic(err)
    }
    topic = fmt.Sprintf("VZCV2X/+/OUT/%s/%s/%s/%d/UPER/SPAT", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)
    err = mqttSubscribeToTopic(topic, 0, mqttConnection)
    if err != nil {
        panic(err)
    }

    // Also subscribe to all Static messages
    mqttSubscribeToAllStaticMessages(mqttConnection)
}

// subscribe to an individual topic
func mqttSubscribeToTopic(topic string, qualityOfService byte, mqttConnection MQTTConnection) error
{
    mqttClient := newMqttClient(mqttConnection)
    token := mqttClient.Subscribe(topic, qualityOfService,
        func(c pahomqtt.Client, msg pahomqtt.Message) {
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
            fmt.Printf("%v: new message on topic: %s", time.Now(), msg.Topic())
            // deserialize OutboundMsg Protobuf here
            protobufMsg := &routedmsgpb.OutboundMsg{}

            if err := proto.Unmarshal(msg.Payload(), protobufMsg); err != nil {
                panic("Failed to unmarshal OutboundMsg")
            }

            // Do something useful here
        })

    token.Wait()
    if token.Error() != nil {
        return token.Error()
    }

    return nil
}

func mqttSubscribeToAllStaticMessages(mqttConnection MQTTConnection) error {
    mqttClient := newMqttClient(mqttConnection)
    token := mqttClient.Subscribe(`REGIONAL/STAT/3/+/+/+/+/+/+/+/+/+/+/UPER/+/+`, 1,
        func(c pahomqtt.Client, msg pahomqtt.Message) {
            handleStaticMsg(msg)
        })

    token.Wait()
    if token.Error() != nil {
        return token.Error()
    }

    return nil
}

// this creates a new TCP connection to the MQTT broker, which could be used for publishing or
subscribing
func newMqttClient(mqttConnection MQTTConnection) pahomqtt.Client {

    clientOptions := pahomqtt.NewClientOptions().SetKeepAlive(20 * time.Second)
    if mqttConnection.tlsConfig != nil {
        uri := fmt.Sprintf("ssl://%v:%d", mqttConnection.brokerHost, mqttConnection.brokerPort)
        clientOptions = clientOptions.AddBroker(uri).SetTLSConfig(mqttConnection.tlsConfig)
    } else {
        uri := fmt.Sprintf("tcp://%v:%d", mqttConnection.brokerHost, mqttConnection.brokerPort)
        clientOptions = clientOptions.AddBroker(uri)
    }
    clientOptions =
clientOptions.SetUsername(mqttConnection.username).SetPassword(mqttConnection.password)
    client := pahomqtt.NewClient(clientOptions)

    if token := client.Connect(); token.Wait() && token.Error() != nil {
        panic(token.Error())
    }
    return client
}

// every 100 milliseconds publish a new Basic Safety Message
func startPublishingBSM(mqttConnection MQTTConnection, clientIdentity ClientIdentity) {
    mqttClient := newMqttClient(mqttConnection)
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    topic := fmt.Sprintf("VZCV2X/3/IN/%s/%s/%s/%d/UPER/BSM", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)

    ticker := time.NewTicker(100 * time.Millisecond)
    go func() {
        for range ticker.C {
            publishBSM(mqttClient, topic, clientIdentity)
        }
    }()
}

// publish a new Basic Safety Message
func publishBSM(mqttClient pahomqtt.Client, topic string, clientIdentity ClientIdentity) {

    var latitude int32
    var longitude int32
    bsm := createNewBSM(&latitude, &longitude, clientIdentity.ID)

    // serialize into Protobuf:

    nowTime := time.Now()
    sendTime := &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)}
    protobufMsg := &routedmsgpb.RoutedMsg{
        Time: sendTime,
        Position: &routedmsgpb.Position{
            Latitude:  latitude,
            Longitude: longitude,
        },
        MsgBytes: bsm,
    }
    protoBytes, err := proto.Marshal(protobufMsg)
    if err != nil {
        panic(err)
    }

    token := mqttClient.Publish(topic, 0, false, protoBytes)
    token.Wait()
    if token.Error() != nil {
        panic(token.Error())
    }

}

func handleStaticMsg(msg pahomqtt.Message) {
    if len(msg.Payload()) > 0 {
        // message has valid content
        // can store on disk and/or in memory
        //processMessage(msg.Payload(), msg.TopicPath())
        //saveFile(MY_DIRECTORY+msg.TopicPath(), msg.Payload())
    } else {
        // this is a deletion
        //processRemoval(msg.TopicPath())
        //removeFile(MY_DIRECTORY + msg.TopicPath())
    }
}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

Appendix D: Example code for Software Entity Types

```
package main

import (
   "bytes"
   "crypto/tls"
   "encoding/json"
   "fmt"
   "io/ioutil"
   "net/http"
   "os"
   "os/signal"
   "strconv"
   "syscall"
   "time"

   pahomqtt "github.com/eclipse/paho.mqtt.golang"

   "github.com/golang/protobuf/proto"
   "github.com/golang/protobuf/ptypes/timestamp"

   // this is my copy of the auto-generated code that the Protocol Buffer compiler created for Go
using RoutedMsg
   "github.com/VerizonVirtualRSU/protobuf/routedmsgpb"
   "github.com/VerizonVirtualRSU/protobuf/staticmsgpb"
)

func main() {

   registrationServerHost := os.Args[1]
   registrationServerPort, _ := strconv.Atoi(os.Args[2])
   mqttBrokerHost := os.Args[3]
   mqttBrokerPort, _ := strconv.Atoi(os.Args[4])

   var tlsConfig *tls.Config
   //tlsConfig = generateTLSConfig(serverCACert)

   clientIdentity := ClientIdentity{
       EntityType:    "VEH",
       EntitySubtype: "PSGR",
       Vendor:        "ACMD",
   }

   // register with the Client Registration interface and get a unique ID back
   register(&clientIdentity, registrationServerHost, registrationServerPort, tlsConfig)
   fmt.Printf("got client ID %d back from Client Registration Service\n", clientIdentity.ID)

   mqttConnection := MQTTConnection{
       brokerHost: mqttBrokerHost,
       brokerPort: mqttBrokerPort,
       username:   "your-username",
       password:   "your-password",
       tlsConfig:  tlsConfig,
   }

   // subscribe to topics
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    mqttSubscribe(mqttConnection, clientIdentity)

    // set up timed function to publish BSM once per second
    startPublishingBSM(mqttConnection, clientIdentity)

    // prevent application from closing until SIGTERM is issued
    signalChannel := make(chan os.Signal, 1)
    signal.Notify(signalChannel, syscall.SIGINT, syscall.SIGTERM)
    <-signalChannel

    // deregister with the Client Registration interface prior to shutdown
    deregister(registrationServerHost, registrationServerPort, tlsConfig, clientIdentity)

    fmt.Println("Shutting down")

}

type ClientIdentity struct {
    EntityType    string
    EntitySubtype string
    Vendor        string
    ID            int
}

type RegistrationResponse struct {
    // unique ID assigned to client
    ID int
}

// register with the Client Registration Server and get a unique ID back
// This is a REST interface (POST request)
func register(clientIdentity *ClientIdentity, host string, port int, tlsConfig *tls.Config) int {

    registrationRequestJson := fmt.Sprintf(`
    {
        "ClientInformation":
        {
            "EntityType": "%s",
            "EntitySubtype": "%s",
            "VendorID": "%s"
        }
    }
    `, clientIdentity.EntityType, clientIdentity.EntitySubtype, clientIdentity.Vendor)

    // send the POST request to Client Registration Service
    var restBaseUrl string
    if tlsConfig == nil {
        restBaseUrl = fmt.Sprintf("http://%s:%d", host, port)
    } else {
        restBaseUrl = fmt.Sprintf("https://%s:%d", host, port)
    }

    req, err := http.NewRequest("POST", restBaseUrl+"/registration",
bytes.NewBufferString(registrationRequestJson))
    if err != nil {
        panic(err.Error())
    }
    req.Header.Set("Content-Type", "application/json")

    restClient := &http.Client{}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    if tlsConfig != nil {
        restClient = &http.Client{
            Transport: &http.Transport{
                TLSClientConfig: tlsConfig,
            },
        }
    }
    resp, err := restClient.Do(req)
    if err != nil {
        panic(err.Error())
    }
    if resp.StatusCode < 200 || resp.StatusCode >= 300 {
        panic(fmt.Sprintf("bad status code in response from POST to /registration: %s",
resp.Status))
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(fmt.Sprintf(fmt.Sprintf("difficulty reading in response from POST to /registration,
err:%s", err)))
    }

    var response RegistrationResponse
    if err := json.Unmarshal(body, &response); err != nil {
        panic(fmt.Sprintf("couldn't unmarshal json response from POST to /registration, err:%s",
err))
    }

    clientIdentity.ID = response.ID

    return clientIdentity.ID
}

type MQTTConnection struct {
    brokerHost string
    brokerPort int
    username   string
    password   string
    tlsConfig  *tls.Config
}

func mqttSubscribe(mqttConnection MQTTConnection) {
    // maybe we subscribe to all incoming BSMs from vehicles
    var topic string
    topic = fmt.Sprintf("REGIONAL/DYN/4/+/+/+/+/+/+/+/+/VEH/+/+/UPER/BSM")
    err := mqttSubscribeToTopic(topic, 0, mqttConnection)
    if err != nil {
        panic(err)
    }
}

// subscribe to an individual topic
func mqttSubscribeToTopic(topic string, qualityOfService byte, mqttConnection MQTTConnection) error
{
    mqttClient := newMqttClient(mqttConnection)
    token := mqttClient.Subscribe(topic, qualityOfService,
        func(c pahomqtt.Client, msg pahomqtt.Message) {
            fmt.Printf("%v: new message on topic: %s", time.Now(), msg.Topic())
            // deserialize OutboundMsg Protobuf here
            protobufMsg := &routedmsgpb.OutboundMsg{}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
            if err := proto.Unmarshal(msg.Payload(), protobufMsg); err != nil {
                panic("Failed to unmarshal OutboundMsg")
            }

            // Do something useful here
        })

    token.Wait()
    if token.Error() != nil {
        return token.Error()
    }

    return nil
}

// this creates a new TCP connection to the MQTT broker, which could be used for publishing or
subscribing
func newMqttClient(mqttConnection MQTTConnection) pahomqtt.Client {

    clientOptions := pahomqtt.NewClientOptions().SetKeepAlive(20 * time.Second)
    if mqttConnection.tlsConfig != nil {
        uri := fmt.Sprintf("ssl://%v:%d", mqttConnection.brokerHost, mqttConnection.brokerPort)
        clientOptions = clientOptions.AddBroker(uri).SetTLSConfig(mqttConnection.tlsConfig)
    } else {
        uri := fmt.Sprintf("tcp://%v:%d", mqttConnection.brokerHost, mqttConnection.brokerPort)
        clientOptions = clientOptions.AddBroker(uri)
    }
    clientOptions =
clientOptions.SetUsername(mqttConnection.username).SetPassword(mqttConnection.password)
    client := pahomqtt.NewClient(clientOptions)

    if token := client.Connect(); token.Wait() && token.Error() != nil {
        panic(token.Error())
    }
    return client
}

func publishSingleMessage(clientIdentity ClientIdentity, mqttClient pahomqtt.Client, msgContent
[]byte, msgLatitude int32, msgLongitude int32) {
    // serialize into Protobuf:

    nowTime := time.Now()
    sendTime := &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)}
    protobufMsg := &routedmsgpb.RoutedMsg{
        Time: sendTime,
        Position: &routedmsgpb.Position{
            Latitude:  msgLatitude,
            Longitude: msgLongitude,
        },
        MsgBytes: msgContent,
    }
    protoBytes, err := proto.Marshal(protobufMsg)
    if err != nil {
        panic(err)
    }

    topic := fmt.Sprintf("VZCV2X/3/IN/%s/%s/%s/%d/UPER/BSM", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
    token := mqttClient.Publish(topic, 0, false, protoBytes)
    token.Wait()
    if token.Error() != nil {
        panic(token.Error())
    }

}

func publishAutopublishMessage(clientIdentity ClientIdentity, mqttClient pahomqtt.Client,
msgContent []byte, msgLatitude int32, msgLongitude int32) {
    // serialize into Protobuf:

    nowTime := time.Now()
    msgEndTime := nowTime.Add(time.Hour * 24)
    sendTime := &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)}
    protobufMsg := &routedmsgpb.AutoPublishAddMsg{
        MsgBytes:    msgContent,
        MsgType:     "RSA",   // this could be something else...in this case we'll imagine it's an
RSA
        Id:          "RSA_1", // must just be unique for the publishing client for the given msgType
        Description: "Work Zone",
        PositionOpt: &routedmsgpb.Position{
            Latitude:  msgLatitude,
            Longitude: msgLongitude,
        },
        FrequencySecOpt: 5,
        StartTime:       &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)},
        EndTime:         &timestamp.Timestamp{Seconds: msgEndTime.UTC().Unix(), Nanos:
int32(msgEndTime.UTC().UnixNano() % 1e9)},
    }
    protoBytes, err := proto.Marshal(protobufMsg)
    if err != nil {
        panic(err)
    }

    topic := fmt.Sprintf("VZCV2X/3/IN/%s/%s/%s/%d/UPER/AUTOPUB_ADD", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)
    token := mqttClient.Publish(topic, 0, false, protoBytes)
    token.Wait()
    if token.Error() != nil {
        panic(token.Error())
    }

    // could listen for AUTOPUB_ADD_ACK here....
}

func publishStaticMessage(clientIdentity ClientIdentity, mqttClient pahomqtt.Client, msgContent
[]byte, msgLatitude int32, msgLongitude int32) {
    // serialize into Protobuf:

    nowTime := time.Now()
    msgEndTime := nowTime.Add(time.Hour * 24)
    sendTime := &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)}
    protobufMsg := &staticmsgpb.StaticAddMsg{
        MsgBytes:    msgContent,
        MsgType:     "MAP",              // this could be something else...in this case we'll
imagine it's a MAP
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited

```
        Id:          "MAP_4th_and_Main", // must just be unique for the publishing client for the
given msgType
        Description: "MAP at 4th and Main",
        Position: &staticmsgpb.Position{
            Latitude:  msgLatitude,
            Longitude: msgLongitude,
        },
        StartTime: &timestamp.Timestamp{Seconds: nowTime.UTC().Unix(), Nanos:
int32(nowTime.UTC().UnixNano() % 1e9)},
        EndTime:   &timestamp.Timestamp{Seconds: msgEndTime.UTC().Unix(), Nanos:
int32(msgEndTime.UTC().UnixNano() % 1e9)},
    }
    protoBytes, err := proto.Marshal(protobufMsg)
    if err != nil {
        panic(err)
    }

    topic := fmt.Sprintf("VZCV2X/3/IN/%s/%s/%s/%d/UPER/STATIC_ADD", clientIdentity.EntityType,
clientIdentity.EntitySubtype, clientIdentity.Vendor, clientIdentity.ID)
    token := mqttClient.Publish(topic, 0, false, protoBytes)
    token.Wait()
    if token.Error() != nil {
        panic(token.Error())
    }

    // could listen for STATIC_ADD_ACK here....

}
```

This document does not reflect any product or service offered by Verizon. It is strictly for test and informational purposes only. Verizon confidential and proprietary. Unauthorized disclosure, reproduction or other use prohibited