Product Development

# FNV2-SOA Gateway Detailed Design

## Version 0.9

Version Date: June 19, 2019

**UNCONTROLLED COPY IF PRINTED**

**FORD CONFIDENTIAL**

# Table of Contents

# 1.Glossary

| | Description |
|---|---|
| **API** | Application Programming Interface |
| **ALM** | Application Lifecyle Manager |
| **ECG** | Enhanced Central Gateway |
| **ECU** | Engine Control Unit |
| **VNM** | Vehicle Network Manager |
| **SOA** | Service Oriented Architecture |
| **IPC** | Inter Process Communication |
| **GPB** | Google Protocol Buffer |
| **RPC** | Remote Procedure Call |

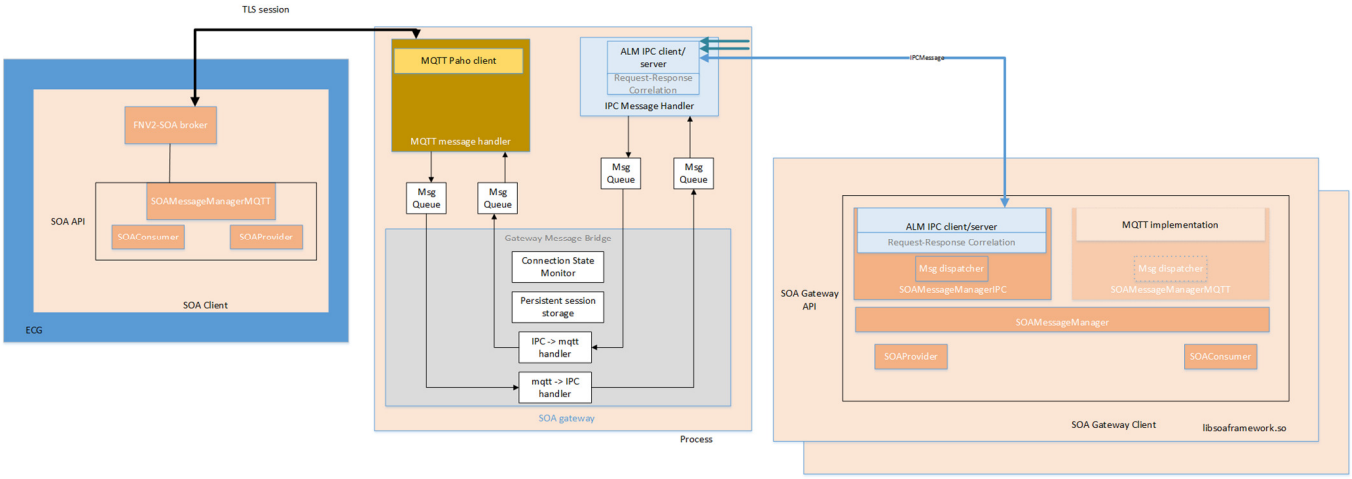
# 2. Introduction

The SOA Gateway component is present on each ECU communicating with the ECG. Its primarily role is to provide a single point of contact for ECU incoming and outgoing messages. Its architecture is flexible enough to accommodate various OSes and platforms including but not limited to SYNC and TCU. This document describes the internal structure of the Gateway particularly focusing on the IPC and security.

# 3. Overall architecture

## 3.1 IPC

The IPC is the communication channel between the Gateway and its clients called SOA Gateway clients. On the ECG, the transport layer is MQTT but on the ICUs where the Gateway is implemented, the ALM IPC is used. The Gateway clients use the SOA API which is implemented relying on IPC rather than MQTT. This public API is the same whatever transport protocol is used (MQTT or IPC) and this is what is already supported on the ECG.




### 3.1.1 SOAMessageManager

To abstract out the underlying protocol, the SOAMessageManager interface is part of the SOA library. On the ECG, the SOAMessageManagerMQTT is selected whereas on the ECU side, the SOAMessageManagerIPC is. The developer can instantiate the Message Manager MQTT or IPC.

The light blue boxes on the diagram above show the location of the IPC components within the SOA Gateway API. This ALM IPC is mainly RPC oriented meaning that the IPC client can send a request to the IPC server which will return the response. This behavior is appropriate for SOA Gateway client as most of the SOA API as publish, subscribe and RPC require a status to be returned.

To pass the client command (publish, subscribe, etc.) to the gateway, the SOaIpcMessage GPB is used with the following structure:

**IPCMessage**

```
syntax = "proto2";


import public "SoaIpcConnectionOptions.proto";
import public "SoaMessageWrapper.proto";


package fnv.soa.gateway;


message SoaIpcMessage{
    // The command to transfer over IPC (e.g. publish)
    required CommandType command = 1;
```

```
    // Service endpoint. The endpoint to be used by consumers to deliver service
requests.
    optional string endpoint = 2;

    // SOA Message
    optional SoaMessageWrapper soaMessageWrapper = 3;

    // connection options.
    optional SoaIpcConnectionOptions connectionOptions = 4;

    // Current IPC message structure version.
    required string version = 5;

    enum CommandType {
        PUBLISH = 0;
        SUBSCRIBE = 1;
        UNSUBSCRIBE = 2;
        CONNECT = 3;
        DISCONNECT = 4;
    }
}
```

**SOAMessageWrapper**

```
syntax = "proto2";

package fnv.soa.gateway;

// SOA message wrapper to wrap the SOA message to be passed over the SOA Gateway
message SoaMessageWrapper {
    required bytes SoaMessage = 1;
}
syntax = "proto2";

package fnv.soa.gateway;
```

```
// Connection options for Gateway Client
message SoaIpcConnectionOptions {
    required string serverFilePath = 1;
    required string clientFilePath = 2;
    optional bool cleanSession = 3;
    optional bytes keepAlive = 4;
    optional bool lastWillAndTestament = 5;
}
```

structure description:

- command: one of the command the Message manager supports which are PUBLISH, SUBSCRIBE, UNSUBSCRIBE, CONNECT, DISCONNECT.
- endpoint: for some of the commands like PUBLISH and SUBSCRIBE, the parameters of the command include the endpoint which needs to be passed to the Gateway. For the command not requiring an endpoint, (e.g. DISCONNECT), this can be omitted.
- soaMessageWrapper: this wraps the SOA message as the Gateway does not need to deserialize it and look into it. If the command does not support it, the message is omitted.
- version: the version of the structure of the IPC message

Once the SOA command is packaged into the SoaIpcMessage GPB, the client SoaMessageManagerIPC calls the ipcClient.sendRequest() to send the message over the IPC link which is going to be received by the Gateway side and transferred to the Gateway Message Bridge.

### 3.1.2 Unix domain sockets

The ALM-IPC is based on domain sockets. Once the socket is created, it is bound to a unique file path using the `SOCKADDR_UN` structure below:

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
       unsigned short int sun_family; /* AF_UNIX */
       char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

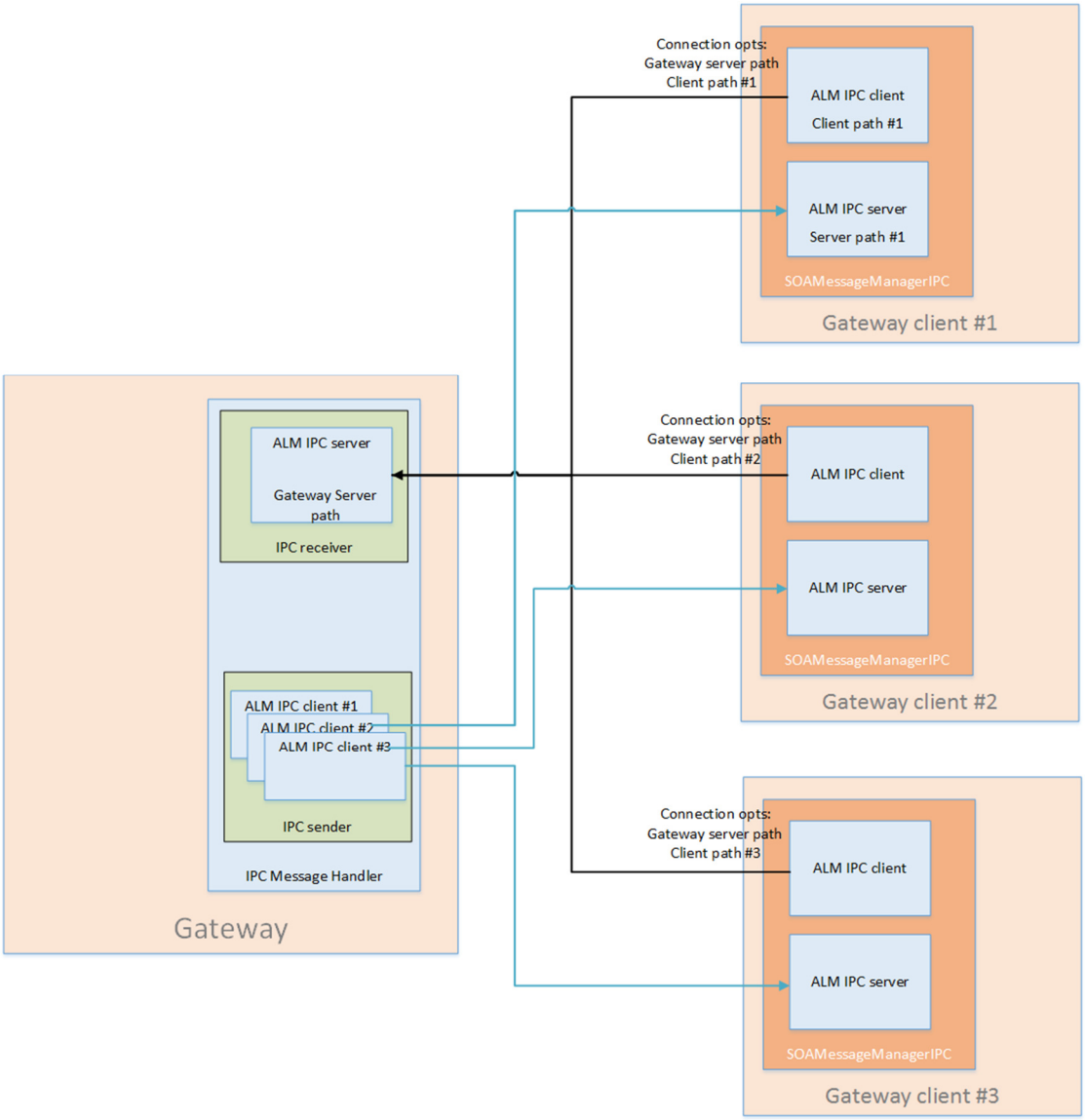
The sun_path is a unique null-terminated pathname string identifying the socket. The Gateway IPC server receiving the transaction from the client will have a single pathname all the client will use to connect. Each of the client will have a unique path to be able for the server to differentiate them.

When the socket is bound to a unique name by using the `bind` function, a file on the filesystem is created using the sun_path from the `SOCKADDR_UN`.

### 3.1.3 Initial connection

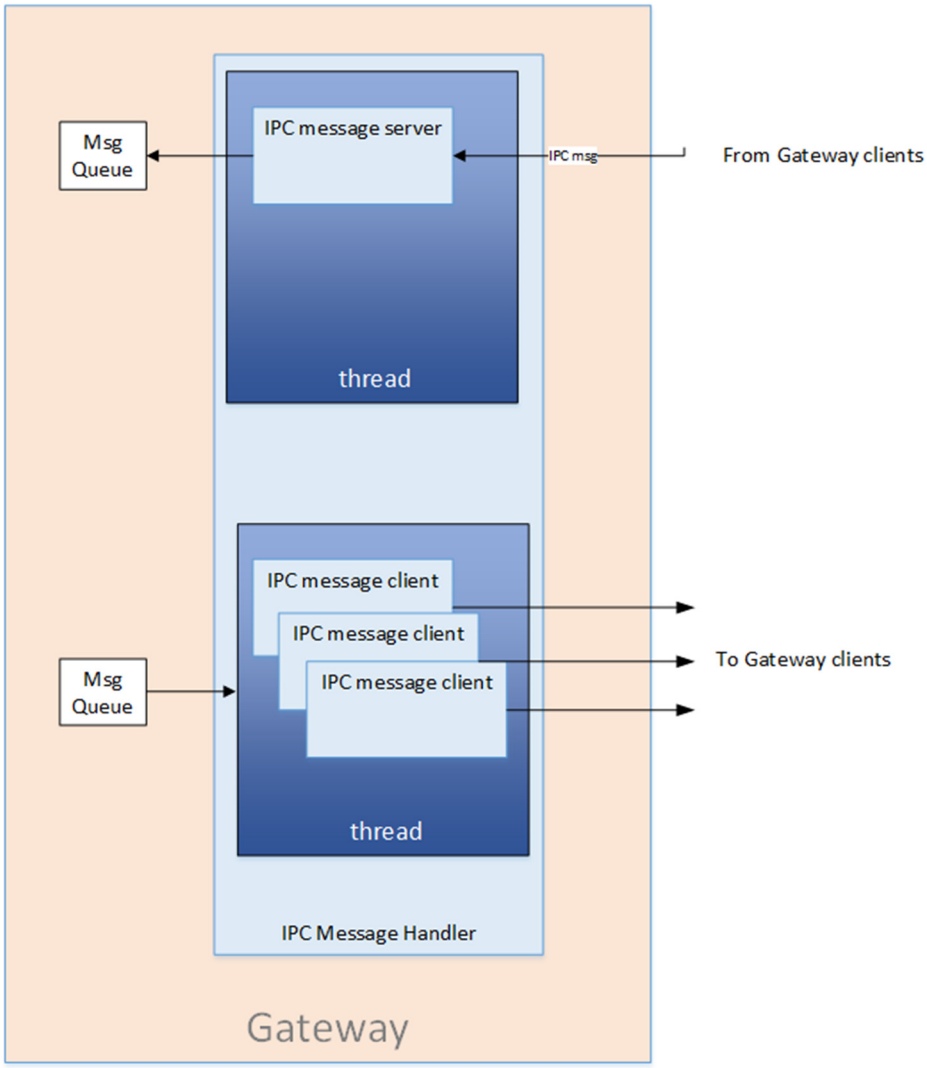
Each of the Gateway clients have they own IPC client and server. The client connects to the IPC server on the Gateway side passing in the connection option structure the Gateway IPC server identification (the Gateway server file path) as well as its own client file path (Client paht #n)

The IPC message handler on the Gateway side will create a dedicated client only if a subscribe is received. The file path of the server on the gateway client is derived from the client file path.



A single thread is used for the IPC server on the Gateway side as once a message is received, it is immediately push to the message queue connected to the message bridge.

The IPC client is also comprised of a single dedicated thread. As soon as a message is available from the message queue coming from the bridge, it is pull out and sent to the appropriate gateway client. Multiple IPC clients will be created on the Gateway byt only one will run at a time.




### 3.1.4 Gateway Message bridge

The Gateway Message Bridge is in charge of the transfer between MQTT and IPC protocol. To achieve this task, multiple sub components are defined.

### serialization/deserialization

- MQTT to IPC

The gateway receives and send commands over MQTT from/to the ECG. This is received by the MQTT message handler which is MQTT client. Each message received is forwarded to the MQTT to IPC message handler in the bridge in charge of the conversion and the mapping between endpoint and IPC clients.

- IPC to MQTT

When a command is sent from the Gateway client to the Gateway, it is received by the IPC message handler and forwarded to the IPC to MQTT message handler in the bridge.

- Message bridge

The message bridge includes the two components to converter IPC to MQTT and vice versa. A third component, the bridge main handler covers the mapping between endpoint and Gateway client and include support for connection management and persistence sessions.

### mapping tables

The bridge will maintain two tables: **Results** table and **Subscription** table.

The **Results** table will use the message ID as a key and map that to the client file path (this is the m_ipcClientIdentifier field in class IpcData).

- An entry is created by the IpcMqttHandler when a PUBLISH, CONNECT, DISCONNECT, SUBSCRIBE, or UNSUBSCRIBE command is received from the IpcHandler.
- The entry is removed by the MqttIpcHandler when status for that entry is returned to the IpcHandler.
- All entries for a client ID are removed by the IpcMqttHandler when a DISCONNECT command is received for that client.

The **Subscription** table will use the endpoint (topic name) as a key and map that to a list of client IDs.

- An entry is modified/created by the IpcMqttHandler when a SUBSCRIBE command is received from IPC.
- An entry is modified/removed by the IpcMqttHandler when an UNSUBSCRIBE command is received for a specific client ID.
- All entries for a client ID are removed by the IpcMqttHandler when a DISCONNECT command is received for that client (unless the session is persistent, see below).

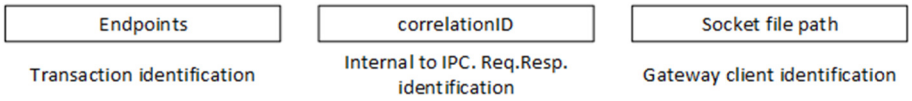
When MqttData for subscription clients is received by the MqttIpcHandler, it will retrieve the list of all clients subscribed to that particular topic. An IpcData instance is created for each client to receive the payload. Each instance is forwarded separately to the IpcHandler. This means that the bridge is managing the dispersal of data from one publisher to multiple subscribers.

In the event of an unexpected disconnect (i.e. client crash), all subscriptions will be removed regardless of persistence setting.

### client identification

Transactions exchanged between the Gateway client, Gateway and ECG need to be identified to be able to correctly route between the different Gateway clients.

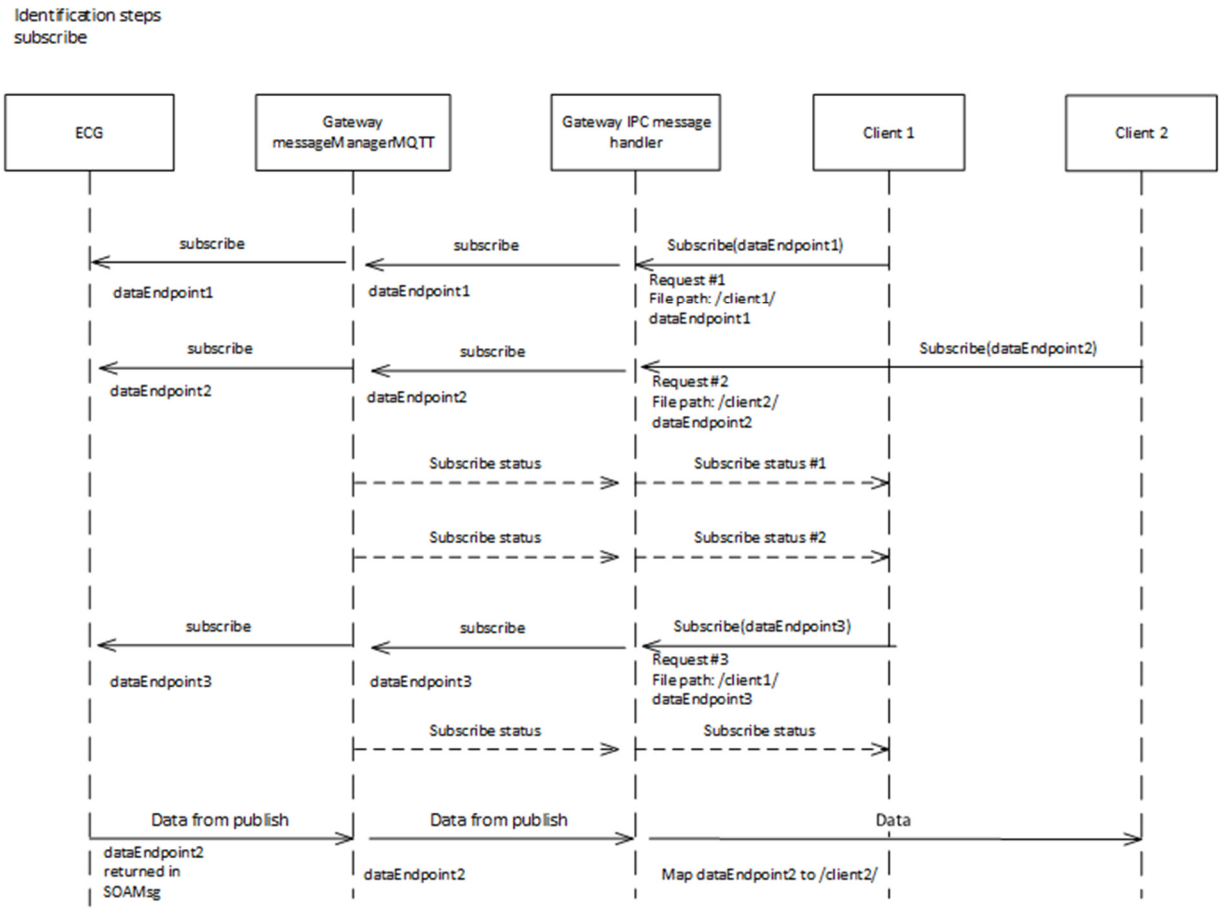
The three following identifiers are available for this purpose with for each of them, their identification purpose:




There are three types of identification required in the Gateway:

- The IPC is based on RPC type of communication. For each IPC message sent, a response (or ACK) is expected. The correlationID is used internally by the ALM-IPC API to send the response back matching the message sent. The correlationID is only used between the IPC client and server and not exposed to SOA messages.
- To map data received by the Gateway corresponding to subscribes. When the Gateway receive data, it needs to map this data to the appropriate client which subscribed to the data endpoint.
- For a given subscribe, the Gateway clients listening to it.

When the Gateway client connects to the Gateway, the SOA Message Manager does not use the same connection options for IPC as for MQTT (SOA IPC ConnectionOptions). Whereas a user name must be provided when connecting to the broker for identification purpose, the IPC uses a different mechanism. The client ID is not provided by the client per say but by the IPC server in the Message Manager on the Gateway which gets the file path identifying the Unix Domain socket of the connecting client. Once the IPC link is established and the file path of the client identified, any requests from the client will be tagged with its file path by the Gateway.




- The Gateway client sends a subscribe request to the Gateway
- The Gateway receives the request and maps the subscribe endpoint to the client file path
- The request is translated to an actual MQTT subscribe and the status of the subscribe is returned to the Gateway MessageManagerIPC
- The Gateway MessageManagerIPC returns to the Client the status of the request based on the correlationID. The request status is returned in the same order as the requests. On the sequence diagram above, back to back subscribes are sent. The first request status to come back is the status of the first subscribe.
- Once data is published and reached the Gateway, the MQTT message handler will pass the data on to the message bridge which will correlate the dataEndpoint to the client path to find which client to send the data to.

### persistent session storage

Each MQTT client when connecting to the broker can set the persistent session bit (cleanSession in the connectionOptions). If set, when the client reconnect, the broker will restore the client subscriptions which does not need to subscribe again. This feature is also implemented in the Gateway. This capability is supported at connection time when a client connect to MQTT. However, as multiple IPC clients are connected to the Gateway and a single MQTT connection is available with the ECG, the broker cannot handle this feature for the Gateway.

The proposal is to implement this feature on the Gateway itself and not rely on the broker. The Gateway will store all subscriptions for connected clients setting the "clean session" flag to false (in IpcData) when they connect. When a client has a persistent session, once this client reconnects, it does not need to subscribe again to the same topics/endpoints which are restored by the Gateway. This feature does not keep the missed messages when the client was off line but just stores the subscriptions. The subscriptions will be deleted in the event that the client connection fails as detected by the IPC KeepAlive functionality.

This capability could be enabled and disabled in a Gateway configuration file the same way is it done on the Mosquitto broker.

### 3.1.5 Connection State Monitor

The Gateway has 2 types of connections and needs to keep track of the state of each of them.

- IPC

The Gateway client can decide to disconnect calling the SOA disconnect API. In that case, only the IPC is going to be disconnected between the client and the Gateway. The MQTT connection with the ECG will be unaffected if other clients still have an active IPC connection. If all the Gateway clients have initiated a disconnect of the IPC, then the Gateway in turn will send a disconnect to the MQTT broker. If an IPC client is disconnected but the disconnect is not initiated by the client (e.g. in case of a crash or client restart), the MQTT connection will not be disconnected.

If any incoming message is for a disconnected client, the message is simply dropped if a persistent session for that client is not set.

If not client are connected or the IPC link is disconnected, no message will be sent to the IPC message handler.

If for any reason the IPC connection is lost from the Gateway side (i.e. not initiated by the client as a Gateway IPC server crash), the client needs to be notified. The ALM-IPC API has to include a mechanism to receive a call back (CONNECTION_LOST) if the IPC server is down/disconnected. The proposal would be to implement a similar mechanism as what is supported on MQTT with the KEEP_ALIVE timeout. If no message is exchanged over the IPC link, the client would send a ping to the server. No response from the server would mean a broken connection and a call the the CONNECTION_LOST callback.

- MQTT

if the MQTT connection is lost with the broker, the Gateway will report this new state to the Connection Status Monitor which will notify the Gateway client sending a connection lost message. The other option is to disconnect all the IPC clients so they will be notified by the IPC API if a KEEP_ALIVE mechanism is implemented as described above. This latter option is similar to what is supported on MQTT side. This is the recommended option as the implementation on the client side will be independent of the type of the transport used (MQTT/IPC).

If a pending message coming from the ECG is in the Gateway with a disconnected MQTT state, the message will be sent before shutting down the IPC connection. If this is a message coming from one of the client, it will be dropped as the client will be notified of the connection interruption.

- ICU power state

The ECU can also be in a power state preventing any TCP/IP connection. The Connection State Monitor will check the power state of the ECU and if it is in a mode preventing to connect MQTT interface (for instance because the TCP/IP stack is down), the Gateway will not try to reconnect automatically to the ECG and will wait for the appropriate power state to reconnect.

### 3.1.6 Connection/disconnection notification

The FNV2 Vehicle Network Manage has a requirement to be notified when an ECU has successfully (or unsuccessfully) connected to the SOA broker ( ECG-4533 - FNV SOA API: Gateway connection notification to FNV2-VNM Closed ).

The following three scenarios are supported by the Gateway:

- Connection

  The ECU via the Gateway successfully connects to the ECG broker. Once the TLS session is established, the Gateway publish the connection state to the ECG including the IP address of the connected ECU (can be obtained from the ECU VNM)

- Disconnection

  There are 2 distinctive scenarios which need to be addressed:

  - 
    - Graceful disconnect

    All the Gateway clients have initiated a disconnect. The Gateway before disconnecting from the broker will publish the disconnection event including the IP address of the ECU.

  - Ungraceful disconnect

    The disconnect what not done purposely which can occur in a crash of the Gateway for instance. When initiating the connection, the Gateway sets the Last Will and Testament message which is received by the VNM if the Gateway is disconnected and did not initiated the disconnect (via the SOA API). The message can include the IP address as this is known at connection time.

- Unsuccessful connection

It may happen that the Gateway tries to connect and fails (e.g. bad TLS credentials, ECG down, etc.). In the case of a connection failure, the Gateway will store the connection attempt and the reason of the failure and will send the list of failure once the first successful connection is made.

| Connection event | publishing endpoints |
|---|---|
| Succesful connection | SERVICES/DATA/GATEWAY/CONNECTIONSTATE |
| Graceful disconnect | SERVICES/DATA/GATEWAY/<br>CONNECTIONSTATE |
| Ungraceful disconnect | LWT message<br>SERVICES/LWT/GATEWAY/ |
| Uncessful connection | SERVICES/DATA/GATEWAY/<br>CONNECTIONSTATE |

**SoaGatewayConnectionFailedReason**

```
syntax = "proto2";

package fnv.soa.framework.protobuf;

// SOA message to broadcast connection information to VNM
message SoaGatewayConnectionState{
        // The connection state
        required ConnectionState state = 0;

        // ECU name e.g. TCU, SYNC. Same name as what is provided in the CommonName
field of the TLS Certificat
    required string EcuName= 1;

        // IP address
    optional string IpAddress = 1;

        // connection retries reason
    optional SOAGatewayConnectionFailedReason SoaConnectionFailedReason= 2;

        // Current IPC message structure version.
```

```
    required string version = 3;

        enum ConnectionState {
        CONNECTED= 0;
            DISCONNECTED= 1;
        }
}
```

**SoaGatewayConnectionFailedReason**

```
syntax = "proto2";

package fnv.soa.framework.protobuf;

// SOA Gateway message to list the unsuccessful connection tentative

message SOAGatewayConnectionFailedReason {
    // List of connection failure reason
    repeated ConnectionFailedReason reason= 0;
        repeated string IpAddress= 1;

        enum ConnectionFailed {
        NOT_RESPONDING= 0;
            TLC_CREDENTIALS= 1;
                UNKOWN= 2;
        }
}
```

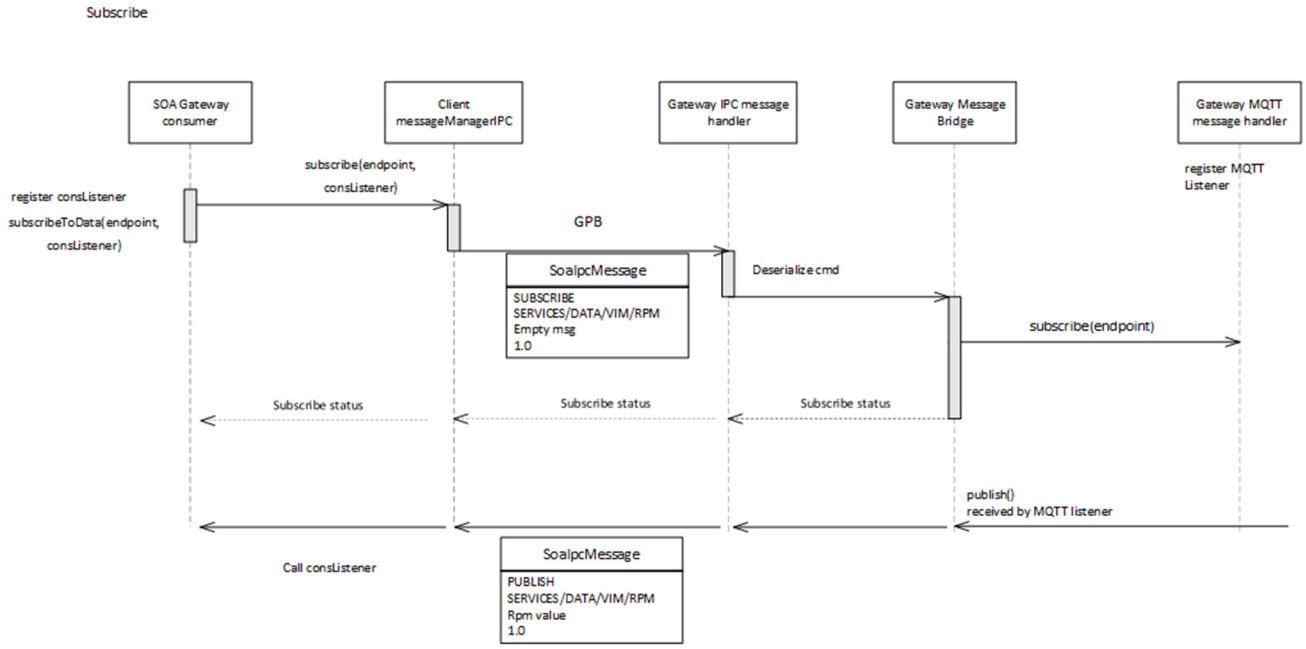
### 3.1.7 Message flow

The two scenarios represented on the diagrams below are for a subscribe and a publish from the Gateway client side.

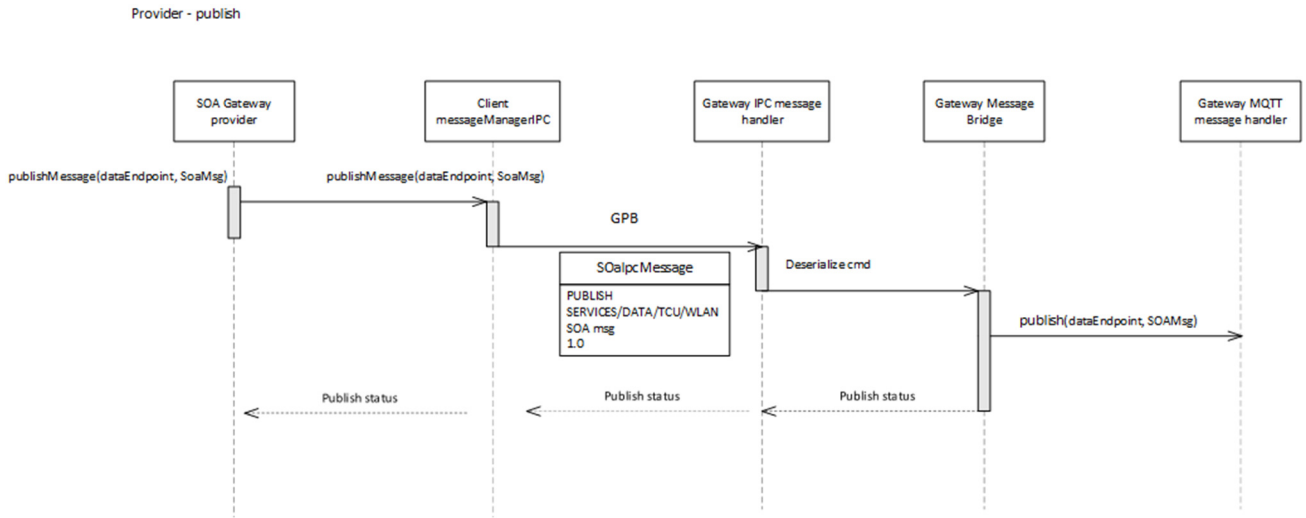
The subscribe is initiated by the client and is no different than what happens on the ECG side. The client calls the subscribeToData() public API specifying the endpoint and the listener to receive the response (ConsListener). The API call is an actual call to the subscribe() private API of the Client SoaMessageManagerIPC. Instead of directly calling the MQTT Paho library, the subscribe API translates the call to an IPC message including the command (SUBSCRIBE) and the topic. This message is received by the Gateway through its IPC message handler and forwarded to the Gateway Message Bridge. The bridge is in charge of translating the IPC message to a MQTT message with the same parameters (endpoint). The MQTT message manager then do the actual transaction through the Paho library.

Data (from publishes) is received by the MQTT Message Handler. The mapping between topic and Gateway client is done by the message bridge.

A single listener is used in the MQTT message handler to subscribe to all topics from all Gateway Clients. This MQTT listener is the main entry point for all messages matching the subscribes of the client.




The sequence for the publish is similar to the subscribe described above. The publishMessage() call endup in the Client SoaMessageManagerIPC and is translated to an IPC message sent over the IPC link to the Gateway. The Gateway IPC message handler receives the IPC message and send it to the bridge for further processing and translation to an actual MQTT publish call done in the MQTT Message Handler.

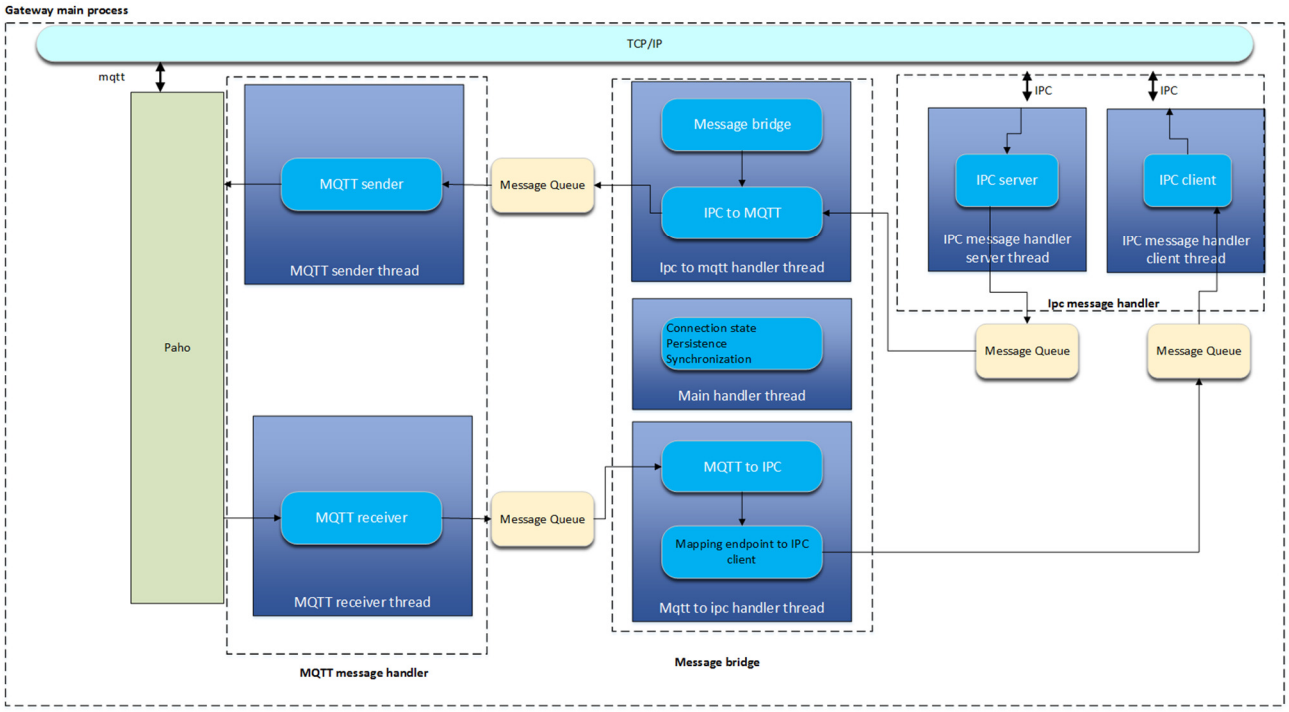


### 3.1.8 Synchronous/Asynchronous calls

The Gateway client can use synchronous or asynchronous SOA API calls which are packaged into the GPB and sent over IPC. Once received by the IPC message handler, the notion of synchronous or asynchronous transaction is not preserved as the Gateway will map all the requests to asynchronous calls only. This is required as the Gateway is connected to multiple clients and a synchronous call from the Gateway would block all the other message processing from the other clients as the Gateway Message Bridge is implemented as a single thread.

## 3.2 Process and threading model

The IPC API is a blocking API meaning that if the IPC server is receiving a command, the client.sendRequest() on the client side will block. To minimize the possibility of blocking clients, the IPC server on the Gateway is run in its own thread (IPC message handler thread). Once the command is received, it is immediately posted to a message queue to release the server and enable a new command to be received.




## 3.3 Security

The other purposes of the Gateway in addition to bridging MQTT and IPC domains is security. The client connecting to the ECG via the Gateway must be authorized to do so and must have the permissions to access a given service as described in the SOA Security Architecture HLD.

The main methods to cover the above two requirements are the extraction of a client identifier appended to the topic string to be used by the broker ACL and the setup of the TLS session with the MQTT broker on the ECG.

### 3.3.1 Unix credentials and identifier

Once the client connects to the IPC server in the Gateway, the credentials of the client can be obtained with the getSockOpt() call as shown below.

```
struct ucred client_creds;
socklen_t szCreds = sizeof(client_creds);
memset(&client_creds, 0, szCreds);

int status = getsockopt(socketFileDesc, SOL_SOCKET, SO_PEERCRED, &client_creds,
&szCreds);
if (status == 0) {
        // credentials available: (long) client_creds.pid, (long)
client_creds.uid, (long) client_creds.gid);
} else {
    // Error. errno is set with the corresponding error
}
```

Those credentials are then used to compose the **clientIdentifier** defined as: **"<NodeID>_<UID>"**

with the NodeID being TCU or SYNC and the UID, the uid credential returned by the getsocopt(). The NodeID is provided by the CommonName in the TLS certificate.

The client identifier is then appended to all outgoing message endpoints.

### 3.3.2 TLS session

The Gateway is seen as a MQTT client from the Broker point of view and need to be configured to use SSL.
The MQTTAsync_SSLOptions is provided for this purpose as it set the private key, CA certificate and other SSL related parameters.

```
SoaConnectionOptions::SharedPtr connOpts =
SoaConnectionOptions::createConnectionOptions();
MQTTAsync_SSLOptions sslopts = MQTTClient_SSLOptions_initializer;
connOpts->setSSL(&ssl_opts);
```

The keys will be exported as a virtual file system so the Gateway can get access to the information as regular files:

/dev/keymgr/SOA_Authenticate_Private_key.pem

/dev/keymgr/SOA_Authentication_Public_cert.pem

/dev/keymgr/SOA_TCU_CA_Cert.pem

On Linux, the Key Manager will exports the keys as a device using the FUSE FS. On QNX a similar mechanism is provided through the Resource Manager.