# fnv::soa: (c) Writing a SOA Service Consumer (from ECG)



## Intro

An FNV SOA Service Consumer implementation leverages the SoaConsumer class to:

- connect to the network,
- send service requests,
- receive service responses,
- subscribe and unsubscribe to asynchronous event data,
- receive subscribed asynchronous event data messages,
- request the status of a service,
- subscribe and unsubscribe to service status updates,
- (coming soon) - get a list of services which handle a given command ID,
- (coming soon) - request on-demand data broadcasts

## Main Features

### Send a request to a provider and (optionally) receive a response

Facilitates creating a request message, sending the request to a service provider, and receiving the response.

### Subscribe to event data messages and receive those messages

Facilitates subscribing to data messages sent by providers and receiving those messages.

Facilitates unsubscribing to data messages.

### Get the status of a service

Facilitates getting the operational status of a service provider once or subscribing to status changes.

Facilitates unsubscribing to status updates.

## Requirements

You need to know your endpoint(s)

- For the MQTT transport mechanism, endpoints are MQTT topics.
- SoaClientEndpoint class is a container class used to hide the details of the syntax
- SoaClientEndpoint objects can be created at compile time from base endpoint strings using **SOA Framework helper macros** (not yet available in R0.0.4.1)

    Example: a base endpoint string might be VIM/CANPDUSERVICE

    A request endpoint string created from the base endpoint would be SERVICES/REQUEST/VIM/CANPDUSERVICE  This is the endpoint zero or more SoaConsumer's sends a remote call request to and a SoaProvider listens on

    A response endpoint string created from the base endpoint would be SERVICES/RESPONSE/VIM/CANPDUSERVICE/<UNIQUE_ID>  This is the endpoint a SoaProvider sends a remote call response to and the SoaConsumer listens on

        - note: UNIQUE_ID is generated by the SOA Framework

    A data endpoint string created from the base endpoint would be SERVICES/DATA/VIM/CANPDUSERVICE  This is the endpoint zero or more SoaConsumer's listens on and a SoaProvider sends a broadcast message to

**Topic naming**

A SoaConsumer endpoint topic string used to create a SoaConsumer object ***MUST*** start with **SERVICES/RESPONSE/**

A provider endpoint topic string to which a remote call request message is sent (via remoteCall(...)) ***MUST*** start with **SERVICES/REQUEST/** . NEVER send a remoteCall request to an endpoint that starts with SERVICES/RESPONSE

Data broadcast message endpoint topic strings ***MUST*** start with SERVICES/DATA/

**Endpoint syntax**

- Endpoint topic strings DO NOT start with a forward slash
- Endpoint topic strings ARE all uppercase
- Endpoint topic strings ARE case sensitive
- Endpoint topic strings MUST NOT include an MQTT topic wildcard symbol. ( # or + )
- The first topic level in a topic string is SERVICES  note this word both starts and ends with an 'S'
- The empty string ("") is NOT a valid endpoint topic string.

**No Bad Endpoints**

Methods which return a pointer to an endpoint object (i.e. shared_ptr<SoaClientEndpoint>) return a nullptr if the endpoint would be invalid.

**IMPORTANT** to know if you call toString() on a returned endpoint object.

**Compile-time C++ constexpr helper functions are available**

"C++-improved" equivalents to C #define macros are available for automating the creation of SoaClientEndpoint objects using only application specific information. These constructs free the application developer from having to explicitly include MQTT topic prefixes such as SERVICES /DATA/ and SERVICES/REQUEST/ . The following constexpr methods can be used to create a C-style (char []) string at ***compile time*** by having the compiler concatenate the correct topic prefix with your application-specific suffix. These are static methods

**Convenience methods**

```
// for creating a service provider topic string --> SERVICES/REQUEST/  is prepended to suffix (used
when registering- or making a request to a service provider)
char [] SoaClientEndpoint::createRequestEndpointName(const char[] suffix)
// for creating a data broadcast topic string --> SERVICES/DATA/ is prepended to suffix (used when
publishing- or subscribing to broadcast data)
char []  SoaClientEndpoint::createDataEndpointName(const char[] suffix)
// for creating a consumer response topic string --> SERVICES/RESPONSE/ is prepended to suffix (used
when creating a consumer)
char []  SoaClientEndpoint::createResponseEndpointName(const char[] suffix)
```

**Example**

```
// build my service endpoint topic C-string SERVICES/REQUEST/ONDEMAND_FT1 at compile time
// while the C++ std::string is created at run-time from it
std::string DATA_TOPIC1 = std::string(SoaClientEndpoint::createRequestEndpointName( "ONDEMAND_FT1" ));
// endpoints can be created using the message manager's factory method
SoaClientEndpoint::SharedPtr DATA_ENDPOINT1 = messageManager->createClientEndpoint( DATA_TOPIC1 );
// ...
```

You need to know your message formats

- Message payloads are binary data wrapped in C++ std::string objects, formatted using Google protobufs
- The protobuf definitions associated with services are available in the source code repository to consumer developers
- A service's request message format is used by consumers making requests
- A service's response message format is used by consumers processing the response data
- A service's data message format is used by consumers listening for data broadcasts

You need to know your required connection options for the MQTT broker connection

- The options are set in a container class SoaConnectionOptions
- These are largely the values defined in MQTTClient_connectionOptions (See MQTTClient.h)
- keepAliveInterval  (in seconds, used to time ping messages when normal client communication is absent)
- cleanSession (boolean, clean previous session state data on the server upon reconnection)
- connectTimeout (in seconds, time for the connect to complete)
- username  (used for SSL)
- password  (used for SSL)
- string clientID  each client of the MQTT broker needs a unique client identifier. Set this in your code during the development phase.
- connection lost listener  a listener object that implements the onConnectionLost(string cause) callback
- auto reconnect (a boolean)  if ungracefully disconnected, should SOA automatically try to reconnect?
- reconnection listener  a listener object that implements the onReconnect( SoaErrorCode status ) callback
- connectListener  a listener object that implement the onConnect( SoaErrorCode status ) callback for the non-blocking connectAsync() status only

plus ...

- boolean isResponseChannel  *( not currently used )*

Refer to the section Connection APIs for more info

## Connection Options Defaults

| Option | Default Value | Notes |
|---|---|---|
| keepAliveInterval | 60 seconds | |

| | | |
|---|---|---|
| cleanSession | true | |
| connectTimeout | 30 seconds | |
| username | nullptr | |
| password | nullptr | |
| clientID | "CLIENTID_NOT_SET" | |
| isResponseChannel | false | |
| auto-reconnect | false | Set to true if you want SOA to attempt to reconnect upon an ungraceful disconnect |
| reconnection time options | reconnect timeout = 1 hour<br><br>minimum retry interval = 0.5 seconds<br><br>maximum retry interval = 1 minute<br><br>retry interval multiplier = 2.0 | These arguments define the functioning of the non-blocking connection algorithm. The algorithm will try to connect immediately upon being triggered. If no error but the server is not available, the algorithm will retry after a calculated interval. The timeout indicates the maximum time from start to end for all retries. The minimum interval is the wait time before retrying after first connection attempt. The multiplier changes the interval before the next retry. The maximum interval caps the calculated next interval. More details at Background Connection Retry Task |
| connectionLostListener | nullptr | (optional) if provided, the SOA framework will call this when there is an ungraceful disconnect |
| reconnectionListener | nullptr | (optional) if provided, the SOA framework will call this with the status of the reconnection attempt when the connection retry algorithm completes. Required for app to explicitly call reconnect(), optional for auto-reconnect |
| connectionListener | nullptr | (optional) Required to use the non-blocking method connectAsync(). The SOA framework will call this with the status of the connection attempt when the connection retry algorithm completes. |

# Choices

## Blocking vs. non-blocking methods

All SOA Framework functionality is provided in either blocking or non-blocking versions. Methods providing non-blocking versions end with "Async". Blocking and non-blocking methods can be used together in a single application as required by the developer's design.

Blocking methods generally return a SoaErrorCode when the status is known, and most non-blocking methods return void immediately. Some blocking methods and one non-blocking method return a shared pointer to a SoaActionResult<DataType> template type which aggregate a SoaErrorCode with an appropriate DataType. The non-blocking method returns an error if no cached value is available.

Example: SoaActionResult<SoaMessage>::SharedPtr (which is a shared_ptr<SoaActionResult<SoaMessage>>)

**Threading for blocking and non-blocking methods**

The SOA Framework executes all public SoaConsumer methods on a single **worker thread** shared among all SoaProvider and SoaConsumer instances. Calling a SOA method places a work task on a FIFO queue to be run-to-completion on the **worker thread** . In this way, all SOA methods are executed **serially** in the order they are called. Blocking methods block until a result is available, while non-blocking versions return immediately and their results are received in a user-provided callback. Result callbacks and message listener callbacks are executed on a single **callback thread** shared among all SoaProvider and SoaConsumer instances.

**No calling blocking methods from the Callback thread**

**IMPORTANT!** Blocking methods cannot be called from a callback thread. If a developer creates code such that a blocking method is called from a callback thread, the method will return immediately returning an **ERROR_ILLEGAL_CALL_THREAD** error code. If a developer wants to call a SOA API from a callback, they can and must use the non-blocking version.

### Blocking methods:

SoaErrorCode **initialize()**

SoaErrorCode **subscribeToData**(...)

SoaErroCode **unsubscribeToData**(...)

SoaErrorCode **subscribeToServiceStatusUpdate**(...)

SoaErroCode **unsubscribeToServiceStatusUpdate**(...)

SoaActionResult<SoaMessage>::SharedPtr **remoteCall**(...)

~~SoaActionResult<SoaServiceStatus>::SharedPtr **requestServiceStatus**(...)~~ **NOT SUPPORTED**

SoaActionResult<SoaServiceMetadataList>::SharedPtr **getCommandHandlers**(...)

### Non-blocking methods:

void **subscribeToDataAsync**(...)

void **unsubscribeToDataAsync**(...)

void **subscribeToServiceStatusUpdateAsync**(...)

void **unsubscribeToServiceStatusUpdateAsync**(...)

void **remoteCallAsync**(...)

~~void **requestServiceStatusAsync**(...)~~ **NOT SUPPORTED**

void **getCommandHandlersAsync**(...)

~~SoaActionResult<SoaServiceStatus>::SharedPtr **requestServiceStatusCached**(...)~~ **NOT SUPPORTED**

### Blocking methods:

| Blocking Method | What it does | Relative time to complete | Comments |
|---|---|---|---|
| SoaErrorCode **initialize()** | connects to the MQTT broker, enables calling (is a required post-instantiation step) | quick | This method connects the client to the broker and sends a subscription request used for responses to remote calls. If the connection cannot be established, this method will fail with an error code. If the connection is already established, the connection sub-task is skipped.<br><br>It is highly recommended to call SoaMessageManager::connectAsync() before calling initialize() to avoid failing on this method, BUT you MUST wait for the successful connection status to be indicated via the asynchronous callback to the SoaConnectionListener. You MUST NOT call this method after calling connectAsync() without waitiing for the successful connection status.<br><br>this may be deprecated in the future |
| SoaErrorCode **subscribeToData**(...) | subscribes to MQTT topic | quick | |

| | | | |
|---|---|---|---|
| SoaErroCode **unsubscribeToData**(...) | unsubscribes to MQTT topic | quick | |
| SoaErrorCode **subscribeToServiceStatusUpdate**(...) | networks with the ServiceManager | extended for networking with ServiceManager | |
| SoaErroCode **unsubscribeToServiceStatusUpdate**(...) | networks with the ServiceManager | extended for networking with ServiceManager | |
| SoaActionResult<SoaMessage>::SharedPtr **remoteCall**(..., timeout) | subscribes to MQTT topic,<br><br>sends MQTT message, receives response,<br><br>unsubscribes to MQTT topic | extended for networking provider and waiting for response | waits for response |
| SoaErrorCode **remoteCall** (...) | sends MQTT message | quick | no RPC response |
| ~~SoaActionResult<SoaServiceStatus>::SharedPtr **requestServiceStatus**(...)~~ | ~~networks with the ServiceManager~~ | ~~extended for networking with ServiceManager~~ | **NOT SUPPORTED** |
| SoaActionResult<SoaServiceMetadataList>::SharedPtr **getCommandHandlers**(...) | networks with the ServiceManager | extended for networking with ServiceManager | |

### Non-blocking methods:

| Non-blocking Method | What it does | Relative time to complete | Comments |
|---|---|---|---|
| void **subscribeToDataAsync**(...) | subscribes to MQTT topic | quick | call returns immediately, action status callback is called as soon as result of sending subscribe message is known (guaranteed to be called before the first data broadcast arrives) |
| void **unsubscribeToDataAsync**(...) | unsubscribes to MQTT topic | quick | call returns immediately, , action status callback is called as soon as result of sending unsubscribe message is known |
| void **subscribeToServiceStatusUpdateAsync**(...) | networks with the Service Manager | extended for networking with ServiceManager | call returns immediately, action status callback is called as soon as result of this action is known |
| void **unsubscribeToServiceStatusUpdateAsync**(...) | networks with the Service Manager | extended for networking with ServiceManager | call returns immediately, , action status callback is called as soon as result of this action is known |

| | | | |
|---|---|---|---|
| void **remoteCallAsync**(..., timeout) | subscribes to MQTT topic,<br><br>sends MQTT message, receives response,<br><br>unsubscribes to MQTT topic | extended for networking provider and waiting for response | call returns immediately, listener receives response<br><br>action status callback is called when the result of this action is known. If the publish of the remote call request is successful, the callback will be called when the response arrives or a timeout occurs. If a timeout occurs, the binding with the user's remote call listener is cancelled, and it is guaranteed to NOT be called with a remote call response. Any status other than NO_ERROR indicates the user's remote call listener will not be called, The action status callback is guaranteed to be called before the remote call response. |
| void **remoteCallAsync**(...) | sends MQTT message | quick | call returns immediately no RPC response, action status callback is called as soon as result of publishing the remote call request message is known |
| ~~void **requestServiceStatusAsync**(...)~~ | ~~networks with the Service Manager~~ | ~~extended for networking with ServiceManager~~ | **NOT SUPPORTED** |
| void **getCommandHandlersAsync**(...) | networks with the Service Manager | extended for networking with ServiceManager | call returns immediately, action status callback is called as soon as result of this action is known, and is guaranteed to be called before the command handlers listener is called |
| ~~SoaActionResult<SoaServiceStatus>::SharedPtr **requestServiceStatusCached**(...)~~ | ~~returns cached status if available~~ | ~~quick~~ | **NOT SUPPORTED** |

Almost all of the public methods, other than those used to get or set the blanket timeout, are limited by a timeout to protect the application from hanging.

Timeouts are specfied in milliseconds.

Timeouts can be specified:

- per method call
  - always for methods whose response is dependent on a remote network element (ex. remoteCall(...) which waits for a response)
  - by user choice of alternative methods whose response is not dependent on a remote network element (ex. subscribing for data messages)
- as a blanket timeout applied to most methods whose response is not dependent on a remote network element
  - developer chooses to set the blanket timeout or use the default by not setting the timeout

The assigned blanket timeout can be queried. The time remaining before timeout cannot be queried at runtime.

### Timeout Causes:

A method can time out due to the following reasons:

- waiting for a network response - Ex. waiting for a response to a remote call request.
- delay in executing the method task (See Threading for blocking and non-blocking methods)

**Invalid Timeout Values**

A timeout parameter value MUST be greater than zero ( > 0 ) to be valid. An invalid timeout value will be rejected by all methods to which it is passed as an argument, causing the method to fail with return value SoaErrorCode::ERROR_PARAM_BAD

**Typical Timeout Values**

At the time of writing, typical timeout values have not been established. The default blanket timeout is 1 second (1000 ms) in release 0.0.4.1.

**Timeouts - seconds vs milliseconds**

NOTE: Timeout values are in milliseconds EXCEPT SoaConnectionOptions timeouts, which are in seconds.

## Waiting for Responses:

This choice is generally NOT yours to make as a consumer developer. This is defined by the service provider. The provider application may or may not send a response to a request.

If a service sends a response to a request, you will know this when you develop the consumer application. Even if you choose to ignore the response, you SHOULD use the version of remoteCall(...)/remoteCallAsync(...) which expects a response, and implement a response listener callback. Otherwise, the service may ignore your request altogether. If in doubt, assume you need to send your consumer endpoint in the request message.

**Sending a response or not sending a response**

Services that wait for requests to act on may or may not send response messages as a reaction to a request type.

In either case the service will either **always** respond or **never** respond, depending on the contract (consumers need to know which it is). Consumers making the request either always expect a response or always don't expect a response.

## Remote Calls:

There are two distinctly different blocking versions of remoteCall(...), and two distinctly different non-blocking counterparts; remoteCallAsync (...). This collection of methods make use of the timeout argument differently than the un/subscribe... methods.

- **remoteCall**(..., **timeout**) and **remoteCallAsync**(..., **timeout**) methods expect a response, and they include a timeout argument to protect the consumer application in the event a response is not received in the expect amount of time.
- **remoteCall**(...) and **remoteCallAsync**(...) methods are for one-way requests. No response is expected, or waited for. To avoid confusion when selecting these methods, a custom timeout argument is not included. The blanket timeout value is applied to these methods.

**Blanket timeout applies to One-Way remoteCall* methods**

The blanket timeout value is applied to One-Way remoteCall... methods. These don't take timeout arguments. Only remoteCall... methods which expect a response have a timeout argument which applies to receiving the response.

## Service Status Info:

Do you need to know if a service is available?

You can subscribe for updates to the status of a service provider.

When a service provider registers its service, a retained message is broadcast by the provider that it is available. "Retained" means the MQTT broker stores it and re-broadcasts it whenever it gets a new subscription for the topic. When a service provider unregisters its service, an unretained message is broadcast by the provider that is not available, and the retained status of the previous "available" message is cancelled at the MQTT broker.

Subscribe to service status updates for a specific service. This enables both synchronous and asynchronous access to the service status via the SOA Framework.

Call subscribeToServiceStatusUpdate(...) or subscribeToServiceStatusUpdateAsync(...) to start the subscription.

Once subscribed:

- A status update message will be sent to you immediately only if the MQTT broker has a retained message (i.e. only if the service is available) with the current operational status: RUNNING
- Your defined service status update listener callback method will be called asynchronously on the callback thread whenever a subscribed status update is received.

The subscription will persist with status updates being sent to your status update listener until it is terminated. Unsubscribing may not terminate the implicit subscription.

Call unsubscribeToServiceStatusUpdate(...) or unsubscribeToServiceStatusUpdateAsync(...) to end the subscription.

## Matrix of Choices:

| | Blocking | Non-Blocking |
|---|---|---|
| Blanket Timeout | subscribeToData(...)<br>unsubscribeToData(...)<br>subscribeToServiceStatusUpdate(...)<br>unsubscribeToServiceStatusUpdate(...) | subscribeToDataAsync(...)<br>unsubscribeToDataAsync(...)<br>subscribeToServiceStatusUpdateAsync(...)<br>unsubscribeToServiceStatusUpdateAsync(...)<br>requestServiceStatusCached(...) |
| Custom Timeout | subscribeToData(..., timeout)<br>unsubscribeToData(..., timeout)<br>subscribeToServiceStatusUpdate(..., timeout)<br>unsubscribeToServiceStatusUpdate(..., timeout) | subscribeToDataAsync(..., timeout)<br>unsubscribeToDataAsync(..., timeout)<br>subscribeToServiceStatusUpdateAsync(..., timeout)<br>unsubscribeToServiceStatusUpdateAsync(..., timeout) |
| No Reply | remoteCall(...) | remoteCallAsync(...) |
| Reply Expected | remoteCall(..., timeout)<br>requestServiceStatus(..., timeout)<br>getCommandHandlers(..., timeout) | remoteCallAsync(..., timeout)<br>requestServiceStatusAsync(..., timeout)<br>getCommandHandlersAsync(..., timeout) |

## Data Structures, Listeners and Callbacks

The following SOA Framework classes are used:

- SoaConsumer
  - This class provides the functionality documented on this page.
- SoaClientEndpoint
  - This class abstracts the identify of a message source and/or destination.
  - Over the current MQTT transport layer, this class contains a topic string.
- SoaMessage
  - The SOA Framework layer's network message wrapper class.
  - The application payload is rather opaque to the SOA Framework.
  - The application creates SoaMessage objects to send to the network.
  - The application receives a SoaMessage object when a request is received.
  - See Message Structures
- SoaServiceDirectoryManager
  - An object used to get or track the status of a dynamic service.
  - SoaConsumer requests for service status are processed by the service directory manager.
- SoaServiceStatus
  - An enum class defining service status values such as available, not_available, running.
- SoaDataListener
  - An abstract class which needs to be realized and implemented by the consumer developer to listen for service event data.
  - It defines the signature of the callback method to be called when subscribed data is received.
  - A reference to an object of this class is provided by the application to the SOA Framework when subscribing for event data from a service.
- SoaRemoteCallListener
  - An abstract class which needs to be realized and implemented by the consumer developer to receive a service response asynchronously after making a remote call (i.e a service request) to a service.
  - It defines the signature of the callback method to be called when the remote call response message is received.
  - A reference to an object of this class is provided by the application to the SOA Framework when making an asynchronous remote call which provides a response.
- SoaServiceStatusListener
  - An abstract class which needs to be realized and implemented by the consumer developer.
  - It defines the signature of the callback method to be called when a service status update message is received.
  - A reference to an object of this class is provided by the application to the SOA Framework when subscribing for service status updates.
- SoaCommandHandlersListener
  - An abstract class which needs to be realized and implemented by the consumer developer.
  - It defines the signature of the callback method to be called when info is received about services which handle a given command ID.
  - A reference to an object of this class is provided by the application to the SOA Framework when making an asynchronous requesting for a list of services which handle a command ID.
- SoaSubscribeToDataContext

        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking subscribeToDataAsync(...) method.
        - This object includes references to all of the data used to call the subscribeToDataAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaRemoteCallContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking remoteCallAsync(...) method where a response is expected.
        - This object includes references to all of the data used to call the remoteCallAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaOneWayRemoteCallContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking remoteCallAsync(...) method where a response is not expected.
        - This object includes references to all of the data used to call the remoteCallAsync(...) method as a means to provide context to the action result listener callback method.
    - ~~SoaRequestServiceStatusContext~~ **NOT SUPPORTED**
        - ~~An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking requestServiceStatusAsync(...) method.~~
        - ~~This object includes references to all of the data used to call the requestServiceStatusAsync(...) method as a means to provide context to the action result listener callback method.~~
    - SoaSubscribeToServiceStatusContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking subscribeToServiceStatusUpdateAsync(...) method.
        - This object includes references to all of the data used to call the subscribeToServiceStatusUpdateAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaUnsubscribeToDataContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking unsubscribeToDataAsync(...) method.
        - This object includes references to all of the data used to call the unsubscribeToDataAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaUnsubscribeToServiceStatusContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking unsubscribeToServiceStatusUpdateAsync(...) method.
        - This object includes references to all of the data used to call the unsubscribeToServiceStatusUpdateAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaGetCommandHandlersActionContext
        - An object of this class is created by the SOA Framework and passed as an argument to the application's action result listener for the non-blocking getCommandHandlersAsync(...) method.
        - This object includes references to all of the data used to call the getCommandHandlersAsync(...) method as a means to provide context to the action result listener callback method.
    - SoaActionResultListener<ContextType>
        - A class template to allow the Consumer developer to implement a context specific action result callback listener class for the different non-blocking methods.
        - A class which realizes this template includes a callback method onActionReceived(SoaErrorCode, shared_ptr<ContextType>) which is called with the result of the non-blocking method.

The SoaConsumer class is dependent on these additional classes:

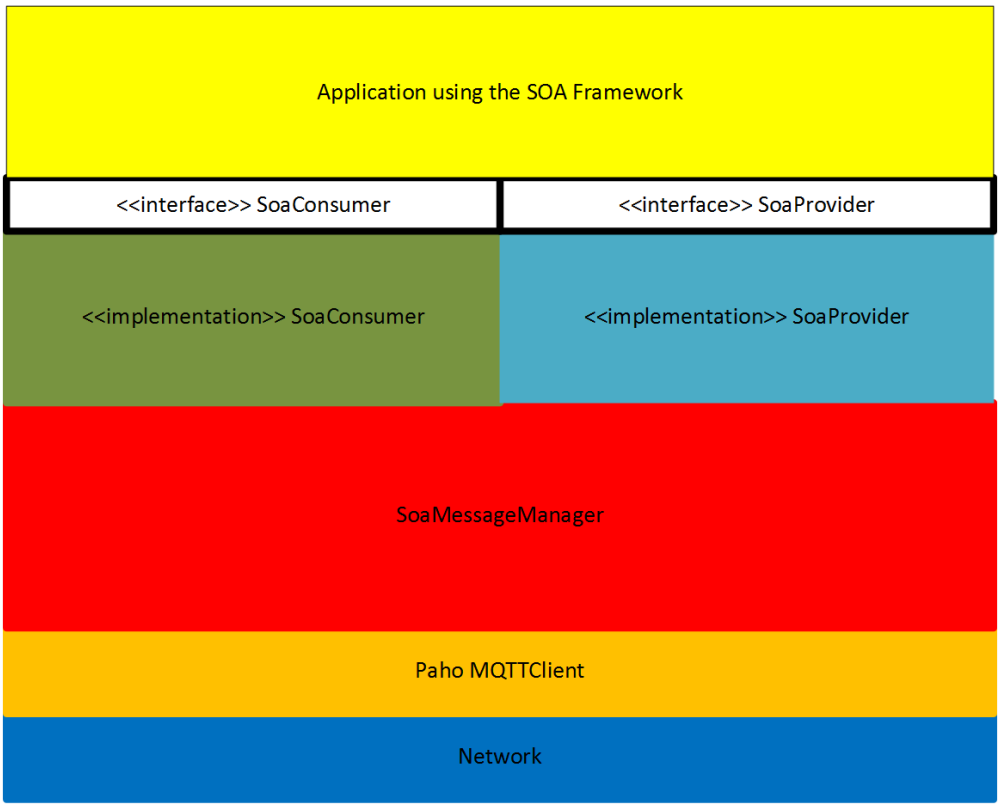
- SoaMessageManager
    - This class is mostly hidden from the SOA Consumer developer, except for its creation.
    - It provides an agnostic transport client (MQTT or IPC) and manages the use of it
- SoaConnectionOptions
    - A set of configuration data settings used for connecting to the MQTT broker or to the SOA Gateway via IPC.
- SoaErrorCode
    - A enum class defining the error values returned by the SOA Framework.
    - **SoaErrorCode::NO_ERROR** is what you always want to see
- ~~SoaLoggerControl~~ This functionality now disabled. Use telemetryctrl to manage logs via the "soa" tag
    - ~~Useful during development. This provides a means of enabling the SOA Framework logger.~~
    - ~~Calling SoaLoggerControl::On() at the start of your application will enable SOA Framework logging.~~

See the section Required Include Paths for details on files to include.

**Use a single Message Manager**

A single message manager is needed per client process. It is designed and meant to be shared across all of your SoaProvider and SoaConsumer instances. The only reason to create multiple message managers is to support different connection options.

## SOA Framework Architecture Overview




**Method Parameters**

As a general rule, SOA Framework methods take three types of parameters:

1) Primitives like uint32_t, int32_t or bool

2) Listener and Callback objects are passed by reference. The application developer is responsible for managing the life cycle of the object correctly.

3) Other objects, such as SoaMessage, SoaClientEndpoint are passed as pointers wrapped in std::shared_ptr<> classes.

- these classes have a scoped typedef called **SharedPtr** to simplify shared pointer syntax. Ex. you can use **SoaMessage::SharedPtr** in place of std::shared_ptr<SoaMessage>

# Message Structures

## Message Types

There are three basic types of messages:

- data (broadcast) message
- request message
- response message

A special data message used by the SOA library is the LWT message (Last Will and Testament)

**Get the sender's identity**

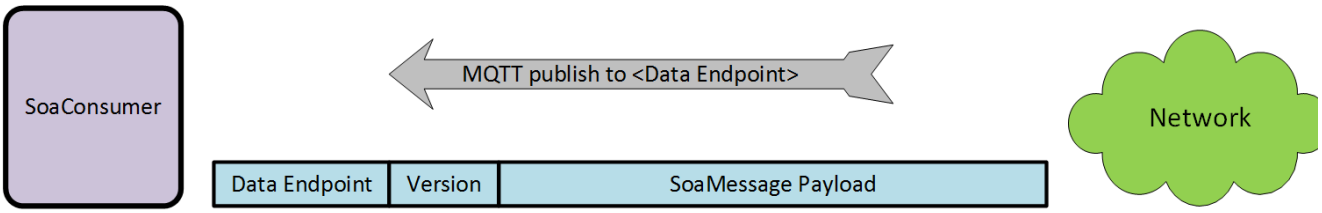
All SoaMessages which are published to the MQTT broker by the SOA library are automatically populated with the sender's ECU ID and UID before being sent. This cannot be avoided. Those fields can be examined via public methods once received. (see getSenderEcuId() and getSenderUid() )

**Data (broadcast) messages** are meant to be received by zero or more consumers,

- only contain
  - a data endpoint (the topic being listened to), can be used by consumer at the application level to route received messages
  - version, provided by the provider at the application level
  - and the message payload

No other message header fields are needed, and would only make the message larger.

- Received by the consumer application in the SoaDataListener callback method onDataReceived(SoaMessage)



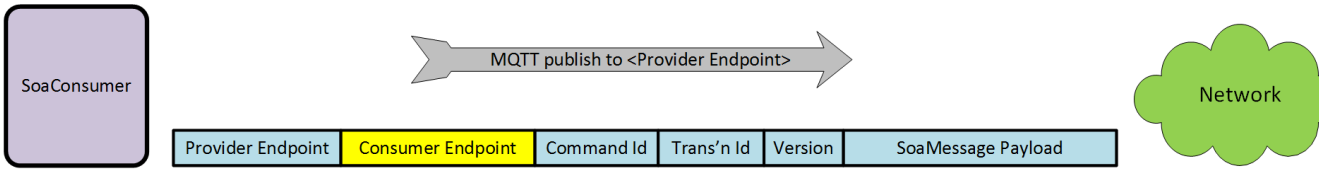

SoaMessage Data Message

**Request messages** are sent by consumers to providers make a request to a service. These are meant to be received by exactly one provider.

- only contain
  - a provider endpoint (the same as the topic the service provider is listening to)
  - a consumer endpoint (the topic the consumer is listening to for a response) **OPTIONAL**: missing when no response expected
    - Your consumer endpoint is auto populated in the request message by the SOA library before it is published to the MQTT broker. DO NOT populate this field with any data when using the 2-way remote call API. The SOA layer will overwrite anything stored there.
  - command ID, indicates a service detail; service dependent
  - transaction ID, provided by the consumer at the application level to coordinate responses against requests
  - version, provided by the consumer at the application level
  - and the message payload

No other message header fields are needed, and would only make the message larger.

- Created by the consumer application by calling the SoaMessage::createRequestMessage() factory method, and subsequently populating the payload field.



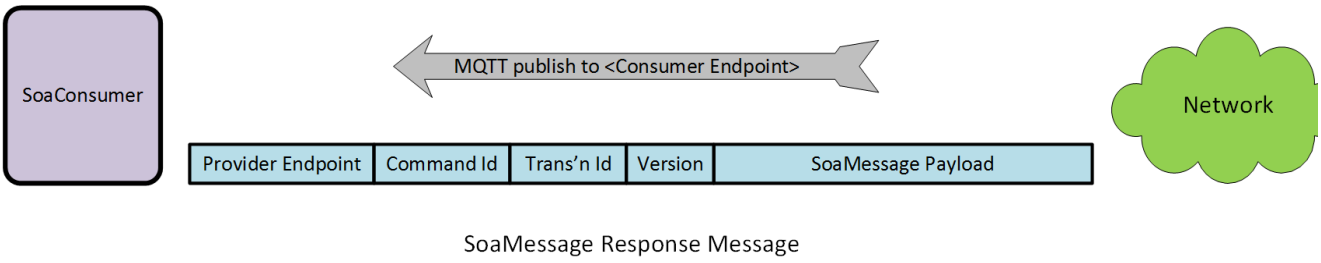

SoaMessage Request Message

**Response messages** are sent by providers to consumers as a reaction to receiving a request. These are meant to be received by exactly one consumer.

- only contain
  - a provider endpoint (the same as the topic the service provider is listening to)
  - command ID, copied from request message; service dependent
  - transaction ID, copied from request message, used by the consumer to coordinate responses against requests
  - version, provided by the provider at the application level
  - and the message payload

No other message header fields are needed, and would only make the message larger.

- Received by the consumer application in the SoaRemoteCallListener callback method onResponseReceived(SoaMessage)




SoaMessage Response Message

**No Consumer Endpoint in a response message**

**IMPORTANT!** The response message DOES NOT include a consumer endpoint. Calling getConsumerEndpoint() on the response message, or on a request message without a consumer endpoint, will return a **nullptr**.

### LWT messages

Last Will and Testament messages are a part of MQTT. The SOA library registers an LWT for every SOA client process when it connects to the broker (directly or via gateway). The LWT is stored in the MQTT broker (for ECG clients) or gateway (for gateway clients). Gateways also register an LWT with the broker for themselves only. The LWT message registered by the SOA library is a SoaMessage with a Google protobuf payload defined by fnv/idl/soa/framework/soa_lwt.proto. If the broker ungracefully loses connection with the client, the broker broadcasts to all clients subscribing to the topic SERVICES/LWT . If a gateway client crashes (this is the only trigger), the gateway will broadcast the client's LWT when it receives an OSSSM crash info message for the crash.

The LWT message contains the following identifying information about the SOA client process: clientId (<ECU_ID>_<UID>, ex ecg_80), ecuId (ex. ecg ), process ID

### Message Fields

Intended use of SoaMessage fields.

- ProviderEndpoint = endpoint that sends a response message or data message, or endpoint to which a request message is sent - this is NOT modified by the SOA layer
- DataEndpoint = endpoint to which a data (broadcast) message is sent - this is NOT modified by the SOA layer
- ConsumerEndpoint = endpoint to which a response is sent - (note: this field is populated/overwritten by the SoaConsumer when sending a responsive remoteCall (2-way) request message.) It must be used by the provider to send a remote call response.
- commandID = application level string used as a message discriminator on the receiving end - this is NOT modified or read by the SOA layer
- transactionID = application level string which can be used to correlate request and response messages at the application level - this is NOT modified or read by the SOA layer
- transactionError = enum: **NO_ERROR, GENERAL_ERROR, VERSION_MISMATCH, COMMAND_NOT_SUPPORTED, *PROVIDER_CLOSED,** * **PROVIDER_DISCONNECTED.** (* these are used by onDemandBroadcasting)
- transactionErrorDetailes = application level string which can be used to elaborate a transaction error. This is also populated by some internal response mechanisms where messages are originated by the SOA library)
- version = application level string intended to be used by receiver to identify sender application version or message payload version before parsing the message payload, or similar use - this is NOT modified or read by the SOA layer
- sender ECU ID = the sender's ECU ID (ex. ec, tcu, sync) This cannot be set by SOA users. It is auto poplated in every SoaMessage. It can be read by the receiver.
- sender UID = the sender's UID. This cannot be set by SOA users. It is auto poplated in every SoaMessage. It can be read by the receiver.
- payload = application level data - this is NOT modified or read by the SOA layer

## Creating a SoaClientEndpoint object

Use the method ...

```
SoaClientEndpoint::SharedPtr SoaMessageManager::createClientEndpoint( const std::string & topic )
```

to create an endpoint of the correct type (MQTT or IPC). The message manager was created with the connection options object you provided, which implicitly defined the transport type.

## Connection APIs

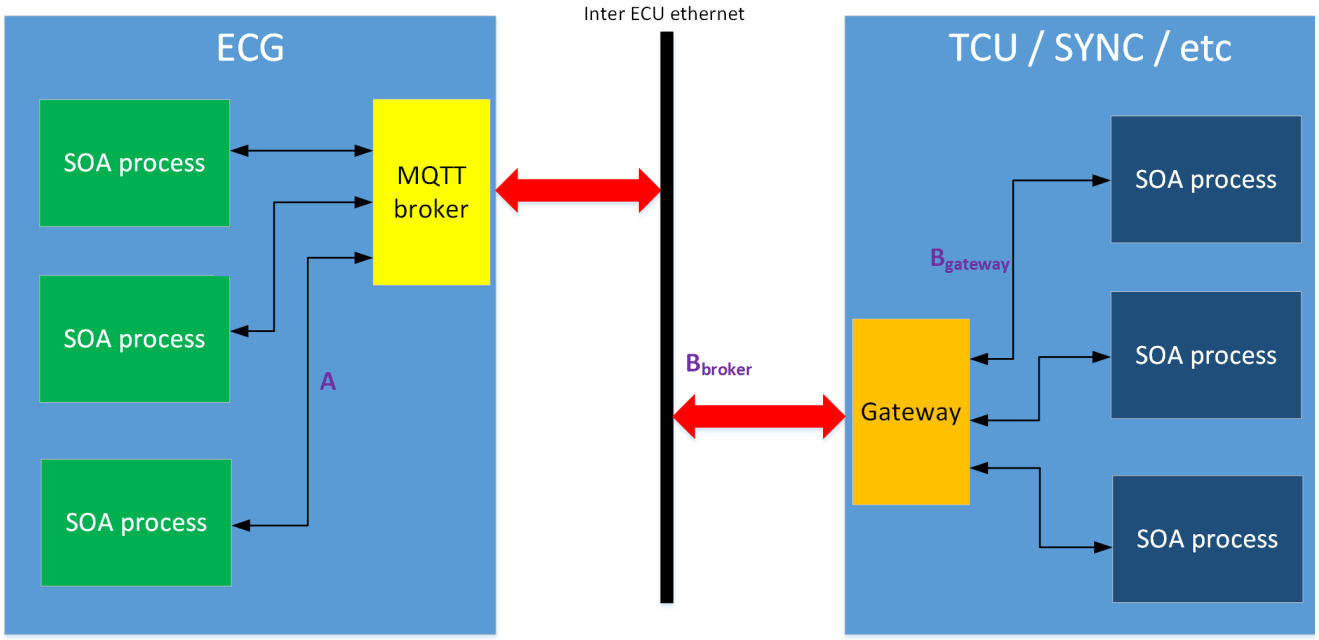
Connecting to the network and initializing the SoaConsumer object are the first things you need to do for or with the object. **BEWARE!** There are differences for ECG processes and Gateway clients.

There are a few choices on how to to this.

Calling or triggering any of these methods while any of these methods is already active in the background will cause the second call to fail with ERROR_WRONG_STATE.

The connection state of the message manager can be retrieved using **SoaMessageManager::getConnectionState()**. Returns one of { **CONNECTED, DISCONNECTED, CONNECTING, CONNECTION_LOST** }

Gateway clients running on non-ECG ECUs include two additional states: { ..., **REGISTERING_TO_GW, REGISTERED_TO_GW }**.




"A" connection: ECG processes connect directly to broker via UDS

"B" connection: non-ECG processes connect to broker through their gateway. Connections have two legs. 1) register to the gateway, 2) gateway connects to broker

| Method | ECG process | non-ECG process |
|---|---|---|
| SoaConsumer:: initialize() | • causes message manager to connect to broker if not already connected, then subscribes for consumer's endpoint (used to receive remoteCall() responses.<br>• if the message manager cannot connect to the broker, this call will fail<br>• do not attempt to call initialize() for a SoaConsumer once it has been successfully initialized.<br>• WARNING: This method is no longer recommended. Use initializeAsync(...) instead because it supports the onInitialized(...) and onDeinitialized(...) callbacks to let the app know when the SoaConsumer can be used in the wake of lost connection/reconnection. (race conditions exist if relying on onConnectionLost(...)/onReconnected(...) alone, and more so with SoaConsumer than with SoaProvider because the consumer sends an MQTT subscribe msg before becoming initialized) | • causes message manager to register to gateway BUT DOES NOT trigger the gateway to connect to broker. If connected to broker, then subscribes for consumer's endpoint (used to receive remoteCall() responses.<br>• if the gateway is down, this method will fail<br>• if gateway is not connected to broker when called, this method will fail. |
| SoaConsumer:: initializeAsync( listener ) | • NEW and RECOMMENDED<br>• causes message manager to attempt to connect to broker.<br>• upon connection to broker, initialization completes, and calls back to listener object's onInitialized(...) method<br>• do not attempt to call initializeAsync(...) for a SoaProvider once it has been successfully initialized, including after onDeinitialized(...) is called<br>• may be called before or after connectAsync() | • causes message manager to attempt to register to gateway BUT DOES NOT trigger the gateway to connect to broker. If connected to broker, then subscribes for consumer's endpoint (used to receive remoteCall() responses.<br>• if the gateway is down, it will continue to reattempt register to gateway<br>• when gateway is eventually connected to broker, subscribes for consumer's endpoint<br>• upon successful subscription, calls back to listener object's onInitialized (...) method<br>• do not attempt to call initializeAsync (...) for a SoaConsumer once it has been successfully initialized, |

| | | |
|---|---|---|
| | | including after onDeinitialized(...) is called<br>• may be called before or after connectAsync() |
| SoaMessageManager::connectAsync() | • explicit request to trigger connection attempt<br>• can be called before calling SoaConsumer::initialize().<br>• notification is via the connection listener<br>• once notified of successful connection, initialize() will succeed | • this is the ONLY API to trigger the gateway to attempt to connect to the broker<br>• this method will first attempt to register to the gateway<br>• once registered to the gateway, onConnect(...) is called with SoaErrorCode::NO_ERROR__GATEWAY_CONNECTED_ONLY<br>• once the gateway is connected to the broker, onConnect(...) is called with SoaErrorCode::NO_ERROR<br>• once connected to the broker, SoaConsumers which have called initializeAsync(...) will complete initialization.<br>• if another process has already triggered the gateway to connect to the broker, this process will receive connection status change notifications |
| SoaMessageManager::reconnect() | • explicit request to trigger connection attempt<br>• if connection is lost, connection attempts will be repeated until connected as per the configured reconnection algorithm (in connection options)<br>• notification is via the reconnection listener | • NOT SUPPORTED<br>• the gateway always tries to reconnect if disconnected connectAsync() has been called, and disconnectGlobalAsync() has not been called more recently. |
| SoaMessageManager::disconnect() | • synchronous call to disconnect from broker | • synchronous call to de-register from gateway<br>• if this process is the only gateway client, the gateway will disconnect from the broker |
| SoaMessageManager::disconnectAsync() | • asynchronous call to disconnect from broker | • asynchronous call to disconnect from gateway<br>• if this process is the only gateway client, the gateway will disconnect from the broker |
| SoaMessageManager::<br><br>disconnectGlobalAsync() | NOT SUPPORTED | • forces the gateway to disconnect from the broker<br>• must be registered with the gateway to call this method (either through a call to connectAsync() or initializeAsync()<br>• DOES NOT cause processes registered with the gateway to be deregistered<br>• HOWEVER all SoaConsumer and SoaProvider objects will have their onDeinitialized(...) callback called. They cannot use SOA APIs until onInitialized(...) is call again. They DO NOT call initializeAsync() again |
| SoaMessageManager::connect() | • only supported when called AFTER the message manager has already connected to the broker using one of the other methods<br>• use SoaConsumer::initialize() to synchronously connect and initialize | • DO NOT USE THIS METHOD |

## SoaConsumer::initialize()

It is highly **recommended to NOT use this method**. Use initializeAsync(...) instead.  The blocking initialize() method may be deprecated.

Your application may include more than one instance of the SoaConsumer class. Every instance must call initialize(). The initialize() method is a blocking method which executes on the worker thread. It calls an internal SoaMessageManager::connect(...) method and performs initialization tasks.

- initialize() is a blocking method which returns a SoaErrorCode enum
- if it returns NO_ERROR, it has successfully connected and initialized the object.
- it may fail to connect due to configuration, resource or network reasons, in which case initialization is not done and it returns an error
- if the message manager is already connected, it accepts this connection and continues to initialize the object. This is true for all SoaConsumer and SoaProvider objects that share a message manager
- initialize may fail after successfully connecting
- This method connects the client to the broker and sends a subscription request used for responses to remote calls. If the connection cannot be established, this method will fail with an error code. If the connection is already established, the connection sub-task is skipped.
- It is highly recommended to call SoaMessageManager::connectAsync() before calling initialize() to avoid failing on this method, BUT you MUST wait for the successful connection status to be indicated via the asynchronous callback to the SoaConnectionListener. You MUST NOT call this method after calling connectAsync() without waitiing for the successful connection status

## SoaConsumer::initializeAsync( SoaInitializedConsumerListener & )

As of SOA R0.67.5 this API is available on all ECUs, including ECG. It is highly recommended use this instead of the blocking initialze() method BECAUSE this method supports the SoaInitializedConsumerListener.

Your application may include more than one instance of the SoaConsumer class. Every instance must call initializeAsync(). The initializeAsync() method is a non-blocking method. It attempts to register to the gateway and then allows initialization tasks to complete when connected to the broker.

- initializeAsync() is a non-blocking method which also returns a SoaErrorCode enum if not able to attempt registration to the gateway
- if it returns NO_ERROR, it has initiated a background attempt to register to the gateway and prepare for completing initialization once connected to the broker.
- it may fail to attempt registration due to configuration reasons, in which case initialization is not done and it returns an error
- if the message manager is already connected, it accepts this connection and continues to initialize the object. This is true for all SoaConsumer and SoaProvider objects that share a message manager
- initialization may fail after successfully connecting

### SoaConsumer::SoaInitializedConsumerListener

- Create a class derived from SoaInitializedConsumerListener that realizes the pure-virtual callback interfaces:
  - onInitialized( SoaErrorCode, SoaConsumer * )  an error code of NO_ERROR means the SoaConsumer referenced by the pointer is able to be used. Each time this method is called, the consumer will need to subscribe to all of its data endpoints anew.
  - onDeinitialized( SoaErrorCode, SoaConsumer * )  when this method is called, the SoaConsumer referenced by the pointer is no longer able to be used. DO NOT call initializeAsync() again for this consumer. When onInitialized(...) is subsequently called, treat that call as a first time call. Re-establish subscriptions, etc.

## SoaMessageManager::connectAsync()

**Differences between ECG and non-ECG processes**

On ECG this tries to connect to broker according to the customizable reconnect algorithm configured in the connection options. On non-ECG ECUs, this tries to register to the gateway and then instruct the gateway to connect to the broker.

As stated above, the message manager may be connected before calling SoaConsumer::initialize(). As of SOA Framework library version SOA-R0.66.0, you can call connectAsync() to connect the message manager before calling SoaConsumer::initialize() or initializeAsync(...)

connectAsync() will repeatedly attempt to connect in the background. It can be called if the connection state is DISCONNECTED, CONNECTION_LOST or CONNECTED.

**On non-ECG ECUs**, the gateway does not attempt to connect to the broker on the ECG unless instructed to do so by one or more of its clients (non-ECG SOA processes are also known as "gateway clients").  connectAsync() is the ONLY method that instructs the gateway to attempt to connect to the broker. It can be called if already connected. This method registers the process to the gateway, followed by requesting the gateway to connect to the broker. To force the gateway to disconnect from the broker, a connected client can call disconnectGlobalAsync(). If not requested to disconnect from the broker but the gateway unwillingly loses connection to the broker, the gateway will automatically attempt to reconnect to the broker based on client requests. As long as there exists one or more registered clients that have previously called connectAsync() not followed by disconnectGlobalAsync(), those clients will instruct the gateway to attempt reconnection when they receive the connection lost notification.  Clients remember if connectAsync() vs disconnectGlobalAsync() was called more recently.

- connectAsync() is a non-blocking method which returns a SoaErrorCode enum immediately

- if it returns NO_ERROR, this indicates all pre-checks have passed and connection is being attempted on a separate thread (not the worker thread and not the callback thread)
- if it returns any other error code, the background connection task was not started
- it returns immediately with an error if no connection listener was provided via the connection options
- if already connected when called, this method returns NO_ERROR, AND the connection callback is called signalling NO_ERROR
- connection establishment is signaled by a NO_ERROR status being passed to your derived SoaConnectionListener class' onConnect( SoaErrorCode ) callback method.
- onConnect(...) will be called when the connection attempts stop due to success, timeout or an error indicating it need not keep trying
- for non-ECG ECUs only, onConnect(...) will be called with a NO_ERROR__GATEWAY_CONNECTED_ONLY status to indicate the process is registered to the gateway. This may not be particularly useful information for all clients.

**Unique not an error SoaErrorCode**

When creating your onConnected(...) callback method, expect and treat the value SoaErrorCode:: NO_ERROR__GATEWAY_CONNECTED_ONLY as a non-error notification. Ignore this event if not useful to your client.)

**Caveat** : You CANNOT call SoaConsumer::initialize() from your onConnect(...) callback because it is called on the callback thread.

**Bad mix of methods**

You MUST NOT call SoaConsumer::initialize() before onConnect(...) is called if using connectAsync() to pre-establish the connection.

More info available at Programmer Guide: General Info: SOA Asynchronous Connections

### Background Connection Retry Task

The background connection retry task can be customized via SoaConnectionOptions::setReconnectionTimeOptions( timeout, initialRetryWaitTime, maxRetryWaitTime, waitTimeMultiplier )

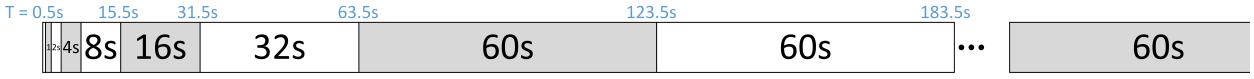
When auto-reconnect is triggered or when connectAsync() or reconnect() are called, the background connection retry task attempts to connect immediately. Each time its attempt fails due to the server being unavailable it waits before retrying again. When it discovers the next retry would be scheduled beyond the timeout it terminates and calls the re/connection listener with an error of ERROR_TIMEOUT. The method setReconnectionTimeOptions(...) is used to customize the retry task. The wait time can be constant or it can grow with each failure. The wait time between attempts can be capped at a maximum wait time, after which the wait time would be constant.

- timeout = the maximum total time to try to reconnect - specified as integral milliseconds
- initialRetryWaitTime = the time to wait after the first attempt before attempting to connect a second time - specified as integral milliseconds
- waitTimeMultiplier = the factor by which the last retry wait time is multiplied to calculate the next retry wait time - specified as a floating point value >= 1.0
- maxRetryWaitTime = the maximum time to wait before attempting to connect again - specified as integral milliseconds

setReconnectionTimeOptions( ... ) returns an error if the customization arguments are out of range or conflict with each other.

Example showing retry intervals for:

timeout = 3600000 = 1 hour, initialRetryWaitTime = 500 = 0.5 seconds, maxRetryWaitTime = 60000 = 60 seconds, waitTimeMultiplier = 2.0




## SoaMessageManager::reconnect()

**ECG Only**

Only supported for MQTT connection options.

**ECG Only**

Not supported for IPC connection options (ex. TCU or SYNC).

This method is reserved for explicitly trying to reconnect after the message manager was ungracefully disconnected. You may call this onConnectionLost( string cause ) was called AND auto-reconnect is false, or after onReconnect( SoaErrorCode error ) was called where error != NO_ERROR.

reconnect() will repeatedly attempt to connect in the background. It can be called if the connection state is CONNECTION_LOST or CONNECTED.

- reconnect() is a non-blocking method which returns a SoaErrorCode enum immediately
- if it returns NO_ERROR, this indicates all pre-checks have passed and connection is being attempted on a separate thread (not the worker thread and not the callback thread)
- if it returns any other error code, the background connection task was not started
- it returns immediately with an error if no reconnection listener was provided via the connection options
- if already connected when called, this method returns NO_ERROR, AND the reconnection callback is called signalling NO_ERROR
- connection establishment is signaled by a NO_ERROR status being passed to your derived SoaReconnectionListener class' onConnect( SoaErrorCode ) callback method.
- onReconnect(...) will be called when the connection attempts stop due to success, timeout or an error indicating it need not keep trying

More info available at Programmer Guide: General Info: SOA Asynchronous Connections

### SoaMessageManager::connect()

This blocking method is reserved for explicitly connecting again only after disconnect() was called. It cannot be called as a means of making an initial connection. This method is only useful for TCU.

connect() will NOT repeatedly attempt to connect. It can be called if the connection state is DISCONNECTED, AND the message manager was previously in a CONNECTED state.

In version SOA-R0.63.0, we introduced some new connection semantics. With the changes, we now support TCU to disconnect when power is about to be removed from ECG and SYNC or these ECUs will effectively be reset. Following the disconnect, when appropriate and the ECG MQTT broker is once again available, TCU may connect. Disconnection is done by calling SoaMessageManager::disconnect() and subsequent reconnection is done by calling SoaMessageManager::connect().

Note that connect() WILL NOT WORK to initially connect the SoaConsumer or SoaProvider objects. Applications MUST invoke initialize() on these objects to establish their initial connection state. A TCU call to SoaMessageManager::connect() can only be made after a call to disconnect(), and only impacts those objects which have called initialize() at some point in the past.

ECG and SYNC applications do not call disconnect() except when terminating. They never call connect(). They cannot call disconnect() followed by connect().

## SoaMessageManager::disconnect() and disconnectAsync()

- On ECG, these methods will disconnect() the client from the broker
- On non-ECG ECUs, these methods will only deregister the client from the gateway. When the last gateway's last client deregisters, the gateway will disconnect from the broker.
- For all clients calling disconnect() ... if disconnecting but not destroying self and not shutting down, then calling disconnectAsync() is OK. But if disconnecting as part of a shutdown routine, the preferred API is the synchronous disconnect() method as this allows the task to be completed properly before the process deallocates all other resources.

## SoaMessageManager::disconnectGlobalAsync()

- Not supported and not meaningful on ECG.
- This method is a "master" process API for telling the gateway to disconnect from the broker. This should not be called by processes that are not aware of the connection needs of other SOA processes on the ECU.

**Master/Slave and Peer/Peer modes**

The SOA Framework DOES NOT track or understand any "master" process, nor does it process requests differently for processes that operate in a master/slave model vs a peer/peer model. However, all SOA processes running on an ECU **MUST** operate under the same model.

# Usage Models

### Master/Slave

- One entity of one process running on the ECU is responsible for asking the gateway to connect to the broker. This would be considered the master. All of the SoaConsumers and SoaProviders would be slaves with respect to their connection to the broker. Only the master calls SoaMessageManager::connectAsync() and SoaMessageManager::disconnectGlobalAsync(). Only the master should care about the status of the connection, The slaves receive notification of onInitialized(...) and onDeinitialized(...) coinciding with the connection controlled by the master

### Peer/Peer

- There is no single entity on the ECU that knows more or cares more about the connection status than any other entity on the same ECU. Each gateway client process would call SoaMessageManager::connectAsync() as required. When disconnecting, these processes are only concerned with disconnecting themselves, so would call SoaMessageManager::disconnectAsync() or SoaMessageManager::disconnect(), while NONE of the clients would ever call SoaMessageManager::disconnectGlobalAsync()

## A Note about QoS (MQTT Quality of Service)

QoS support has a very high performance hit on the broker so QoS configuration is not currently supported by the SOA Framework, with no plans to supported it. It is widely recommended to use QoS >0 mainly for poor performance or unreliable network and this is not our case. The SOA Framework unconditionally uses QoS = 0.

## Building a Consumer Application

**IMPORTANT info about listener lifecycles to avoid crashes**

Be sure to read the section at Listener Lifecycles Caveat to avoid potential crashes if listener objects are destroyed at run time.

### Basic Steps

a. ~~Optionally enable SOA Framework logging.~~
b. Create your endpoint(s)
c. If subscribing to event data messages, create your SoaDataListener class(es) and object(s)
d. If subscribing to service status updates, create your SoaServiceStatusListener class(es) and object(s)
e. If making a non-blocking service request, create your SoaRemoteCallListener class(es) and object(s)
f. If making a non-blocking request for a list of command handlers asynchronously, create your SoaCommandHandlersListener class(es) and object(s)
g. If using non-blocking methods, create SoaActionResultListener<ContextType> classes and objects
h. Create a SoaConsumer; this includes creating a SoaMessageManager and connecting your application to the MQTT broker.
i. Set a non-default blanket timeout if needed
j. Subscribe for data messages
k. Subscribe for service status updates
l. Make a remote call, block waiting for a response
m. Make a remote call providing a response, do not block waiting for a response
n. Make a remote call which does not provide a response
o. Synchronously get subscribed service status without blocking
p. Request a service status and block waiting for the status
q. Request a service status without blocking for the status to be received asynchronously
r. Get a list of services which handle a given command ID

**SoaErrorCodeUtil**

You can call SoaErrorCodeUtil::getString( SoaErrorCode error) or SoaErrorCodeUtil::getCString( SoaErrorCode error ) to get a readable representation of an error code.

### Details

a. ~~Optionally disable SOA Framework debug level logging - only impacts SOA debug level logs. All other log levels are logged unconditionally~~ .This functionality now disabled. Use telemetryctrl to manage logs via the "soa" tag

**Disable SOA Framework debug logging**

```
int main(){
    SoaLoggerControl::Off();  // is on by default
        // ... rest of the code ...


}
```

- ~~Enable by calling SoaLoggerControl::On(true) // already on by default~~ deprecated/disabled
- ~~Disable by calling SoaLoggerControl::Off() or SoaLoggerControl::On(false)~~ deprecated/disabled
- ~~Check if enabled by calling bool SoaLoggerControl::isEnabled()~~ deprecated. always returns true

a. Create your endpoint topics. Endpoints can be created at run time using the factory method available from the message manager.

**Create your endpoints**

```
const std::string CONSUMER_TOPIC     = SoaClientEndpoint::createResponseEndpointName(
"MY_ENDPOINT") ; // SERVICES/RESPONSE/ is prepended at compile time
const std::string SERVICE_TOPIC      = SoaClientEndpoint::createRequestEndpointName
("THE_SERVICE_ENDPOINT");  // SERVICES/REQUEST/ is prepended at compile time
const std::string DATA_TOPIC         = SoaClientEndpoint::createDataEndpointName
("THE_DATA_ENDPOINT");  // SERVICES/DATA/ is prepended at compile time
const std::string CONSUMER_SDM_TOPIC = SoaClientEndpoint::createResponseEndpointName
("MY_SDM_ENDPOINT");  // SERVICES/RESPONSE/ is prepended at compile time
SoaClientEndpoint::SharedPtr myConsumerEndpoint;
SoaClientEndpoint::SharedPtr serviceEndpoint;
SoaClientEndpoint::SharedPtr dataEndpoint;
SoaClientEndpoint::SharedPtr consumerSdmEndpoint;
```

a. If subscribing to event data messages, create your SoaDataListener class(es) and object(s)

**Implementing a Data Listener**

```
mutex dataMessageMutex;
condition_variable cvd;
...
class DataListener : public SoaDataListener{
        void onDataReceived(SoaMessage::SharedPtr dataMessage) {
                lock_guard<mutex> lock(dataMessageMutex);
                dataMsg = dataMessage;
                cvd.notify_one();
        }
};
```

- Create classes derived from the interface class SoaDataListener
- Implement the pure virtual method onDataReceived()
- This will be called on the Callback thread when a data message is received for a subscribed data endpoint.

a. If subscribing to service status updates, create your SoaServiceStatusListener class(es) and object(s)

**Implementing an Service Status Listener**

```
SoaServiceStatus serviceStatusA = NOT_AVAILABLE;
SoaServiceStatus serviceStatusB = NOT_AVAILABLE;
...
class StatusListener : public SoaServiceStatusListener {
        void onStatusChanged(const SoaClientEndpoint::SharedPtr endpoint, SoaServiceStatus status)
{
                if (endpoint == serviceEndpointA) {
                        serviceStatusA = status;
                } else if (endpoint == serviceEndpointB) {
                        serviceStatusB = status;
                }
        }
};
```

- Create classes derived from the interface class SoaServiceStatusListener
  - You are free to implement separate listener classes or instantiate separate instances for each service whose status is being watched
  - Or common listener instantiations can be used, while demultiplexing messages based on service endpoint
- Implement the pure virtual method onStatusChanged()
- This will be called on the Callback thread when a service status update message is received for a subscribed service status.

b. If making a non-blocking service request, create your SoaRemoteCallListener class(es) and object(s)

**Implementing a Remote Call Response Listener**

```
mutex responseMessageMutex;
condition_variable cvr;
```

```
...
class RemoteCallListener : public SoaRemoteCallListener{
        void onResponseReceived(SoaMessage::SharedPtr responseMessage) {
                lock_guard<mutex> lock(responseMessageMutex);
                respMsg = responseMessage;
                cvr.notify_one();
        }
};
```

- Create classes derived from the interface class SoaRemoteCallListener
- Implement the pure virtual method onResponseReceived()
- This will be called on the Callback thread when a response message is received for remote call.

c. If making a non-blocking request for a list of command handlers asynchronously, create your SoaCommandHandlersListener class(es) and object(s)

**Implementing a Command Handlers List Listener**

```
class CommandXyzHandlersListListener : public SoaCommandHandlersListener {
        void onCommandHandlersListUpdated(SoaServiceMetadataList::SharedPtr handlers) {
                updateListOfHandlers("XYZ", handlers);
        }
};
```

- Create classes derived from the interface class SoaCommandHandlersListener
- Implement the pure virtual method onCommandHandlersListUpdated()
- This will be called on the Callback thread when an update message is received for the set of services which handle a specific command ID

.

a. If using non-blocking methods, create SoaActionResultListener<ContextType> classes and objects

**Implementing Action Result Callbacks**

```
class DataActionListener : public SoaActionResultListener<SoaSubscribeToDataContext>{
        virtual void onActionCompleted(SoaErrorCode error, shared_ptr<SoaSubscribeToDataContext>
actionContext){
        SoaClientEndpoint::SharedPtr endpoint = actionContext -> getDataEndpoint();
                string dataEndpoint = "<empty>";
                if (endpoint != nullptr) {
                        dataEndpoint = endpoint->toString();
                }
                cout << "Data subscribe completed for endpoint " << dataEndpoint << " with status:
" << (error == SoaErrorCode::NO_ERROR ? "no error" : ("error= " + std::to_string((int)error)) <<
endl;
        }
};


class ServiceStatusActionListener : public
SoaActionResultListener<SoaSubscribeToServiceStatusContext>{
        virtual void onActionCompleted(SoaErrorCode error,
shared_ptr<SoaSubscribeToServiceStatusContext> actionContext){
        SoaClientEndpoint::SharedPtr endpoint = actionContext -> getProviderEndpoint();
                string providerEndpoint = "<empty>";
                if (endpoint != nullptr) {
                        providerEndpoint = endpoint->toString();
                }
                cout << "Service status update subscribe completed for endpoint " <<
providerEndpoint << " with status: " << (error == SoaErrorCode::NO_ERROR ? "no error" : ("error= "
+ std::to_string((int)error)) << endl;
        }
};


class RemoteCallActionListener : public SoaActionResultListener<SoaRemoteCallContext>{
        virtual void onActionCompleted(SoaErrorCode error, shared_ptr<SoaRemoteCallContext>
actionContext){
```

```
        SoaClientEndpoint::SharedPtr endpoint = actionContext -> getProviderEndpoint();
                string providerEndpoint = "<empty>";
                if (endpoint != nullptr) {
                        providerEndpoint = endpoint->toString();
                }
                cout << "Remote call completed for endpoint " << providerEndpoint << " with
status: " << (error == SoaErrorCode::NO_ERROR ? "no error" : ("error= " + std::to_string((int)
error)) << endl;
        }
};


class CommandHandlerActionListener : public
SoaActionResultListener<SoaGetCommandHandlersActionContext>{
        virtual void onActionCompleted(SoaErrorCode error,
shared_ptr<SoaGetCommandHandlersActionContext> actionContext){
        string commandId = actionContext -> getCommandId();
                cout << "Get command handlers completed for command ID " << commandId << " with
status: " << (error == SoaErrorCode::NO_ERROR ? "no error" : ("error= " + std::to_string((int)
error)) << endl;
        }
};
```

- Create classes derived from the interface template class SoaActionResultListener<ContextType> instantiated using the correct context types for the non-blocking methods
- Implement the pure virtual method onActionCompleted()
- This will be called on the Callback thread when the result of the method is known.

> **ActionResult callbacks executed before message listeners**
>
> An action result listener is executed for a method call before a message listener is executed for the same method call.

a. Create a SoaConsumer; this includes creating a SoaMessageManager and connecting your application to the MQTT broker.

**Instantiating a SoaConsumer**

```
SoaConsumer::SharedPtr initializeConsumer(){
        //Set connection options
    // This establishes an MQTT connection. IPC clients create IPC connection options
        SoaConnectionOptions::SharedPtr connectionOptions = SoaConnectionOptions::
createConnectionOptions();
        connectionOptions -> setClientId( consumerClientId );   // needed during development
phases
        connectionOptions -> setDebugBrokerUrl( brokerURL );    // needed during development phases
        connectionOptions -> setConnectionTimeOut( 2 );         // in seconds

        // Instantiate message manager and service directory manager (this is common for MQTT and
IPC)
        messageManager = SoaMessageManager::createMessageManager( *connectionOptions );
    // if endpoint not already created, can create them now using the factory available from the
message manager
    myConsumerEndpoint  = messageManager->createClientEndpoint( CONSUMER_TOPIC );
    serviceEndpoint     = messageManager->createClientEndpoint( SERVICE_TOPIC );
    dataEndpoint        = messageManager->createClientEndpoint( DATA_TOPIC );
    consumerSdmEndpoint = messageManager->createClientEndpoint( CONSUMER_SDM_TOPIC );
        SoaServiceDirectoryManager::SharedPtr svcDirMan = SoaServiceDirectoryManager::create(
messageManager, consumerSdmEndpoint );

        //Create consumer
        SoaConsumer::SharedPtr consumer = SoaConsumer::createSoaConsumer( messageManager,
svcDirMan, myConsumerEndpoint);

        if (consumer == nullptr) {
                cout << "consumer creation failed: bad parameter(s)" << endl;
        } else {
        // must call initialize to connect the message manager to the MQTT broker AND subscribe to
enable remote calls
                SoaErrorCode rc = consumer -> initialize();
```

```
                if (rc  != SoaErrorCode::NO_ERROR){
                        cout << "consumer initialization failed: Error = " << SoaErrorCodeUtil::
getString( rc ) << endl;
                        consumer = nullptr;
                }
        }
        return consumer;
}
```

- Create a SoaConnectionOptions object using the factory method
- Set required connection options
  - during development, you need to provide a client ID which is unique among all clients connecting to the MQTT broker
  - during development, you need to explicitly specify the URL for your MQTT broker
- Instantiate a SoaMessageManager object using the SoaConnectionOptions object
- Instantiate a SoaServiceDirectoryManager object using the message manager and a unique endpoint to listen for service updates.
- Instantiate a SoaConsumer object using the SoaMessageManager and SoaServiceDirectoryManager objects, and your consumer endpoint.
  - If any of the parameters are bad the factory will return a nullptr
- Once the SoaConsumer is created, you MUST call initialize() as a post instantiation step to connect to the MQTT broker and enable remote calling

**Connection options are copied to the message manager**

Set all connection options prior to instantiating the message manager. The options are copied to the message manager. The options object can be destroyed after creating the message manager. The created message manager will not know about any changes to made to the options object.

**Each SoaConsumer must be initialized**

If your code has already initialized another instance of SoaConsumer or SoaProvider associated with the same message manager, the connection will have already been established. But each instance of SoaConsumer needs to be initialized by calling initializeAsync(...) or initialize().

a. Set a non-default blanket timeout if needed

**Setting a blanket timeout**

```
SoaConsumer::SharedPtr consumer = ...

// set a non-default blanket timeout
// as long as the new timeout > 0, the setTimeout() method will succeed
SoaErrorCode rc = consumer -> setTimeout( 2000 );  // 2 seconds
```

- Call setTimeout() on the SoaConsumer instance.
- All SoaConsumer methods which do not take a timeout parameter will use this timeout value
- If not set, the default blanket timeout will apply

a. Subscribe for data messages

**Subscribe for data messages**

```
DataListener  dataListener;
...

SoaErrorCode subscribeStatus = consumer -> subscribeToData(DATA_ENDPOINT, dataListener);
```

- Call either the blocking or non-blocking version of subscribeToData(...) with the endpoint for which the data will be published, a data listener object reference, an action-result listener if non-blocking, and an optional custom timeout.
- The blocking version is shown here.
- Data subscriptions are simple operations which do not take any time

a. Subscribe for service status updates

**Subscribe for service status changes**

```
StatusListener              statusListener;
ServiceStatusActionListener statusActionListener;
...
subscribeToServiceStatusUpdateAsync( serviceEndpoint, statusListener, statusActionListener, 3000 );
```

- Call either the blocking or non-blocking version of subscribeToData(...) with the endpoint for which the data will be published, a data listener object reference, an action-result listener if non-blocking, and an optional custom timeout.
- The non-blocking version is shown here, with a custom timeout.
    - If the operation does not complete within the timeout, it is cancelled and the action-result listener is called with a timeout error.

a. Make a remote call (i.e. send a request message), block waiting for a response

**Send a request and wait for a response**

```
// my application version as it relates to the remote call
string GetRpmVersion = "1.0";
// create the request message using the factory method
string requestCommand = "GET_RPM";
uint32_t transactionVal = 0x1000;
...


string transactionId = std::to_string( getNextTransactionVal() );
SoaMessage::SharedPtr request =  SoaMessage::createRequestMessage( serviceEndpoint,
requestCommand, GetRpmVersion , transactionId );
request->setPayload("...");  // some remote calls may not need a payload. The command ID may be
enough.
// block this thread until a response is received
SoaActionResult<SoaMessage>::SharedPtr remoteCallResult = consumer -> remoteCall(serviceEndpoint,
request, 5000);

SoaMessage::SharedPtr response = nullptr;
if (remoteCallResult -> getError() == SoaErrorCode::NO_ERROR) {
        response = remoteCallResult -> getResult();
    ...
}
```

- Create a request message using the SoaMessage factory method (a version which takes a consumer endpoint argument has or will be deprecated)
    - Optionally specify a commandID, application-level version value and/or an application-level transactionID; as you require
- Set the payload on the request message
- Use the version of remoteCall which takes a timeout argument and returns a SoaActionResult<SoaMessage> type
    - The result (NO_ERROR or error code) can be extracted from the return value
    - The response message can be extracted from the return value if there is one
    - The blocking method remoteCall(...) CANNOT be called from the callback thread
    - if attempted, the error code returned will be SoaErrorCode::ERROR_ILLEGAL_CALL_THREAD

a. Make a remote call providing a response, do not block waiting for a response

**Send a request, get response asynchronously**

```
// my application version as it relates to the remote call
string GetRpmVersion = "1.0";
// create the request message using the factory method
string requestCommand = "GET_RPM";
uint32_t transactionVal = 0x1000;


RemoteCallListener remoteCallListener;;
RemoteCallActionListener remoteCallActionListener;
...


string transactionId = std::to_string( getNextTransactionVal() );
SoaMessage::SharedPtr request =  SoaMessage::createRequestMessage( serviceEndpoint,
requestCommand, GetRpmVersion , transactionId );
```

```
request->setPayload("...");  // some remote calls may not need a payload. The command ID may be
enough.
// don't block this thread waiting for a response, result goes to RemoteCallActionListener,
response message goes to RemoteCallListener
consumer -> remoteCallAsync( serviceEndpoint, request, remoteCallListener,
remoteCallActionListener, 5000 );
```

- Create a request message using the SoaMessage factory method (a version which takes a consumer endpoint argument has or will be deprecated)
    - Optionally specify a commandID, application-level version value and/or an application-level transactionID; as you require
- Set the payload on the request message
- Use the version of remoteCallAsync which takes a SoaRemoteCallListener argument, a SoaActionResultListener<> argument and a timeout argument
    - The result of making the request is delivered the SoaActionResultListener<SoaRemoteCallContext> callback method
    - If the error code is NO_ERROR, the response message is delivered to the SoaRemoteCallListener callback after this callback returns, else no response message will be received
    - The non-blocking method remoteCallAsync(...) CAN be called from the callback thread

a. Make a remote call which does not provide a response

**Send a request - no response is expected**

```
// my application version as it relates to the remote call
string GetRpmVersion = "1.0";
// create the request message using the factory method
string requestCommand = "GET_RPM";
uint32_t transactionVal = 0x1000;
...


string transactionId = std::to_string( getNextTransactionVal() );
SoaMessage::SharedPtr request =  SoaMessage::createRequestMessage( serviceEndpoint,
requestCommand, GetRpmVersion , transactionId );
request->setPayload("...");  // some remote calls may not need a payload. The command ID may be
enough.
// send the request
SoaErrorCode result = consumer -> remoteCall( serviceEndpoint, request );

if (result == SoaErrorCode::NO_ERROR) {
        ...
}
```

- Create a request message using the SoaMessage factory method (a version which takes a consumer endpoint argument has or will be deprecated)
    - Optionally specify a commandID, application-level version value and/or an application-level transactionID; as you require
- Set the payload on the request message
- Use the version of remoteCall which does not take a timeout argument, and returns a SoaErrorCode
    - The result (NO_ERROR or error code) can be extracted from the return value
    - The blocking method remoteCall(...) CANNOT be called from the callback thread
    - if attempted, the error code returned will be SoaErrorCode::ERROR_ILLEGAL_CALL_THREAD

a. ~~Synchronously get subscribed service status without blocking~~ **NOT SUPPORTED**

**Subscribe for service status changes**

```
SoaServiceStatus serviceStatusA = NOT_AVAILABLE;
...


// we have already subscribed for service status updates
// we can be running on any thread
SoaActionResult<SoaServiceStatus>::SharedPtr serviceStatusResult = consumer ->
requestServiceStatusCached( serviceEndpointA );


if (serviceStatusResult -> getError() != SoaErrorCode::NO_ERROR) {
        serviceStatusA = serviceStatusResult -> getResult();
}
```

- ~~Get the current service status value on demand from the service manager without blocking if the value is not cached~~
  - ~~If the service status is not known to the SOA Framework, an error code is returned, and the status value is undefined~~
  - ~~The result (NO_ERROR or error code) can be extracted from the return value~~
  - ~~The service status can be extracted from the return value using getResult() if there was no error~~
  - ~~This method, requestServiceStatusCached(), CAN be called from the callback thread~~

a. ~~Request a service status and block waiting for the status~~ **NOT SUPPORTED**

**Request the status of a service provider**

```
SoaServiceStatus serviceStatusA = NOT_AVAILABLE;
...


SoaActionResult<SoaServiceStatus>::SharedPtr serviceStatusResult = consumer -> requestServiceStatus
( serviceEndpointA, 5000 );


if (serviceStatusResult -> getError() != SoaErrorCode::NO_ERROR) {
        serviceStatusA = serviceStatusResult -> getResult();

}
```

- ~~Get the current service status value on demand from the service manager while blocking if the value is not cached~~
  - ~~The result (NO_ERROR or error code) can be extracted from the return value~~
  - ~~The service status can be extracted from the return value using getResult() if there was no error~~
  - ~~If an internal subscription for the status of the identified service has been created by a subsequent call, this method returns quickly with the cached status~~
  - ~~The blocking method requestServiceStatus(...) CANNOT be called from the callback thread~~
  - ~~if attempted, the error code returned will be SoaErrorCode::ERROR_ILLEGAL_CALL_THREAD~~

a. ~~Request a service status without blocking for the status to be received asynchronously~~ **NOT SUPPORTED**

**Request the status of a service provider without waiting**

```
SoaServiceStatus serviceStatusA = NOT_AVAILABLE;
StatusListener statusListenerA;
ServiceStatusActionListener statusActionListenerA;
...


consumer -> requestServiceStatusAsync( serviceEndpointA, statusListenerA, statusActionListenerA,
5000 );
```

- ~~Get the current service status value asynchronously on demand from the service manager~~
  - ~~The result of making the request is delivered the SoaActionResultListener<SoaSubscribeToServiceStatusContext> callback method~~
  - ~~If the error code is NO_ERROR, the service status is delivered to the SoaServiceStatusListener callback after this callback returns, else no service status will be received~~
  - ~~The non-blocking method requestServiceStatusAsync(...) CAN be called from the callback thread~~

a. Get a list of services which handle a given command ID

**Get a list of services which handle a command**

```
string requestCommand = "GET_RPM";

SoaActionResult<SoaServiceMetadataList>::SharedPtr listOfServices SoaConsumerInternal::
getCommandHandlers(requestCommand , 5000 );
```

- This functionality is available in a blocking and non-blocking version

# Gateway crash management

## For a consumer

In case of a crash or a non responsive gateway, either SLM or Stability Monitor will restart it automatically. The gateway will reconnect with the existing clients before the crash and will send a notification on the connection lost listener. However, the client will have to manage this case as all the subscriptions will be lost. There are few steps which are recommended to deal with this scenario:

- Define a connection lost listener. This listener is optional but in this case, it is needed if the client wants to restore its capability to communicate over SOA. The onConnectionLost (const std::string& cause) should be implemented.
- In the connection lost listener, check the connection lost reason. The following strings are used:
  - "lost"
  - "forced"
  - "terminated"
  - "gateway reset"

To handle the gateway crash, only the last one above is of interest.

- Resubscribe to the consumer topic: one of the first topic needed is the consumer endpoint used to receive response from remote calls. This is done in the initializeAsync()
- Resubscribe to any other subscribetoData() or subscribeToDataAsync() you may have done before the crash.

Once all those steps are implemented, even if the gateway crashes, the client will be able to resume its normal operation.

### For a service provider

see how to write a service provider: Gateway crash management

### For an onDemand broadcast consumer

TBD

## Shutdown Strategy

- To be added ...

## Required Include Paths

SOA Framework code is accessed via the namespace fnv::soa::framework.

Example. fnv::soa::framework::SoaConsumer

The required header files are found in the repo **FNV/soa** in the directory:

```
fnv/soa/include/soa/framework
```

Include example:

```
#include <soa/framework/soa_consumer.hpp>
```

Header file names are all-lowercase, underscore-separated conversions of the camel case class names they contain.

Example:

| Class Name | Path and Filename |
|---|---|
| SoaConsumer | `fnv/soa/include/soa/framework/soa_consumer.hpp` |