fnv::soa: (e) Writing a SOA Service Provider (from ECG)

Intro

An FNV SOA Service Provider implementation leverages the SoaProvider class to:

- connect to the network,
- register a service with the Service Manager (a separate process co-resident on the platform),
- wait for requests,
- act on requests,
- and to blindly publish event data for whichever consumers are listening for it.
- (coming soon) - automation of on-demand data broadcasts

Main Features

Register a service, receive service requests and respond to them

Facilitates registering a service with the service manager and binding a request message listener.

Facilitates receiving the service request messages from consumers and optionally sending response messages to the requester.

Facilitates unregistering the service.

Publish event data

Facilitates sending data messages on defined topics.

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 SoaProvider endpoint topic string used to register a service (via registerService(...)) ***MUST*** start with **SERVICES/REQUEST/** . NEVER register a service against an endpoint that starts with SERVICES/RESPONSE/

A consumer endpoint topic string to which a response is sent (via remoteCallResponse(...)) ***MUST*** start with **SERVICES/RESPONSE/** . The consumer endpoint is always provided in the request message.

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
- Your protobuf definitions need to be available in the source code repository to developers of consumers of your service
- Your service's request message format needs to be known to consumers making requests
- Your service's response message format needs to be known to consumers processing the response data
- Your service's data message format needs to be known to consumers listening for your 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 minimum retry interval = 0.5 seconds maximum retry interval = 1 minute 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 return a SoaErrorCode when the status is known, and non-blocking methods return void immediately.

Threading for blocking and non-blocking methods

The SOA Framework executes all public SoaProvider 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:

Blocking Method	What it does	Relative time to complete	Comments
SoaErrorCode **initialize()**	connects to the MQTT broker (is a required post-instantiation step)	quick	This method connects the client to the broker. 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 this may be deprecated in the future
SoaErrorCode **publishMessage**(...)	sends MQTT message	quick	when responding to remote call requests, use remoteCallResponse() instead
SoaErroCode **registerService**(...)	networks with the ServiceManager to register the service, subscribes to MQTT topic	extended for networking with ServiceManager	
SoaErrorCode **unregisterService**(...)	networks with the SerivceManager to unregister the service, unsubscribes to MQTT topic	extended for networking with ServiceManager	
SoaErrorCode **remoteCallResponse**(...)	sends MQTT message	quick	publishMessage() is not guaranteed to have the same semantics as this method

Non-blocking methods:

Non-blocking Method	What it does	Relative time to complete	Comments
void **publishMessageAsync**(...)	sends MQTT message	quick	call returns immediately, action status callback is called as soon as result of sending the message is known
void **registerServiceAsync**(...)	networks with the ServiceManager to register the service, subscribes to MQTT topic	extended for networking with ServiceManager	when responding to remote call requests, use remoteCallResponseAsync() instead call returns immediately, action status callback is called as soon as result of this action is known
void **unregisterServiceAsync**(...)	networks with the SerivceManager to unregister the service, unsubscribes to MQTT topic	extended for networking with ServiceManager	call returns immediately, action status callback is called as soon as result of this action is known
void **remoteCallResponseAsync**(...)	sends MQTT message	quick	publishMessageAsync() is not guaranteed to have the same semantics as this method call returns immediately, action status callback is called as soon as result of sending the message is known

Timeouts:

All of the public methods, except 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.registering a service is dependent on the Service Manager service)
  - by user choice of alternative methods whose response is not dependent on a remote network element (ex. publishing a message)
- as a blanket timeout applied to all 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. registering a service involves communication with the Service Manager service.
- 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.

Sending Responses:

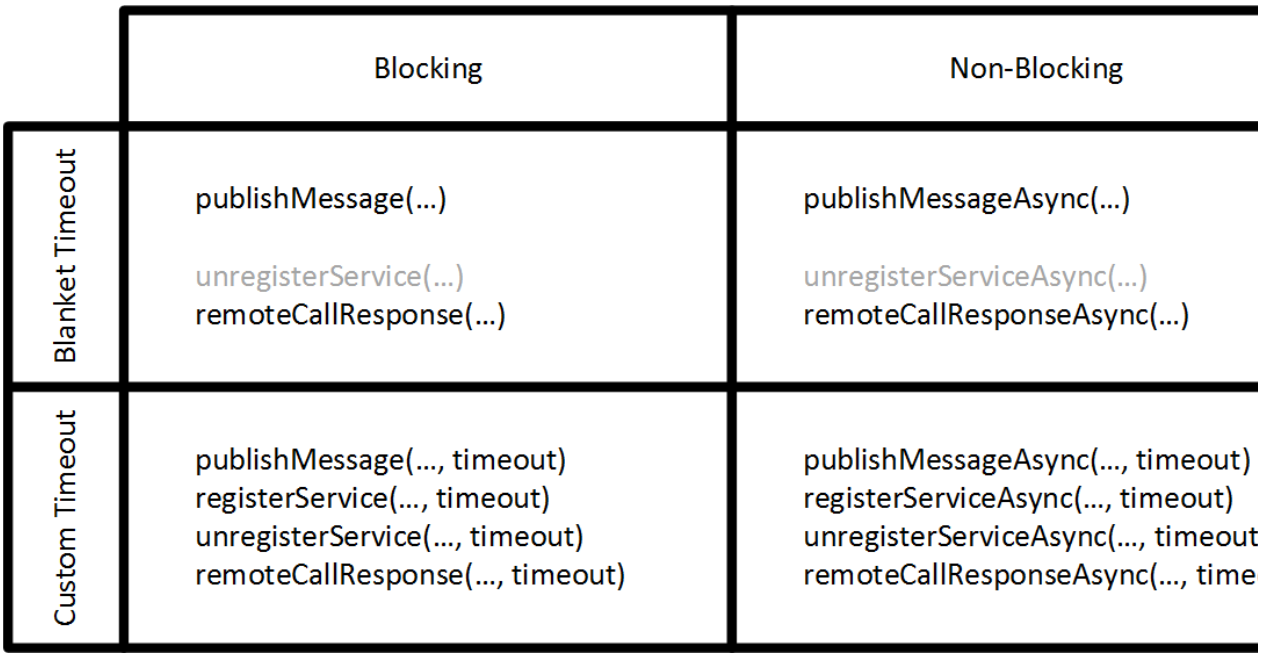
This choice is defined by the service you are providing. Your provider application may or may not be such that a response message needs to be sent to the requesting consumer.

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 (the consumers need to know which it is). Consumers making the request either always expect a response or always don't expect a response.

Matrix of Choices:



Data Structures, Listeners and Callbacks

The following SOA Framework classes are used:

- SoaProvider
  - 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
- SoaServiceRequestListener
  - An abstract class which needs to be realized and implemented by the provider developer.
  - It defines the signature of the callback method to be called when a service request message is received.
  - A reference to an object of this class is provided by the application to the SOA Framework when registering a service endpoint.
- SoaPublishMessageContext
  - 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 publishMessageAsync(...) method.
  - This object includes references to all of the data used to call the publishMessageAsync(...) method as a means to provide context to the action result listener callback method.
- SoaRegisterServiceContext
  - 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 registerServiceAsync(...) method.
  - This object includes references to all of the data used to call the registerServiceAsync(...) method as a means to provide context to the action result listener callback method.
- SoaUnregisterServiceContext
  - 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 unregisterServiceAsync(...) method.
  - This object includes references to all of the data used to call the unregisterServiceAsync(...) method as a means to provide context to the action result listener callback method.
- SoaActionResultListener<ContextType>
  - A class template to allow the provider 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 SoaProvider class is dependent on these additional classes:

- SoaMessageManager
  - This class is mostly hidden from the SOA provider 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.

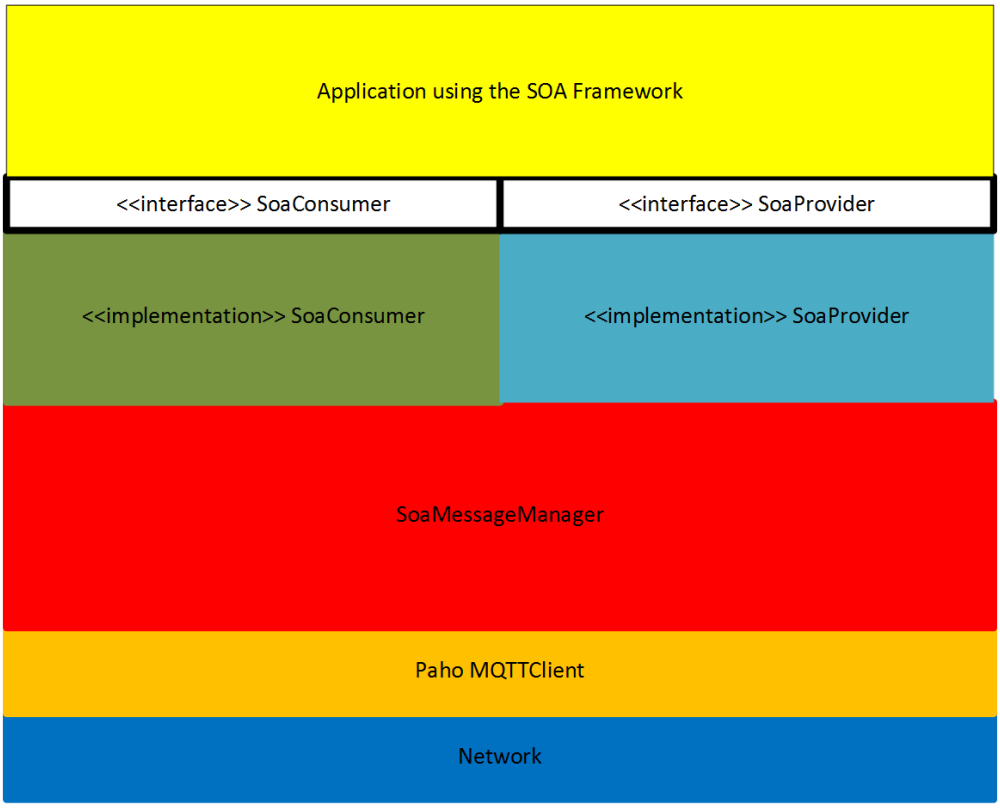
- SoaErrorCode
  - A enum class defining the error values returned by the SOA Framework.
  - **SoaErrorCode::NO_ERROR** is what you always want to see
- ~~SoaLoggerControl~~ This functionality now disabled. Use telemetryctrl to manage logs via the "soa" tag
  - ~~Useful during development. This provides 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 tree 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

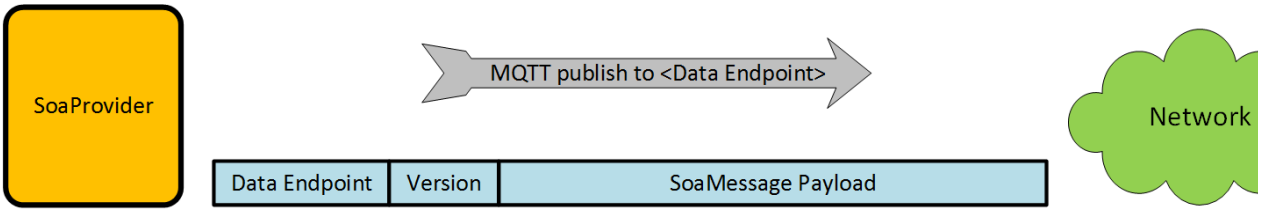
All SoaMessages which are published to the MQTT broker by the SOA library are automatically populated with the sender's ECU ID and UID before being sent. This cannot be avoided. Those fields can be examined via public methods once received. (see getSenderEcuId() and getSenderUid())

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.

- Created by the provider application by calling the SoaMessage constructor and explicitly populating the message fields



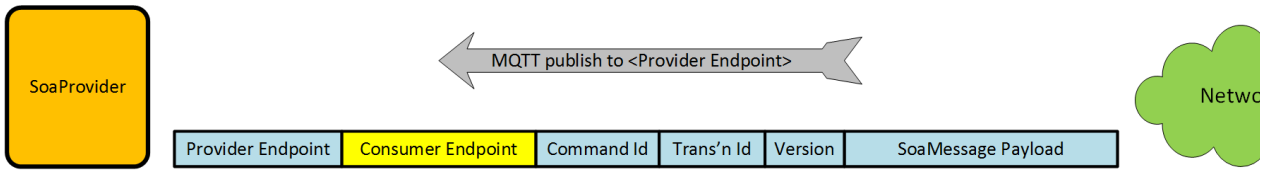
SoaMessage Data Message

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
  - 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.

- Received by the provider application in the SoaServiceRequestListener callback method onRequestReceived(SoaMessage)



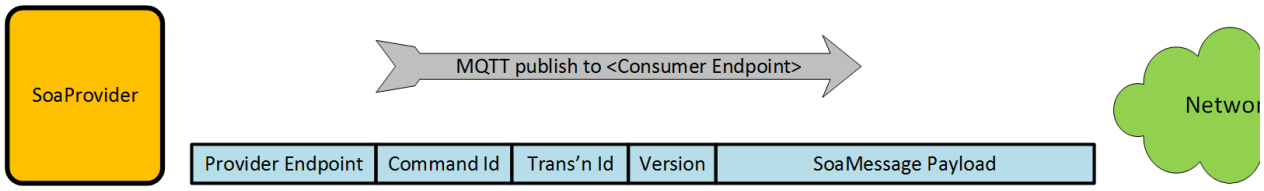
SoaMessage Request Message

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.

- Created by the provider application from the request message by calling the SoaMessage::createResponseMessage() factory method, and subsequently populating the payload field.



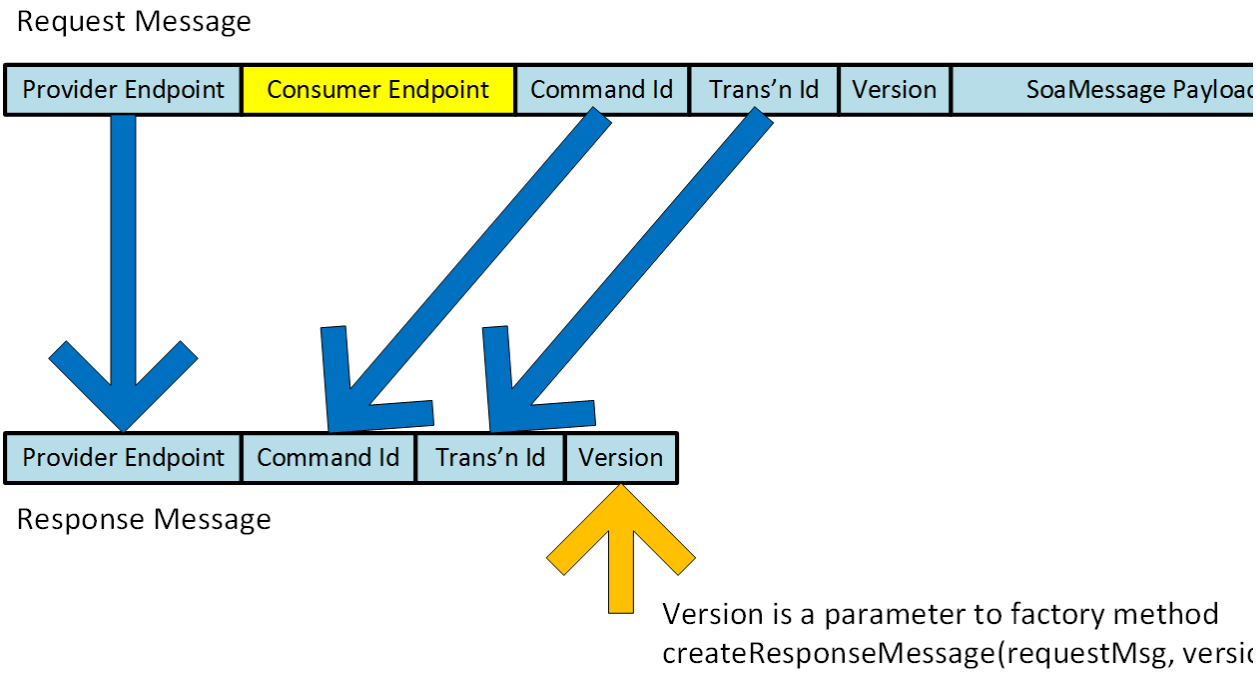
SoaMessage Response Message

Creating a response message from a request message:

- call the factory method:

```
SoaMessage::SharedPtr SoaProvider::createResponseMessage(SoaMessage::SharedPtr requestMessage,
const string& version)
```

- The factory method will create a new pre-populated SoaMessage object as follows:
- The provider only needs to add the SoaMessage payload



No Consumer Endpoint in a response message

IMPORTANT! The response message created from the request 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**. To send the response to the consumer, get the response endpoint directly from the request message using getConsumerEndpoint()

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 remoteCall 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.

It is recommended to use the constexpr methods defined above

Connection APIs

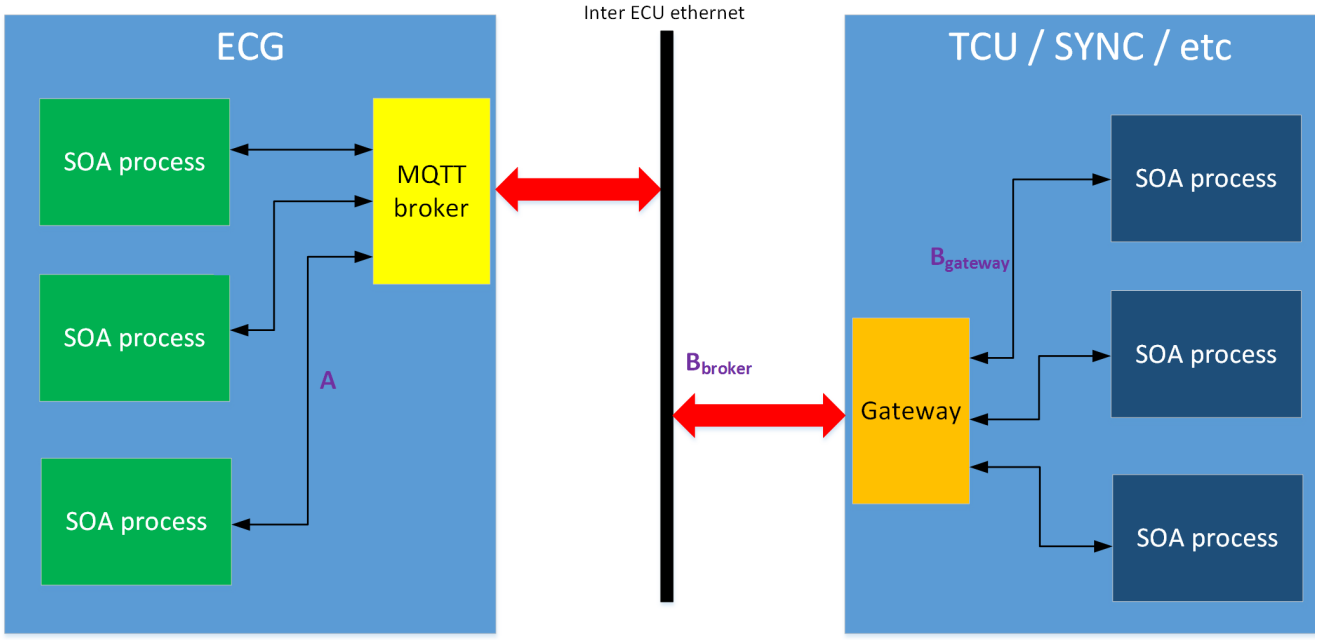
Connecting to the network and initializing the SoaProvider object are the first things you need to do for or with the object. **BEWARE!** There are differences for ECG processes and Gateway clients.

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 on ECG.

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

SoaProvider::initialize()	• causes message manager to connect to broker if not already connected. • if the message manager cannot connect to the broker, this call will fail • do not attempt to call initialize() for a SoaProvider once it has been successfully initialized. • 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 SoaProvider can be used in the wake of lost connection/reconnection. (race conditions exist if relying on onConnectionLost(...)/onReconnected(...) alone)	• causes message manager to register to gateway BUT DOES NOT trigger the gateway to connect to broker. • if the gateway is down, this method will fail • if gateway is not connected to broker when called, this method will fail.
SoaProvider::initializeAsync(listener)	• NEW and RECOMMENDED • causes message manager to attempt to connect to broker. • upon connection to broker, initialization completes, and calls back to listener object's onInitialized(...) method • do not attempt to call initializeAsync(...) for a SoaProvider once it has been successfully initialized, including after onDeinitialized(...) is called • 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 the gateway is down, it will continue to reattempt register to gateway (1 hour) • upon connection to broker, initialization completes, and calls back to listener object's onInitialized(...) method • do not attempt to call initializeAsync(...) for a SoaProvider once it has been successfully initialized, including after onDeinitialized(...) is called • may be called before or after connectAsync()
SoaMessageManager::connectAsync()	• explicit request to trigger connection attempt • can be called before calling SoaProvider::initialize(). • notification is via the connection listener • once notified of successful connection, initialize() will succeed	• this is the ONLY API to trigger the gateway to attempt to connect to the broker • this method will first attempt to register to the gateway • once registered to the gateway, onConnect(...) is called with SoaErrorCode::NO_ERROR__GATEWAY_CONNECTED_ONLY • once the gateway is connected to the broker, onConnect(...) is called with SoaErrorCode::NO_ERROR • once connected to the broker, SoaProvider which have called initializeAsync(...) will complete initialization. • 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 • if connection is lost, connection attempts will be repeated until connected as per the configured reconnection algorithm (in connection options) • notification is via the reconnection listener	• NOT SUPPORTED • 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 • 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 • if this process is the only gateway client, the gateway will disconnect from the broker

SoaMessageManager:: disconnectGlobalAsync()	NOT SUPPORTED	• forces the gateway to disconnect from the broker • must be registered with the gateway to call this method (either through a call to connectAsync() or initializeAsync() • DOES NOT cause processes registered with the gateway to be deregistered • 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 • use SoaProvider::initialize() to synchronously connect and initialize	• DO NOT USE THIS METHOD

SoaProvider::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 SoaProvider class. Every instance must call initialize() or initializeAsync(...). 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. 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

SoaProvider::initializeAsync(SoaInitializedProviderListener &)

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 SoaInitializedProviderListener.

Your application may include more than one instance of the SoaProvider 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

SoaProvider::SoaInitializedProviderListener

- Create a class derived from SoaInitializedProviderListener that realizes the pure-virtual callback interfaces:
  - onInitialized(SoaErrorCode, SoaProvider *) an error code of NO_ERROR means the SoaProvider referenced by the pointer is able to be used. Each time this method is called, the consumer will need to regsiter all services anew.
  - onDeinitialized(SoaErrorCode, SoaProvider *) when this method is called, the SoaProvider 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 service regsitrations, 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 SoaProvider::initialize(). As of SOA Framework library version SOA-R0.66.0, you can call connectAsync() to connect the message manager before calling SoaProvider::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 you 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 SoaProvider::initialize() from your onConnect(...) callback because it is called on the callback thread.

Bad mix of methods

You MUST NOT call SoaProvider::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)

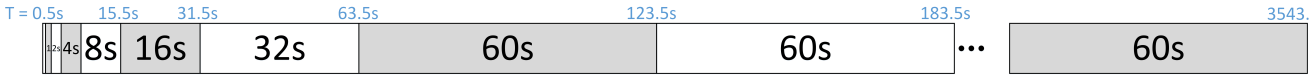
When auto-reconnect is triggered or when connectAsync() or reconnect() are called, the background connection retry task attempts to connect immediately. Each time its attempt fails due to the server being unavailable it waits before retrying again. When it discovers the next retry would be scheduled 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 explicilty 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 Provider 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 servicing requests, create your SoaServiceRequestListener class(es) and object(s)
d. If using non-blocking methods, create SoaActionResultListener<ContextType> classes and objects
e. Create a SoaProvider; this includes creating a SoaMessageManager and connecting your application to the MQTT broker.
f. Set a non-default blanket timeout if needed
g. If servicing requests, register your service
h. Run your app:
    i. Prepare to service requests - i.e. your main application thread needs to remain alive to provide an application life cycle
    ii. If sending blind data broadcasts, publish these messages
i. Respond to a request

SoaErrorCodeUtil

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

Details

1. ~~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

Enable SOA Framework logging

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


}
```

- ~~Enable by calling SoaLoggerControl::On(true)~~ 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

2. Create your endpoints

Create your endpoints

```
const SoaClientEndpoint::SharedPtr SERVICE_ENDPOINT = SoaClientEndpoint::createClientEndpointMqtt
("SERVICES/REQUEST/MY_SERVICE_ENDPOINT");
const SoaClientEndpoint::SharedPtr DATA_ENDPOINT    = SoaClientEndpoint::createClientEndpointMqtt
("SERVICES/DATA/MY_DATA_ENDPOINT");
```

- Methods *will be available* by EOM July 2017 to avoid the need to include "SERVICES/REQUEST/" and "SERVICES/DATA/" prefixes

3. If servicing requests, Create a SoaServiceRequestListener class and object

Implementing a Service Request Listener

```
class RequestListener : public SoaServiceRequestListener{
        void onRequestReceived(SoaMessage::SharedPtr request) {
                lock_guard<mutex> lock(providerMutex);
                cout << "sync_rpc_provider_test: adding " + request->getRawPayload() + " request to
queue"<< endl;
                requestQueue.push(request);
                cout << "sync_rpc_provider_test: queue has " + to_string(requestQueue.size())  + "
requests" << endl;
                cout << "sync_rpc_provider_test: notifying new request received" << endl;
                conditionVariable.notify_one();
        }
};
```

- Create classes derived from the interface class SoaServiceRequestListener
- Implement the pure virtual method onRequestReceived()
- This will be called on the Callback thread when a request sent to your topic is received.

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

Implementing an Action Result Callback

```
class RegisterServiceActionListener : public SoaActionResultListener<SoaRegisterServiceContext>{
        virtual void onActionCompleted(SoaErrorCode error, shared_ptr<SoaRegisterServiceContext>
actionContext){
                cout << "Called on action complete call back" << 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.

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

Instantiating a SoaProvider

```
SoaProvider::SharedPtr initializeProvider(){
        // Set connection options
    //
        SoaConnectionOptions::SharedPtr connectionOptions = SoaConnectionOptions::
createConnectionOptions();
        connectionOptions -> setClientId( providerClientId );   // needed during development phases
        connectionOptions -> setDebugBrokerUrl( brokerURL );    // needed during development phases
        connectionOptions -> setConnectionTimeOut( 2 );         // in seconds

        // Instantiate message manager
        messageManager = SoaMessageManager::createMessageManager( *connectionOptions );

        //Create provider
        SoaProvider::SharedPtr provider = SoaProvider::createSoaProvider(messageManager);

    if (provider == nullptr) {
        cout << "provider creation failed: bad parameter(s)" << endl;
    } else {
                // post instantiation initialize to connect to MQTT broker
                SoaErrorCode error = provider -> initialize();
                if (error != SoaErrorCode::NO_ERROR){
                        cout << "initialize failed. Error was " << SoaErrorCodeUtil::getString() << endl;
                        provider = nullptr;
                }
        }

        return provider;
}
```

- 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 SoaProvider object using the SoaMessageManager object
- Once the SoaProvider is created, you MUST call initialize() as a post instantiation step to connect to the MQTT broker

> **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.

> **BEWARE: now call initialize() to connect**
>
> As of SOA-R0.52 as per Git tag, you MUST call SoaProvider::initialize() to connect to the MQTT broker instead of calling SoaMessageManager::connect()

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

6. Set a non-default blanket timeout if needed

Setting a blanket timeout

```
SoaProvider::SharedPtr provider = ...

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

   - Call setTimeout() on the SoaProvider instance.
   - All SoaProvider methods which do not take a timeout parameter will use this timeout value
   - If not set, the default blanket timeout will apply
7. If servicing requests, register your service

Register your service

```
// an instance of the request listener is instantiated
RequestListener              requestListener;
// an instance of an action result listener is instantiated
RegisterServiceActionListener registerActionListener;


...


SoaProvider::SharedPtr provider = ...

// register my service
provider -> registerServiceAsync(SERVICE_ENDPOINT, requestListener, registerActionListener, 2000 );
```

   - You need a request listener implemented to service requests
   - Have a defined service endpoint that consumers will send requests to.
   - Have an action result listener implemented if using the non-blocking version
   - Your service endpoint will be registered with the Service Manager service, allowing consumers to query your service's availablity
      - Because registration has a dependency on network communication, the blanket timeout is not applied. An explicit timeout is required.
8. Run your app
   a. Prepare to service requests - i.e. your main application thread needs to remain alive to provide an application life cycle
   b. If sending blind data broadcasts, publish these messages

Sending a data broadcast

```
// an instance of an action result listener is instantiated
DataPublishActionListener publishActionListener;
...
// an instance of a message object is instantiated
SoaMessage::SharedPtr dataMessage = make_shared<SoaMessage>();
SoaProvider::SharedPtr provider = ...



while (serviceActive) {

        string messagePayload = prepareMessage( getCurrentData() );
        // set the message payload (overloaded setters accept a string argument or a google::protobuf::
Message argument)
        dataMessage -> setPayload( messagePayload );
        provider -> publishDataAsync( DATA_ENDPOINT, dataMessage, publishActionListener );


        periodicDelay();        // wake up at a correct time to next broadcast periodic data
}
```

- Have an action result listener implemented if using the non-blocking version
- Have a SoaMessage object to send
- Set the payload on the message
- Publish the message

9. Respond to a request

Respond to a request

```
// This code is NOT running on the callback thread because it uses the blocking version of
remoteCallResponse
// create a new response message from the request message if we got a consumer endpoint
SoaClientEndpoint::SharedPtr responseEndpoint = requestMessage -> getConsumerEndpoint();
if (responseEndpoint != nullptr) {
        SoaMessage::SharedPtr responseMessage = SoaMessage::createResponseMessage(requestMessage,
"1.0");

        string responsePayload = buildResponse( requestMessage );   // application defined method for
the example
        responseMessage -> setPayload( responsePayload );

        SoaErrorCode rc = provider->remoteCallResponse( responseEndpoint, responseMessage );
}
```

- Verify the request message contains a consumer endpoint
- Instantiate a response message from the request message
- Set the payload on the response message
- Send the response
    - The non-blocking method remoteCallResponseAsync(...) may be called from the callback thread
    - The blocking method remoteCallResponse(...) CANNOT be called from the callback thread
        - if attempted, the error code returned will be SoaErrorCode::ERROR_ILLEGAL_CALL_THREAD

Gateway crash management

For a service provider

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 provider topic: one of the first topic the service provider needs to re-subscribe to is its endpoint to be able to receive remote calls from consumers. This subscription is done when calling the registerService(SERVICE_ENDPOINT, requestListener, timeout)

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

For a service consumer

see how to write a consumer: Gateway crash management

For an onDemand broadcast provider

TBD

Shutdown Strategy

- To be added ...

Required Include Paths

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

Example. fnv::soa::framework::SoaProvider

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_provider.hpp>
```

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

Example:

Class Name	Path and Filename
SoaProvider	`fnv/soa/include/soa/framework/soa_provider.hpp`