fnv::soa: (f) Writing a SOA on-demand data broadcast Consumer or Provider (from ECG)

Table of Contents

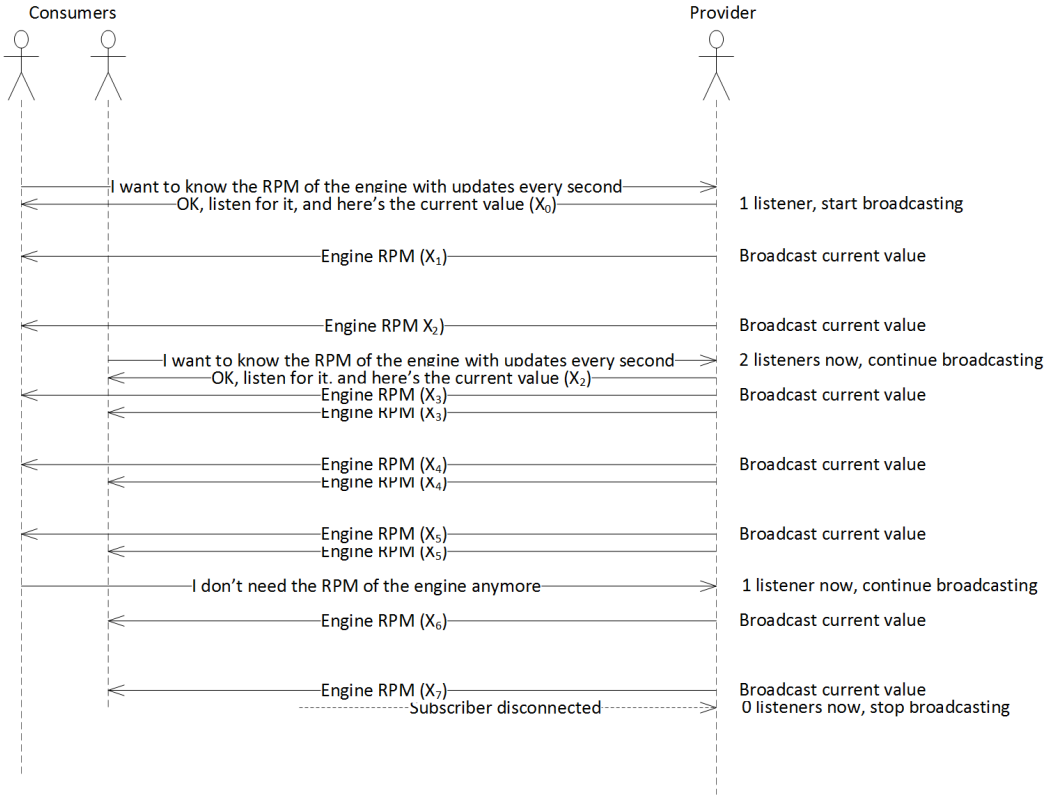
Intended Audience

You are on this page because you either want to create an on-demand data broadcast consumer or an on-demand data broadcast provider. These are polar opposites, but can be better documented together because each relies on the other.

Intro to On-Demand Data Broadcast Feature

- This feature facilitates the implementation of a pub-sub pattern for cooperating data providers and consumers..
- The on-demand data broadcast feature allows for resource usage reduction - reduce network bandwidth, reduce CPU usage - by avoiding broadcasts when no consumer is listening, by facilitating a publication / subscription model between consumers and providers over MQTT.
- Data providers may always have data to share; which could be broadcast unconditionally on well known topics to consumers who need it. This may be at regular intervals (ex. every 1 second) or it may be only when the data value changes.
  But providers on the MQTT transport system don't know if any consumer is listening unless a consumer subscription reaches the provider, and equally important, if a listening consumer stops listening for any reason including unsubscribing or disconnecting.
- In more complex scenarios, some consumers may only care about the data if it exceeds a particular value. Passing the decision making to the provider allows for another means of reducing resource usage.
- The feature is built into the FNV SOA Framework. It is built on top of- and built into the SoaConsumer and SoaProvider classes.

High Level Flow Diagram

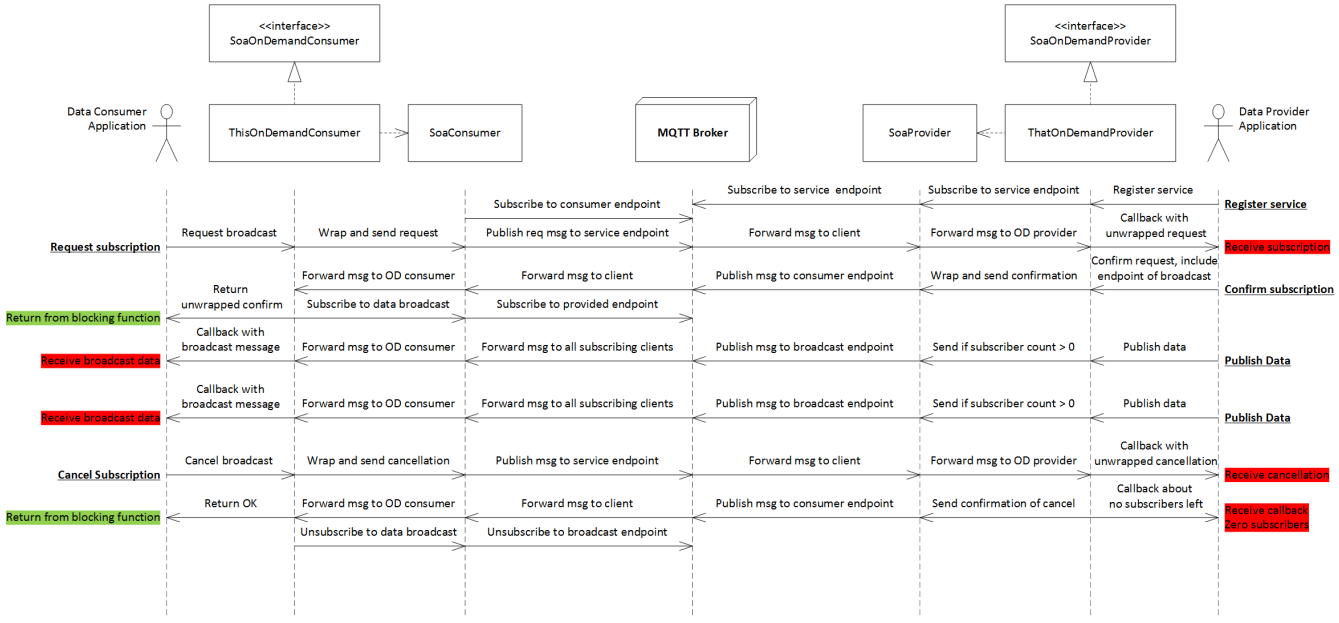


Feature Implementation Flow Diagram

The following UML-based flow diagram includes a more detailed view of a similar flow with one consumer as implemented by this feature.

The simple rectangles are classes within each application. The 3D box is the MQTT broker

Feature interfaces are represented by labels to the left and right. The **underlined bold** labels represent the basic inward interfaces offered by this feature. The highlighted labels represent response data from the SOA framework. **Green** labels are blocking function returns, while **red** labels are asynchronous callbacks.



Feature Usage Note

The previous two diagrams show how users use this feature.

The high level view shows that the provider developer can create a contract for consumers to subscribe for specfic data broadcasts rather independent of the SOA Library on-demand data broadcast feature, as long as the contract is constrained to the basic subscribe/response, publish and cancel flow. When using this feature, the messages defined in the contract are forwarded as expected, untouched by the SOA Framework with only very limited exceptions.

This feature efficiently augments those messages with metadata to allow the feature to manage subscriptions for the provider. (See notations in the Feature Implementation Flow Diagram for "Wrap and Send"). This also facilitates notification of ungraceful disconnects.

The metadata used by this feature is piggybacked on the user control messages, but not (currently) on the data broadcast messages

Main Features

On Demand Provider

- Register a service as an on-demand provider against a well known endpoint
- Receive subscription requests from on-demand consumers
- Use your own service contract for requests, cancellations, acknowledgements and publications wrt message format and content
    - Acknowledgement message can but need not include current data values
    - Cancellation messages are optional within non-optional cancellation requests
- Provide dynamic endpoints for consumers to subscribe to; confirm subscriptions
- Publish data on the wire ONLY when there are consumers (transmission automatically suppressed when no subscribers)
- Receive notifications of cancelled subscriptions
- Receive notifications when no subscribers remaining on a per publication endpoint basis
- Zero subscriber count notification includes subscribers which ungracefully disconnect
- Blocking and non-blocking API available
- Realized SoaOnDemandProvider interface class "is a" provider supporting a single provider endpoint, servicing multiple dynamic broadcast endpoints
- Supports unlimited concurrent dynamic data endpoints per on-demand provider
- Supports unlimited concurrent subscribers per dynamic endpoint

On Demand Consumer

- Send a subscription request to on-demand provider
- Use service provider's contract for requests, cancellations, acknowledgements and publications wrt message format and content
- Receive subscription acknowledgement/denial including current data as per service message contract
    - No need to explicitly subscribe to on-demand subscription endpoint
- Cancel subscription
    - No need to explicitly unsubscribe from on-demand subscription endpoint
- Receive subscription data messages
- Receive notification if provider ungracefully disconnects, auto-unsubscribed from dynamic endpoint
- Only blocking API is available (as of 2017/12/1) for requests and cancellations
- Realized SoaOnDemandConsumer interface class "is a" consumer supporting a single provider endpoint and single data listener callback
- Only one subscription supported at a time by a SoaOnDemandConsumer object

Requirements

On Demand Provider

This API has been created to implement an on-demand provider object that represents a single provider entity. Whereas the SoaProvider class instantiates as an object that facilitates multiple providers, offering users the ability to register multiple services and binding multiple arbitrarily defined and instantiated listener objects to accomodate multiple services and asynchronous API action status listeners, the SoaOnDemandProvider class is an interface class for realizing an application-defined class which, when instantiated, only allows for a single service provider, and can listen for a single data endpoint at a time.

The provider endpoint (ex. SERVICES/REQUEST/your_service_name_here) is injected via the constructor of the base interface class. This is the endpoint consumers will send subscription requests to.

The on-demand provider needs a pre-initialized SoaProvider object at time of instantiation as well.

The user must implement a number required callback methods for basic notifications, and optionally implement a few action status callback methods needed only if the asynchronous API is used.

- MUST subclass SoaOnDemandProvider
- SoaOnDemandProvider constructor requires pre-initialized SoaProvider object and On-Demand provider endpoint
- MUST implement callback onBroadcastReceived(...) to get subscription requests
- MUST implement callback onBroadcastCancelled(...) to be notified of a subscription cancellation
- MUST implement callback onBroadcastNoActiveListeners(...) to be notified when no there are no more listeners for a dynamic endpoint
- Register the on demand service after successful instantiation
- Acknowledge or deny an subscription request using the provided SoaHandle within a limited time

    - subscription requests are internally cached when received, awaiting acknowledgement; as an entry in a cache it will be aged out in a short time
- Optionally implement the action status callback method (onRegisterStatus(...), onSetStatus(...), onPublishStatus(...)) for the associated non-blocking "async" method
- Optionally call SoaOnDemandProvider::publishMessge(...) or SoaOnDemandProvider::publishMessgeAsync(...) instead of the SoaProvider versions to take advantage of message send suppression when no subscribers exist for the broadcast data endpoint

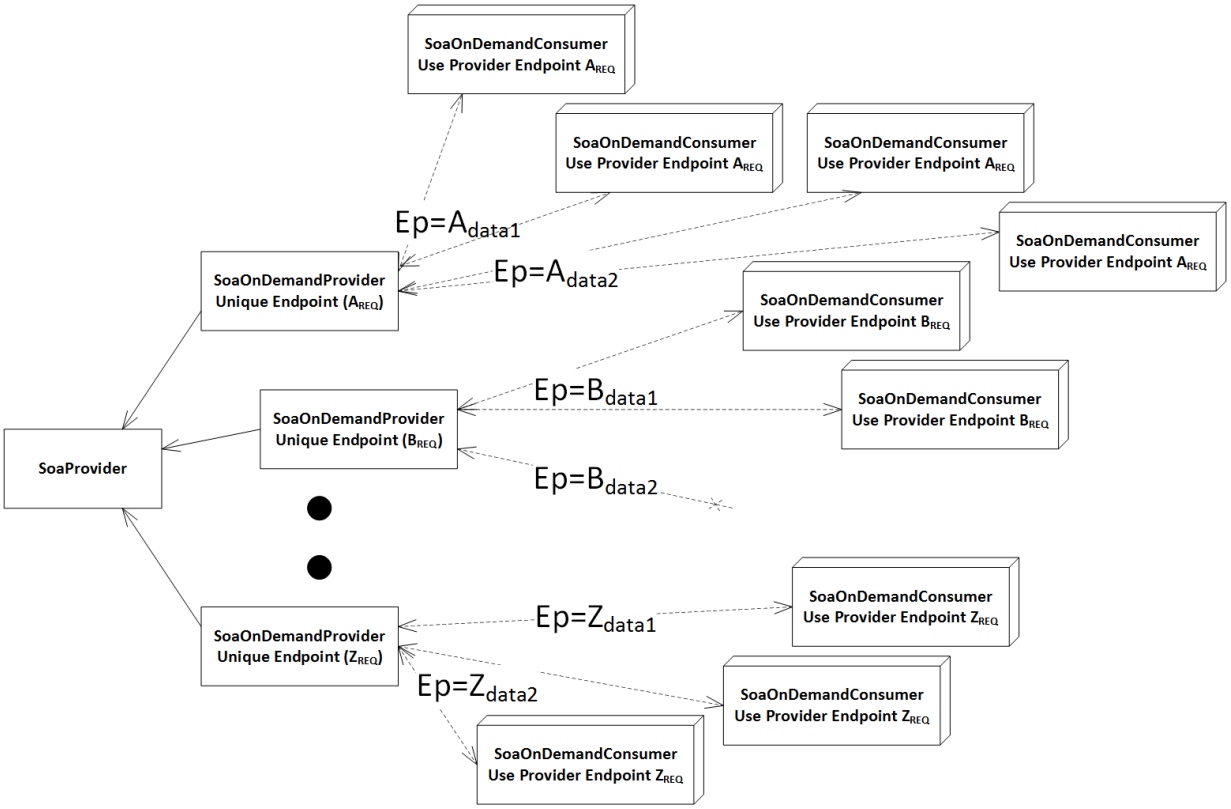
> **request message consumerEndpoint field is reserved**
>
> The consumer endpoint field in the request message (ex. assigned SoaMessage::setConsumerEndpoint(...)) is reserved by this feature, and any data assigned to it by the consumer will be destroyed.

SoaOnDemandProvider relationships

One SoaProvider object facilitates multiple derived SoaOnDemandProviders objects

Each SoaOnDemandProvider object listens for requests on a unique endpoint (Ex. A), and broadcasts data to multiple data endpoints (ex. Ep=A_{data2})

Each data endpoint can have zero or more subscribers.



On Demand Consumer

This API has been created to implement an on-demand consumer object that represents a single consumer entity. Whereas the SoaConsumer class instantiates as an object that facilitates multiple consumers, offering users the ability to send remote procedure call request messages to multiple services and binding multiple arbitrarily defined and instantiated listener objects to accomodate multiple data subscriptions and asynchronous API action status listeners, the SoaOnDemandConsumer class is an interface class for realizing an application-defined class which, when instantiated, only allows for a single subscription by a single consumer.

The provider endpoint (ex. SERVICES/REQUEST/their_service_name_here) is injected via the constructor of the base interface class. This is the endpoint the consumer sends its subscription request to.

The on-demand consumer needs a pre-initialized SoaConsumer object at time of instantiation as well.

The user must implement a number required callback methods for basic notifications (and optionally implement a few action status callback methods needed only if the asynchronous API is used, when available.)

- MUST subclass SoaOnDemandConsumer

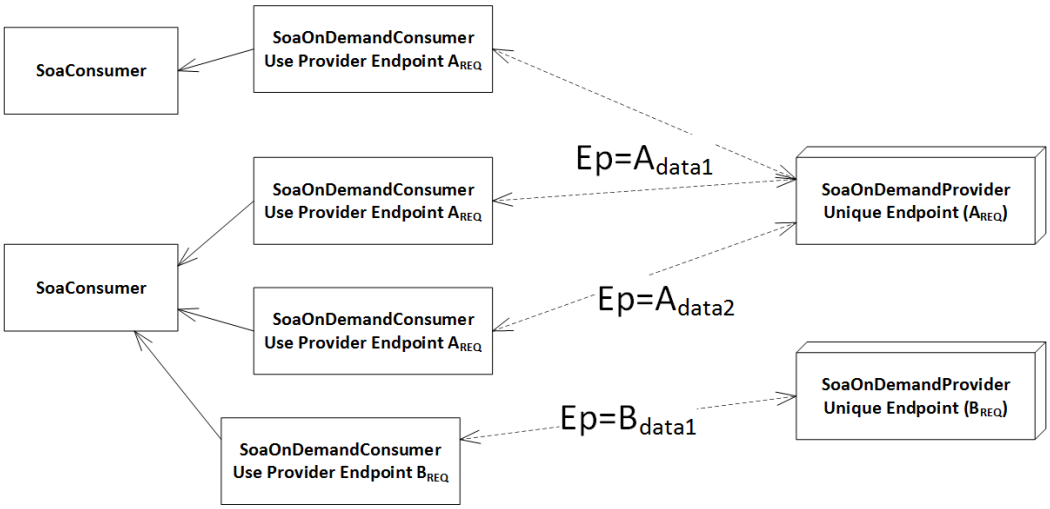
- SoaOnDemandConsumer constructor requires pre-initialized SoaConsumer object and the On-Demand provider endpoint
- MUST implement callback onDataReceived(...) as per the provider's definition of data broadcast message
- Create a subscription request message as per the provider's definition
- Request a subscription to data broadcasts from the provider
- Cancel its subscription when no longer needed
- Optionally implement the action status callback method (onRequestStatus(...), onCancelStatus(...)) for the associated non-blocking "async" method, when supported

SoaOnDemandConsumer relationships

One SoaConsumer object facilitates multiple derived SoaOnDemandConsumer objects

Each SoaOnDemandConsumer can only ever make requests to a single SoaOnDemandProvider; provider endpoint is injected in the constructor (Ex. A_{REQ})

A SoaOnDemandConsumer supports only a single subscription at a time; a single data endpoint (ex. Ep=A_{data1}), data is routed to its solitary data message callback.



Building an On-Demand Client

On Demand Provider

Basic steps

Expanding on the interactions described in the introducation, these are the basic steps in building an on-demand provider

- Implement a class derived from SoaOnDemandProvider - refer to the requirements section for related information
- Implement all the requirements to instantiate and initialize a SoaProvider object - refer to the programmer's guide for SoaProvider
- In the application thread, instantiate your derived SoaOnDemandProvider with the SoaProvider instance and your on-demand provider's endpoint as arguments
- Register your on-demand provider to enable receiving requests
- Wait for requests
- Handle requests by setting up subscriptions = acknowledging them with the request-specfic data endpoint and a response message, or denying them
- Publish your data to the appropriate data endpoints

Example (Async)

On Demand Provider

```
// my_on_demand_provider.hpp
// WARNING! This code has been exerpted and modified from an existing larger source file.
// This text has not been compiled
// Some code had been intentionally omitted for brevity


#include <soa_client_endpoint.hpp>
#include <soa_error_code.hpp>
#include <soa_on_demand_provider.hpp>
```

```
namespace framework = fnv::soa::framework;

using SoaClientEndpoint = framework::SoaClientEndpoint;
using SoaMessageManager = framework::SoaMessageManager;
using SoaProvider = framework::SoaProvider;
using SoaMessage = framework::SoaMessage;
using SoaHandle = framework::SoaHandle;
using SoaErrorCode = framework::SoaErrorCode;
using SoaSetOnDemandBroadcastContext = framework::SoaSetOnDemandBroadcastContext;
using SoaPublishMessageContext = framework::SoaPublishMessageContext;
using SoaConnectionOptions = framework::SoaConnectionOptions;
using SoaOnDemandProvider = framework::SoaOnDemandProvider;

namespace myprovider {


class MyOnDemandProvider : public SoaOnDemandProvider {
    public:
        typedef std::shared_ptr< MyOnDemandProvider > SharedPtr;

        MyOnDemandProvider (SoaMessageManager::SharedPtr msgMgr, SoaProvider::SharedPtr provider,
SoaClientEndpoint::SharedPtr serviceEndpoint) :
            SoaOnDemandProvider(provider, serviceEndpoint),
            mp_msgMan(msgMgr),
            m_data1PublishThread((pthread_t)-1),
            DATA_TOPIC1(std::string(SoaClientEndpoint::createRequestEndpointName("ONDEMAND_FT1"))),
            DATA_ENDPOINTX(mp_msgMan->createClientEndpoint(DATA_TOPIC1)),
            DUMMY_ENDPOINT(mp_msgMan->createClientEndpoint("_DUMMY_EP_")),
            m_data1(0)
        {}

        virtual ~MyOnDemandProvider ();

        // callbacks for external events - triggered by consumers
        virtual void onBroadcastRequestReceived(SoaMessage::SharedPtr requestMessage, SoaHandle requestHandle);

        virtual void onBroadcastNoActiveListeners(SoaClientEndpoint::SharedPtr dataEndpoint);

        virtual void onBroadcastCancelled(SoaMessage::SharedPtr cancelMessage);

        // callbacks for local events - triggered by local code
        virtual void onRegisterStatus(SoaErrorCode error);

        virtual void onSetStatus(SoaErrorCode error, std::shared_ptr<SoaSetOnDemandBroadcastContext> context);

        virtual void onPublishStatus(SoaErrorCode error, std::shared_ptr<SoaPublishMessageContext> context);

        int startThread();
        void joinThread();

    private:
        SoaMessageManager::SharedPtr mp_msgMan;
        pthread_t m_data1PublishThread;

        const std::string DATA_TOPIC1;
        const SoaClientEndpoint::SharedPtr DATA_ENDPOINTX;
        const SoaClientEndpoint::SharedPtr DATA_ENDPOINTY;
        const SoaClientEndpoint::SharedPtr DUMMY_ENDPOINT;
        const int DATA1_PUBLISH_PERIOD = 3; // seconds

        int m_data1;    // THE DATA TO BROADCAST
                        // not included in this code, this value changes over time

        std::string getData1() {
            return std::to_string(m_data1);
        }


        // for publishing data values
        static void * data1Publisher(void * context);
```

```
        void publishLoop(SoaClientEndpoint::SharedPtr, std::function<std::string(void)>, int period);
};


} // namespace myprovider

// ---------------------------------------------------------------------
// my_on_demand_provider.cpp


#include "my_on_demand_provider.hpp"


namespace myprovider {

// callbacks - external events
void MyOnDemandProvider::onBroadcastRequestReceived(SoaMessage::SharedPtr requestMessage, SoaHandle
requestHandle) {
    // instead of queuing the request, can use the async method here
    SoaClientEndpoint::SharedPtr dataEndpoint = nullptr;

    // handle the request
    if (requestMessage != nullptr) {
        bool isRequestValid = true;
        std::string initialValue;
        if (requestMessage->getRawPayload() == "subscription_request_for_data_X") {
            dataEndpoint = DATA_ENDPOINT_X;
            initialValue = getDataX();
        } else if (requestMessage->getRawPayload() == "subscription_request_for_data_Y") {
            dataEndpoint = DATA_ENDPOINT_Y;
            initialValue = getDataY();
        } else {
            isRequestValid = false;
        }

        SoaMessage::SharedPtr responseMessage = SoaMessage::createResponseMessage(requestMessage, appVersion);
        if (isRequestValid) {
            responseMessage->setPayload(initialValue);

            // acknowledge the request
            setOnDemandBroadcastAsync(dataEndpoint, responseMessage, requestHandle, 1000);
        } else {
            // deny the request
            responseMessage->setTransactionError(SoaMessage::TransactionError::GENERAL_ERROR);
            setOnDemandBroadcastAsync(DUMMY_ENDPOINT, responseMessage, requestHandle, 1000);
        }
    }
}

void MyOnDemandProvider::onBroadcastNoActiveListeners(SoaClientEndpoint::SharedPtr dataEndpoint) {
    std::cout << ("No listeners for " + dataEndpoint->toString() + "\n");
}

void MyOnDemandProvider::onBroadcastCancelled(SoaMessage::SharedPtr cancelMessage) {
    std::cout << ("Subscription cancelled\n");
}

// callbacks - application events
void MyOnDemandProvider::onRegisterStatus(SoaErrorCode error) {
    if (error != SoaErrorCode::NO_ERROR) {
        std::cout << ("Failed to register service: " + SoaErrorCodeUtil::getString(error) + "\n");
    }
}

void MyOnDemandProvider::onSetStatus(SoaErrorCode error, std::shared_ptr<SoaSetOnDemandBroadcastContext>
context) {
    if (error == SoaErrorCode::NO_ERROR) {
        std::cout << ("Added subscription for " + context->getDataEndpoint()->toString() + "\n");
    } else {
        std::cout << ("Subscription not added: " + context->getResponseMessage()->getRawPayload() +
                      ", Error: " + SoaErrorCodeUtil::getString(error) + "\n");
```

```
    }
}

void MyOnDemandProvider::onPublishStatus(SoaErrorCode error, std::shared_ptr<SoaPublishMessageContext> context)
{
    if (error != SoaErrorCode::NO_ERROR) {
        std::cout << ("publishAsync result: " + SoaErrorCodeUtil::getString(error) + "\n");
        std::cout << ("=> msg payload = " + context->getMessage()->getRawPayload() + "\n");
    }
}

// publish at regular intervals given by the period
void MyOnDemandProvider::publishLoop(SoaClientEndpoint::SharedPtr dataEndpoint, std::function<std::string(void)
> getData, int period) {
    SoaMessage::SharedPtr message = std::make_shared<SoaMessage>();
    SoaClientEndpoint::SharedPtr providerEndpoint = mp_msgMan->createClientEndpoint(getServiceEndpoint()-
>toString());
    message->setProviderEndpoint(providerEndpoint);
    std::chrono::seconds periodSeconds(period);

    std::cout << ("starting publishLoop: " + std::to_string(period) + " seconds\n");
    while ( !m_shutdown ) {

        message->setPayload(getData());
                // only sent out on the wire if there is at least one subscriber
        publishMessageAsync(dataEndpoint, message);

        {
            // time our publishes by waiting for the length of the period for a shutdown event
            std::unique_lock<std::mutex> lock(m_shutdownMutex);
            m_shutdownCondvar.wait_for(lock, periodSeconds);
        }
    }

    std::cout << "leaving publishLoop\n";
}

//static
//  thread executable for publishing data
void * MyOnDemandProvider::data1Publisher(void * context) {
    MyOnDemandProvider * myODP = static_cast<MyOnDemandProvider*>(context);

    std::function<std::string(void)> functor = [myODP]() {
            return myODP->getData1();
    };

    if (myODP) {
        std::cout << "MyOnDemandProvider: data1Publisher thread started\n";
        myODP->publishLoop(derivedODP->DATA_ENDPOINTX, functor, derivedODP->DATA1_PUBLISH_PERIOD);
    } else {
        std::cout << "MyOnDemandProvider: can't run data1Publisher\n";
    }
    return nullptr;
}



int MyOnDemandProvider::startThread() {
    // publish the data on regular intervals
    return pthread_create(&m_data1PublishThread, nullptr, &data1Publisher, this);
}

void MyOnDemandProvider::joinThreads() {
    std::cout << "MyOnDemandProvider waiting to join threads\n";
    pthread_join(m_data1PublishThread, nullptr);
    std::cout << "MyOnDemandProvider data publisher joined\n";
}

} // namespace myprovider
```

```
int main() {
    // set up dependencies of the on-demand provider
    SoaConnectionOptions::SharedPtr connOpts = SoaConnectionOptions::createConnectionOptions();
    connOpts->setClientId(m_providerClientId);
    connOpts->setDebugBrokerUrl(brokerURL);

    SoaMessageManager::SharedPtr msgMgr = SoaMessageManager::createMessageManagerMqtt(*connOpts);

    // create provider
    SoaProvider::SharedPtr provider = SoaProvider::createSoaProvider(msgMgr);

    // connect to broker
    if (provider->initialize() != SoaErrorCode::NO_ERROR) {
        std::cout << "failed to initialize provider\n";
        return -4;
    }

    myprovider::MyOnDemandProvider::SharedPtr onDemandProvider;

    // instantiate and register my on-demand provider
    onDemandProvider = std::make_shared<myprovider::MyOnDemandProvider>(msgMgr, provider, SERVICE_ENDPOINT);
    onDemandProvider->registerOnDemandDataServiceAsync(1000);

        // start thread to wait for requests, handle requests, broadcast the data
    onDemandProvider->startThread();
    // wait to stop
    onDemandProvider->joinThread();
}
```

On Demand Consumer

Basic steps

Expanding on the interactions described in the introduction, these are the basic steps in building an on-demand consumer

- Implement a class derived from SoaOnDemandConsumer - refer to the requirements section for related information
- Implement all the requirements to instantiate and initialize a SoaConsumer object - refer to the programmer's guide for SoaConsumer
- In the application thread, instantiate your derived SoaOnDemandConsumer with the SoaConsumer instance and the on-demand provider's endpoint as arguments
- Request a data broadcast subscription by sending a request message to the provider
- Wait for data messages
- Handle receiving subscribed data
- Cancel the subscription
- After a subscription has been terminated (via cancellation or loss of provider), the on-demand consumer may request a new subscription from the same provider

Example

On Demand Consumer

```
// my_on_demand_consumer.hpp
// WARNING! This code has been exerpted and modified from an existing larger source file.
// This text has not been compiled
// Some code had been intentionally omitted for brevity


#include <soa_client_endpoint.hpp>
#include <soa_error_code.hpp>
#include <soa_on_demand_consumer.hpp>

namespace framework = fnv::soa::framework;

using SoaClientEndpoint = framework::SoaClientEndpoint;
using SoaMessageManager = framework::SoaMessageManager;
using SoaConsumer = framework::SoaConsumer;
```

```
using SoaMessage = framework::SoaMessage;
using SoaErrorCode = framework::SoaErrorCode;
using SoaConnectionOptions = framework::SoaConnectionOptions;
using SoaOnDemandConsumer = framework::SoaOnDemandConsumer;
using SoaErrorCodeUtil = framework::SoaErrorCodeUtil;
using SoaServiceDirectoryManager = framework::SoaServiceDirectoryManager;
using SoaLoggerControl = framework::SoaLoggerControl;


namespace myconsumer {


const std::string SERVICE_TOPIC = std::string(SoaClientEndpoint::createRequestEndpointName("ONDEMAND_FT"));
const std::string CONSUMER_SDM_TOPIC = std::string(SoaClientEndpoint::createResponseEndpointName
("UNIQUE_SDM_ENDPOINT"));

class MyOnDemandConsumer : public SoaOnDemandConsumer {
    public:
        typedef std::shared_ptr<DerivedOnDemandConsumer> SharedPtr;

        MyOnDemandConsumer(SoaMessageManager::SharedPtr msgMgr, SoaConsumer::SharedPtr consumer,
SoaClientEndpoint::SharedPtr serviceEndpoint) :
            SoaOnDemandConsumer(consumer, serviceEndpoint),
            mp_msgMan(msgMgr),
            m_execThread((pthread_t)-1)
        {
            // nothing else to do
        }
        virtual ~MyOnDemandConsumer() = default;
        virtual void onDataReceived(SoaErrorCode error, SoaMessage::SharedPtr message);

        int start();
        int join();
    private:
        SoaMessageManager::SharedPtr mp_msgMan;
        pthread_t m_execThread;

        static void * run(void * context);
        void execute();
};


} // namespace myconsumer




// ------------------------------------------------------------------------------------------
// my_on_demand_consumer.cpp


#include "my_on_demand_consumer.hpp"

namespace myconsumer {

// handle the incoming data broadcasts
void MyOnDemandConsumer::onDataReceived(SoaErrorCode error, SoaMessage::SharedPtr message) {
    if (error == SoaErrorCode::NO_ERROR) {
        std::cout << ("Consumer got data: " + message->getRawPayload() + "\n");
    } else {
        std::cout << ("Consumer got error code: " + SoaErrorCodeUtil::getString(error) + "\n");
    }
}

// request the subscription
void MyOnDemandConsumer::execute() {
    std::cout << "Consumer starting\n";

    SoaErrorCode error = SoaErrorCode::NO_ERROR;
    // subscribe to data
```

```
    std::cout << ("Consumer: subscribe to data\n");
    SoaMessage::SharedPtr request = SoaMessage::createRequestMessage(mp_providerEndpoint, "subscribe", "v1",
"t1");
    request1->setPayload("subscription_request_for_data_X");
    framework::SoaActionResult<SoaMessage>::SharedPtr initial = requestDataBroadcast(request, 2000);

    error = initial->getError();
}

// static
//  thread executable for running an on-demand onsumer
void * MyOnDemandConsumer::run(void * context) {
    MyOnDemandConsumer* consumer = static_cast<MyOnDemandConsumer*>(context);
    if (consumer) {
        consumer->execute();
    }
    return nullptr;
}


// create and start the consumer thread
int MyOnDemandConsumer::start() {
    int stat = pthread_create(&m_execThread, nullptr, &run, this);
    return stat;
}

// wait for the consumer to finish
int MyOnDemandConsumer::join() {
    int stat = pthread_join(m_execThread, nullptr);
    return stat;
}

} // namespace myconsumer

int main() {
    // set up dependencies of the on-demand consumer
    SoaConnectionOptions::SharedPtr connOpts = SoaConnectionOptions::createConnectionOptions();
    connOpts->setClientId(m_consumerClientId);
    connOpts->setDebugBrokerUrl(brokerURL);
    // create the message manager
    SoaMessageManager::SharedPtr msgMgr = SoaMessageManager::createMessageManagerMqtt(*connOpts);
    // create the endpoints
    SoaClientEndpoint::SharedPtr SERVICE_ENDPOINT = msgMgr.createClientEndpoint(SERVICE_TOPIC);
    SoaClientEndpoint::SharedPtr CONSUMER_SDM_ENDPOINT = msgMgr.createClientEndpoint(CONSUMER_SDM_TOPIC);
    SoaClientEndpoint::SharedPtr consumerEndpoint = msgMgr.createClientEndpoint(RESPONSE_EP);

    SoaServiceDirectoryManager::SharedPtr svcDirMgr = SoaServiceDirectoryManager::create(msgMgr,
CONSUMER_SDM_ENDPOINT);
    // create consumer
    SoaConsumer::SharedPtr consumer = SoaConsumer::createSoaConsumer(msgMgr, svcDirMgr, consumerEndpoint);

    // connect to the broker
    if (consumer->initialize() != SoaErrorCode::NO_ERROR) {
        std::cout << "failed to initialize consumer\n";
        return -4;
    }

    myconsumer::MyOnDemandConsumer::SharedPtr onDemandConsumer;

    // instantiate my on-demand consumer
    onDemandConsumer = std::make_shared<myconsumer::MyOnDemandConsumer>(msgMgr, consumer, SERVICE_ENDPOINT);

        // start thread to make a subscription request and wait for data
    onDemandConsumer->start();
    // wait to stop while receiving the on-demand data broadcast messages
    onDemandConsumer->join();
}
```

Code Samples

A set of demo applications can be accessed in github.

Three applications were created for SOA-R0.57

- Blocking consumer
  C++ Blocking On Demand Consumer source file (.cpp)
  C++ Blocking On Demand Consumer header file (.hpp)
- Blocking provider
  C++ Blocking On Demand Provider source file (.cpp)
  C++ Blocking On Demand Provider header file (.hpp)
- Non-blocking provider
  C++ Non-blocking On Demand Provider source file (.cpp)
  C++ Non-blocking On Demand Provider header file (.hpp)

The consumer app is executed against either of the provider apps. Only run one of the providers at a time. Any number of consumer apps may be executed concurrently.

In the functional tests, a single consumer app will create 3 TestConsumers, each of which creates 2 SoaOnDemandConsumers connected to MQTT via a SoaConsumer, and each TestConsumer starts after a defined delay. Common on-demand subscriptions with the provider will start and stop and overlap. The provider supports 2 different on demand requests. It generates 2 types of data in the form of periodically incrementing values. Broadcasts of these data types are done using different periods, and different from the modifying periods.

Using a command line parameter, the consumer can be configured to run 1, 2 or 3 TestConsumers in the hard coded pattern. The pattern is described in detail in the consumer header file.

API

Timeouts are in milliseconds

General note: All timeout arguments here are in milliseconds and must be greater than 0

class SoaOnDemandProvider

- You need to derive your own on-demand class from this abstract class to get access to these public member functions

Include the following header file ...

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

Method	What it does	Parameters	Comments
SoaOnDemandProvider(provider, serviceEndpoint)	protected constructor	SoaProvider::SharedPtr **provider** ▪ a pre-initialized SoaProvider SoaClientEndpoint::SharedPtr **serviceEndpoint** ▪ where consumers send their subscription requests to	You must call this in the initializer list of your derived constructor if the arguments are bad, the object will not be useable and calls to the member functions will return ERROR_MALFORMED_OBJECT
SoaErrorCode **registerOnDemandDataService**(timeout)	subscribes to your service endpoint	int32_t **timeout** ▪ same as for SoaProvider::registerService(...)	requests sent by consumers to your service endpoint trigger your onBroadcastRequestReceived() callback blocking method - cannot be called from the callback thread returns an error code

SoaErrorCode **setOnDemandBroadcast**(dataEndpoint, responseMessage, requestHandle, timeout)	sends a response to the consumer and initiates a consumer-tracker for the endpoint	SoaClientEndpoint::SharedPtr **dataEndpoint** ▪ dynamic endpoint for this data subscription - must not be nullptr SoaMessage::SharedPtr **responseMessage** ▪ application-specific message as per your defined interface - must not be nullptr SoaHandle **requestHandle** ▪ identifier received by callback onBroadcastRequestReceived(...) int32_t **timeout** ▪ same as for SoaProvider::remoteCallResponse(...)	You must call this or the async version to acknowledge or deny the subscription request. To confirm a request, provide all of the arguments, and the message must have no transaction error or a value of SoaMessage::TransactionError::NO_ERROR To deny a request, provide a response message with a transaction error value set other than NO_ERROR The response message is application specific - it may or may not include an initial or current data value The requestHandle is used to correlate the confirmation with a request blocking method - cannot be called from the callback thread returns an error code
void **registerOnDemandDataServiceAsync**(timeout)	subscribes to your service endpoint	int32_t **timeout** • same as for SoaProvider::registerServiceAsync(...)	non-blocking method - can be called from the callback thread The status of this call is received in onRegisterStatus(...)
void **setOnDemandBroadcastAsync**(dataEndpoint, responseMessage, requestHandle, timeout)	sends a response to the consumer and initiates a consumer-tracker for the endpoint	SoaClientEndpoint::SharedPtr **dataEndpoint** • dynamic endpoint for this data subscription - must not be nullptr SoaMessage::SharedPtr **responseMessage** • application-specific message as per your defined interface - must not be nullptr SoaHandle **requestHandle** • identifier received by callback onBroadcastRequestReceived(...) int32_t **timeout** • same as for SoaProvider::remoteCallResponse(...)	You must call this or the blocking version to acknowledge or deny the subscription request. To confirm a request, provide all of the arguments, and the message must have no transaction error or a value of SoaMessage::TransactionError::NO_ERROR To deny a request, provide a response message with a transaction error value set other than NO_ERROR The response message is application specific - it may or may not include an initial or current data value The requestHandle is used to correlate the confirmation with a request non-blocking method - can be called from the callback thread The status of this call is received in onSetStatus(...)
SoaErrorCode **publishMessage**(endpoint, message, timeout = *default*)	publish a message conditional on there being a subscriber to the endpoint	SoaClientEndpoint::SharedPtr **endpoint** ▪ the dynamic endpoint to send the data to SoaMessage::SharedPtr **message** ▪ application specific message containing the subscribed data int32_t **timeout** ▪ optional - same as for SoaProvider::publishMessage(...)	This method only sends a message if the endpoint is being tracked as a dynamic endpoint and there are one or more subscribers to it blocking method - cannot be called from the callback thread returns an error code

void **publishMessageAsync**(endpoint, message, timeout = *default*)	publish a message conditional on there being a subscriber to the endpoint	SoaClientEndpoint::SharedPtr **endpoint** ▪ the dynamic endpoint to send the data to SoaMessage::SharedPtr **message** ▪ application specific message containing the subscribed data int32_t **timeout** ▪ optional - same as for SoaProvider:: publishMessage(...)	This method only sends a message if the endpoint is being tracked as a dynamic endpoint and there are one or more subscribers to it non-blocking method - can be called from the callback thread The status of this call is received in onPublishStatus(...)
void **onBroadcastRequestReceived**(requestMessage, requestHandle)	notification of a request for a data broadcast subscription	SoaMessage::SharedPtr **requestMessage** ▪ application specific data broadcast request message SoaHandle **requestHandle** ▪ a SOA Framework value used to correlate a broadcast confirmation with this request	You must implement this member function This method executes on the callback thread Use the requestHandle when calling setOnDemandBroadcast[Async](..)
void **onBroadcastNoActiveListeners**(dataEndpoint)	event notification that there are no longer any subscribers for the endpoint	SoaClientEndpoint::SharedPtr **dataEndpoint** ▪ the endpoint for which no more subscribers exist	You must implement this member function This method executes on the callback thread
void **onBroadcastCancelled**(cancelMessage)	event notification that a subscriber has cancelled their subscription	SoaMessage::SharedPtr **cancelMessage** ▪ application specific cancellation message - this will never be nullptr	You must implement this member function This method executes on the callback thread This method will not be called if the consumer cancels with a null cancel message. In that case, the tracked subscriber count is descreased without notification, unless the count reaches zero The message dataEndpoint field contains the data endpoint for the subscription If the subscriber count related to this subscription reaches zero. onBroadcastNoActiveListeners(...) will be called after this method
void **onRegisterStatus**(error)	notification of result of async register method	SoaErrorCode **error** ▪ the status of the call to registerOnDemandDataServiceAsync(...)	This method executes on the callback thread Error values are the same as for the blocking version
void **onSetStatus**(error, context)	notification of result of async request ack method	SoaErrorCode **error** ▪ the status of the call to setOnDemandBroadcastAsync(...) std::shared_ptr <SoaSetOnDemandBroadcastContext> **context** ▪ the context includes the arguments passed to setOnDemandBroadcastAsync(...)	This method executes on the callback thread Error values are the same as for the blocking version
void **onPublishStatus**(error, context)	notification of result of async publish message method	SoaErrorCode **error** ▪ the status of the call to publishMessageAsync(...) std::shared_ptr <SoaPublishMessageContext> **context** ▪ the context includes the arguments passed to publishMessageAsync(...)	This method executes on the callback thread Error values are the same as for the blocking version

SoaClientEndpoint::ConstSharedPtr **getServiceEndpoint**()	getter for configured service endpoint		returns a const pointer to the service endpoint the on demand provider was instantiated with this can be used if needed for context

class SoaOnDemandConsumer

Include the following header file ...

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

Method	What it does	Parameters	Comments
SoaOnDemandConsumer(provider, serviceEndpoint)	protected constructor	SoaConsumer::SharedPtr **consumer** • a pre-initialized SoaConsumer SoaClientEndpoint::SharedPtr **serviceEndpoint** • where to send the subscription request	You must call this in the initializer list of your derived constructor if the arguments are bad, the object will not be useable and calls to the member functions will return ERROR_MALFORMED_OBJECT
SoaActionResult<SoaMessage>::SharedPtr **requestDataBroadcast**(requestMessage, timeout)	send a subscription request message to the configured on-demand provider endpoint	SoaMessage::SharedPtr **requestMessage** ▪ application specific data broadcast request message int32_t **timeout** ▪ same as for SoaConsumer::remoteCall(...)	blocking method - cannot be called from the callback thread The request message cannot be nullptr The consumerEndpoint field in the request message is reserved for this feature. Any endpoint data added by the user will be destroyed.
SoaErrorCode **cancelDataBroadcast**(cancelMessage, timeout)	send a cancel message to the configured on-demand provider endpoint	SoaMessage::SharedPtr cancelMessage ▪ application specific data broadcast request message int32_t **timeout** ▪ same as for SoaConsumer::remoteCall(...)	blocking method - cannot be called from the callback thread The cancel message can be nullptr, but the consumer application must only provide it as nullptr if allowed by provider's requirements
void **requestDataBroadcastAsync**(requestMessage, timeout)	send a subscription request message to the configured on-demand provider endpoint		not yet supported
void **cancelDataBroadcastAsync**(cancelMessage, timeout)	send a cancel message to the configured on-demand provider endpoint		not yet supported
void **onDataReceived**(error, message)	event notification of receiving subscribed data or an unrequested termination of the subscription	SoaErrorCode **error** ▪ indicates if there the message includes subscribed data, or the subscription has been terminated SoaMessage::SharedPtr **message** ▪ application specific message containing the subscribed data	This method executes on the callback thread If error is NO_ERROR, message contains subscribed data Otherwise, the error will be ERROR_PROVIDER_DISCONNECTED or ERROR_PROVIDER_CLOSED In both error cases, the subscription is terminated, and the on-demand consumer is available to try to again subscribe to data when it is known the provider is again accessible
void **onRequestStatus**(error)	notification of result of async broadcast request method		not yet supported

void **onCancelStatus**(error)	notification of result of async subscription cancel method		not yet supported
SoaClientEndpoint:: ConstSharedPtr **getServiceEndpoint**()	getter for configured service endpoint		returns a const pointer to the service endpoint the on demand provider was instantiated with this can be used if needed for context
SoaMessage:: ConstSharedPtr **getRequestMessage**()	get the request message		returns a const pointer to the request message sent to the provider this can be used if needed for context
SoaMessage:: ConstSharedPtr **getCancelMessage**()	get the cancel message		returns a const pointer to the cancel message sent to the provider this can be used if needed for context

request message consumerEndpoint field is reserved

The consumer endpoint field in the request message (ex. assigned SoaMessage::setConsumerEndpoint(...)) is reserved by this feature, and any data assigned to it by the consumer will be destroyed.

Header Files

These header files are needed for this feature in addition to those for fundamental SOA Framework classes. (Links to github)

soa/framework/soa_on_demand_consumer.hpp

soa/framework/soa_on_demand_provider.hpp

soa/framework/soa_set_on_demand_broadcast_context.hpp

Ungraceful Disconnect Support

An ungraceful disconnect of the partnering client in the data broadcast subscription is realized differently by consumers and providers.

Disconnected On Demand Provider

- If an on-demand provider object should be destroyed while connected, the destructor will generate a broadcast message received by all subscribers indicating the provider has closed.
    - The on-demand consumer's onDataReceived(...) callback will get called with an error value ERROR_PROVIDER_CLOSED
    - The subscription is terminated for the subscriber, which can then send a new subscription request to the same provider when appropriate
- If an on-demand provider suddenly terminates or otherwise gets disconnected from the MQTT broker, a last will and testament message will be sent to all connected subscribers.
    - The on-demand consumer's onDataReceived(...) callback will get called with an error value ERROR_PROVIDER_DISCONNECTED
    - The subscription is terminated for the subscriber, which can then send a new subscription request to the same provider when appropriate
    - As a connection can be shared by multiple SoaProviders, each potentially supporting multiple on-demand providers, all impacted on-demand consumers will be notified about their specific subscription

Disconnected On Demand Consumer

- If an on-demand consumer object should be destroyed while connected prior to the application calling the cancelDataBroadcast(...) method, the destructor will call cancelDataBroadcast()
    - This will appear to the on-demand provider as a cancellation, and not as an ungraceful disconnect
- If an on-demand consumer suddenly terminates or otherwise gets disconnected from the MQTT broker, a last will and testament message will be sent to its on-demand provider
    - The on-demand provider's subscriber database will be updated about the lost subscriber
    - The on-demand provider will NOT be notified of a cancelled subscription.
    - The on-demand provider will be notified if a subscriber count reaches zero (0) via the onBroadcastNoListeners(...) callback
    - As a connection can be shared by multiple SoaConsumers, each potentially supporting multiple on-demand consumers, all impacted on-demand providers will be notified about the respective subscriptions

Restrictions / Constraints

On Demand Provider

- "Regular" SoaConsumer objects without involving the SoaOnDemandConsumer class CANNOT request subscriptions from SoaOnDemandProviders

On Demand Consumer

- "Regular" SoaProvider objects without involving the SoaOnDemandProvider class CANNOT fulfill subscription request from SoaOnDemandConsumers

Not Supported

This feature does not support multiple on-demand consumer objects associated with a common SoaConsumer to subscribe to a common data endpoint provided by a provider.

As long as the subscribed endpoints are different, there is no problem.

This is due to the unique data values needed to track subscriptions in the provider's subscriber data base. This use case should not be required.