Participant Publication/Subscriber, Readers and Writer Instantiation

The application uses RTI's System Designer with the TMS idl files to standup all entities via XML App Create along with RTI's Dynamic Data feature.

Topics and Implementation Plans

The following is a list of 19 different topics and the way this design intends to address them.

TMS Keyed SampleId and DDS Resource usage - As defined in the TMS specification, SampleId is always a unique combination of DeviceId plus a monotonically incrementing Sequence Number. This means that each sample will take a unique memory resource (receiver cache) vs. placed as a sample in the same cache. This will require a DDS Dispose operation after each request is received and dispatched.

Reader Topic filters

Reader topics are keyed by SampleID, which is comprised of the DeviceID and a "SequenceNumber." The DeviceID is unique to each device and retrieved at runtime via your devices internal interface. It will be used to programmatically apply filters to the 5 reader topics defined below so that the device needs only concern itself with requests and responses that are directed to *this* device and not deal with all such traffic in the system.

With the current list of supported Topics, RequestResponse will have 4 Topics (1 Subscriber and 3 Publishers associated with the 1 Command Publish and 3 Command Subscriber topics listed in Appendix B.)

General Topic Design - There are five design patterns associated with the 19 identified topics resulting in 13 required threads. They are identified as follows:

Published Topics (14) and Patterns (5) (and threads (8 w/ 3 optional))**: (x, y, z notation shown)**

> **Publish Once (3, 1, 3 (optional))** - A topic that is published once prior to the main loop with Publish Last QoS Profile enabling late joiners. These include `DeviceAnnouncement, DevicePowerPortList, and Device Grounding`. Because these are published only once in the main loop, attaching a Writer Event monitoring thread is optional.
>
> **Periodic Publish(3, 1, 3)** - A topic, once enabled, is sent out at a regular interval. These include the three defined as Fixed Rate (`Heartbeat, DevicePowerMeasurementList, EngineStart.`
>
> **On Change Publish (4, 1, 4)** - A topic that is its status changes is published. These include `DevicePowerStatusList, LoadSharingStatus, SourceTransitionState, and FingerPrintNickname` topics.
>
> **In Response Publish(3, 1, 0)** - Request Response to a received Command Topic. These are published within the associated subscriber's thread context.
>
> **Command Request Publish(1, 1, 1)** - In this example, the `MicrogridMembershipRequest` will be sent at startup. Presumably, it would

normally be sent based on some operator control (i.e., powering up the device). Note this topic is actually defined as an Advisory topic, meaning the generator may join the grid without permission (1st paragraph page 208 of Draft TACTICAL MICROGRID 12 COMMUNICATIONS AND CONTROL document dated 1 February 2021.) While advisory, to be a good citizen, this design will respect the `MicrogridMembershipApproval` Response `MMR_COMPLETE`. `MMR_COMPLETE` will be used gate (enable and disable) periodic and On Change Publish topics.

Receive Topics (5) and Patterns[1] (1) (and 5 threads)**:**

All 5 Receive Topics fit within a single receive pattern. These include the three Commands Expected to be sent to this device: `LoadSharingRequest`, `SourceTransitionRequest`, and `FingerprintNicknameRequest`. Each of these requires a simple `RequestResponse` to be sent with the `DeviceID` and `SequenceNumber` of the request populated in the response along with the response code. The fourth received `MicrogridMembershipOutcome` sent as a result of an on-change from the controller. The fifth received topic is the `RequestResponse` associated with the `MicrogridMembershipRequest`. The sequenceNumber within the SampleID is used to match the response to a given request. This `RequestResponse` may be used to update a status panel on the device for convenience, but no action results or are necessary since in the current design `MicrogridMembershipRequest` continuing to be periodically sent until `MicroGridMembershipApproval` is `MMR_COMPLETE`.

TMS / DDS interaction with your internal system variables:

Generally, a TMS request, or local operation, results in an internal system variable change. When a TMS request, the variable change should be done in the TMS request command thread context. An equivalent TMS “shadow variable” should also be maintained, but not changed at the time of the internal system variable modification. The main loop should be continually updating non-“OnChange” TMS variables and checking for internal “OnChange” variables to be different than the TMS shadow variables. If this is detected, the variables should be made equivalent and an OnChange TMS message should be triggered.

Thread Interaction

The only thread interaction involves the five receive topics as follows:

- The three Command Requests - can indirectly cause an On Change event with the triggers an OnChange publish topic.
- `MicroGridMembershipApproval MMR_COMPLETE` (Or not) will gate the enable/ disable functions associated with all periodic or On change topics.

[1] This could be considered two design patterns, a straight Receiver and a Receiver that has a common code block to generate the response (In implementation, likely just pass in a ‘send response flag’ in the thread control block.)

- `MicrogridMembershipRequest` RequestResponse may optionally update a localized display panel on the device.

Lock / Atomic Operation Considerations

The current threading model is simple and has little interaction between threads. The design has no thread-safety / locks or synchronization between threads. There may be On Change Data that must be consistent among data members associated with a given topic before being presented to the controller (i.e., published). Care should be taken to ensure all of the On Change topic data is consistent before triggering the associated On Change topic publication. If this is an issue, data should be updated via multiple non-atomic internal operations within the main loop. After specific internal updates and variables are changed, any TMS related variables that require a change should trigger On Change topic publication. If needed, all of the variables should be copied out to the topic, and the topic On Change triggered.

Response and Request Sequence handling

As defined, requests and responses are required to have a keyed SampleId composed of the deviceId (Fingerprint) and a sequence number that monotonically increments with any request. The device unique Sequence number allows this device to correlate responses to requests (especially if there is more than one outstanding - which is always possible if a request or response is lost.

The request and response pattern is both sent and received. As a generator, we will both issue requests and receive them and need to ensure correlation with the response received or sent respectively.

Sent Responses: Correlating sending responses to specific requests is fairly easily handled. Here, each response will be issued in the context of the request. That is, upon receiving a request, the handler thread for that request will turn the response around within the handler thread as it receives the request (this handler may also do work within the device to carry out the request - which in turn may change some data that will trigger an On-Change topic to be issued.

Received Responses. Responses are generally used to acknowledge the receipt of a command and might be used to post the state of the response to an operator display. Response correlation cannot be done in the context of the sending request thread (since we have no way to know if the request got lost and may want to issue a new request). This design will keep the last ten request sequenceNumbers, with the enum of the topic (a paired ReqQEntry) in a circular queue to handle correlation. When a

response arrives, it will be modulo looked upon the circular queue. If it's not there, the response will be dropped and ignored. If it is in the circular queue, it can be correlated with the command and optionally used to update a display (or potentially take other programmatic operations like retry etc.)

Further this enqueueing of the requested sequence number and topic_enum will be done by the `RequestSequenceNumber::getNextSequnceNo` (enum TOPICS_E) function (used when writing the request topic sample in the main loop). To ensure only this function performs the enqueueing, the `ReqCmdQ` write function is private and the `RequestSequenceNumber` class has been declared a 'friend' object (giving it private access). This means that a sequence being written to the `ReqCmdQ` can not be a response being processed (read from the queue) since the `Request topic` sample for that sequence has not been written yet. By writing the `sequence_number` to the queue entry before the enum and reading the `sequence_number` last, if the sequence number matches the request_response we are processing we necessarily have a paired enum. The only race condition that could occur is that we get an enum that would have paired, but is overwritten by the main_loop request topic being sent, returning us a sequence that does not match our `request_response sequence.` In this case we discard the response as designed (for further information read all the comments around these two objects, `RequestSequenceNumber` and `ReqCmdQ`.)

Appendix A - Relevant Data Structures (Reference)

```
struct ReplyStatus {

    /** Indicator of success or failure. Intended to support automatic
handling. Values from 100 to 599 are as defined in the IETF RFCs.
Values outside that range are reserved for future versions of this
standard. Selected values are defined as constants. */
    unsigned long code;

    /** Short textual description of the status code intended for
human operators. */
    string<MAXLEN_reason> reason;
}; // struct ReplyStatus

/** The request has succeeded. */
const unsigned long REPLY_OK = 200;

/** Some value in the request was invalid. */
const unsigned long REPLY_BAD_REQUEST = 400;

const unsigned long REPLY_METHOD_NOT_ALLOWED = 405;

const unsigned long REPLY_CONFLICT = 409;

const unsigned long REPLY_GONE = 410;

const unsigned long REPLY_PRECONDITION_FAILED = 412;

const unsigned long REPLY_REQUEST_ENTITY_TOO_LARGE = 413;

const unsigned long REPLY_INTERNAL_SERVER_ERROR = 500;

const unsigned long REPLY_NOT_IMPLEMENTED = 501;

const unsigned long REPLY_SERVICE_UNAVAILABLE = 503;

/** Request is valid, and authorization is required before processing.
*/
const unsigned long REPLY_PENDING_AUTHORIZATION = 600;


struct SampleId {

    /** Identity of the device that sent this message. */
    Fingerprint deviceId;

    /** A number that is unique to each SampleId sent by this device.
It commonly starts at 0 and increments for each generated SampleId,
but such behavior is not required. For example, a pseudo-random
sequence may be used to avoid leaking information about how many
identities have been generated. */
```

```
    unsigned long long sequenceNumber;
}; // struct SampleId


struct RequestResponse {

    /** Copy of the corresponding requestId. */
    @key SampleId relatedRequestId; //@key

    /** Indication of success or failure. */
    ReplyStatus status;
}; // struct RequestResponse

enum MicrogridMembership {
    MM_UNINITIALIZED, // Uninitialized value.
    MM_JOIN, // Join or stay in the microgrid.
    MM_LEAVE // Leave or stay out of the microgrid.
}; // enum MicrogridMembership

struct MicrogridMembershipRequest {

    /** Identity of this request */
    @key SampleId requestId; //@key

    /** The device requesting a membership change. May be a
platformId, indicating that all deviceIds on the platform are
affected. */
    Fingerprint deviceId;

    /** The desired membership state. */
    MicrogridMembership membership;
}; // struct MicrogridMembershipRequest

enum MicrogridMembershipResult {
    MMR_UNINITIALIZED, // Uninitialized value.
    MMR_REPLACED, // New request received before preparation for this
request was complete. Device should discard this request and wait for
the new request to complete.
    MMR_COMPLETE, // Completed preparation for the requested action.
No significant issues were encountered.
    MMR_BLOCKED // Preparation could not be completed without a
significant negative impact such as shedding load. Operator should
resolve the issue before proceeding.
}; // enum MicrogridMembershipResult

struct MicrogridMembershipApproval {

    /** Identity of this request */
    @key SampleId requestId; //@key

    /** Copy of the corresponding
MicrogridMembershipRequest.requestId. */
    SampleId relatedRequestId;
```

```
    /** Device (or platform) that has been approved. */
    Fingerprint deviceId;

    /** Requested state. */
    MicrogridMembership membership;

    /** Indicate whether the microgrid is ready to complete this
request. */
    MicrogridMembershipResult result;

    /** Short, human-readable text description of a blocked request.
This should summarize to the operator what the issue is (e.g.
``blocked by interruptible load'', ``blocked by critical load'', or
``manual action required''). */
    string<MAXLEN_hint> hint;
}; // struct MicrogridMembershipApproval
```

Appendix B - List of Example Topics

(With Filter Param for Device)

For subscribed topics we only want topics with our ID in the right place

1. DeviceAnnouncement (once, durability) - In example only pub so Filter n/a
2. Heartbeat (periodic) - In example only pub Filter n/a
3. MicrogridMembershipRequest (sent to MSM from device) - Elicits RequestResponse from MSM In example only pub Filter n/a
4. MicrogridMembershipOutcome (from MSM to device) - sent “On Change” pattern from MSM - Filter for relatedRequestId.deviceId (this is the requesters ID and the message is directed back to us.) == our deviceId
5. SourceTransitionRequest (sent to device from MSM) - Elicits RequestResponse from Device
6. SourceTransitionState (From device to MSM) - sent “On Change” pattern to Device - pub only filter n/a
7. RequestResponse (Receive from MSM)
8. RequestResponse (Send to MSM)

Appendix C - “To Do” list

The following is a list of enhancements and optimizations in no particular order.

1. Put in real QoS (especially durability - currently we have to start the manager then the device)
2. Get rid of redundant #define tms_TOPIC* in tmsTestExampleApp.h - instead use a topic_array of pointers to the static const DDS_Char * as defined in lines 25-78
3. In the Applications, use an array each of DDSDynamicDataWriter and DDSDynamicReader pointers and iterate through the lookup of writers and readers, indexing the array by TOPIC_ENUM (each array of ~50 topics will be partially filled depending upon the readers and writers needed).
4. Do the same as #3 for needed dynamic data handles that need to be created.
5. Do the same for the thread ids and pthead_cancel(threadId)
6. I don’t think the same can be done in creating the threads (but 3-5 should clean up over 150 lines of code and will accommodate the handling of additional topics without new code) .
7. Add a callback to user handler if a enqueued response has not been handled and is about to be overwritten (requires adding a flag to the ReqCmdQ structure and setting it on the ResponseRequest Topic thread.) - Callback would be handled in the main loop context since that’s the only place allowed to issue a request by this design.
8. Add DDS filters for receiving any topics from the MSM (we don’t want all request or responses in the system, just the ones for our device).
9. Dispose or delete the key’d requests and responses as they arrive. By TMS definition, they are all uniquely key’d and will continually be added to the reader caches and saved (effectively creating a resource leak).
10. Investigate Objectifing Threading: Thread processing and handlers seems a bit too much c/ procedural vs. Threading Objects with generic handlers and inheritance to specific handlers.

https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/PROFILE_QosPolicy__DDS_Extension__.htm

```
const char *url_profiles[1];

    DDSDomainParticipantFactory *factory =
DDSDomainParticipantFactory::get_instance();
    DDS_DomainParticipantFactoryQos factoryQos;

    retcode = factory->get_qos(factoryQos);
    if (retcode != DDS_RETCODE_OK) {
        std::cerr << "ERROR: Can't get factor QoS profile " <<
std::endl << std::flush;
        goto tms_app_main_just_return;
    }
    url_profiles[1] = "file: ~/Github/tms/tmsApp/model/
tmsTestApp.xml";

    factoryQos.profile.url_profile.from_array(url_profiles, 1);

    factoryQos.profile.ignore_resource_profile =
DDS_BOOLEAN_TRUE;
    factory->set_qos(factoryQos);
    /*
    rti::core::QosProviderParams params =
        dds::core::QosProvider::Default()-
>default_provider_params();

    std::vector<std::string> url_profiles = {
        "file: ~/Github/tms/model/tmsTestApp.xml"
    };

    params.url_profile(url_profiles[1]);
    params.ignore_resource_profile(true);
```

```
    dds::core::QosProvider::Default()-
>default_provider_params(params);
    */

    participant = DDSTheParticipantFactory->
            create_participant_from_config(
                                "TMS_ParticipantLibrary1::TMS
Device Participant1");
```