Remote Device Management Service Low Level Design

Primary Owner ots-dse-hw-team (LDAP)

Last modified 2 seconds ago by traykeli.

Remote Device Management Service Low Level Design

Overview / Problem Statement

The Remote Device Management Service (RDMS) is a shared service responsible for managing connectivity to and from GSF HSE developed devices. It leverages AWS IoT Core and its rule based message routing capabilities to deliver messages sent from devices to their respective application service. Additionally, since MQTT is an asynchronous messaging protocol, RDMS provides a synchronous abstraction layer called the "Command Bridge" API which is used by application services to send messages down to devices.

Requirements

To provide these functionalities, integrating devices with RDMS comes with a few requirements:

1. Devices must publish their messages according to the following MQTT topic structure:
    - **[TOPIC PREFIX]** {product_name}/{product_location}/{product_id}/
    - The topic prefix consumes 3 of the 7 allowed levels. This leaves 4 topic levels for use by the application service.
2. Devices must publish status/health messages to a reserved heartbeat topic:
    - **[RESERVED TOPIC]** {product_name}/{product_location}/{product_id}/rdms/heartbeat
    - The publish interval may be specified by the device. 5 minutes is recommended.
3. Devices must subscribe to a reserved command topic:
    - **[RESERVED TOPIC]** {product_name}/{product_location}/{product_id}/rdms/cmd
    - As part of the command payload, a response callback topic can be supplied. If supplied, the device must publish the result of the command to that topic.

Process Flow Overview

Sending Messages

During device onboarding, an application service can be granted permission to receive messages from an MQTT topic filter associated with its device family (e.g a topic filter of rek/# or deadbolt/# can be configured to send messages to the Return Experience Kiosk Service or Deadbolt Backend Service, respectively). When a device publishes a message to that topic,

AWS IoT Core Rules will route the message to the application service's desired integration point (SNS is recommended, but any supported integration can be used).

Device Connectivity
When a device publishes a message to its reserved heartbeat topic, AWS IoT Rules routes the message to an AWS IoT Events detector model. This model continually evaluates / restarts the heartbeat timer and writes the device's heartbeat data to a DynamoDB table. This heartbeat information includes the last known connection time, device firmware version, etc. and can be subsequently queried by application services.

Sending Commands
When the Command Bridge Lambda receives a request from an application service, if a response was requested, the function generates a random topic name (e.g. a simple UUID) and subscribes to that ad hoc callback topic. It then publishes the request's message to the reserved command topic and waits for the device to respond on the callback topic. Once the response is received, it returns the result to the application service caller.

Out of Scope for Design Document (still in scope for feature)

- *Authentication of SCAR-HSE-Devices with IoT Core*. Since RDMS is in charge of remote device management, we are acting under the assumption that these devices are already properly authenticated. The Authentication process itself is not trivial and will require a design document of its own, it is relevant since a large portion of RDMS will be dealing with the underlying services (IoT Core) which will establish that authentication.
- *Access and Storage of Device Telemetry data*. While the high level design scopes out the creation and storage of telemetry data, low level design details are not fully scoped out in this document.
- *CA cycling and hardware/firmware image updates*. This could be an extension to the base design of RDMS, but low-level details are not fully scoped out in this document.

Assumptions

- Each HSE Device is already properly provisioned and authenticated with IoT Core (another design/spike on this later)

Key Concepts

- IoT Rules -- Enables HSE IoT devices to interact with AWS services
- IoT Thing -- An AWS service used to manage IoT Devices
- Pub-Sub Messaging -- Asynchronous communication between services and/or devices

Proposed Solution

The purposed solution is to leverage a Java Coral Lambda Service in combination with IoT Core services to manage the creation of and communication to SCAR-HSE-Devices. The Command Bridge Lambda will do the majority of the heavy lifting by leveraging IoT Rules, IoT Things, and IoT Events which will manipulate the asynchronous nature of pub/sub communication protocols and make it appear to clients that these calls are synchronous/normal HTTPS request. This design also relies on a SCAR-HSE-ProtoBuffService Lambda which will handle the encoding/decoding of messages that are too large to be transferred over the air (OTA) for any given message constraints.

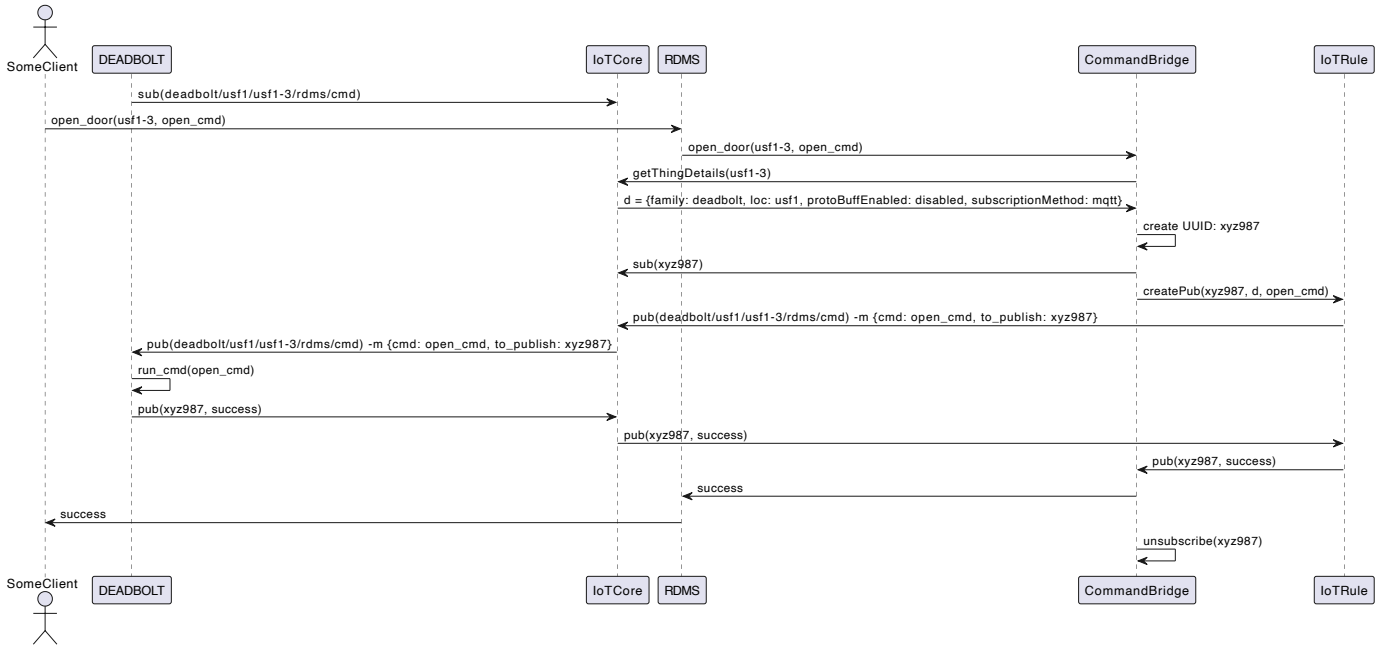
Key Acronyms

- *SCAR-HSE-Device (**SHED**)*: An IoT device built in house by our team that will communicate with this service (e.g. Deadbolt Locker, Rek Kiosk, etc). Assume that each SHED is properly authenticated with IoT Core and is associated with an IoT Thing Type.
- *Remote Device Management Service (**RDMS**)*: Main services that interacts directly with SHEDs.
- *Command Bridge Lambda (**CML**):* Lambda which handles the transporting/translating of messaging between SHEDs and clients

Key Flows

DEADBOLT deadbolt/+/+/rdms/cmd Flow

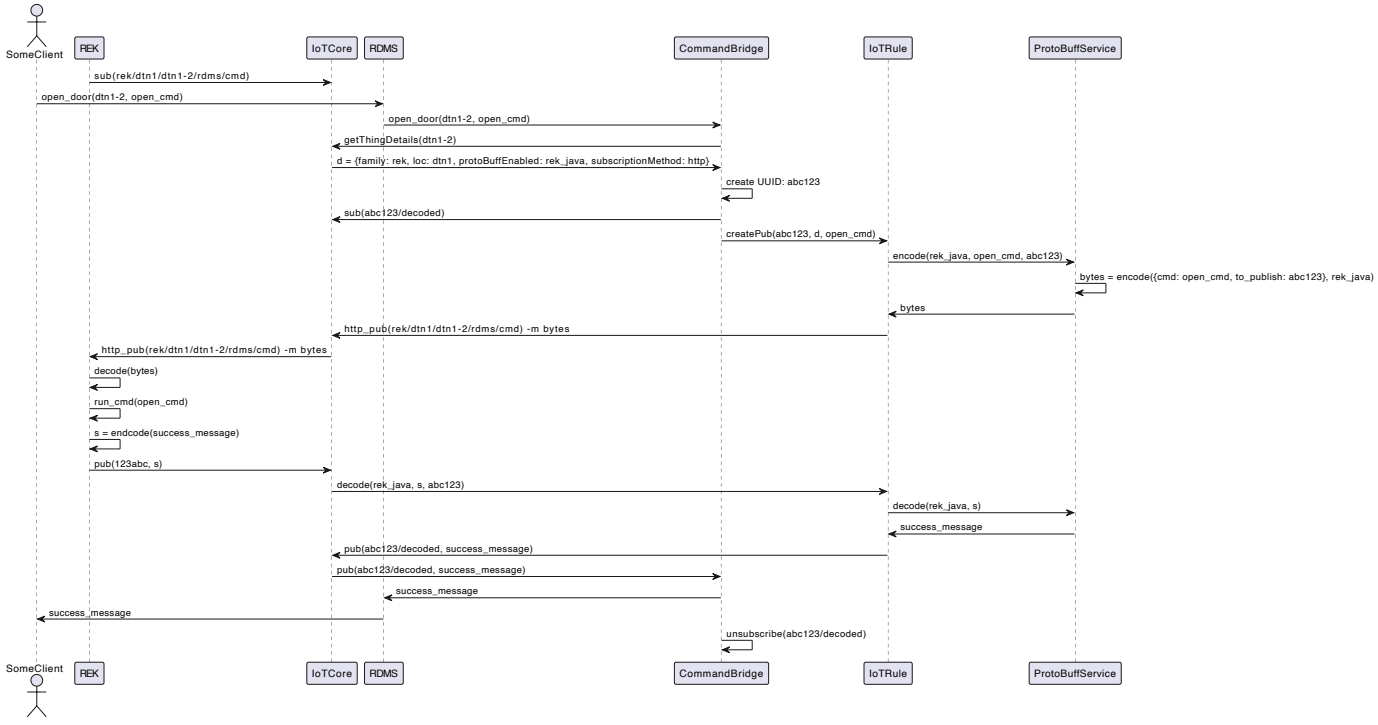
- Assuming that Android Application will act as the gatekeeper between IoT and PCB connection (for the scope of this document)
- Uses MQTT for data transport method
- Does not have message size constraints



edit

REK rek/+/+/rdms/cmd Flow

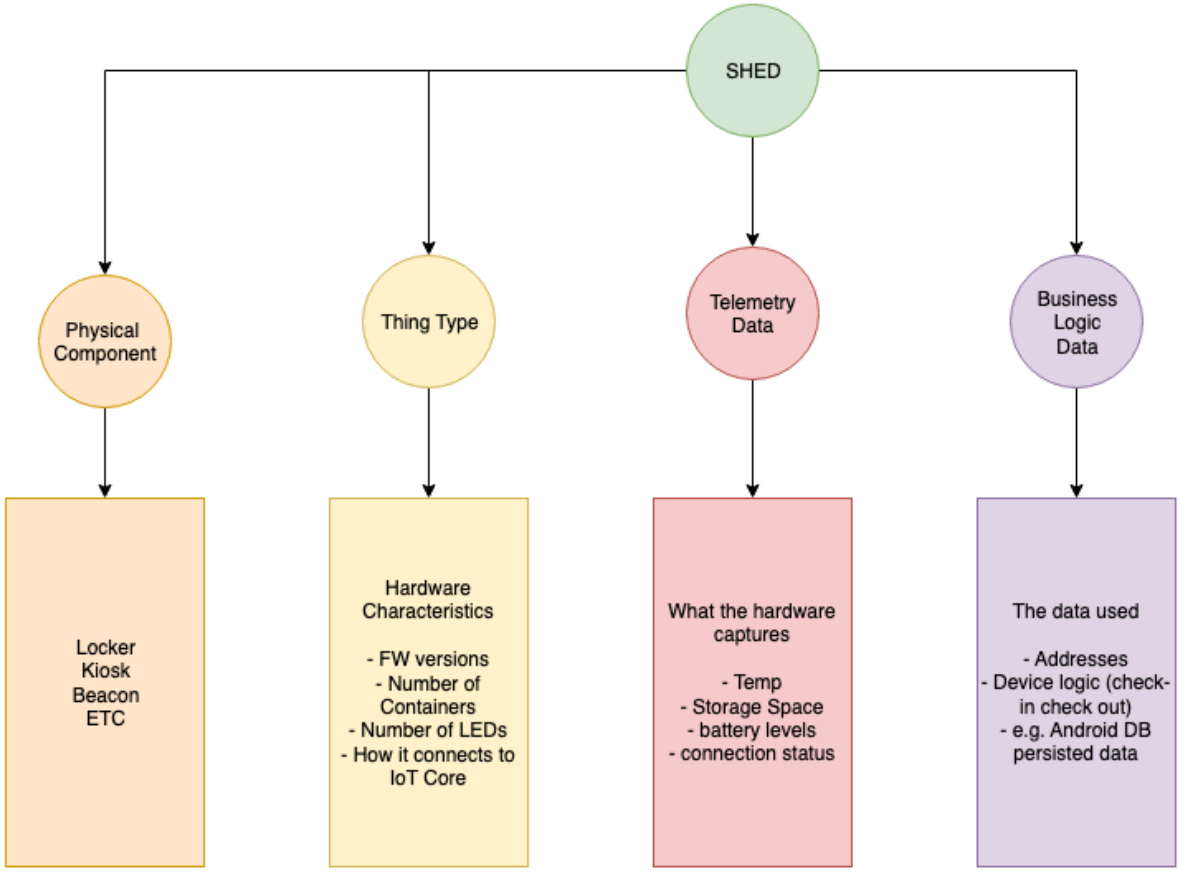
- Assume that REK SHED in diagram has encoding/decoding built into the application that matches what is defined with IoT Thing type
- Since LoRa is the communication method with this SHED, it is limited on pub/sub capabilities (it can publish, but not subscribe). RDMS spoofs the subscription by sending data to the SHED via the IoT Wireless API. This data sent will act as a subscription command sending all the necessary information to perform commands (i.e. open a door)
- There is also message size constraints so this diagram introduces the use of protocol buffers to reduce the size of the message being passed OTA.



edit

Data Model

SHED Visual Aid



IoT Registry Thing Types

AWS IoT provides thing repositories used for managing IoT devices. RDMS will leverage this model in creating different IoT device things types per device family. This enables different decision to be made by the **Command Bridge Lambda** by handling IoT messages different based off each "thing type".

Thing Type	Description	Properties
Deadbolt Locker	The available attribute for all Deadbolt lockers	**Attributes:** View below (searchableAttributes) **Name Value Value:** deadbolt
Rek Kiosk	The available attributes for all REK Kiosks	**Attributes:** View below (searchableAttributes) **Thing Name Value:** rek

```
const DeadboltThing = {
  thingTypeName: "deadbolt",
  thingTypeProperties: {
    searchableAttributes: [
      "screenSerialNumber",
```

```
const RekThing = {
  thingTypeName: "rek",
  thingTypeProperties: {
    searchableAttributes: [
      "mainControlModuleFrimwareVersion",
```

```
      "pcbBoardCount",
      "deviceStorageType",
      "pcbBoardFirmwareVersion",

     "controlBoardFirmwareVersion",
      "protobuffEnabled",
      "dataListenMethod",
    ],
    thingTypeDescription: "Smart
device storage solution",
  },
  thingTypeMetadata: {
    deprecated: false,
    creationDate: 1468423800950,
  },
};
```

```
      "mainControlModuleHardwareVersion",
      "connectedExpansionBoards",

    "connectedExpansionBoardsHardwareVersion",
      "bleMode",
      "temperature",
      "batteryState",
      "binLedCount",
      "binLedTypes",
      "binLockCount",
      "binLockStatus",
      "binCapacity",
      "barcodeScannerStatus",
      "protobuffEnabled",
      "dataListenMethod",
    ],
    thingTypeDescription: "Device which handles
package returns",
  },
  thingTypeMetadata: {
    deprecated: false,
    creationDate: 1468423800950,
  },
};
```

*note that the following attributes are shared across all **SHEDs**

- ***dataListenMethod:*** Values can be 'http' or 'mqtt'. This value determines how **CBL** should publish a message to the given **SHED.** There are two main methods of data transport for our current **SHED** designs, MQTT and HTTP. As we develop more **SHEDs** and expand the business, other methods of data transport may be included and extended, but at the time of writing this document only HTTP and MQTT will be supported. (View payload object below for an example).
- ***protobuffEnabled***: Let's the system know whether a protocol buffer file should be used (i.e. if the payload needs to be compressed before sending it over bluetooth due to size constraints). The value of the attribute value will be structured: "{thingTypeName}_{language}" which will be used in a **SCAR-HSE-ProtoBuffService** Lambda which handles the serialization/deserialization of incoming and outgoing messaging between AWS services and IoT devices. If the message does not need to be serialized/deserialized then the default value of "disabled" will be supplied. (An example will be provided in **Creating Rek Things**).

Creating Deadbolt Things

`$ aws iot create-thing --thing-name "deadbolt-usf2-1" --thing-type-name "deadbolt" --attribute-payload "${deadboltPayload}"`

Deadbolt Payload example:

```
const deadboltPayload = {
  attributes: {
    screenSerialNumber: "B1A30045C",
    pcbBoardCount: "8",
    deviceStorageType: "TC-56",
    pcbBoardFirmwareVersion: "1.0.2",
    controlBoardFirmwareVersion: "0.0.9",
```

```
    protobuffEnabled: "disabled",
    dataListenMethod: "mqtt",
  },
};
```

Creating Rek Things

`$ aws iot create-thing --thing-name "rek-dtn6-1" --thing-type-name "rek" --attribute-payload "${rekPayload}"`

Rek Payload example:

```
const rekPayload = {
  attributes: {
    mainControlModuleFrimwareVersion: "1.0.2",
    mainControlModuleHardwareVersion: "1.3.4",
    connectedExpansionBoards: "boardA, boardB, boardC", // CSV of different boards
    connectedExpansionBoardsHardwareVersion:
      "boardA=1.1.1, boardB=1.3.4, boardC=2.4.8", // CSV of boards with associated
hardware version
    bleMode: "enabled", // enabled | disabled
    temperature: "86F", // in fahrenheit
    batteryState: "4", // UNKNOWN = 0; CRITICAL = 1; LOW = 2; MEDIUM = 3; HIGH = 4;
    binLedCount: "25", // amount of LED lights
    binLedTypes: "type1, type2, type3", // CSV of different LED types
    binLockCount: "4", // amount of bin locks, can be variable
    binLockStatus: "bin1=closed, bin2=open, bin3=closed, bin4=closed", // ascending
left to right order
    binCapacity: "86", // percent full based on range to nearest package, i.e. closer
package => higher number
    barcodeScannerStatus: "enabled", // enabled | disabled
    protobuffEnabled: "rek_java", // name of protobuff family
    dataListenMethod: "http",
  },
};
```

Updating Things

```
aws iot update-thing --thing-name "ExistingThing"
                     --thing-type-name "SomeHseThing" --attribute-payload
"${SomeHseThingPayload}"
```

Entity Types

Entity Type	Description	Properties
Device Thing Heartbeat Data	Standard device health metrics unique to each Thing Type. Returned on a 5 minute cadence to the {product_name}/{product_location}/{product_id}/rdms/heartbeat topic	**deviceData:** JSON blob specific to Thing (view below)
Device Thing	Irregular System errors affecting health metrics that can be thrown at anytime (client side event) and reported to the	**errorData:** JSON blob specific to Thing

Exception Data	{product_name}/{product_location}/{product_id}/rdms/heartbeat topic	(view below)

SHED Interfaces for Heartbeat Entities

Each SHED should report to the heartbeat topic on a five minute cadence with the following interfaces implemented as messages (JSON equivalent).

```
interface ShedHeartbeatTelemetryData {
    deviceId: string;
    deviceData: any;
    timeOfLastConnection: string | number;
    error: ShedHeartbeatTelemetryData | null;
}

interface ShedHeartbeatTelemetryError {
    message: string,
    errorData: any
}
```

Examples of Deadbolt and Rek Heartbeat Telemetry Data

```
const DeadboltHeartbeatTelemetryData = {
  deviceId: "deadbolt-dtn1-6",
  deviceData: {
    deviceCount: 26,
    shelves: [
        {
            mac: 'B213000A',
            status: 'online',
            door: 'open'
        },
        {
            mac: 'B213000B',
            status: '(offline)',
            door: 'closed'
        }
    ],
    mqttConnectionToPcb: 'connected' //
connected | disconnected
  },
  timeOfLastConnection: 1677280194,
  error: null,
};
```

```
const RekHeartbeatTelemetryData = {
  deviceId: "rek-dtn1-6", //
matches thing device ID
  deviceData: {
    bins: [
        {
            mac: 'J112000F5',
            temperature: 85,
            binCapacity: 56
        },
        {
            mac: 'J112000W5',
            temperature: 85,
            binCapacity: 12
        },
        {
            mac: 'J112000F7',
            temperature: 85,
            binCapacity: 96
        },
    ],
    batteryLevel: 1
  },
  timeOfLastConnection: 1677280194,
  error: null,
};
```

Leveraging IoT Detector Model

The DynamoDB RDMS table will be populated from IoT Event actions based off of the data returned from the heartbeat topic. This is accomplished through the IoT Detector Model. In order for connection to be established each SHED must implement JSON input files mapping to the device specific **deviceData** attribute in the **ShedHeartbeatTelemetryData** interface, as well as JSON input files for **errorData** in the **ShedHeartbeatTelemetryError** interface.

Each SHED must also define a detector model JSON file which uses the input JSON files defined earlier. This model consists of states that an IoT device can be in. Each state has the substates *onEnter, onInput,* and *onExit.* Each substate can have events which fire off actions, it is through this model that the SHED devices will populate the RDMS table. Here is a great step by step guid that goes into the weeds of creating a detector model: here

Deadbolt Heartbeat Detector Model

*note: It is implied that at each heartbeat (good or bad), there is an attached IoT Rule mapping to a dynamoDB event which updates the database, implementation of this can either be through an SNS topic OR through the internal IoT Core library Event. A spike should be performed to determine which method makes better sense.

States: [Normal, Error, MqttDisconnect]

Initial State: Normal

- Normal
  - State conditions
    - Goes to Error state when $Input.deviceData.error reports anything other than null
    - Goes to MqttDisconnect state when $Input.deviceData.mqttConnectionToPcb reports disconnected
  - Events
    - publishes IoT Rule (SNS topic/SIM ticket) when a given shelf reports offline (e.g. $Input.deviceData.shelves.filter(shelf => shelf.status == '(offline)')) for three consecutive heartbeats
    - publishes IoT Rule (SNS topic/SIM ticket) when $Input.deviceData.error reports anything other than null
    - publishes IoT Rule (SNS topic/SIM ticket) when $Input.deviceData.mqttConnectionToPcb reports disconnected for three consecutive heartbeats
- Error
  - State conditions
    - Goes to Normal state when $Input.deviceData.error reports null && at least one shelf from $Input.deviceData.shelves reports online
  - Events
    - publishes IoT Rule (SNS topic) when device returns to Normal
- MqttDisconnect
  - State conditions
    - Goes to Normal state when $Input.deviceData.mqttConnectionToPcb reports connected for three consecutive heartbeats
    - Goes to Error state when $Input.deviceData.error reports anything other than null
  - Events
    - publish IoT Rule (SNS topic) when returning to normal

Rek Heartbeat Detector Model

States: [Normal, Error, FullCapacity, LowBattery]

Initial State: Normal

- Normal
  - State conditions
    - Goes to Error state when $Input.deviceData.error reports anything other than null
    - Goes to FullCapacity state when $Input.deviceData.bins all report binCapacity above 95
    - Goes to LowBattery state when $Input.deviceData.batteryLevel reports level 1
  - Events
    - publishes IoT Rule (SNS topic/SIM ticket) when a given bin reports a temp above a certain threshold (e.g. $Input.deviceData.bins.filter(bin => bin.temperature > 85)) for three consecutive heartbeats
    - publishes IoT Rule (SNS topic/SIM ticket) when $Input.deviceData.error reports anything other than null
    - publishes IoT Rule (SNS topic/SIM ticket) when navigating to FullCapacity

- Error
  - State conditions
    - Goes to Normal state when $Input.deviceData.error reports null && $Input.deviceData.batteryLevel reports > 1
  - Events
    - publishes IoT Rule (SNS topic) when device returns to Normal
- FullCapacity
  - State conditions
    - Goes to Normal state when $Input.deviceData.bins reports at least one bin containing less than 70 capacity for three consecutive heartbeats
    - Goes to Error state when $Input.deviceData.error reports anything other than null
  - Events
    - publish IoT Rule (SNS topic) when returning to normal
- LowBattery
  - State conditions
    - Goes to Normal state when $Input.deviceData.batteryLevel reports > 1
    - Goes to Error state when $Input.deviceData.error reports anything other than null
  - Events
    - publish IoT Rule (SNS topic) when returning to normal

Data In Transit

Protocol Buffers

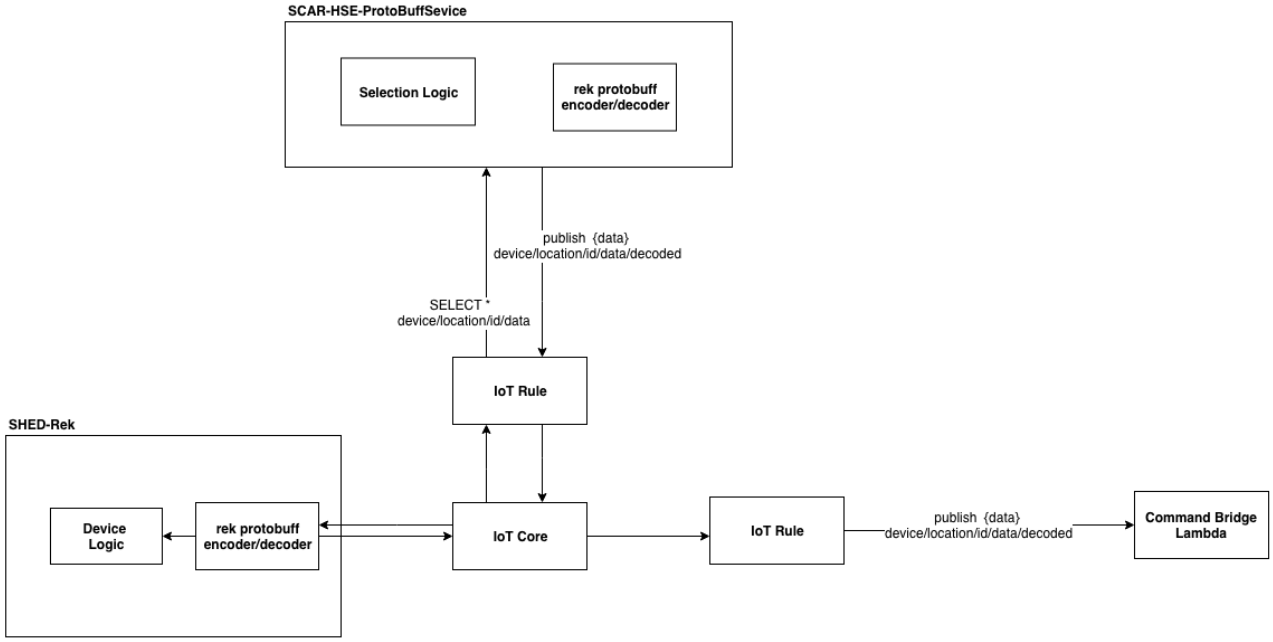
Each SHED will specify whether or not it requires protocol buffers for message transfer. Protocol Buffers compresses message sizes and allows for smaller payload communication (which may need to exist between any given SHED and RDMS). This is done upon SHED creation by supplying a value to protobuffEnabled. The value is a simple string in the following manner "{thing_type}_{language}". For Rek this would be "rek_java".

This entails two things:

1. That both the client and the **SCAR-HSE-ProtoBuffService** have a compiled version of the Rek .proto files in the specified language.
2. That subscribers to a given SHED topic with protobuffEnabled will append "/decoded" which will be published by the **SCAR-HSE-ProtoBuffService** implying that the payload is ready for business logic use

If a SHED does not need to compress messages, the default string of "disable" will be supplied upon SHED creation.

Diagram of protobuff data exchange:



Rek .proto files proposals for Heartbeat data:

```
message HealthMessage {
    optional sint32 environmental_sensor = 1;
    optional SysErrorCode sys_err = 2;
    optional BatteryLevel battery_level = 3;
    optional uint32 capacity = 4;
    optional bool motion = 5;
    optional uint32 weight = 6;
    optional uint32 lock_mask = 7;
    optional uint32 io_status = 8;
    optional uint32 timestamp = 9;

    enum SysErrorCode {
        NONE = 0;
        POWER_FAILURE = 1;
        SENSOR_FAILURE = 2;
    }

    enum BatteryLevel {
        UNKNOWN = 0;
        CRITICAL = 1;
        LOW = 2;
        MEDIUM = 3;
        HIGH = 4;
    }
}
```

Data Storage

DynamoDB is the proposed storage solution for device telemetry data and device Heartbeat. Note that Thing Type will also be stored/managed in IoT Core Registry. Being that each Device Thing will constantly be publishing Telemetry data via IoT

Core RDMS will be write-heavy.

Supported Access Patterns

Access Pattern	Query Type	Required Parameters
Get heartbeat by ID	1 base table query	device ID
Get heartbeat by Thing Type	1 base table query	Thing Type
Get exception by ID	1 base table query	device ID
Get exception by Thing Type	1 base table query	Thing type
Get heartbeat by location ID	1 GSI	location ID
Get exception by location ID	1 GSI	location ID

Base Table

Primary key		Attributes		
Partition key: pk	Sort key: sk			
DEADBOLT#USF2-1	EXCEPTION#1677013894	location	ttl	data
		LOCATION#USF2	3354027950	{ sysError: 1, message: "Application Crashed" }
	EXCEPTION#1677013994	location	ttl	data
		LOCATION#USF2	3354029950	{ sysError: 1, message: "MQTT Disconnected" }
	HEARTBEAT#1677014079	location	ttl	data
		LOCATION#USF2	3354028182	{ sysError: 0, pcbBoardConnectionStatus: [{mac1: "online"}, {mac2: "online"}, {mac3: "(offline)"}, {mac4: "online"}, {mac5: "online"}, {mac6: "online"}, {mac7: "online"}, {mac8: "online"},], controlBoardMqttConnection: "connected" }
REK#DTN1-5	EXCEPTION#1677073894	location	ttl	data
		LOCATION#DTN1	3354048182	{ environmentalSensor: 8, sysErr: 1, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }
	HEARTBEAT#1677084079	location	ttl	data
		LOCATION#DTN1	3354028182	{ environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }
	HEARTBEAT#1677088079	location	ttl	data
		LOCATION#DTN1	3354048182	{ environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }

GSI 1

Primary key		Attributes		
Partition key: gsi1pk	Sort key: sk			
LOCATION#DTN1	EXCEPTION#1677073894	pk	data	ttl
		REK#DTN1-5	{ environmentalSensor: 8, sysErr: 1, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }	3354048182
	HEARTBEAT#1677084079	pk	data	ttl
		REK#DTN1-5	{ environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }	3354048182
	HEARTBEAT#1677088079	pk	data	ttl
		REK#DTN1-5	{ environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false, lock_mask: 0, io_status: 0, timestamp: 1674248345 }	3354028182

Data Retention

All Heartbeat and Exception data is retained for 2 years (configured via TTL)

APIs

SendCommandToDevice (response requested)

Request

```
{
  "deviceId":"deadbolt-usf2-1",
  "cmd": "mosquitto_pub -h controller -p 8883 -t \"shelf-v1/<MAC>/shelf-door-release\" -
m \"release\"",
  "requestResponse": true
}
```

Response

```
{
  "topic": "1a83fc73-fac5-48ea-b86f-e2fd79fac0b6",
  "message": "successfully executed command for usf2-1",
}
```

*note that for SendCommandToDevice the cmd object can be system level commands or higher level (i.e. for deadbolt, communication to Android could be at a software level to get back the device count stored in RoomDB). While this is out of the scope for this document, the RDMS system is flexible enough to pass commands if both clients agree on a contract for the data being requested and the command being delivered. Future refactors will include the creation of a DeadboltCommandClient/RekCommandClient that could be used on both ends (caller/device) so this contract is fulfilled

SendCommandToDevice (no response requested)

Request

```
{
  "deviceId":"deadbolt-usf2-1",
  "cmd": "mosquitto_pub -h controller -p 8883 -t \"shelf-v1/<MAC>/shelf-door-release\" -
m \"release\"",
  "requestResponse": false
}
```

Response

```
{
  "message": "successfully delivered command",
}
```

GetDeviceHeartbeatById

Request

```
{
  "deviceId":"deadbolt-usf2-1"
}
```

Response

```
{
  data: {
    sysError: 0,
    pcbBoardConnectionStatus: [
      { mac1: "online" },
      { mac2: "online" },
      { mac3: "(offline)" },
      { mac4: "online" },
      { mac5: "online" },
```

```
      { mac6: "online" },
      { mac7: "online" },
      { mac8: "online" },
    ],
    controlBoardMqttConnection: "connected",
    timestamp: 1674248345,
  }
}
```

GetDeviceExceptionById

Request

```
{
  "deviceId":"deadbolt-usf2-1",
}
```

Response

```
{
  data: { sysError: 1, message: "MQTT Disconnected" }
}
```

GetDeviceHeartbeatByLocation

Request

```
{
  "location":"dtn1",
}
```

Response

```
{
  data: [
    { environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false,
lock_mask: 0, io_status: 0, timestamp: 1674248775, deviceId: "dtn1-5" },
    { environmentalSensor: 8, sysErr: 0, batteryLevel: 4, capacity: 0, motion: false,
lock_mask: 0, io_status: 0, timestamp: 1674248345, deviceId: "dtn1-4" }
  ]
}
```

GetDeviceExceptionByLocation

Request

```
{
  "location":"dtn1",
}
```

Response

```
{
  data: [
    { environmentalSensor: 8, sysErr: 1, batteryLevel: 1, capacity: 0, motion: false,
lock_mask: 0, io_status: 0, timestamp: 1674248345, deviceId: "dtn1-5" },
```

```
    { environmentalSensor: 8, sysErr: 2, batteryLevel: 4, capacity: 0, motion: false,
lock_mask: 0, io_status: 0, timestamp: 1674242345, deviceId: "dtn1-2" }
  ]
}
```

Pros / Cons

Pros	Cons
• Great flexibility for sending any command via bridge lambda design • IoT Core ecosystem is rich and provides a plethora of tools catered to telemetry data, the assumption is these services should be more efficient than using other services to handle the data • Clear set of rules for expansion to new devices OOTB with IoT Core	• Heavy compute load for the Lambda which may impact costs • Lack of support for L2 constructs and CDK in IoT core (experimental, we'd be pioneers in the space) • No enforcement of data between clients. The clients (sender and receiver) must enforce their own data checks and contracts.

Alternate Solution 1

The alternate solution borrows heavily from the proposed solution by leveraging the same data model, storage, and API design. Where it differs is in how the compute components are structured. Rather than having a one large package that hosts the Coral service code, as well as the event-driven lambda code, the event driven pieces are split out into a separate package.

Pros / Cons

Pros	Cons
• High CDK support for L2 constructs surrounding SNS topics	• May not be as efficient as the prior solution • Design is not catered to leveraging IoT core to its full potential • Carries the rest of the other cons as well

Metrics & Monitoring

Detail what metrics we'll track and how we'll monitor system health.

- How will we track how much this feature is being used?
- How do we know our service is working?
- How will we know our service is failing?
- Alarms?

Detail what metrics the business will track and how we'll ensure they have the needed data to support measuring success.

- What are the business reporting metrics needed to ensure launch success and adoption? Do these require incremental tech asks to support?

Security

Detail the security requirements.

- What is the data classification for any new data points?
- Do we need to obfuscate or encrypt anything?
- How will we manage client access?
- Will infosec review be required?
- What is the worst case scenario if there is a data breach, DDoS attack, etc?

Spend

Detail the high-level cost comparisons between solutions using IMR.

- Very roughly, what do we expect monthly spend to be (refer to IMR)
- How do alternate solutions compare cost-wise?
- https://w.amazon.com/index.php/IMR
- https://w.amazon.com/bin/view/IMR/2021_IMR_Rate_Card/

Roll Out Plan

How will we roll this feature out?

- Are there any constraints or client breaking concerns?
- Does it need to go behind a Weblab?
- Are we rolling out the feature in phases?
- Are we rolling this feature out at different dates by region?
- How will we control feature gating?

Story Breakdown

Story	High level description
Create CommandBridgeLambda - RemoteDeviceManagementService	TBD
Create Deadbolt Heartbeat Detection Model - RemoteDeviceManagementCDK	TBD
Create REK Heartbeat Detection Model - RemoteDeviceManagementCDK	TBD
Create RDMS DDB Table - RemoteDeviceManagementCDK	TBD
Create cmd endpoint - RemoteDeviceManagementService	TBD
Create Deadbolt JITP template - TBD (ProvisioningWebsite?)	TBD
Create ProtocolBufferLib	TBD

FAQs

Exactly as the title sounds, outline frequently asked questions here. Each question should be it's own subsection with the content of that section providing the answer. See example below.

How should I format an FAQ entry?

Like this.

Open Questions

1. What are our data retention requirements? Can we implement DDB TTL after 2 years?

References

Bulleted list of links to useful information.

Review Notes

- Things relation tree for data model vs family and devices
- Look deeper into outbound protobuff lambda (redesign in diagram)
- Add middle ground to transport from SHED to IoT Core (in diagram)
- REK --> LoRa --> LNS (no managed by us) ---> IoT Core (via MQTT)
    - specify gateway which we manage for REK
- Make data flow more clear coming from device (looks like it is going straight into IoT Rule action)
- Explore the MQTT bridge option for direct comms between PCB board
    - performance benefits
    - FW connection to PCB boards to the internet
- CommandBridge subscribe/publishing on IoT core immediately instead of going through IoT Rule
- Protobuff config file storage / should be a service NOT just a Lambda
- PcbBoard versioning Thing type
- Double check the thing types searchable attribtues
- Subscription Method Thing type (how to not shoot ourselves in the foot i.e. Deadbolt HTTP)
- Add a "when to update" Thing model
- Separate Thing data from telemetry data
- Throughput of IoT core vs raw DynamoDB
- Generalize to only Error: e.g. Goes to Error state when $Input.deviceData.shelves all report (offline)
- RDMS Table Logic
- Generic Warning state (super flexible)
- Protocol buffer version introduced IN CASE of .proto file changes
- Support thing type query
- Add note to add to Andes
- Array of data for API
- API (by time range)
- API Model by business use case (i.e. is device online?)
-

Team Review [Date]

Notes

Bulleted list of general meeting notes.

- Did we all agree on anything?
- Were any concerns raised?

Action Items

Bulleted list of specific actions the team identified that needed to be addressed.

- Mark each action as complete as you close them.

Appendices

Any additional information to supplement the design. Each item should be entered as a sub heading under this section per the Appendix A example below.

Appendix A: [Information on Something]

Example appendix entry.

Tags: