In-Band Manageability Framework

Developer Guide

July 2021

Intel Confidential

Document Number: 626909-2.8

Contents

Tables

Revision History

Date	Revision	Description
July 2021	2.8	Updated TPM usage and Platform OS assumptions. Updated security information.
May 2020	2.1.1	EIS 2.2 release.

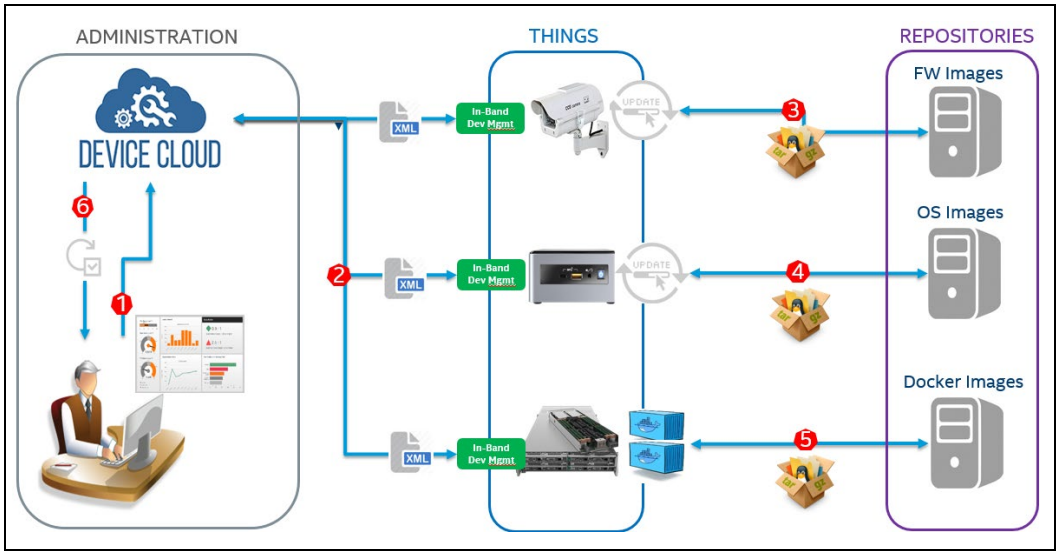
§

1.0 Introduction

In-Band Manageability Framework (Turtle Creek) is a software running on Edge IoT Device, which enables an administrator to perform critical Device Management operations over-the-air remotely from the cloud. It also facilitates publishing of telemetry and critical events and logs from the Edge IoT device to the cloud enabling the administrator to take corrective actions if, and when necessary. The framework is designed to be modular and flexible, ensuring scalability of the solution across preferred Cloud Service Providers (for example, Azure* IoT Central, Telit DeviceWISE, ThingBoard.io, and so on).

Some of the key advantages of Intel's In-band Manageability solutions are:

1. Out-of-box cloud support: Azure IoT Central, Telit DeviseWise, ThingsBoard.io.
2. Single interface to handle OS, FW and Application (Docker container) updates.
3. Scalable across Intel x86 (Intel Atom® and Intel® Core™) architectures SoCs and on Vision platforms from Intel.



This document provides detailed instructions on how to provision a device with **Azure IoT Central**.

The Device Management use-cases covered by the In-Band Manageability Framework are listed in the table below:

Use-cases	Notes
Update	• System (OS), Software-over-the-air (SOTA) • Firmware-over-the-air (FOTA) • Application-over-the-air (AOTA)
Telemetry	• System attributes • Events • Devices States • Usage data
Recovery	• Rollback post updates • System Reboot/Shutdown

Embedded within the In-Band Manageability Framework are features that ensure Security and Diagnostics aspects:

Features	Notes
Security	• ACL for trusted repositories • Mutual TLS authentication between services • TPM to store framework secrets
Diagnostics	• Pre and Post OTA update checks • Periodic system checks

1.1 Purpose

This Developer Guide provides the reader instructions on how to navigate and build Turtle Creek framework source code. It also provides information that Manageability solution developers will find useful, for example:

- Turtle Creek configuration file composition
- Enable logging
- Adding new Platform support for FW update capability
- Adding support to a new Cloud Backend and Communicating with Turtle Creek framework

1.2 Audience

This guide is intended for

- Manageability Solution developers to extend/modify Turtle Creek framework.

- System Integrators administrating devices running In-Band Manageability framework.

1.3 Terminology

Table 1. Terminology

Term	Description
AOTA	Application Over the Air (Docker)
BIOS	Basic Input Output System
Device	A device is any equipment that is installed to be monitored or controlled in a building. Examples of devices include light switches, thermostats, cameras, other mechanical loads, chillers, cooler, and so on.
FOTA	Firmware Over the Air
FW	Firmware
IoT	Internet of Things
OS	Operating System
OTA	Over-the-air
SMBIOS	System Management BIOS
SOTA	Software Over the Air (OS update)

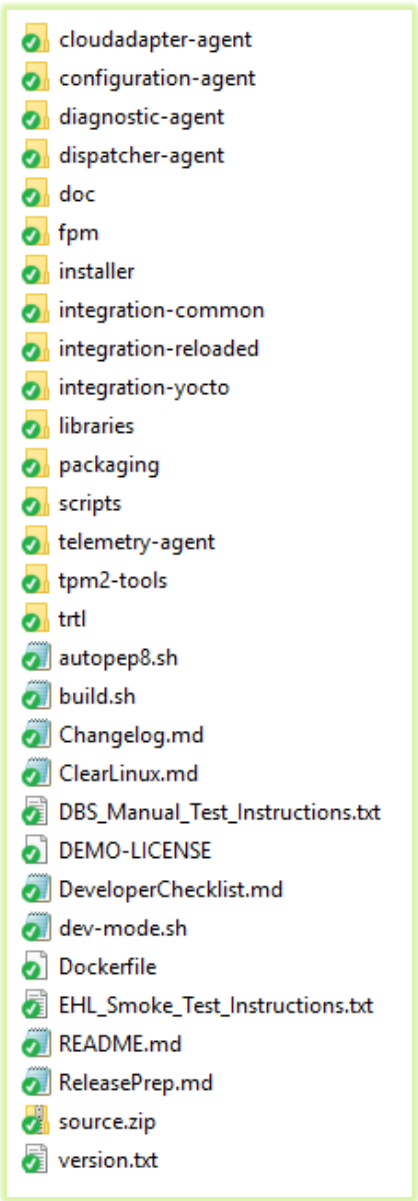
§

2.0 Source Overview

In-Band Manageability has five different agents namely cloudadapter, configuration, diagnostic, dispatcher, telemetry. Each agent has its own unique responsibility to handle different things.

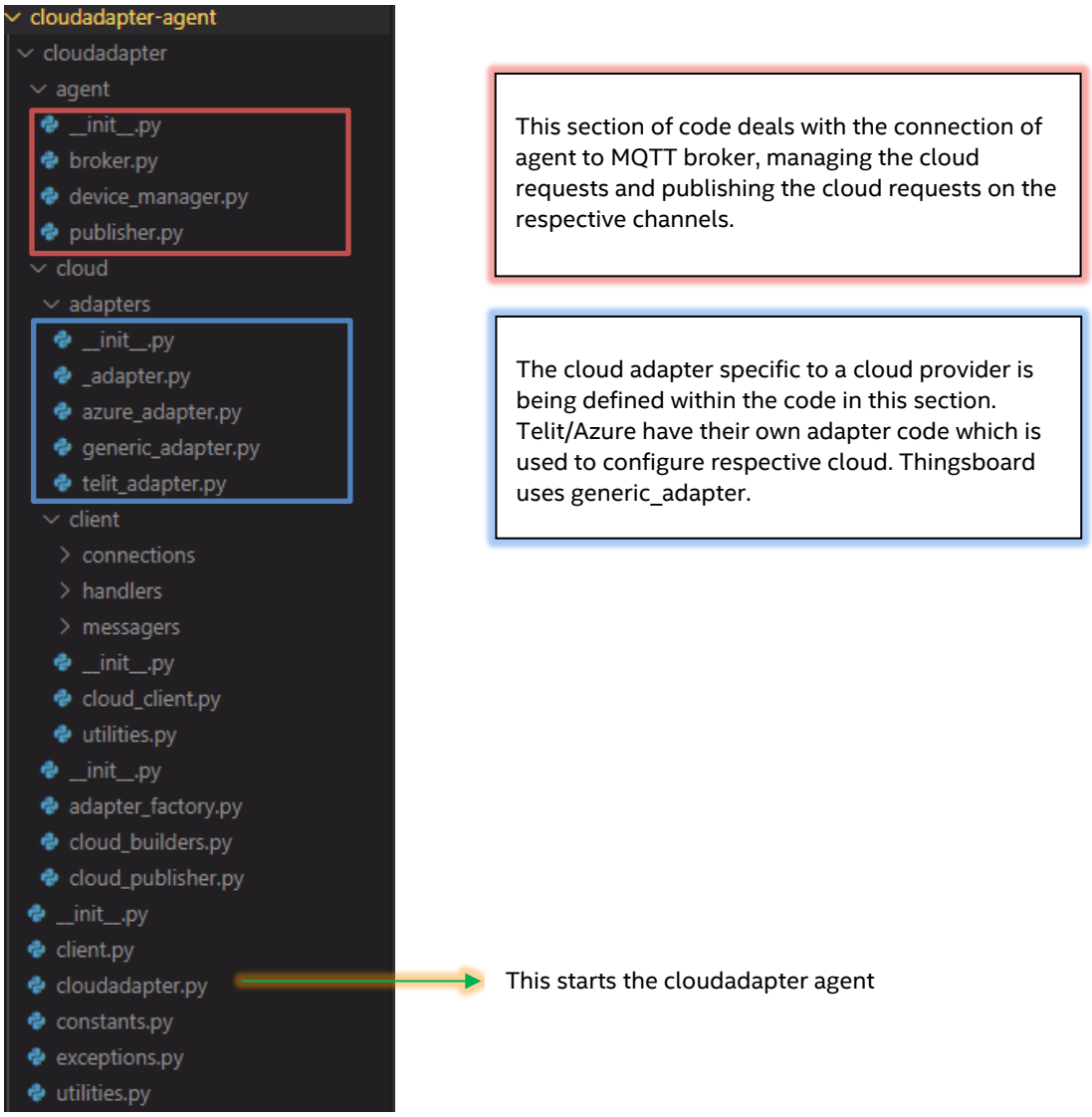
Each agent communicates with the other agents by using the MQTT list below in Section 8.

The source structure of turtle-creek is as shown in the diagram below:



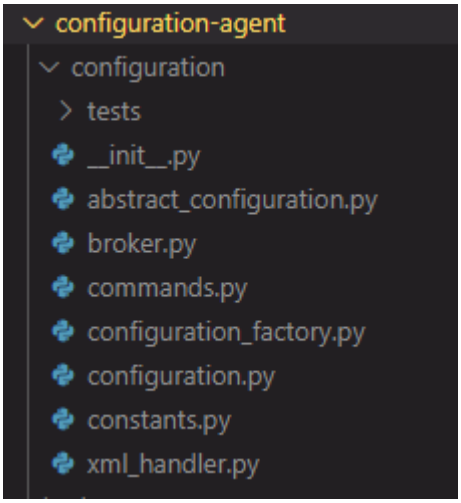
2.1 Agents Overview

Cloudadadapter-agent: This agent is responsible for all the communication between the device and the cloud. It handles all the operations related to connections to the cloud, publishing messages to the cloud, receiving messages from the cloud etc. Whenever a request for an OTA is made the message is first received by the cloudadapter agent and then passes it over to the dispatcher agent via MQTT channel for performing the request.



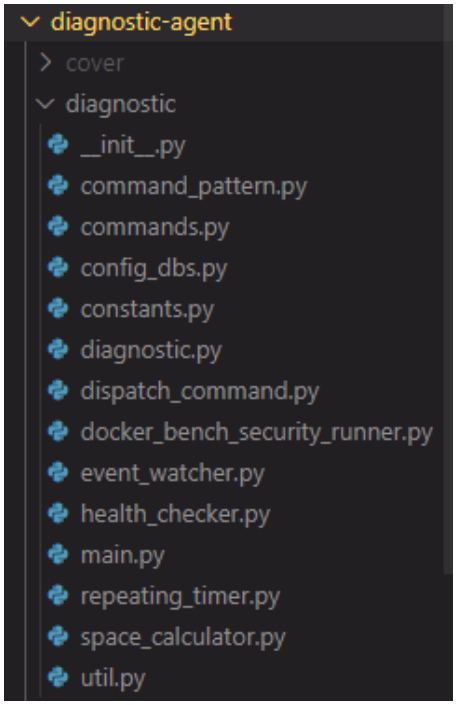
The file *cloudadapter-agent/cloudadapter/constants.py* contains all the MQTT subscription and publishing channels used by cloudadapter-agent to communicate with other agents.

Configuration-agent: This agent is responsible for publishing the config parameter values across all other agents. The parameters being used by this agent are from */etc/intel_manageability.conf* file. The descriptions of the parameters are available in the USER guide. To configure and use a new parameter, refer Section 4.



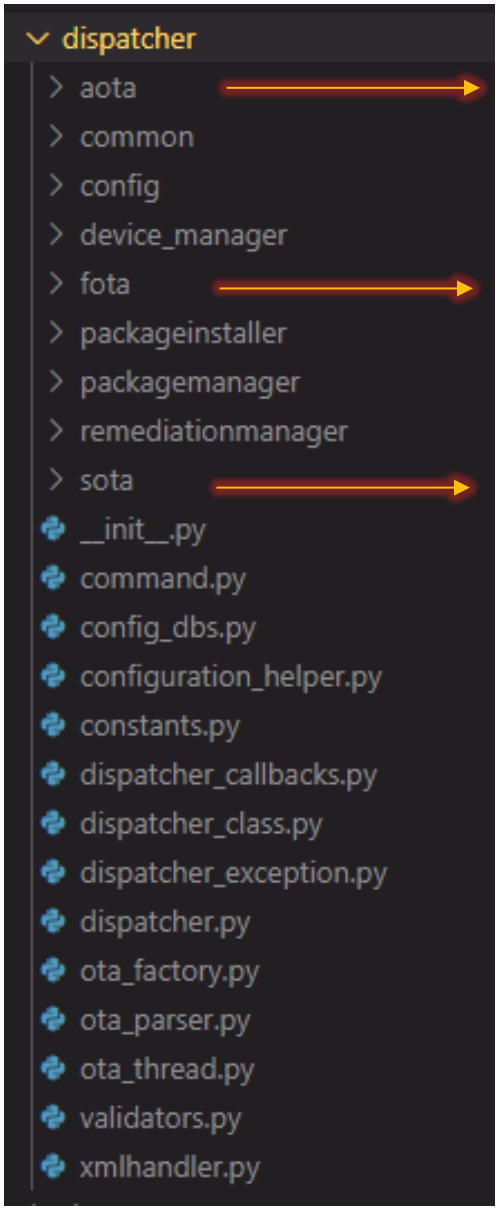
The *broker.py* handles all the config updates that are to be performed on the system.
The *configuration.py* starts the configuration agent.
The *xml_handler.py* contains the necessary functions required to modify the XML conf file.
The *constants.py* contains all the MQTT subscription and publishing channels used by configuration-agent.

Diagnostic-agent: Diagnostic agent monitors and reports the state of critical components of the framework. This agent is responsible for performing all the diagnostic checks like system health checks. It requires software checks before installation, network checks, docker stats, docker-bench-security checks as such. These checks will be performed at timed intervals. These timed intervals can be altered by changing the interval seconds within the */etc/intel_manageability.conf* file using configuration updates from cloud via button click or manifest update. Once the checks are completed, the result message is published to the cloud as telemetry.



The *command_pattern.py* consists of all the commands/checks that are being handled by the diagnostic agent.
The *constants.py* contains all the MQTT subscription and publishing channels used by diagnostic-agent.
The *dispatch_command.py* dispatches correct command/checks based on the request.
The *docker_bench_security_runner.py* runs the DockerBenchSecurity checks on the docker containers and images, while the *event_watcher.py* watches for events from Docker.
The *repeating_timer.py* creates a timer that repeats for a given interval specified by the time-based checks.
The file *constants.py* contains all the MQTT subscription and publishing channels used by diagnostic-agent to communicate with other agents.

Dispatcher-agent: The dispatcher-agent's role is to dispatch and perform the received commands/operations from the cloud on the device. It is responsible for determining what kind of request is received from the cloud and it invokes the respective commands/threads that would perform the desired operation. Once the operation is performed, the status of the operation will be published to the cloud by this agent.



This section handles the AOTA operations upon AOTA thread creation by *ota_thread.py*

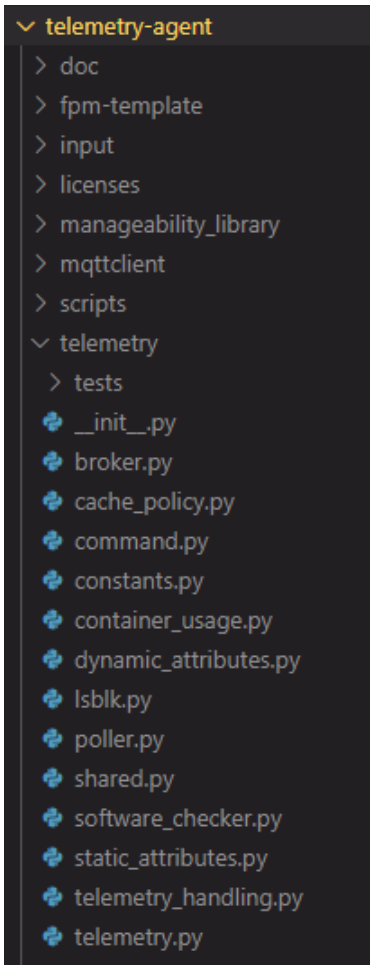
This section handles the FOTA operations upon FOTA thread creation by *ota_thread.py*

This section handles the SOTA operations upon SOTA thread creation by *ota_thread.py*

The *dispatcher_class.py* contains handles an OTA request from cloudadapter agent and then creates a thread respective to the OTA triggered. This code uses a factory pattern to determine the correct instance of the class to be used. A concrete instance of a class is derived from *ota_factory.py*.

The file *constants.py* contains all the MQTT subscription and publishing channels used by dispatcher-agent to communicate with other agents.

Telemetry-agent: The Telemetry agent publishes all the system's static and dynamic telemetry to the cloud.



The broker.py initializes the agents publish/subscribe channels.

The *container_usage.py* has code that gets the container stats on a device.

The *dynamic_attributes.py* contains functions that retrieve dynamic telemetry information such as disk_usage, cpu percentage, network telemetry, and available memory,

The *static_attributes.py* have function that gets the device's static telemetry information such as cpu_id, disk information, and total physical memory,

The *telemetry_handling.py* is responsible for calling the necessary telemetry events upon time-intervals and then publishing the information on to the cloud and other agents when needed.

The file *constants.py* contains all the MQTT subscription and publishing channels used by telemetry-agent to communicate with other agents.

2.2 Run Agents via Source Code

To run the agents after modifying the source code and test the source code, the developer is required to run the script *dev-mode.sh* located under the **iotg-manageability** directory.

For a developer to run and test the modified code, the following requirements are required on the device the developer is working on:

- Prior to developer running the code from source, Turtle Creek should be installed, running in binary mode, and provisioned to the cloud.

The *dev-mode.sh* script checks the environment, installs all the required dependencies, disables, and stops the active Turtle Creek agents.

If the network needs proxy, the developer may need to add the respective proxy information in **line 92** for pip within the *dev-mode.sh* to install the dependencies. If no proxy is required, the proxy parameter needs to be removed from the script.

Once the necessary changes are made to the code, the developer is required to open one terminal for each associated agent. On each terminal, run the following commands:

1. `cd ~/iotg-manageability/inbm/<agent>`
2. execute tests

Additionally, the user can enable logging from the agent terminal using the command

```
make logging LEVEL=DEBUG
```

or refer to Section 5 to enable logging prior to running the Turtle Creek via source code.

§

3.0 Build instructions

Developers can build Turtle Creek executables if source is provided as part of the release package.

To successfully build Turtle Creek code, the user would need to execute the following commands, to make sure the scripts have the executable access:

```
cd turtlecreek/source

find . -type f -iname configure -exec chmod +x {} \;

find . -type f -iname "*.sh" -exec chmod +x {} \;

chmod -R 755 trtl/scripts/
```

The user should be able to build the source from the directory **turtlecreek/source** using the command.

Note: Docker needs to be installed on the system to build the code.

```
sudo ./build.sh
```

or the following command can also be used for better build performance:

```
sudo DOCKER_BUILDKIT=1 ./build.sh
```

When the build is complete, the build output can be found in the **turtlecreek/source/output** folder.

§

4.0 Configuring Framework

There may be scenarios where new configurations are required to be added to extend the functionality of the Turtle Creek framework. For example, if a developer plans to add a new health check telemetry code within the framework, and wants to configure the */etc/intel_manageability.conf* to accommodate this health check tag with a certain value, follow the steps:

1. The user needs to first edit the configuration base file at

```
~/turtle-creek/configuration-agent/fpm-template/etc/intel_manage
ability.conf
```

2. The xsd schema that validates the above file must also accommodate the new tag, else the schema validation would fail. The schema is located at

```
~/turtle-creek/configuration-agent/fpm-template/usr/share/
configuration-agent/turtle_creek_schema.xsd
```

3. Once the changes are made, the code can now use the new parameter value added to the conf file.
4. To test the changes the developer can build the entire source code using the build instructions mentioned in Section 3, and then uninstalling and reinstalling the turtlecreek from output folder after the build is complete.

(Or)

Copy the conf file in step 1 to */etc/intel_manageability.conf* and *xsd_schema* file in step 2 to */usr/share/turtle_creek_schema.xsd* and then run the agents via source code. To run the modified source code, refer Section 2.2.

§

5.0 Security

Security is a key aspect for any software solution to consider, especially for IoT devices that would be deployed in the field. The below section details out the various techniques, measures, assumptions, and recommendations we made when designing Turtle Creek.

5.1 OS Hardening

Turtle Creek, a user space application, relies on the underlying OS to have inbuilt security capabilities. Below is a list of solution that is recommended to help harden the security of any OS.

Secure Boot: An industry standard now, where the Platform Firmware (BIOS) checks signature of the boot chain software, for example, UEFI drivers, EFI applications, and OS Kernel. If the signature is valid, the BIOS proceed with the boot chain. This mechanism ensures that boot chain components were not altered at rest and hence can guarantee to certain extent the integrity and authenticity of the boot components.

AppArmor: AppArmor is an access control mechanism in Linux kernel, which confines a program to resources that are set in its profile. AppArmor binds access control attributes to Program and not to Users. AppArmor profiles are loaded during boot by the kernel and consist of policies that the Program would be subjected to while accessing the resources listed in it.

5.2 Turtle Creek Hardening

In addition to the OS hardening recommendations, Turtle Creek also ensures a Secure solution by following the below mechanisms.

5.2.1 AppArmor Profiles

All the Turtle Creek frameworks services/tools have an associated AppArmor profile, which gets enforced when the framework is installed on a platform. These profiles define the access that Turtle Creeks executables have on the underlying platform resources (file system), ensuring only certain directories are readable/writeable, thereby reducing the risk of corrupting the platform by accessing an unauthorized resource via Turtle Creek.

These profiles can be found under */etc/apparmor.d/usr.bin.<turtle creek services>*

5.2.2 Access Control List

Turtle Creeks services communicate with each other over MQTT in localhost. MQTT being a pub/sub protocol client publish and receive information on predefined “topics”. Controlled access to these topics is critical to ensure that an unauthorized MQTT client is not able to eavesdrop or publish incorrect/garbage data to legitimate clients. Access control is achieved by setting up ACL list in the MQTT Broker configuration. Access control specifies which topic each of the MQTT clients is authorized to read and write to.

ACL list is defined in */etc/intel-manageability/public/mqtt-broker/acl.file*

5.2.3 MQTT over TLS Support

To further protect the confidentiality of data being transmitted between Turtle Creek services, the session between the MQTT Broker and the services is established over mutual TLS. During the provisioning phase of Turtle Creek on a platform, the provisioning script sets up the MQTT broker and the Turtle Creek services with X509 cert and keypairs to facilitate mTLS session. The certs and keys are located under:

Certificate: */etc/intel-manageability/public/<>*

Key: */etc/intel-manageability/secret/*

5.2.4 Trusted Repo List

The OTA command consists of a URL from where Turtle Creek fetches the update (FW, OS, Application) package. To ensure that a package is only fetched from a trusted location, Turtle Creek maintains a list of URLs tagged as – *<trustedRepositories>* in the configuration file.

The trustedRepositories URL’s can be found in: */etc/intel_manageability.conf*

5.2.5 Signature Verification on OTA Packages

To ensure that the OTA packages are not modified or corrupted, Turtle Creek also employs signature verification of the downloaded OTA package against an OTA cert that the users can enroll during the provisioning step. Turtle Creek verifies the signature of the OTA package being sent in the Manifest against the signature generated using the enrolled OTA cert.

When the signature matches, Turtle Creek proceeds with the update command, else it deletes or removes the OTA package from the platform.

5.2.6 Manifest Schema Checks

As Turtle Creek accepts OTA commands in the format of XML string, it enforces strict Manifest Schema checks to ensure that the OTA commands meet a predefined requirement of tags, fields, data lengths etc. This ensures that no unwanted data or tags are injected in the OTA commands.

5.2.7 Docker Bench Security

As Turtle Creek users can also deploy Containers, Turtle Creek uses the Docker Bench Security (DBS) to enhance the security of the platform. The Docker Bench Security is a script that checks for common best practices around deploying Docker containers in production.

Configuration that enforces DBS can be found in */etc/intel_manageability.conf* under *<dbs>* tag.

In production, it is recommended that the DBS be set to ON.

5.2.8 Platform TPM usage

Turtle Creek services communicate with each other over TLS secured MQTT sessions. The certificates used for this communication are created during the provisioning phase of the framework where the private/public certs and keys are generated using OpenSSL API's. While provisioning, Turtle Creek creates a small file system that writes all the generated private keys.

As these certs or keys are considered secrets, they are kept encrypted on the disk. The encryption is done using a randomly generated passphrase which Turtle Creek stores in TPM.

Turtle Creek uses slot **0x81001231** on the platform TPM as it is unlikely to conflict with any other programs TPM usage. If this slot is used by any other program, then we would need to assign a new slot for Turtle Creek.

This can be done by modifying the file: */usr/bin/**tc-get-tpm-passphrase***

```
STORE_SLOT="0x81001231"
```

§

6.0 Enable Logging

Upon starting the Turtle Creek services, the following agents will connect the device to the cloud.

Agents: dispatcher, diagnostic, telemetry, configuration, cloudadapter

To enable debug messages, the user could configure *logging.ini* files for each agent by changing **ERROR** to **DEBUG** with a text editor.

Note: These *logging.ini* files are located at */etc/intel-manageability/public/<agent-name>-agent/logging.ini*

For example: If the logging needs to be enabled for the dispatcher agent,

- `sudo vi /etc/intel-manageability/public/<agent-name>-agent/logging.ini`
- Change the value **ERROR** to **DEBUG**
- Restart the agent using sudo systemctl restart dispatcher.

1. To reload the modified files to display debug logs, restart the agents with the command below as root:

```
Command-line: sudo systemctl restart dispatcher diagnostic
telemetry configuration cloudadapter
```

2. To view logs of a particular agent, run the following command:

```
Command-line: journalctl -fu <agent-name>
```

Quick Tip: If the logging needs to be enabled on all the agents, the following command can be used to enable logging. Once the command is executed, follow step 1 to restart all agents.

```
sudo sed -i 's/level=ERROR/level=DEBUG/g' /etc/intel-manageabilit
y/public/*/logging.ini
```

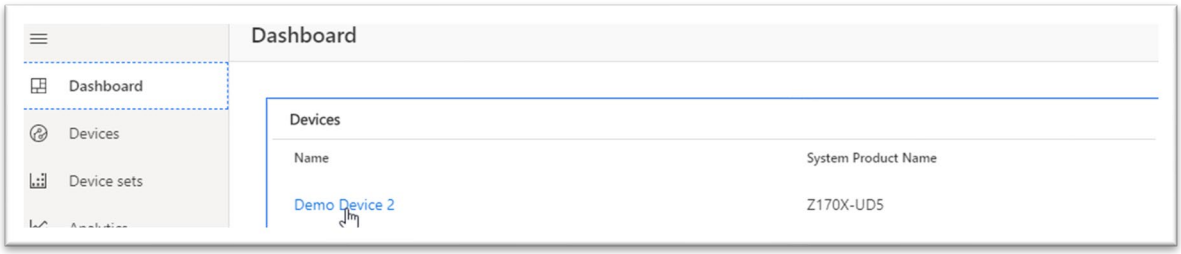
§

7.0 OTA Updates via Manifest

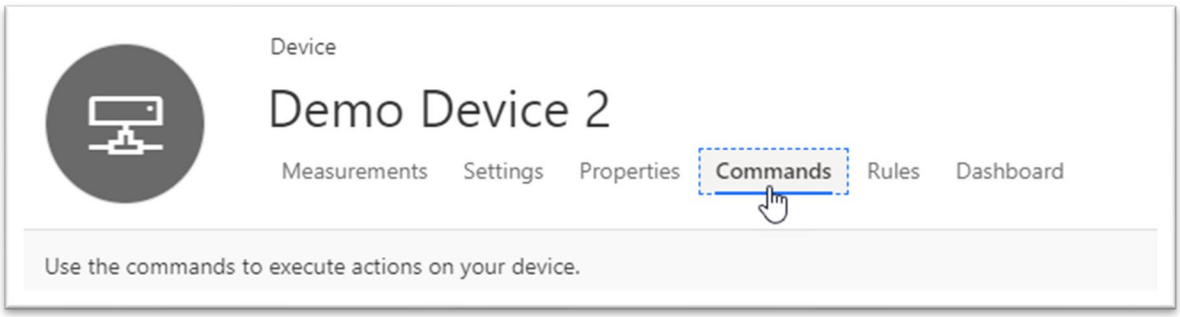
Manifest is an XML string that contains important information about the update to be executed. Any OTA update type can be done via the Manifest Update, by entering XML text to update the Endpoint.

To trigger Manifest updates:

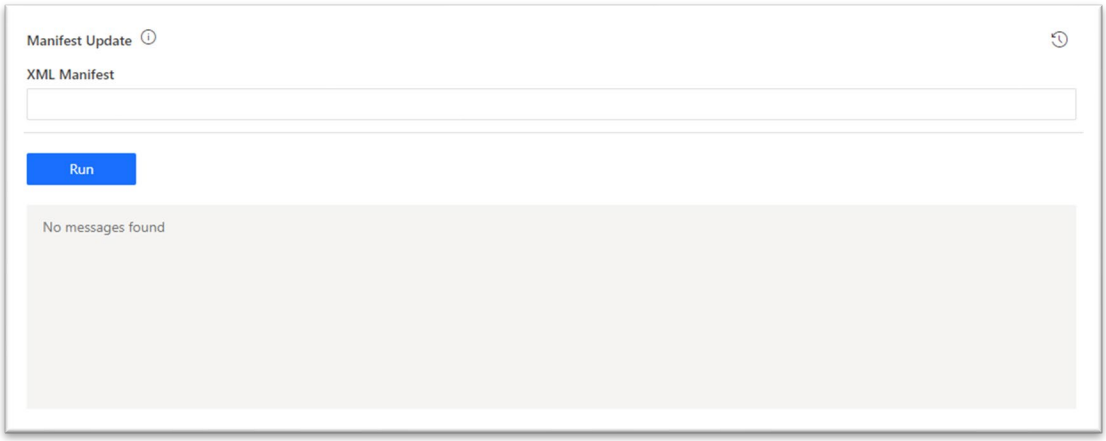
1. Click **Dashboard** tab to select Edge Device. Then, click the device name.



2. Select **Commands** tab.



3. Scroll the page to the text area named **Manifest Update**.



7.1 Manifest Rules

- All tags marked as **required (R)** in the manifest examples below must be in the manifest. Any tags marked as **optional (O)** can be omitted.
- The start of a section is indicated as follows **<manifest>**.
- The end of a section is indicated by **</manifest>**. All sections must have the start and the matching end tag.
- Remove spaces, tabs, comments and so on. Make it a single continuous long string. Example: **<xml ...><manifest><ota><header>...</ota><manifest>**
- Parameter within a tag cannot be empty.
  Example: **<description></description>** is not allowed.

7.2 AOTA Updates

Supported AOTA commands and their functionality:

The supported ***docker*** commands are as indicated in the following table:

***docker* Command**	**Definition**
Import	Importing an image to the device
Load	Loading an image from the device
Pull	Pulls an image or a repository from a registry
Remove	Removes docker images from the system
Stats	Returns a live data stream for all the running containers

The supported ***docker-compose*** commands are as stated in the following table:

***docker-compose* Command**	**Definition**
Up	Deploying a service stack on the device
Down	Stopping a service stack on the device
Pull	Pulls an image or a repository from a registry
List	Lists containers
Remove	Removes docker images from the system

List of AOTA commands NOT supported:

Docker-Compose	Import
	Load
	Stats
Docker	Up
	Down
	List

Fields in the AOTA form:

Field	Description
App	Docker or Docker-compose
Command	Docker-Compose supported operations: Up, Down, Pull, List and Remove. Docker supported operations: Load, Import, Pull, Remove and Stats
Container Tag	Name tag for image/container.
Docker Compose File	Field to specify the custom yaml file for docker-compose command. Example: *custom.yml*
Fetch	Server URL to download the AOTA container *tar.gz* file **Note:** If the server requires username/password to download the file, you can provide in server username/ server password
Server Username/ Server Password	If server needs credentials, we need to specify the username and password
Version	Each container will have a tag with the version number. You are recommended to use this version number under version in the AOTA trigger. Command: sudo docker images. See image below to see result of this command
Docker Registry Docker Registry Username/Password	Specify Docker Registry if accessing any registry other than the default *index.docker.io*. Optional fields Docker Registry Username/Password can be used to access docker private images in AOTA through docker and docker-compose up, pull commands.

7.2.1 AOTA Manifest Parameters

Tag	Example	Required/ Optional	Notes
<?xml version='1.0' encoding='utf-8'?>		R	XML header
<manifest>	<manifest>	R	
<type></type>	<type>ota</type>	R	Always OTA
<ota>	<ota>	R	
<header>	<header>	R	
<id></id>	<id>yourid</id>	O	

<name></name>	<name>SampleAOTA</name>	O	
<description></description>	<description>Yourdescription</description>	O	
<type></type>	<type>aota</type>	R	
<repo></repo>	<repo>remote</repo>	R	
</header>	</header>	R	
<type>	<type>	R	
<aota name="">	<aota name="text">	R	Text must follow XML standards
<cmd></cmd>	<cmd>up</cmd>	R	Valid values: Up, down, load, import, pull, list, remove, stats
<app></app>	<app>docker</app>	R	Valid values: Docker, btrfs, compose
<fetch></fetch>	<fetch> http.yoururl.com<fetch>	R	Trusted repo + name of package http://server name/AOTA/ container tar.gz
<file></file>	<file>custom.yml</file>	O	User can specify the name of custom yaml file to use with docker-compose
<version></version>	<version>0.7.6</version>	O	Update Package version.
<signature></signature>	<signature>96e92d</signature>	O	Signature of package – signed checksum of package. Recommended for security purposes
<containerTag></containerTag>	<containerTag>Modbusservice</containerTag>	R	Name of container image
<username></username>	<username>user</username>	O	Username credentials of the server where the package is hosted for downloads
<password></password>	<password>ps</password>	O	Password credentials of the server where the package is hosted for downloads

<docker_username></docker_username>	<docker_username>usr</docker_username>	O	Docker Username credentials of the private registry where docker images reside
<docker_password></docker_password>	<docker_password>psw d</docker_password>	O	Docker password credentials of the private registry where docker images reside
<docker_registry></docker_registry>	<docker_registry>hub.intel.docker.com</docker_registry>	O	Docker registry URL of any private registry where the
			required docker images reside.
</type>	</type>	R	
</ota>	</ota>	R	
</manifest>	</manifest>	R	

7.2.2 Docker manifest examples

7.2.2.1 Example of docker image import manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name='samplerpm'><cmd>import</cmd><app>docker</app><fetch>yourul
/hdcrpmlite.tgz</fetch><version>1.0</version><containerTag>hdcrpm
lite:1</containerTag></aota></type></ota></manifest>
```

7.2.2.2 Example of docker image load manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name='samplerpm'><cmd>load</cmd><app>docker</app><fetch>yourul/c
offee.tgz</fet
ch><version>1.0</version><containerTag>coffee</containerTag></aot
a></ type></ota></manifest>
```

7.2.2.3 Example of docker pull manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name='modbusservice'><cmd>pull</cmd><app>docker</app><version>1.0
</version><con tainerTag>hello-
world</containerTag></aota></type></ota></manifest>
```

7.2.2.4 Example of docker remove manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name='modbusservice'><cmd>remove</cmd><app>docker</app><version>1
.0</version>< containerTag>hello-
world</containerTag></aota></type></ota></manifest>
```

7.2.2.5 Example of docker stats manifest

```
<?xml version="1.0" encoding="utf-
8"?><manifest><type>ota</type><ota><header><type>aota
</type><repo>remote</repo></header><type><aota name="sample-
rpm"><cmd>stats</cmd><app>docker</app></aota></type></ota></manif
est>
```

7.2.3 Docker-Compose Manifest Examples

7.2.3.1 Example of docker-compose up manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name='samplerpm'><cmd>up</cmd><app>compose</app><fetch>yoururl/si
mplecompose.tar.gz</fetch><version>2.0</version><containerTag>sim
plecompose</containerTag></aota></type></ota></manifest>
```

7.2.3.2 Example of 'docker-compose -f <custom.yml> up' manifest

```
<?xml version="1.0" encoding="utf-
8"?><manifest><type>ota</type><ota><header><type>aota</type><repo
>r emote</repo></header><type><aota
name="samplerpm"><cmd>up</cmd><app>compose</app><fetch>https://ap
idev.devicewise.com:443/file/5dcbfbe114c9786cd10c6d84/simplecompo
se.tar.gz</fetch><file>custom.yml</file><containerTag>simplecompo
se</containerTag><username>username</username><password>XX
XXX</password></aota></type></ota></manifest>
```

7.2.3.3 Example of docker-compose down manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><type>aota</type><repo
> remote</repo></header><type><aota
name='modbusservice'><cmd>down</cmd><app>compose</app><fetch>http
://ubitartifactory-or.intel.com/artifactory/bmpv2-test-
orlocal/bmp_075_update/modbusservice.tar.gz</fetch><version>1.0</
version><containerTag>modbusservice</containerTag></aota></type><
/ota></manifest>
```

7.2.3.4 Example of 'docker-compose -f <custom.yml> down' manifest

```
<?xml version="1.0" encoding="utf-
8"?><manifest><type>ota</type><ota><header><type>aota</type><repo
>remote</repo></header><type><aota
name="samplerpm"><cmd>down</cmd><app>compose</app><file>custom.ym
l</file><container Tag>simple-
compose</containerTag></aota></type></ota></manifest>
```

7.2.3.5 Example of docker-compose pull manifest

```
 <?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><id>sampleId</id><name
> Sample AOTA</name><description>Sample AOTA manifest
file</description><type>aota</type><repo>remote</repo></header><t
ype><aota name='sample-docker-compose-
up'><cmd>pull</cmd><app>compose</app><fetch>http://127.0.0.1:80/s
imp le-
compose.tar.gz</fetch><version>1.0</version><containerTag>simplec
ompose</containerTag></aota></type></ota></manifest>
```

7.2.3.6 Example of 'docker-compose -f <custom.yml> pull' manifest

```
<?xml version="1.0" encoding="utf-
8"?><manifest><type>ota</type><ota><header><type>aota</type><repo
>re mote</repo></header><type><aota
name="samplerpm"><cmd>pull</cmd><app>compose</app><fetch>https://
apidev.devicewise.com:443/file/5dcbfbe114c9786cd10c6d84/simplecom
pose.tar.gz</fetch><file>custom.yml</file><containerTag>simplecom
pose</containerTag><username>username</username><password>XX
XXX</password></aota></type></ota></manifest>
```

7.2.3.7 Example of docker-compose list manifest

```
 <?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><id>sampleId</id><name
> Sample AOTA</name><description>Sample AOTA manifest
file</description><type>aota</type><repo>remote</repo></header><t
ype><aota name='sample-docker-
composeup'><cmd>list</cmd><app>compose</app><containerTag>simplec
ompose</containerTag></aota></type></ota></manifest>
```

7.2.3.8 Example of docker-compose remove manifest

```
 <?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><id>sampleId</id><name
> Sample AOTA</name><description>Sample AOTA manifest
file</description><type>aota</type><repo>remote</repo></header><t
ype><aota name='sample-docker-
composeremove'><cmd>remove</cmd><app>compose</app><version>1.0</v
ersion><containerTag>simple-
compose</containerTag></aota></type></ota></manifest>
```

7.3 FOTA Updates

To perform FOTA updates, IBVs must supply the SMBIOS or Device Tree info that is unique to each platform SKU. The info must fulfill the vendor, version, release date, manufacturer, and product name that matches the endpoint as shown below.

Prior to sending the manifest the user needs to make sure that the platform information is present within the */etc/firmware_tool_info.conf* file. Refer to Section 7 on how to modify the file and extend the FOTA support to a new platform.

Note: The following information must match the data sent in the FOTA update command for In-Band Manageability Framework to initiate a firmware update process.

Information	Field	Checks
Firmware	Vendor	Exact string match
	Version	Checks if its “unknown”
	Release Date	Checks if the FOTA date is newer than current
System	Manufacturer	Exact string match
	Product Name	Exact string match

To find the firmware and system fields at the endpoint, run the commands below:

Intel x86 UEFI-based products

For UEFI based platforms, the firmware and system information can be found running the following command:

```
Command-line: dmidecode -t bios -t system
```

7.3.1 FOTA Manifest Parameters

Tag	Example	Required /Optional	Notes
<?xml version='1.0' encoding='utf-8'?>		R	
<manifest>	<manifest>	R	
<type></type>	<type>ota</type>	R	Always OTA
<ota>	<ota>	R	

<header>	<header>	R	
<id></id>	<id>yourID</id>	O	
<name></name>	<name>YourName</name>	O	Endpoint Manufacturer Name
<description></description>	<description>YourDescription </de scription>	O	
<type></type>	<type>fota</type>	R	
<repo></repo>	<repo>remote</repo>	O	
</header>	</header>	R	
<type>	<type>	R	
<fota name=''>	**<fota name='text'>**	R	Comply with XML Standards
<fetch></fetch>	**<fetch>http://url:80/BIOSUP DAT E.tar</fetch>**	R	FOTA path created in repository
<signature></signature>	**<signature> </signature>**	O	Digital signature of *.tar file.
<biosversion></biosversion>	**<biosversion>A..ZZZZ.B11.1< /bio sversion>**	R	Verify with BIOS Vendor (IBV)
<vendor></vendor>	**<vendor>VendorName</ven dor>**	R	Verify with BIOS Vendor (IBV)
<manufacturer></manufactur er>	**<manufacturer>BIOS_Manufa ctur er</manufacturer>**	R	In Release Notes supplied by BIOS vendor
<product></product>	<product>BIOS_Product</pro duct > In Release Notes supplied by bios vendor	R	Product Name set by Manufacturer
<releasedate></releasedate>	<releasedate>2017-06-23</releasedate>	R	Verify with BIOS Vendor (IBV)

<tooloptions></tooloptions>	<tooloptions>p/b/n</tooloptions>	O	Verify with BIOS Vendor (IBV)
<guid></guid>	**<guid>**7acbd1a5a-33a4-48c3ab11-a4c33b3d0e56**</guid>**	O	Check for 'System Firmware Type' on running cmd:fwupdate -l
<path></path>	<path></path>	R	
</fota>	</fota>	R	
</type>	</type>	R	
</ota>	</ota>	R	
</manifest>	</manifest>	R	

The following table references each XML tag within a manifest that triggers the FOTA update. Using the following XML tags in the order of description will trigger a FOTA update via Manifest.

7.3.2 Sample FOTA Manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>ota</type><ota><header><id>sampleID</id><name
>Sampl e FOTA</name><description>Sample FOTA manifest
file</description><type>fota</type><repo>remote</repo></header><t
ype><fota
name='sample'><fetch>http://ubuntufota.jf.intel.com:8000/Afulnx+X
041_BIOS.tar</fe
tch><biosversion>5.12</biosversion><vendor>American Megatrends
Inc.</vendor><manufacturer>Default
string</manufacturer><product>Default
string</product><releasedate>2017-
1120</releasedate><path>/var/cache/repositorytool</path></fota></
type></ota></manifest>
```

7.4 SOTA Updates

SOTA flow can be broken into two parts:

1. Pre-reboot - The pre-boot part is when a SOTA update is triggered.
2. Post-reboot - The post-boot checks the health of critical manageability services and takes corrective action.

7.4.1 SOTA Manifest Parameters

Tag	Example	Required/ Optional	Notes
<?xml version='1.0' encoding='utf-8'?>		R	
<manifest>	<manifest>	R	
<type></type>	<type>ota</type>	R	Always OTA
<header>	<header>	R	
<id></id>	<id>Example</id>	O	
<name></name>	<name>Example</name>	O	
<description></description>	<description>Example</description>	O	
<type></type>	<type>sota</type>	R	
<repo></repo>	<repo>remote</repo>	R	
</header>	</header>	R	
<type>	<type>	R	
<sota>	<sota>	R	
<cmd></cmd>	<cmd	R	

	logtofile="Y">update</cmd>		
<username></username>	<username>xx</username>	O	
<password></password>	<password>xxx</password>	O	
<release_date></release_date>	<release_date>2020-01-01</release_date>	R	The release date provided should always be in 'YYYY-MM-DD' format. Also release date should be higher than the platform mender date to proceed to SOTA update.
</sota>	</sota>	R	
</type>	</type>	R	
</ota>	</ota>	R	
</manifest>	</manifest>	R	

7.4.2 Sample SOTA Manifest:

```
<?xml version="1.0" encoding="utf-
8"?><manifest><type>ota</type><ota><header><id>a</id><name>a</n
ame><description>a</description><type>sota</type><repo>remote</
repo></header><type><sota><cmd
logtofile="Y">update</cmd><release-date>2020-
0101</release_date></sota></type></ota></manifest>
```

7.5 Configuration Operations

Configuration update is used to change/retrieve/append/remove configuration parameters value from the Configuration file located at */etc/intel_manageability.conf*. Refer to table below to understand the configuration tags, values, and descriptions.

Default Configuration Parameters:

Telemetry		
Collection Interval Seconds	60 seconds	Time interval after which telemetry is collected from the system.
Publish interval seconds	300 seconds	Time interval after which collected telemetry is published to dispatcher and the cloud
Max Cache Size	100	Maximum cache set to store the telemetry data. This is the count of messages that telemetry agent caches before sending out to the cloud
Container Health Interval Seconds	600 seconds	Interval after which container health check is run and results are returned.
Diagnostic Values		
Min Storage	100 MB	Value of minimum storage that the system should have before or after an update
Min Memory	200 MB	Value of minimum memory that the system should have before or after an update
Min Power Percent	20%	Value of minimum battery percent that system should have before or after update
Mandatory SW	docker, trtl, telemetry	List of software that should be present and are checked for.
Docker Bench Security Interval Seconds	900 seconds	Time interval after which DBS will run and report back to the cloud.
Network Check	true	This configures network check on the platforms based on their Ethernet capability.
Dispatcher Values		
DBS Remove Image on Failed Container	False	Specifies if the image should be removed in the event of a failed container as flagged by DBS.
Trusted Repositories		List of repositories that are trusted and packages can be fetched from them
SOTA Values		

Ubuntu Apt Source	http://linux-ftp.jf.intel.com/pub/mirrors/ubuntu/	Location used to update Ubuntu
Proceed Without Rollback	True	Whether SOTA update should go through even when rollback is not supported on the system.

7.5.1 Configuration Manifest

To send the whole manifest with edited parameters at once,

- Go to **Manifest Update** widget by clicking the **eye** icon next to the device of interest under **Methods**.
- Enter the parameters to be updated along with the path of the element in the system.
- To see the values of parameters, use **Get Element Manifest**.
- To modify the parameters of interest, use the **Set Element Manifest** and edit the values. Use the tag to identify the category of the configuration you are updating. Example *diagnostic* or *telemetry*.
- To overwrite the existing configuration file with a new one then use the **Load Element Manifest**.

The following commands are useful to append, remove values for parameters that have multiple values. Parameters that have multiple values are **trustedRepositories**, **sotaSW** and **ubuntuAptSource**.

- To append to existing value, use **Append Element** manifest.
- To remove part of a value, use **Remove Element** manifest

7.5.1.1 Example of Get Element manifest

Example:

- To set one value: **minStorageMB**
- To set multiple values at once: **minStorageMB;minMemoryMB**

Note: Path takes in keys as an input, with key as the configuration parameter tag, where the value needs to be retrieved. To retrieve multiple values at once, use '**;**' to separate one tag from another as shown above.

```
<?xml  version="1.0"  encoding="UTF-
8"?><manifest><type>config</type><config><cmd>get_element</cmd><c
onfigtype><get><path>minStorageMB;minMemoryMB</path></get></confi
gtype></config></manifest>
```

7.5.1.2 Example of Set Element manifest

Example:

- To set one value: minStorageMB:100
- To set multiple values at once: minStorageMB:100;minMemoryMB:200

Note: Path takes in key value pairs as an input, with key as the configuration parameter tag and value to be set as the value. To set multiple key:value pairs, use "**;**" to separate one pair from another as shown in the example above.

```
<?xml  version="1.0" encoding="UTF-
8"?><manifest><type>config</type><config><cmd>set_element</cmd><c
onfigtype><set><path>minStorageMB:100;minMemoryMB:200</path>
</set></configtype></config></manifest>
```

7.5.1.3 Example of Append Element manifest

Note: Append is only applicable to three configuration tags, for example, **trustedRepositories**, **sotaSW** and **ubuntuAptSource**

Path takes in key value pair format, example: trustedRepositories: https://dummyURL.com

```
<?xml  version="1.0" encoding="UTF-
8"?><manifest><type>config</type><config><cmd>append</cmd><config
type><append><path>trustedRepositories:https://dummyURL.com</path
> </append></configtype></config></manifest>
```

7.5.1.4 Example of Remove Element manifest

Note: *Remove* is only applicable to three configuration tags, for example, **trustedRepositories**, **sotaSW** and **ubuntuAptSource**.

Path takes in key value pair format, example: trustedRepositories: https://dummyURL.com

```
<?xml  version="1.0" encoding="UTF-8"?><manifest><type>c
onfig</type><config><cmd>append</cmd><configtype><remove>
<path>trustedRepositories:https://dummyURL.com</path></remove></c
onfigtype></config></manifest>
```

7.5.1.5 Example of Load Element manifest

Note: The configuration file you provide in Fetch needs to be named as *intel_manageability.conf* file. If you wish to send with signature, tar both the pem file and the *intel_manageability.conf* in a tar file.

```
<?xml version="1.0" encoding="UTF- 8"?><manifest><type>co
nfig</type><config><cmd>load</cmd><configtype><load><fetc
```

```
h>http://ubuntufota.jf.intel.com:8000/turtle_creek.conf</fetch></
load></configtype></config></manifest>
```

7.5.2 Manual Configuration Update:

User can also manually update the parameters of the configuration file within */etc/intel_manageability.conf* instead of triggering a config update from the cloud.

To manually edit the parameter values. The user needs to open */etc/intel_manageability.conf* file in a text editor and edit the parameter values. Then restart the configuration agent using the following command:

```
Command-line: sudo systemctl restart configuration
```

7.6 Power Management

Power Management such as restart, or shutdown of an end device can be triggered using a Manifest as well as Button Click.

7.6.1 Restart via Manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>cmd</type><cmd>restart</cmd></manifest>
```

7.6.2 Shutdown via Manifest

```
<?xml version='1.0' encoding='utf-
8'?><manifest><type>cmd</type><cmd>shutdown</cmd></manifest>
```

§

8.0 Extending FOTA support

In-Band Manageability Framework supports a scalable FOTA solution where triggering FOTA on any new platform is made easy by adding the platform related information to a config file that the framework uses while installing the new firmware.

8.1 Understanding FOTA Configuration File

The FOTA config file is located at */etc/firmware_tool_info.conf*. This file consists of all the platform information of the products that supports the In-Band Manageability FOTA.

If a new platform needs to be supported by the framework, the user needs to add the platform related information in the XML format within this conf file.

The XML format of the conf file looks similar as the following snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<firmware_component>
 <firmware_product name='NUC6CAYS'>
   <bios_vendor>Intel Corp.</bios_vendor>
   <operating_system>linux</operating_system>
   <firmware_tool>UpdateBIOS.sh</firmware_tool>
   <firmware_file_type>bio</firmware_file_type>
 </firmware_product>
</firmware_component>
```

Once the platform information is added, there are no code changes required. This information from the conf file will be used by the code to perform a FOTA update.

8.2 Configuration Parameter Values

The following table helps in understanding what each tag in the firmware configuration file refers to. The **Required(R)/Optional(O)** field associated with each tag represents whether the tag is mandatory or not while adding a new platform information.

Tag	Example	Required /Optional	Notes
<?xml version='1.0' encoding='utf-8'?>		R	
<firmware_component>	<firmware_component>	R	

<firmware_product name='<platform_name>'> Or <firmware_product name='<platform_name>' tool_options='true'> When tool_options is required by the firmware bootloader to install firmware	<firmware_product name='NUC6CAYS'> Or <firmware_product name='Default string' tool_options='true'>	R	This is the platform name. Run the command 'dmidecode –t bios –t system' to view the information
<operating_system>...</operating_system>	<operating_system>linux</operating_system>	R	OS name – Linux/Windows
<firmware_file_type>...</firmware_file_type>	<firmware_file_type>bio</firmware_file_type>	R	Type of the file name – bio, fv, cap etc.
<bios_vendor>vendor_name</bios_vendor>	<bios_vendor>Intel Corp.</bios_vendor>	O	Run the command 'dmidecode –t bios –t system' to view the information
<firmware_tool>firmware_tool_used_for_platform</firmware_tool>	<firmware_tool>UpdateBIOS.sh</firmware_tool>	O	Can be obtained from the vendor
<manufacturer>manufacturer_name</manufacturer>	<manufacturer>Intel Corp.</manufacturer>	O	Run the command 'dmidecode –t bios –t system' to view the information
<firmware_dest_path>location_of_new_firmware_file_to_be_stored<firmware_dest_path>	<firmware_dest_path>/boot/efi/</firmware_dest_path>	O	Only used on the platforms where the firmware update is just to replace the existing firmware file in a path.
<firmware_tool_args>any_additional_args</firmware_tool_args>	<firmware_tool_args>--apply</firmware_tool_args>	O	Additional arguments that follow the firmware tool command to apply firmware
<firmware_tool_check_args>firmware_arguments_to_check_if_tool_exists</firmware_tool_check_args>	<firmware_tool_check_args>-s</firmware_tool_check_args>	O	Additional args that are used to check if a firmware tool exists on system.

8.3 AppArmor Permissions:

When specifying the firmware update related to the tool or script name within the configuration mentioned in Section 7.2, the user needs to make sure that the tool or script path has AppArmor permissions granted to execute the firmware update. To add the information to the AppArmor profile, follow the steps below.

For example, the tool or script used to perform update for an Intel NUC is located at */usr/bin/UpdateBIOS.sh*. In this case, the user needs to make sure that the dispatcher's AppArmor profile has an entry with rix access rights to the script path. If the entry does not exist, the entry is required to be added to the AppArmor profile.

To edit the AppArmor profile:

1. *sudo vi /etc/apparmor.d/usr.bin.dispatcher*
2. Add the entry */usr/bin/UpdateBIOS.sh rix* with a comma at the end and save the file.

```
/usr/bin/UpdateBIOS.sh rix,
```

3. After updating the file, restart the AppArmor using the following command:

```
sudo systemctl restart apparmor
```

§

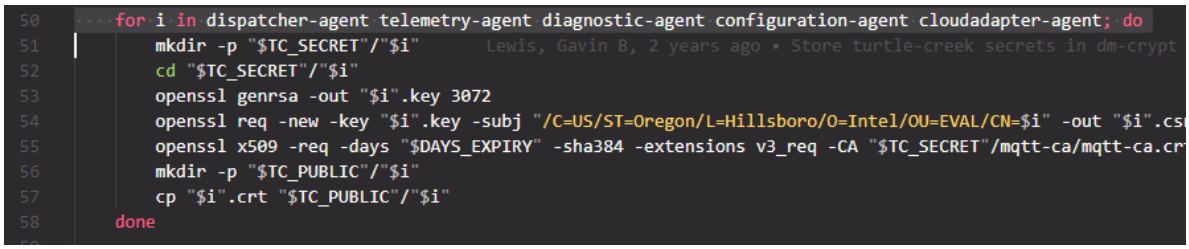
9.0 Creating a New Agent

The framework code base can be extended when there is a requirement to add an additional Agent to perform a designated task.

The following steps with examples given, provide a clear overview on how to setup a new agent.

1. A new folder with name **<agent_name>-agent** should be created under the *~/turtlecreek* directory.
2. Once the source code is added to the folder, mqtt keys need to be generated for the new agent created. To generate mqtt-keys,

   Go to *~/turtle-creek/fpm/mqtt/template/usr/bin/mqtt-ensure-keys-generated* file, and add the new agent name in the for-loop(line 50) shown in the below image.



3. These certificates are stored in

   ```
   /etc/intel-manageability/secret/<agent_name>-
   agent/<agent_name>-agent.crt
   ```

   And the respective keys are stored in

   ```
   /etc/intel-manageability/secret/<agent_name>-
   agent/<agent_name>-agent.key
   ```

4. The above-mentioned paths must be used within the agent code to make sure keys and certs are being pulled in from the right location.
5. Once the code is ready to be built, a service file created for the agent should include the correct group name.

   ```
   ~ /turtle-creek/<agent_name>-agent/fpm-template/etc/systemd/
   system/<agent_name>.service
   ```

   An example of the *dispatcher.service* file located at

   ```
   ~/turtle-creek/dispatcher-agent/fpm-template/etc/systemd/
   system/dispatcher.service
   ```

is shown below highlighting its respective group name.

```
[Unit]
Description=Dispatcher Agent Service
Requires=network.target mqtt.service
After=multi-user.target mqtt.service

[Service]
# ExecStart command is only run when everything else has loaded
Type=idle
User=root
EnvironmentFile=-/etc/environment
EnvironmentFile=-/etc/dispatcher.environment
ExecStart=/usr/bin/dispatcher
RestartSec=5s
Restart=on-failure
WorkingDirectory=/etc/systemd/system/
Group=dispatcher-agent

[Install]
WantedBy=multi-user.target
```

§

10.0 Issues and Troubleshooting

10.1 OTA Error Status

Error Message	Description
COMMAND_FAILURE	Diagnostic agent checks fail to run properly or if diagnostic agent/ config agent is not up when contacted. {'status': 301, 'message': 'COMMAND FAILURE'}
COMMAND_SUCCESS	Post and pre-install check go through. {'status': 200, 'message': 'COMMAND SUCCESS'}
FILE_NOT_FOUND	File to be fetched is not found. {'status': 404, 'message': 'FILE NOT FOUND'}
IMAGE_IMPORT_FAILURE	Image is already present when Image Import is triggered. {'status': 401, 'message': 'FAILED IMAGE IMPORT, IMAGE ALREADY PRESENT'}
INSTALL_FAILURE	Installation was not successful due to invalid package or one of the source file, signature or version checks failed. {'status': 400, 'message': 'FAILED TO INSTALL'}
OTA_FAILURE	Another OTA is in progress when OTA is triggered. {'status': 303, 'message': 'OTA IN PROGRESS, TRY LATER'}
UNABLE_TO_START_DOCKER_COMPOSE	Docker-compose container is not able to be started or spawned etc. {'status': 400, 'message': "Unable to start docker-compose container."}
UNABLE_TO_STOP_DOCKER_COMPOSE	Docker-compose down command was not successful. {'status': 400,
	'message': "Unable to stop dockercompose container."}

UNABLE_TO_DOWNLOAD_DOCKER_COMPOSE	Docker-compose downloaded command failed. {'status': 400, 'message': "Unable to download docker-compose container."}
XML_FAILURE	Result of bad formatting, missing mandatory tag. {'status': 300, 'message': 'FAILED TO PARSE/VALIDATE MANIFEST'}

10.2 Dispatcher-Agent Not Receiving Messages

If the dispatcher-agent does not receive the manifest message from the *cloudadapteragent* after triggering SOTA/FOTA, the current workaround is to remove *mosquitto.db*. This will remove the messages in the MQTT queue:

```
1. sudo systemctl stop mqtt
2. sudo rm /var/lib/mosquitto/mosquitto.db
3. sudo systemctl start mqtt
```

§