WESTERN POWER DISTRIBUTION

OPEN LV

OPENING UP THE SMART GRID

LV Common Application Platform

Public API

ea technology

RIIO NIC NETWORK INNOVATION COMPETITION

Report Title:	LV Common Application Platform Public API
Report Status:	For Publication
Project Ref:	Drawing: 2383 MANUL V04.03.03
Date:	18.10.2017

Document Control

	Name	Date
Prepared by:	Richard Ash	29.09.2017
Reviewed by:	Tim Butler	18.10.2017
Recommended by:	Richard Potter	18.10.2017
Approved:	Dan Hollingworth	18.10.2017

Revision History

Date	Issue	Status
18.10.2017	1.0	For Publication

DISCLAIMER

Neither WPD, nor any person acting on its behalf, makes any warranty, express or implied, with respect to the use of any information, method or process disclosed in this document or that such use may not infringe the rights of any third party or assumes any liabilities with respect to the use of, or for damage resulting in any way from the use of, any information, apparatus, method or process disclosed in the document.

Telephone +44 (0) 1332 827446. E-mail wpdinnovation@westernpower.co.uk

Contents

Table of figures

Table of tables

1. Introduction

The Common Application Platform for LV Networks (LV-CAP) is a software environment which facilitates the implementation of the Smart Grid at the lower distribution voltages. To drive down the cost of deploying Smart Interventions, the platform allows multiple algorithms to be deployed to one set of measurement and data processing hardware. The platform allows these algorithms to be designed and produced by independent third-party developers and packaged as stand-alone Applications which can be easily deployed by the distribution network operator without requiring bespoke software development.

This document details the Application Programming Interface (API) for developers intending to write Applications to run on LV-CAP. LV-CAP uses Docker to overcome dependency problems for third party developers, and helps to maintain and manage containers. It uses a MQTT messaging system for the communication of running containers and has a data storage functionality to persist data. This document has details on how a third-party Application can be set-up, run and interact with the core services on the platform.

2. Glossary

Table 1: Glossary of Terms

Term	Description
ACL	Access Control List, a list of the resources which a specific client may access. Used to control access to topics on the **MQTT broker**.
API	Application Programming Interface – a set of defined interfaces to be used by application developers.
APID	See **Application ID**.
Application	A **Docker Container** suitable for use with LV-CAP in accordance with this API document. All LV-CAP **Applications** are Docker Containers, but not all Docker Containers are suitable for use as LV-CAP Applications.
Application ID	The unique identifier for a specific version of an Application, by combining the **Vendor, Application Name** and **Application Version**. See Section 4.2
Application Name	This is a string which identifies an **Application**. This is chosen at will by the Application developer. See Section 4.2.
Application Version	A string which indicates the version of **Application** in a **Docker Image**. Decimal points may be used to separate version numbers, e.g. 1.2.3. This is chosen by the Application developer. See Section 4.2.
BLOB	Binary Large OBject, a SQL database field which can store an arbitrary array of binary data.
Container Manager	The main process which controls all LV-CAP **Applications**.
Docker	Open source program that allows Linux applications and their dependencies to be packaged as a **Docker Image**.

Docker Container	An isolated environment in which a **Docker Image** is run under **Docker**. Multiple Docker Containers may be created from a single Docker Image and run simultaneously.
Docker Image	A file system image containing a packaged Linux program (with its dependencies) which can be deployed to run on a **Docker** system.
GUID/UUID	A Version 4 GUID/UUID is a universally unique 48-byte identifier which is generated using random numbers. Example:- 821b8e33-4eaa-480e-b205-30fa9572af1a
IID	See Instance ID
Instance	One running copy of an **Application**, which is separate from any other copy of the Application, and has its own independent configuration. See Section 4.2.
Instance ID	String identifying a specific **Instance** of an **Application**, which is unique only on a given LV-CAP system. See Section 4.2.
LV	Low Voltage. Used in this context to refer to the Low Voltage electricity distribution network which delivers power to domestic and commercial customers at 400/230V AC.
LWT	Last Will and Testament, in **MQTT** a message to be sent when to subscribers when a publisher disconnects unexpectedly.
MQTT	(Message Queue Telemetry Transport) a publish subscribe based lightweight messaging protocol, used on top of the TCP/IP protocol.
MQTT Broker	Process which is responsible for distributing messages to interested clients based on the topic of a message. The LV Common Application Platform runs a private instance of an MQTT Broker
MQTT Topic	Identifier within an **MQTT** message used by the broker to allow filtering and direction of messages. All messages are published to a topic, and clients receive them if they are subscribed to the topic.
Vendor	A string which identifies the developer of an **Application**. These are allocated by EA Technology to each party creating **Applications** to run on LV-CAP.

3. Platform Overview

The LV Common Application Platform (LV-CAP) provides a framework for measurements to be made, processed through algorithms, and actions taken based on the results (Figure 1). All of these functions may be undertaken by Applications developed by EA Technology or third parties. LV-CAP provides a number of core services for third party Container developers to utilise. These are:

1. Container management (installation, configuration, starting and running of Applications, including multiple copies and versions.).
2. A Data Marketplace which allows all Applications on the platform to communicate with each other in a uniform manner.

3. A Data Storage mechanism which allows Application outputs to be stored for future use.

All other functionality is provided by Applications, but using standard interfaces so that different implementations can be swapped in and out without affecting other Applications. To achieve this Applications do not communicate directly but rather via the Data Marketplace using the messaging API described in Section 8. This is shown in Figure 2.

A key piece of the provided framework is the Container Manager. The Container Manager has ultimate control over the entire system ensuring everything runs as expected. Apart from the Container Manager, all core services on the platform run as Docker containers which the Container Manager is responsible for starting, stopping and updating. All Applications are packaged within Docker containers which the Container Manager will again start, stop and manage. A Docker container contains a GNU/Linux application and all its library dependencies except the Linux kernel itself. This allows each Docker Image created to be portable, easily updated and independent.

The Container Manager utilises the functionality within Docker to limit and share resources of a running container. This control allows the Container Manager to manage platform resources, giving Applications their requested resources and preventing them from consuming excess resources and starving others of resource. Developers of third party Applications must be aware that their application cannot use the entire resources of the system and that it must share processor, RAM and storage with other Applications running on the system.

As well as managing the start-up and shutdown of Applications, the Container Manager is responsible for ensuring that updated configuration files are delivered to the relevant containers, and that updates to containers are applied. Finally, it checks that Applications are still running correctly, handling any errors returned from Applications and dealing with Applications that have ceased to operate correctly.

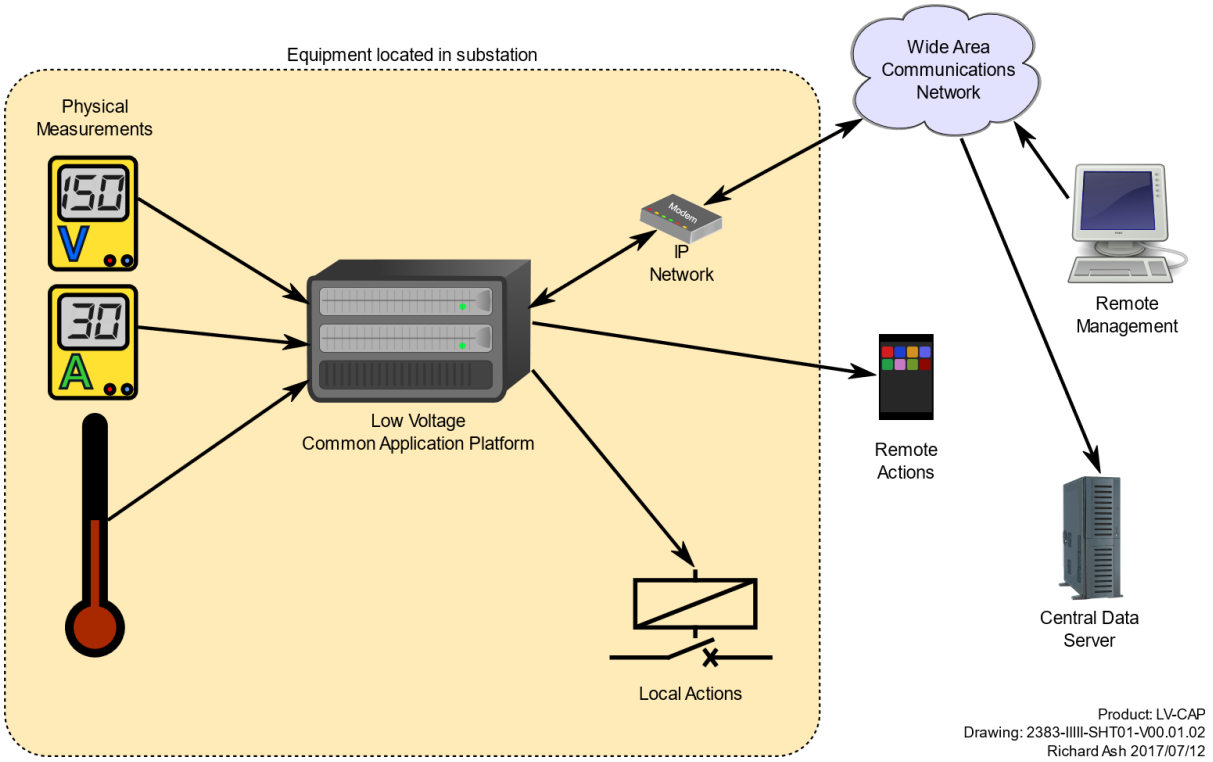

Figure 1 - LV-CAP System Concept

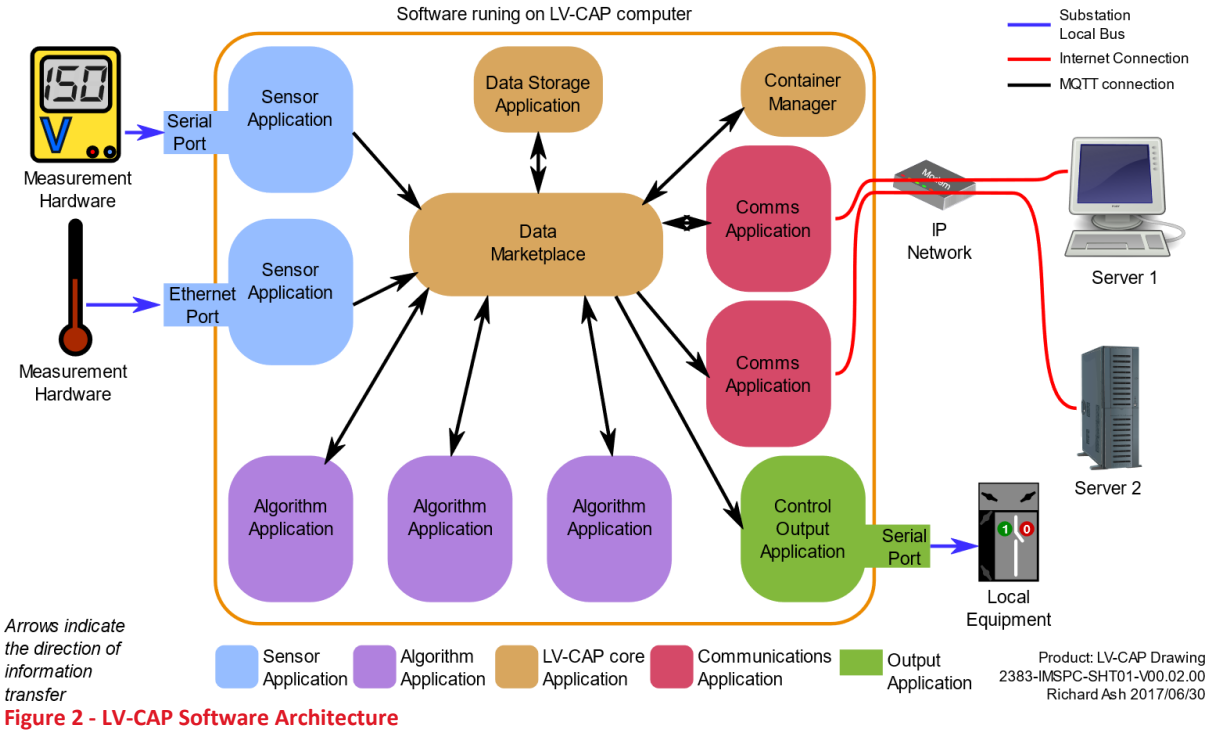

Figure 2 - LV-CAP Software Architecture

All communications between Applications in LV-CAP take place through the Data Marketplace (Figure 2). This uses Message Queue Telemetry Transport (MQTT) to transport messages. An MQTT broker, Mosquitto, is supplied as part of LV-CAP and is used by both core services and third-party containers. The message protocol for communicating on the MQTT broker and connection settings to the broker are documented in Section 8 of this document. Access

Control Lists (ACLs) are used on the MQTT broker to secure it, preventing Applications from publishing on and subscribing to topics they should not. The ACLs are automatically configured by LV-CAP when a new Application is added to the system.

LV-CAP systems are configured with an internal IP network. The Data Marketplace operates on this internal network and all Applications are automatically connected to it. Applications are only connected to external networks when there is a clear requirement for such a connection, and the system administrator has permitted it.

The Data Storage Application provides a database connected to the Data Marketplace. As well as being used by the Container Manager, it stores the outputs of Applications so that they can be subsequently retrieved for external communication or further processing.

Applications running on LV-CAP will generally fulfil one of four roles. Some Applications may fulfil more than one role at the same time.

1. Sensor Applications are responsible for reading data from physical sensor hardware. The data read is sanity checked and published to the Data Marketplace in a standard format. The data is then available to any other Application to subscribe to. The set of sensors provided for any given LV-CAP installation, and hence the Sensor Applications required, will vary depending on the user's requirements. The data format is independent of the measurement hardware so that different supplier's hardware can be used without software alterations outside the related Sensor Application.
2. Algorithm Applications consume data from one or more sensors and perform calculations upon it, for instance calculating the real-time temperature of a Transformer or forecasting the localised demand for energy. The Applications read from the Data Marketplace and publish their outputs back to the Data Marketplace.
3. Output Applications are the mirror image of Sensor Applications. They respond to information on the Data Marketplace (created by Algorithm Applications) by controlling physical hardware connected to the LV-CAP system, for instance carrying out network switching or energy storage.
4. Communications Applications connect the LV-CAP platform to the outside world. LV-CAP provides an IP communications link to the outside world, which Communications Applications use to upload and download data. A Communications Application uploads selected data values from the Data Marketplace to a central data server, or downloads Application images and configuration files from a central management server.

The default Communications Application is provided by Nortech Management Ltd. to communicate with their iHost server product.

4. General Principals

This section includes some principals which have driven the design and operation of the LV-CAP system. An understanding of these will make it easier to navigate and comprehend the rest of this specification.

4.1 Architecture

LV-CAP is designed to work as a loosely-coupled data processing pipeline, in which measurement data from Sensor Applications feeds into one or more Algorithm Applications. The outputs of these Algorithm Applications may feed other Algorithm Applications. Ultimately data reaches either a Communications Application to be sent to an external system, or an Output Application to take local actions on the Smart Grid (Figure 3).

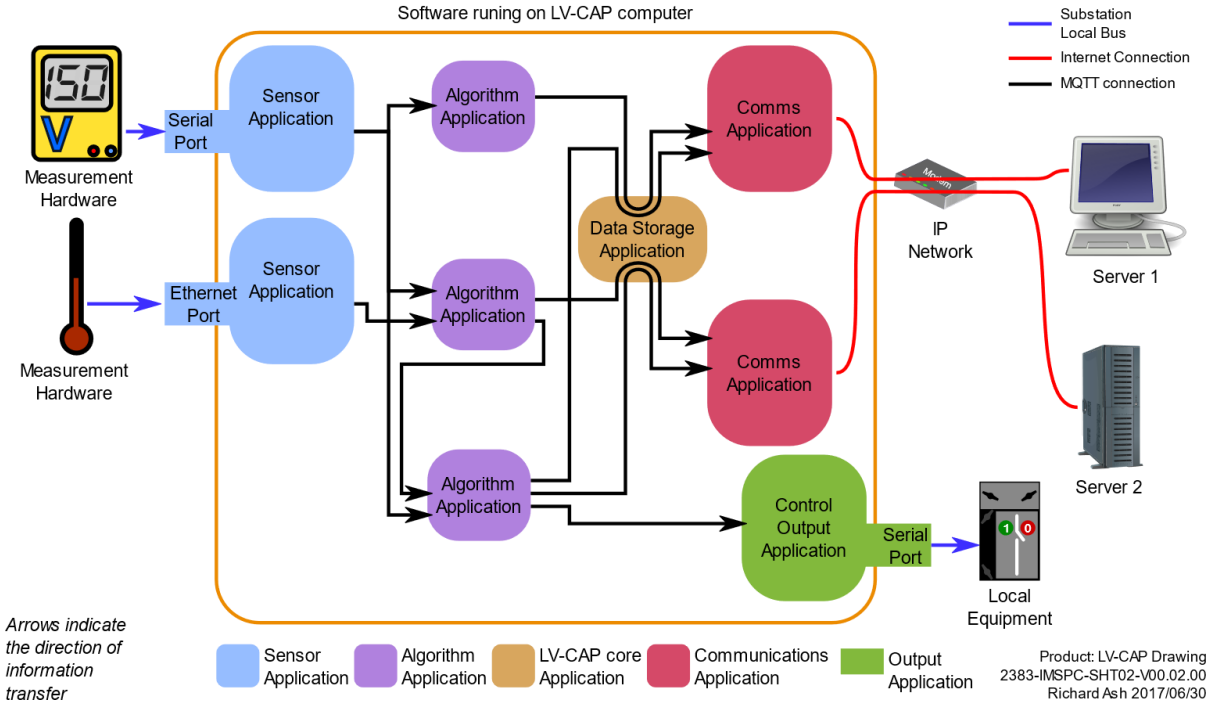

Figure 3 - Data Flow through an example LV-CAP system

Data is pushed through the pipeline from the Sensor Applications towards the outputs. The pipeline runs in approximately real time, although this is not enforced as in a true real-time system. If the workload of the LV-CAP system temporarily exceeds the available processing power then the system will lag behind before catching back up when resources allow.

The pipeline forms a tree structure, with each node being an input or output on a single topic in the Data Marketplace. Any Application may subscribe to any topic to make use of the data found there, with the delivery of messages to the various destinations handled by the MQTT broker. The expectation is that Applications will generally output onto fixed topic names (within their allocated sub-tree), whilst being freely configured (via the Configuration API) to read from whichever input topics the system operator requires.

The MQTT broker forming the Data Marketplace will only buffer a single message on each topic, so Containers must handle their input messages sufficiently quickly to be ready when the next message arrives on a topic. The Data Marketplace does not perform any rate adaption, so Applications need to be prepared to receive their input messages at whatever intervals the upstream Application produces them.

4.2 Application Identification

Applications must have uniquely defined identifiers. These fulfil a number of roles:

- To ensure that Applications will never encounter a name "clash" with another Application.
- To allow multiple copies of the same Application to run simultaneously, with separate configuration settings.
- To allow different versions of the same Application to be installed and run simultaneously.
- To allow system operators to unambiguously specify what Applications are to be run on any given system.

In this section, each word in **bold** is defined in the Glossary at the start of this document. To facilitate the selection and operation of **Applications**, each **Application**'s **Docker Image** has three unique pieces for information associated with it:

1. A **Vendor** string. This is a string which identifies the developer of the **Application**. These are allocated by EA Technology to each party creating **Applications** to run on LV-CAP.
2. An **Application Name** string. This is a string which identifies the **Application**. This is chosen at will by the **Application** developer. It should not container version information.
3. An **Application Version**. This is a string which indicates the version of the **Application**. Decimal points may be used to separate version numbers, e.g. 1.2.3. This is chosen by the **Application** developer.

This information enabled a system operator to specify exactly what **Application** they wish to run on LV-CAP. There are a number of constraints on the above fields which must be satisfied when they are chosen:

- The **Vendor** and **Application Name** must be valid Docker Names (see Reference 4):
  - Composed of valid ASCII characters.
  - Restricted to lower case letters, digits and hyphens (no underscores or periods).
  - May not start with a dash.
- The **Application Version** has the same restrictions as the **Vendor** and **Application Name** with the addition that periods are permitted in the **Application Version** only.
- Each release or update of an Application must have a unique combination of **Vendor**, **Application Name** and **Application Version**.

- For an Application to be successfully updated, the update must have an **Application Version** which Docker considers to be different to the existing Application's **Application Version**.
- For compatibility, the total length of the **Vendor**, **Application Name** and **Application Version** must be less than 44 characters.

The **Vendor**, **Application Name** and **Application Version** are combined to form the **Application ID** <**APID**>. When used as a file name or **Topic Name** then these sections are separated with an underscore:

`<Vendor>_<Application Name>_<Application Version>`

When used as a tag for a **Docker Image** then they are combined according to the usual docker convention:

`<Vendor>/<Application Name>:<Application Version>`

The <**APID**> identifies a specific Application executable in a globally unique manner. In the future this will be enforced through the digital signing of **Application Images** and their <**APID**>.

When creating the Docker Image, these fields are specified to the -t option of the docker build command as follows:

`docker build -t <Vendor>/<Application Name>:<Application Version>.`

When an **Application** is to be executed on LV-CAP, a **Docker Container** is created from the **Docker Image**. Each **Docker Container** must have a unique name, and we must support creating multiple Containers from one Image. To enable this a fourth field is used, which is the **Instance**. Instance is a two digit number which is unique to this Instance of an **Application** on the LV-CAP system. **Instance** values are set up in the **Container Manager** configuration by the system operator. The **Instance** value of "00" is special and reserved for use by Applications which cannot have more than one instance running. A single instance of any other Application may use any other value between "01" and "99". There is no requirement for **Instance** numbers to be contiguous, and their numeric value has no significance.

The **Vendor**, **Application Name** and **Instance** are combined to form the **Instance ID** (abbreviated in this document as <**IID**>) in the form

`<Vendor>_<Application Name>_<Instance>`

The <**IID**> identifies a specific instance of an **Application**, which is unique only on a given LV-CAP system, and may use any (specified) version of the **Application**. This is set up through the **Container Manager** configuration file.

Each Application Instance **Docker Container** created on LV-CAP will have the container name in Docker set to the <**IID**>.

The **Application Version** is deliberately omitted from the **Instance ID** so that the name does not change when newer versions of the **Application** are deployed. The **Instance ID** <IID> of a container is used as its handle, to identify the Container's area of file system space, MQTT topic namespace and so on.

4.2.1 Legacy Applications

Applications developed against older versions of this **API** (and the Innovate UK project) were identified by a single **GUID**. This 48-byte opaque string served as both **Application ID** <APID> (although it lacked version information) and **Instance ID** <IID> (although it lacked instance numbers). Applications using this form can still run one instance, using their legacy identifiers.

For these legacy containers, two instances of the same Container with the same GUID will never be run on the same LV-CAP installation. The GUID of a container was used as its handle, to identify the Container's area of file system space, MQTT topic namespace and so on.

4.3 Message Serialisation

All messages transmitted via the Data Marketplace must be serialised in JavaScript Object Notation (JSON). Adding additional white space to JSON payloads to 'pretty print' them is discouraged. All messages sent via the MQTT Broker must be valid JSON.

Standardised JSON object structures should be used wherever possible to maximise interoperability. These are defined in Section 9 of this document.

4.4 Topic Names

All messages exchanged through the Data Marketplace are published on MQTT topics. Whilst this API sets out specific topics for some purposes (e.g. interactions with the Container Manager) it is up to Application authors to choose suitable topics (and especially sub-topics) for the messages which their container produces. In order to use the standard JSON object structures defined in Section 9, unchanging information about the value has to be encoded in the topic name, rather than in the JSON payload itself. This also reduces the transmission of redundant (invariant) information where communications bandwidth is limited.

The MQTT standard itself places few restrictions on the choice of topic names, apart those specified in Section 4.7 of the standard. Within LV-CAP the following restrictions must be complied with to ensure reliable inter-operability between Applications.:

- MQTT topic names shall consist only of ASCII characters (no UTF-8)
- Topic names shall only contain upper and lower case letters, digits, hyphens and underscores
- Spaces are not permitted topic names.
- The maximum length of a complete topic name shall be 220 characters, including '/' separators.

The following additional guidance should be borne in mind when selecting names for MQTT topics, both in Application development (where topic names are fixed) and in system configuration (where topics are defined in the Application configuration file):

- Use topic levels to separate sections of your topic name. E.g. "output/transformer/forecast/4h/capacity" not "output/transformer-forecast_4hCapacity".
- Design in extensibility – it will be disruptive to change existing topic names to allow additional data to be published (e.g. more channels or intermediate calculation results).

- Avoid site-specific topic names like "Melrose Avenue North", which will have to be changed for each site, in favour of generic labels applicable to similar sites like "Feeder 1".
- Use of CamelCase offers more economy in topic characters than separation_with_underscores.

To make it easier for system operators to understand what messages on a topic mean, the following general form of topic names is recommended:

<subtree>/<asset>/<parameter group>/<time>/<parameter>

Not all components will be required for a given topic name and may be omitted. This results in topic names like:

algorithm/data/0ca2eadb-b128-4dff-9bd7-cbb15e21b8b1/Number1Tx/state/hst

sensor/data/eatl_modbusrtusensor_01/Number1Tx/load/A

It is expected that Applications will be flexible and configurable in respect of the topics they subscribe to (inputs), but have fixed or generated topic names for publication (outputs).

4.5 Units

All messages transmitted should have a timestamp (as shown in the preferred JSON formats in section 9). These timestamps are 64-bit UNIX timestamps, defined as the number of seconds since 1st Jan 1970 UTC. Where sub-second resolution is required, the fractional value should be stored as a separate field.

Wherever possible, Applications should use the time stamp fields from incoming messages in preference to referencing system time (explicitly or implicitly). This will make it much easier to test Applications in a reproducible manner by simply replaying a fixed sequence of input messages, regardless of the relationship between message time stamps and system time.

Values are always given in the base SI unit for the quantity being measured or calculated. For example, current is always given in Amps, never in milliamps or kiloamps. Temperatures are given in degrees Centigrade rather than Kelvin (in accordance with common engineering practice).

Metadata for the display of values may be passed between containers via Data Series Metadata Objects described in Section 9.4.

4.6 Text Encoding

UTF-8 is the preferred method of encoding text.

When including non-English text in JSON strings bear in mind that that the double-quote character must be escaped with a backslash, and other escape sequences are used for newline etc. control characters, as per the JSON specification.

4.7 Data Persistence

Applications have two options for persisting data:

1. Data which is output to the Data Marketplace can be stored in the Data Storage Application (Section 7). Any Application can then retrieve this data in the future (up to a time limit imposed by the removal of old data values).
2. Each Application is assigned a filesystem volume. This file system is private to the Application and not visible to any other container on the system. Data can be stored here by the Application, e.g. to save system state or history.

The rest of the Docker Container environment is ephemeral and will be lost when the Application is re-started, either by the Container Manager or because the whole LV-CAP platform is rebooted.

4.8 Data Flow and Valid Flags

LV-CAP is designed to work on the basis that data keeps flowing through the processing pipeline at all times. To support this, the standardised JSON object structures in Section 9 all contain a Valid key. When correct data is not available or cannot be calculated, Applications should continue to output messages to the Data Marketplace in the normal manner, but with the Valid key set to false. Applications subscribed to the topics will then be made aware that time is moving on, but that there is a problem with the data source.

Containers which fulfil the Sensor Application role (Section 3) should continue to output messages at the configured interval under all circumstances. This includes if the sensors are disconnected or producing out-of-range values. When data is not available or out-of-range, the "Valid" member in the output should be set to false. The actual sensor value sent in this case does not matter, because subscribed containers should not use the value when "Valid" is false. The timestamp field must be updated so that the subscribed containers can keep track of time.

Algorithm containers receiving input messages with Valid set to false should not use the Value in the received message, but may rely upon the time stamps. When the input timestamps reach the point that the algorithm is due to provide output, it must do so. It is up to the Application author to decide if there is sufficient Valid data to produce an output or not. If there is insufficient Valid data to produce a new result then the container should output, setting Valid to false and using the timestamp from the most recent input message (whether that message is valid or not).

4.9 Data Priority

The standard JSON formats described in Section 9 provide for a Priority field. This allows the upload of certain messages to be prioritised by Data Upload Applications, based on policy set by the system operator and priority information from Application authors.

Valid Priority values are integers between 1 and 5. A Priority value of 1 is the highest priority and 5 is the lowest priority. Any other Priority value is not valid and is treated the same as if Priority is not specified. These messages with no specified Priority have lower Priority than all messages with a specified Priority.

The data APIs in sections 8.4 and 8.5 support query by Priority.

5. Start-up Procedures

5.1 LV-CAP System Start

5.1.1 Start-up of Core Services

As discussed in section 3, the LV Common Application Framework consists of a number of core services for third party containers to use. These have a defined start-up order and only once these have all started up will any other container be started. The Core Services are started in the following order:

1. Container Manager
2. Data Marketplace
3. Data Storage Application

5.1.2 Start-up of Remaining Applications

The remaining Applications installed on a given LV Common Application Platform will be started automatically, once the platform's framework has successfully started up and entered the running state. All other Applications must be independent of each other (there is no concept of inter-Application dependencies) so that Applications can start in any order. Applications will be started by the Container Manager in order of their installation date (i.e. order of when they were added to Docker's available image list).

5.2 Application Start

Each Application on the LV Common Application Platform must perform certain actions when it is started by the Container Manager. Failure to do so is likely to result in the Application being shut down by the Container Manager.

Upon starting, a third-party Application must perform the following actions in the given order:

1. Connect to the Data Marketplace (see Section 8.1 for the MQTT Broker connection details).
2. Subscribe to topics listed in Table 2**Error! Reference source not found.**
3. Send a configuration request message to the Container Manager via the Data Marketplace.
4. Wait until a response is sent back by the Container Manager. This response will either contain the Application's configuration, or will include an error message if the Container Manager is not aware of any configuration for the Application.
5. If configuration data is received, the Application should process the configuration and apply it internally.
6. If the configuration is valid, the Application can start operating, sending a status update to the Container Manager indicating all is OK.
7. If no configuration is available or there is an error in the configuration, the Application must send a status update to the Container Manager indicating an error:

- Sending STATUS_INITIAL will result in the Container Manager re-sending the configuration file, the Application should stay in a non-operating state awaiting configuration.
- Sending STATUS_ERROR and will cause the Container Manager to restart the Application. See Section 8.2.2 for more information on status messages.

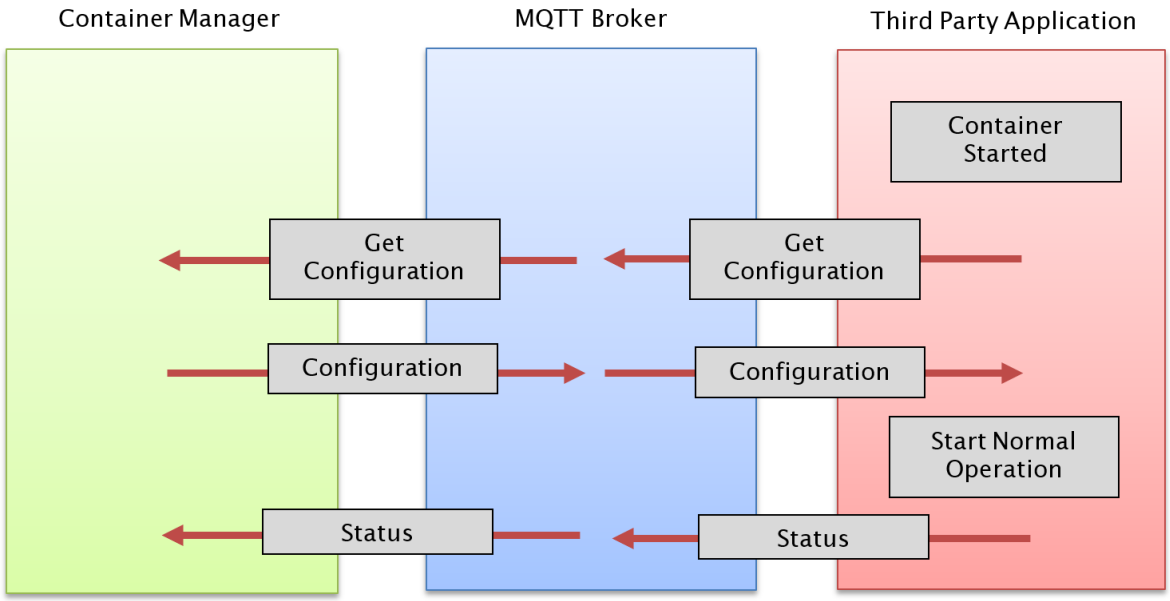

Figure 4 - Application start-up procedure

The above diagram shows the Application start procedure for a successful start.

Due to the Application start order (Section 5.1.2) it is possible that once a container starts its 'normal operation', other Applications it may want to communicate with may not yet be operating. In this situation the Application will have to wait until any dependencies are running. This is not normally a problem because the MQTT protocol allows publishing and subscription to occur in any order, with no requirement for topics to be configured or created in advance.

5.3 Required Subscriptions for all Applications

All Applications must subscribe to the following MQTT message topics in order to interact correctly with the Container Manager and remain running on the platform.

Table 2 - Required Subscriptions by Third Party Applications

Topic	Purpose
status/request	Receives status requests from the Container Manager. Non-response to two consecutive status requests will lead to the Application being restarted without notice by the Container Manager.
config/response/<IID>	Receives Application configuration sent by the Container Manager. The IID is the Application's own unique IID as in section 4.2
command/<IID>	Integer, the command to execute.

6. Shutdown Procedure

Similar to the start-up procedure (Section 5**Error! Reference source not found.**), the LV C ommon Application Platform has a defined shutdown procedure. This shutdown procedure is designed to allow Applications to shut down in a safe manner and avoid any data loss or corruption. The shutdown procedure not only applies to shutting down of the entire platform, but also occurs when an updated Application image is applied. Applications can request their own shutdown if required, but all Applications shall respond to a shutdown request from LV-CAP.

When LV-CAP requests shutdown of an Application, the procedure is defined as:

1. Container Manager sends a shutdown command to the Application via the MQTT broker (see section 8.2.3).
2. The Application handles the notification and performs its own internal shutdown procedure, which may include writing any data to disk, stopping all MQTT subscriptions including that of status requests, and any other work to perform a clean shutdown. Once completed, the Application must respond to the Container Manager using a *status/response* message with the STATUS_SHUT_DWN status.
3. The Container Manager will then shut down the container. If the Container Manager does not receive a status message from the Application to be shutdown which includes the STATUS_SHUT_DWN status for more than 1 minute, the container will automatically be shut down.

If the Application fails to shut down within the “status/request” (default 2 minutes) interval then Container Manager will shut it down forcibly by terminating the process.

In the event of an Application requesting its own shutdown by the Container Manager, the procedure is defined as:

1. The Application performs its own internal shutdown procedure, which may include writing any data to disk, stopping all MQTT subscriptions including that of status requests, and any other work to perform a clean shutdown.
2. Once completed, the Application must send a status to the Container Manager using a *status/response* message with the STATUS_SHUT_DWN status.
3. The Container Manager will then shut down the Application, and added to the stopped Application list. It will not be run again until the Container Manager configuration is altered or the Container Manager is re-started.

If an Application needs to be re-started by the Container Manager, the procedure is:

1. The Application prepares for being restarted, which may include writing any data to disk, stopping all MQTT subscriptions including that of status requests, and any other work to perform a clean restart.
2. Once completed, the Application must send a status to the Container Manager using a *status/response* message with the STATUS_RESTART status.
3. The Container Manager will then shut down the Application and start it back up again immediately.

7. Data Storage

The Data Storage Application allows persistence of data from Applications. Data stored in the Data Storage Application can also be configured by the system administrator to be uploaded by one or more Communications Applications. In this role, the Data Storage Application acts as a buffer so that data is uploaded to its destination reliably, even in the face of unreliable communications links.

The data stored on the platform will be placed in a database which can be accessed via the Data Marketplace (see section 8.3). The output of each Application Instance <IID> will be stored in its own table. This table is created when the Application Instance is first created by the Container Manager. All tables created for Applications will store records with the format documented in Table 3

Table 3 - Containers Table schema

Field	Type	Description
ID	Opaque Integer	Each record stored will be assigned an ID by the Data Storage Application. This integer will be unique amongst the records currently stored in the Data Storage Application, but may be re-used over the life of the LV-CAP system as old data is purged from the database and new records added. There are no guarantees about the numeric value of this identifier.
Timestamp	Integer	The Unix timestamp at which the record was added to the Data Storage Container.
SubTopic	String	Part of the MQTT topic string on the Data Marketplace from which the record came will be stored in this field. Because tables are allocated by the Application Instance ID, the topic string up to the IID would be the same for all records. The common string is not stored, leaving only the Application's sub-topic string. If no sub topics are present this field will be null.
Data	BLOB	This is the MQTT JSON message sent to be stored. It is stored as a BLOB so that no alterations are made to the JSON object.

The payload an Application sends to be stored will retrieved unaltered from the Data Storage Application. The Data Storage Application will set the values of the other fields automatically. The API for retrieving records is documented in Section 8.5. Applications are strongly encouraged to output their data in one of the standard JSON payload formats described in Section 9.

8. Data Marketplace API

The main form of communication between Applications and the LV Common Application Framework is via the Data Marketplace. This section documents the MQTT message topics, their associated payload and includes examples of message payload. The API is broken up

into sections according to the Application roles (Section 3) expected to use them. Applications may (and will) use methods from more than one section of the API.

8.1 MQTT Broker

Each container wishing to operate on the LV Common Application Framework must connect and communicate using the provided MQTT broker. The LV-CAP system uses a secured MQTT broker, in order to support authentication of Application when they connect to the Data Marketplace. The connections settings required are shown in Table 4.

Table 4 - Secured MQTT Broker Settings

Setting	Value
Hostname	marketplace
Port Number	8883
Encryption	TLS v1.2 or higher
Authentication	X509 client certificate
Username	Set to the Application's Application ID
Client ID	Set to the Application's Application ID

EA Technology will operate a TLS Certificate Authority for the LV-CAP system. All client SSL certificates must be signed by this Certificate Authority, which will be trusted by the Data Marketplace. This Certificate Authority certificate will be issued to Application developers for inclusion in Application at build time, so they can authenticate the Data Marketplace.

Client certificates will be signed on request by the certificate authority, with the Common Name (CN) of the certificate set to the Application ID <APID> of the Application they are to be used by (see Section 4.2). This client certificate should be embedded in the Application so that it can be used to connect to the Data Marketplace. The client certificate and associated private key need to be embedded in the Application so that it can connect to the Marketplace. The private key should be encrypted to minimise the risk of it being extracted from the Application by a third party. These is no reason for EA Technology, or any other Application author, to know the Application's private key.

When the Application connects to the Data Marketplace it's certificate will be checked. If valid, and not revoked by the system operator, it will be allowed to connect. Access control lists will then allow the Application to publish on the topics set out in this API. In general, subscriptions will not be restricted.

A new certificate should be obtained whenever an updated version of the Application is produced. This both mitigates the fixed expiry date of certificates, and allows the certificate of specific Application versions to be revoked if the keys are compromised. This will also have to be done when an updated Certificate Authority Root Certificate is required.

8.1.1 Payload Descriptions

JSON does not have a concept of fixed-size (bit width) integers, however implementation in strongly typed languages is made much easier by defining the maximum size of integer fields wherever possible. In this documentation:

- Any key which is shown with type “Integer” will always fit into a 32-bit signed integer.
- Any key which is shown with type “Int64” will always fit into a 64-bit signed integer.

8.1.2 Security and Signing

The present implementation of LV-CAP provides only limited security between Applications, and so requires a high degree of trust in Application authors. To improve this situation in the future, an optional “signature” object has been added to all JSON payloads specified in this API. This member is reserved for the definition (in a future version of this API) of a mechanism for cryptographically signing each JSON payload.

The signing scheme is intended to use public (asymmetric) key cryptography. The source Application will sign all outgoing messages with a private key, which must be kept secret. Destination Applications receiving these messages can use the source Application’s public key (which does have to be kept secret) to verify that the messages received are indeed from the correct source container. It is intended that the public keys will be distributed to the relevant Applications via their configuration data.

A Application which does not implement signature verification will be able to receive future signed messages without modification, because it will ignore the signature object. Applications with signature validation implemented will have to decide on their policy for messages received without signatures.

At some future date, it may become mandatory to sign messages on some critical API topics when communicating with the LV-CAP core components. It will be up to other Application authors at what point they require signed input messages.

The signing of Docker Images will also be added in future to ensure that when system operators specify a particular Application ID <APID> (see section 4.2) only that specific version can be run.

8.1.3 Last Will and Testament

The MQTT broker supplied by the framework supports the Last Will and Testament (LWT) feature. This can be used to define, upon connection, a message which will automatically be sent by the broker to subscribers of the set topic upon the non-clean disconnection of a client. In order to manage the platform all Applications must provide a LWT on their status response topic (Section 8.2.2). The status response sent as the LWT must include the FAILED state within the payload. Applications may also set LWT’s on any topic they desire to inform others of their failed state.

8.1.4 Quality of Service

MQTT provides a Quality of Service (QoS) level feature, which defines how hard a broker or client will work to ensure a message is delivered. More details can be found in section 4.3 of the MQTT Standard.

MQTT QoS is a property of both the publishing and subscribing of a message, so a client can publish a message at any QoS and a client may subscribe to a topic at any QoS. The implemented QoS will be the lowest of the publishing and subscribing QoS levels. There are 3 QoS levels defined in MQTT:

- QoS 0 - At most once. The status request topic has a QoS of 0 as this regular heartbeat is not critical, and must be sent regularly.
- QoS 1 - At least once. The Container Manager sends out commands at QoS 1 as Containers can easily handle receiving the same command more than once.
- QoS 2 - Exactly once. This is used when querying the Data Storage Container, as multiple message delivery could have complex and undesirable affects upon the database.

8.2 LV-CAP Core API

The Core API is responsible for management of Applications. All Applications will need to use the Core API to register with and run on LV-CAP.

8.2.1 Configuration

The Configuration message topic is used to request and distribute configuration to Applications.

QoS: Messages on this these topics must be sent and received with QoS = 1. Applications must cope with multiple copies of their configuration information being delivered.

Retention: Messages sent on these topics must have the retention flag set to false.

Table 5 - Configuration of MQTT topics

Topic	Description	Sender	Receiver	Payload	Notes
config/request/<IID>	Message containing a request from a container to the Container Manager requesting it's configuration	Any Application	Container Manager	{ "Timestamp": <Int64>, "Signature": {} }	No required payload. **Timestamp**: (Optional) Standard LV-CAP timestamp (see Section 4.5) when the configuration was requested. Required in signed payloads to protect against replay attacks. **Signature**: (Reserved) See Section 8.1.2

config/response/<IID>	Message containing updated configuration for a specific Application Instance. Can be a response to a request, or a new set of configuration pushed to a Application Instance.	Container Manager	Application Instance with IID specified in topic name	{
				"Configuration":
				{
				"<Key_1>": <Value_1>,
				"<Key_2>": <Value_2>,
				"<Key_n>": <Value_n>
				},
				"Timestamp": <Int64>,
				"Signature": {}
				}
					Configuration: JSON Object read directly from the Application Instance configuration file. The structure will be different for each Application, as described in Section 9.5.
					Timestamp: (Optional) Standard LV-CAP timestamp (see Section 4.5) when the configuration was requested. Required in signed payloads to protect against replay attacks.
					Signature: (Reserved) See Section 8.1.2

8.2.2 Status

The Status topic is used by the Container Manager to request the status of running Applications. The Container Manager will periodically request the status, and running Applications must respond to the request to confirm that they are operating correctly.

If an Application does not respond or responds with a status other than STATUS_MSG_OK or STATUS_INITIAL (see Table 7), it is considered to have failed the request. After three successive failed status requests the Application will be restarted by the Container Manager. If the Container still fails further status requests to reach a total of 5 consecutive requests, it will be permanently shut down, and this error logged in the database.

QoS: Messages on this these topics must be sent and received with the QoS shown in Table 6

Retention: Messages on this these topics must have the retention flag set to false.

Table 6 - MQTT Status Topic

Topic	QoS	Description	Sender	Receiver	Payload	Notes
status /request	0	Message to request status from all running containers.	Container Manager	All Applications	{ "Timestamp": <Int64>, "Signature": {} }	No required payload. **Timestamp**: (Optional) Standard LV-CAP timestamp (see Section 4.5) when the status was requested. Required in signed payloads to protect against replay attacks. **Signature**: (Reserved) See Section 8.1.2
Status /response/<IID>	1	Message containing a status update from the Application Instance identified by <IID>.	Any Application	Container Manager	{ "Status": <Integer>, "Message": "<message>" "Timestamp": <Int64>, "Signature": {} }	**Status**: (Required) One of the values from Table 7. **Message** (Optional): If the Message string is present it will be sent to the error Database. **Timestamp**: (Optional) Standard LV-CAP timestamp (see Section 4.5**Error! Reference source not found.**) when the status was requested. Required in signed payloads to protect against replay attacks. **Signature**: (Reserved) See Section8.1.2.

The valid status response values are shown in Table 7.

Table 7 - Status Field Values

Status Value	Meaning
1	STATUS_MSG_OK – the Application is running normally.
2	STATUS_MSG_FAIL – the Application has failed. The Container Manager will restart the container. If the key “Message” is present in the JSON object it will be stored in the Data Storage Application as an error message.
3	STATUS_MSG_ERR – the same as STATUS_MSG_FAIL for backwards compatibility.
4	STATUS_SHUT_DWN – the Application has completed its shutdown procedures and is ready to be stopped by the Container Manager. The container will not be restarted unless the Container Manager configuration is altered or the Container Manager is re-started.
5	STATUS_INITIAL – the Application is waiting to receive its configuration (and can do nothing until it does). The Container Manager will resend the Application's configuration.
6	STATUS_RESTART – the Application wishes to be re-started. It has completed any shutdown procedures and saving of state and is ready to be stopped and started again by the Container Manager.

Status values other than STATUS_MSG_OK and STATUS_INITIAL are regarded as failure conditions. If the key “Message” is present in a JSON object with a failure status, the Message string will be stored in the Data Storage Application as an error.

8.2.3 Command

The MQTT command topic allows the Container Manager to send instructions to any running Application.

QoS: Messages on this these topics must be sent and received with QoS = 1

Retention: Messages sent on this topic must have the retention flag set to false.

Table 8 - Commands MQTT

Topic	Description	Sender	Receiver	Payload	Notes
command/<IID>	This is a command sent from the Container manager for the container to execute	Container Manager	Any Application	{ "Command": <Integer>, "Timestamp": <Int64>, "Signature": {} }	**Command**: (Required) One of the command values shown in Table 9 below. **Timestamp**: (Optional) Standard LV-CAP timestamp (see Section 4.5) when the command was issued. Required in signed payloads to protect against replay attacks. **Signature**: (Reserved) See Section 8.1.2.

The command values in Table 9 are currently specified. In the future, more values may be added, so all LV-CAP Applications must check the payload of the message received is the expected value.

Table 9 - Command Topic Command Values

Command Value	Command
1	Shut Down. Currently the only implemented command. All Applications must implement this command. This command is used when an updated Application is deployed. The Container Manager will send a shutdown command for the running Application to stop everything it is doing before re-starting the Application. See Section 6 for more details.

8.2.4 Error

The MQTT error topic allows all containers to log any issue or internal error.

QoS: Messages on these topics must be sent and received with QoS = 1.

Retention: Messages sent on this topic must have the retention flag set to false.

Table 10 - Report Error Topic Table

Topic	Description	Sender	Receiver	Payload	Notes
storage/data /error/<IID>	Topic to log any external or internal errors to storage.	All Applications	Data Storage Application	{ "Errno": <Integer>, "Message": "String", "Timestamp": <Int64>, "Signature": {} }	**Errno: (**Required**)** One of the errno values shown in the **Timestamp**: (Optional) Standard LV-CAP timestamp (see Section **Error! R eference source not found.**) when the command was issued. Required in signed payloads to protect against replay attacks. **Signature**: (Reserved) See Section 8.1.2

Table 11 - Errno Description Table

Command Value	Errno Name	Description
1	ERRNO_JSON_INVALID	Payload from MQTT failed to Parse. Invalid JSON.
2	ERRNO_IO	Input/output Error
3	ERRNO_ACCESS	Permission denied
4	ERRNO_NO_DEVICE	No device found
5	ERRNO_FILE_DIRECTORY	Directory not found

6	ERRNO_MQTT_SUBSCRIPTION	Failed subscription to MQTT Topic
7	ERRNO_MQTT_PUBLISH	Failed Publish, this is only used when trying to publish a payload to. If failed to publish an error message use std::out. This will be saved by the Docker Log files and can be accessed later by Admin.
8	ERRNO_APPLICATION	Process failed due to Application error. The message to accompany this Errno is mandatory.
9	ERRNO_CONFIGURATION	Failed processing the incoming Config. This is if the contents of the configuration expected does not match or has the wrong types. (This could also be ERROR_JSON_INVALID if it's not valid JSON)
10	ERRNO_MQTT_CABLLBACK	Error occurred in the MQTT Call back. This can be when setting up the call back or an error within the call back with an incoming message
11	ERRNO_SENSOR	This can have two applications. The first, for any Sensor Container that has an error with reading a sensor it can output this Errno with the relevant message. The second is for any Algorithm Container reading in the Sensor Payload and the Payload is valid but any of the Key types is incorrect.
12	ERRNO_NETWORK	Error accessing the network.
13	ERRNO_PORT	Error opening or accessing a port.
14	ERRNO_PROFILE	Any Algorithm Application expecting a Profile Payload, the Payload is valid but any of the Key types is incorrect.

8.3 Sensor Data API

Applications which fulfil the Sensor Application role (see Section 3) will publish on the topics in the Sensor Data API. Applications in the Algorithm Application role will often subscribe to these topics to obtain their inputs. This data will not normally be stored.

8.3.1 Sensor Readings

Topics for transferring sensor reading data collected and published by Sensor Applications.

QoS: Messages on this these topics must be sent and received with QoS = 1.

Retention: Messages on this these topics must have the retention flag set to false.

Table 12 - Sensor Reading MQTT messages

Topic	Description	Sender	Receiver	Payload	Notes
sensor/data/ <IID>/<sensorname>	New sensor readings	Sensor Applications	Any Application	Standard Scalar Object Format, Series Object Format or Co-ordinate Object Format.	See Section **9** for details of standard JSON formats.

See Section 4.4 for guidelines on choosing intelligible topic names for message output. Sensor Applications are responsible for publishing data and setting the Valid flag in messages (Section 9) in accordance with the guidelines set out in Section 4.8.

Readings will often be published at fixed time intervals. These intervals will start when the sensor Application receives its configuration, and so may not be aligned to "clock face" times. For instance, if the configuration was received at 09:05:00, setting a time interval of 20 seconds. The Sensor Application will output at 09:05:20 then at 09:05:45 and so on.

Depending on the properties of the sensor Application, there is a possibility that if many sensors have the same interval time and one sensor takes longer to read that this would delay the next sensor and so on. The start of the normal operation for the Sensor Application is most susceptible to this, however after a short time this will reach an equilibrium and each output will be at the prescribed interval. Applications consuming the messages must be equipped to cope with these timing variations.

8.3.2 Sensor Metadata

Topics for transferring sensor metadata published by Sensor Applications. Publishing on these Metadata topics is optional.

QoS: Messages on this these topics must be sent and received with QoS = 1.

Retention: Messages on this these topics must have the retention flag set to true.

Table 13 - Sensor Reading MQTT messages

Topic	Description	Sender	Receiver	Payload	Notes
sensor/data/ <IID>/<sensorname>	New sensor readings	Sensor Applications	Any Application	Standard Scalar Object Format, Series Object Format or Co-ordinate Object Format.	See Section **9** for details of standard JSON formats.

8.4 Algorithm Data API

Applications which fulfil the Algorithm Application role (see Section 3) will publish on the topics described in the Algorithm Data API. Application in the Algorithm Application role may subscribe to these topics to obtain inputs. Applications in the Output Application role will normally subscribe to one or more of these topics to obtain inputs.

Data published on these topics may be stored in the Data Storage Application, depending on the latter's configuration and the "ToStore" flag set by the publishing Application. Only stored data will be available for upload by Communications Applications.

QoS: Messages on this these topics must be sent and received with QoS = 1.

Retention: Messages on this these topics must have the retention flag set to false.

Table 14 - Algorithm Data Table

Topic	Description	Sender	Receiver	Payload	Notes

algorithm/data/<IID>/<subtopic>	The main topic an algorithm Application will publish its data on	Algorithm Application	Any Application	Any valid JSON object.
				
				Applications are strongly encouraged to use one of the standard JSON Object Formats to improve interoperability.
				{
				<Valid JSON Payload>
				}
					Algorithm Applications may output on any sub-topic starting with "algorithm/data/<IID>" (where <IID> is the Application Instance's assigned identifier).

When choosing the sub-topic on which to output data, authors are encouraged to use a descriptive topic name (Section 4.4**Error! Reference s ource not found.**). This makes configuring systems easier and less error prone. For example, transformer capacity forecasts for the available capacity in transformer T1 over the next 4 hours might be output on topic

algorithm/data/<IID>/T1/forecast/4h/capacity

If the JSON payload is to be stored in the Data Storage Application it must have a KEY "ToStore" and the value set to true. If this is not present or is set to false then the data will not be stored. Only stored data will be available for upload by Communications Applications.

Payloads should have a KEY "Timestamp" containing the Unix timestamp the calculation refers to. Where the calculation covers a range of time, this should be the time stamp of the most recent time covered by the calculation.

Algorithm Applications may use optional metadata subtopics in exactly the same way as Sensor Applications, as documented in Section 8.3.2.

8.5 Data Upload API

The Data Upload API provides a means to access the data queued for upload in the Data Storage Application. Applications which fulfil the Communications Application Upload role (see Section 3) will use this API extensively. Applications using this API must be explicitly authorised by

the system operator in the Data Storage Application configuration. A separate (virtual) queue is maintained for each Upload Application of data which is waiting for upload. Once a message has been uploaded the Upload Application must notify this fact back to the Data Storage Application via this API so that the queues can be updated.

This API operates on a pattern of separate topics for requests and response messages. When using this API, Applications should always subscribe to the response topic before publishing a request. This avoids a race between the response and the subscription which may cause the container to miss response messages.

All methods in this API work with the per-Application database tables described in Section 7. The SubTopic and Data columns are set from the received message. The other columns in the table will be set automatically by the Data Storage Application.

QoS: Messages on this these topics must be sent and received with the QoS shown in Table 15.

Retention: Messages on this these topics must have the retention flag set to false.

Table 15 - Communications Table MQTT

Topic	QoS	Description	Sender	Receiver	Payload	Notes
storage/request/newdata/<IID>	2	A request for new data to be uploaded by Upload Application <IID>. The request will search all tables in the database which the Application is permitted to upload from.	Upload Application with identifier <IID>	Data Storage Application	{ "MaxLength": <Integer>, "StartTime": <Int64>, "EndTime": <Int64>, "PreferOldest": <Boolean>, “InstanceID”: <IID> "SubTopic": <String>, "MinPriority": <Int>, "MaxPriority": <Int>, "Timestamp": <Int64>, "Signature": {} }	The request Payload keys are documented in **Error! Reference source not found.**.

storage/response/newdata/<IID>	2	The response to the above request.	Data Storage Application	Upload Application with identifier <IID>	{
					"Status": <Integer>,
					"Response": [
					{
					"TableName": <IID>,
					"TableRows": [
					{
					"ID":<Integer>,
					"Timestamp": <Int64>,
					"SubTopic": <string>,
					"Data": <JSON Object>,
					},
					{rowN}
					]
					},
					{
					"TableName": <IID>,
					“TableRows”: [
					{
					"ID":<Integer>,
					"Timestamp": <Int64>,
					"SubTopic": <string>,
					"Data": <JSON Object>,
					},
					{rowN}
					]
					}
					]
					"Timestamp": <Int64>,
					"Signature": {}
					}
						The response Payload keys are documented in **Error! Reference source not found.**.
						
						If an error occurs then the Payload will still have Status and Response members, but the Response array will be empty.

storage/uploaded/<IID>	1	Indicates that messages have been uploaded by <IID> and they should be removed from the upload queue.	Upload Application with identifier <IID>	Data Storage Application	{
					"NewData": [
					{"<IID>": [<Integer>, <Integer>]},
					{"<IID>": [<Integer>, <Integer>]}
					],
					"Timestamp": <Int64>,
					"Signature": {}
					}
						NewData: (Required) Array of objects, one for each table to be updated.
						<IID>: (Required) Array of opaque integer identifiers of the messages which have been uploaded.
						Timestamp: (Optional) Standard LV-CAP timestamp (see Section **Error! Reference source not found.**) when the update was sent. Required in signed payloads to protect against replay attacks.
						Signature: (Reserved) See Section **Error! Reference source not found.**.

Table 16 - Request Object Keys

Key	Status	Description
MaxLength	Optional	The maximum number of records to be returned from the database. This is subject to an upper limit set in the Data Storage Container configuration (see 8.5.1**Error! Reference source not found.** below). If no value is given then the default value is 100 records.
StartTime	Optional	A UNIX timestamp. Only records added to the Data Storage Application after this time will be returned. If not supplied then records from the start of the database will be returned, unless the operator has imposed a tighter restriction.
EndTime	Optional	A Unix timestamp. Only records added to the Data Storage Application before this time will be returned. If not supplied then records up to the present time are returned.
PreferOldest	Optional	Flag indicating that if there are more than MaxLength records available, the oldest data should be returned rather than the default of returning the newest data.
InstanceID	Optional	String identifying the Application Instance which data should be returned for. Only records which exactly match the given topic will be returned (no wildcards). This constrains the query to only return results from the specified table in the database.

		If this member is an empty string or omitted from the JSON then data from all Application Instances is returned.
SubTopic	Optional	String giving the sub-topic data is required for. This is the sub-topic below algorithm/data/<IID>. Only records which exactly match the given sub-topic will be returned (no wildcards). This string does not include the '/' between the IID and the sub-topic (see example in Figure 7**Error! Reference source not found.**). To retrieve data from all sub-topics, do not include this key in the JSON payload. To request data only from the top-level topic (no sub-topics) then this key must be included in the JSON payload with an empty string value.
MaxPriority	Optional	Integer defining what priority messages are to be returned. If this JSON key is supplied, messages with priority equal to or numerically less than the value only will be returned. The special value of 6 can be used to return only messages which had no Priority value when stored. If neither this JSON key nor MaxPriority is specified then messages of all priorities will be returned.
MinPriority	Optional	Integer defining what priority messages are to be returned. If this JSON key is supplied, messages with priority equal to or numerically greater than the value only will be returned. If neither this JSON key nor MaxPriority is specified then messages of all priorities will be returned. If both keys are supplied then only messages which meet both criteria will be returned.
Timestamp	Optional	A UNIX timestamp when the request was sent. Required in signed payloads to protect against replay attacks.
Signature	Reserved	See Section 8.1.2

Table 17 - Request Object Keys

Key	Status	Description
Status	Always Present	Integer indicating whether the query succeeded or not. See Table 18
Response	If Status = DSC_QUERY_OK	An array of objects containing data from different tables to be uploaded. Always an array even if data is only from one table.
Response/TableName	Always Present	Name of the table (source application Instance ID) the data in this object is from.

Response/TableRows	Always Present	An array of selected rows from the table (array even if only one row is selected). Each object in the array is an individual message from the source table.
Response/TableRows/ID	Always Present	Opaque integer identifier for the message. These have no meaning except as a handle to be passed back to the Data Storage Application when the message has been uploaded. ID values are only unique within a single table, and may be recycled after the database has been cleaned.
Response/TableRows/Timestamp		UNIX timestamp when the message was added to the Data Storage Application (see Section **Error! Reference source not found.**).
Response/TableRows/SubTopic		The subtopic (below algorithm/<IID>) on which this message was published. For backwards compatibility, the sub-topic string returned always starts with a '/' character. Messages with no sub-topic are returned with sub-topic string of '/'. See also the example in Figure 8.
Response/TableRows/Data		The original message JSON object stored in the Data Storage Application.
Timestamp	Optional	A UNIX timestamp when the response was sent. Required in signed responses to protect against replay attacks.
Signature	Reserved	See Section 8.1.2

Table 18 - Response Status Values

Status Value	Code	Description
0		Never sent, an unanticipated error.
1	DSC_QUERY_OK	Query succeeded, the length of the complete results set is less than or equal to MaxLength. The result is returned in the TableRows array. See also DSC_QUERY_MORE.
2	DSC_QUERY_EMPTY	The query was valid, but found no records. The TableRows array will be empty.

3	DSC_QUERY_TABLE_DENIED	The query is against a table (Application Instance) which the sending container is not allowed to access.
4	DSC_QUERY_TOO_LONG	The query requested more data than the Data Storage Application is willing to provide, because the `MaxLength` field value was too large (see 8.5.1below).
5	DSC_QUERY_TOO_BIG	The data requested by the query is too big to fit into the MQTT payload length restriction.
6	DSC_QUERY_TOO_OLD	The data requested in the query is from further in the past than the Data Storage Application is willing to provide.
7	DSC_UNAVAILABLE	The Data Storage Application is unable to respond to this request, either because it is too busy or is in the process of shutting down.
8	DSC_QUERY_MORE	Query succeeded, there are more than `MaxLength results.` The first MaxLength results are returned in the `TableRows` array, but another query is needed to get more values.
9	DSC_QUERY_INVALID	The JSON query object is empty or not valid JSON and cannot be parsed.

The Data Upload API is not designed to be re-entrant. After a request has been made, the container should wait for the response (there may need to be an exceptional time-out in case the Data Storage Application suffers an error). If a second request is made whilst the response is being produced, the response is undefined. The request does not modify the database at all, so if a second identical request is made after the first response is received, the same data will be returned.

The response status value is used to show whether there is more data available than was sent or not. There is no concept of a database cursor or response pagination. As a result, API users who need to upload all the available data must:

1. Request data for upload.
2. Upload the received messages (if any).
3. Update the database to mark the messages as uploaded.
4. Continue querying until a response of DSC_QUERY_EMPTY is received, at which point there is no more data to upload.

Although the JSON format for the "storage/response/newdata/<IID>" topic allows for messages from multiple tables to be sent in one message, this is not guaranteed. The Data Storage Container may opt to return data from only one table or topic in the response (where there is data to retrieve), and return data from other tables/topics when subsequent requests are received.

8.5.1 Limits

The Data Storage Application is a shared resource and excessively large queries have the potential to degrade the performance of LV-CAP for all users. To mitigate this risk, limits are imposed on the queries which will be accepted.

- **Maximum number of records requested in one query**. If `MaxLength` is not set then a limit of 100 records will be applied. A query for 100 records will always be permitted. This limit may be increased (up to a maximum of 15 000) by the LV-CAP operator, but Application should not depend upon larger queries being allowed on any given system. Note that the limit is on the requested size, not the actual number of records found (which is not known when the query is set up). Thus a request for 16 000 records will always fail (DSC_QUERY_TOO_LONG from **Error! Reference source not found.**), even if the table is empty.
- **Maximum query size**. Because the query result is sent as an MQTT message via the Data Marketplace, it is limited to a maximum of 256MB (268,435,455 bytes), as documented in section 2.2.3 of the MQTT 3.1.1 specification. If the query results in a message which is longer than this limit, then an error (DSC_QUERY_TOO_BIG from Table 18.) is returned instead. Applications must request fewer messages to reduce the returned message size below the limit.
- **Maximum data age**. The Data Storage Application will not be able to store data going back in time indefinitely. Old records will be purged by the Data Storage Application to control the database size. To manage database performance the LV-CAP operator may also impose a maximum age on queries. Any request for data older than this age will fail with DSC_QUERY_TOO_OLD from Table 18

8.5.2 Examples

An example query payload requesting the oldest available data for upload, from all Applications, is shown in Figure 5. The query is not signed. Up to 100 records will be returned as there is no maximum length given. This query may fail:

- With status DSC_QUERY_TOO_OLD if the Data Upload Application does not allow queries indefinitely into the past.
- With status DSC_UNAVAILABLE if the Data Upload Application is shutting down or overloaded.
- With status DSC_QUERY_TOO_BIG if the results will not fit in a MQTT packet.

If it succeeds it could give status:

- DSC_QUERY_EMPTY if there is no data to be sent.
- DSC_QUERY_OK if there are between 1 and 100 messages to be sent.
- DSC_QUERY_MORE is there are more than 100 messages to be sent.

```
{
    "PreferOldest": True
}
```

Figure 5 - Example query payload

An example query payload requesting the latest available data from a specific Application Instance is shown in Figure 6**Error! Reference source not found.**. The query is not signed. Up to 50 records will be returned as requested. This query may fail:

- With status DSC_QUERY_TABLE_DENIED if the Data Storage Application does not allow this Data Upload Application to upload data from this Application Instance.
- With status DSC_UNAVAILABLE if the Data Storage Application is shutting down or overloaded.
- With status DSC_QUERY_TOO_BIG if the results will not fit in a MQTT packet.

If it succeeds it could give status:

- DSC_QUERY_EMPTY if there are no messages from this Application Instance.
- DSC_QUERY_OK if there are between 1 and 50 messages to be sent.
- DSC_QUERY_MORE is there are more than 50 messages to be sent.

```
{
    "MaxLength": 50,
    “InstanceID”: "eatl_profiler_04"
}
```

Figure 6 - Example query payload for a specific Application Instance

An example query payload requesting the latest, highest priority, data from a specific topic is shown in Figure 7. The data requested would originally have been published on topic "algorithm/data/eatl_profiler_04/alarm/highhigh" or "storage/data/eatl_profiler_04/alarm/highhigh". The query is not signed. Up to 10 records will be returned as requested. This query may fail:

- With status DSC_QUERY_TABLE_DENIED if the Data Storage Application does not allow this Data Upload Application to upload data from this Application Instance.
- With status DSC_UNAVAILABLE if the Data Storage Application is shutting down or overloaded.
- With status DSC_QUERY_TOO_BIG if the results will not fit in a MQTT packet.

If it succeeds it could give status:

- DSC_QUERY_EMPTY if there are no messages on this topic with priority equal to 1.
- DSC_QUERY_OK if there are between 1 and 10 messages on this topic with priority 1.
- DSC_QUERY_MORE is there are more than 10 messages on this topic with priority 1

```
{
    "MaxLength": 10,
    "InstanceID": "eatl_profiler_04",
    "SubTopic": "alarm/highhigh",
    "MaxPriority": 1
}
```

Figure 7 - Example query payload for a specific topic and priority

The result of a successful run of this query is shown in Figure 8. The sole available record is returned, with the status DSC_QUERY_OK. The TableName field contains the InstanceID of the application which produced the data, which was passed as InstanceID in the query. The TableRows array contains the single record, with its ID, time-stamp and sub-topic (note the leading '/' here for backwards compatibility) which was passed in the query.

```
{
"Status": 1,
"Response": [
    {
      "TableName": "eatl_profiler_04",
      "TableRows": [
        {
        "ID": 101,
        "Timestamp": 1504110640,
        "SubTopic": "/60/busbar/l3/voltage-mean",
        "Data": {"Timestamp":1504110640, "Value":240.0, "Valid":true, "ToStore":true}
        }
      ]
    }
  ]
}
```


Figure 8 - Example response payload for a single match

When the data has been successfully uploaded, the database record can be marked as recorded by sending an uploaded payload like that shown in Figure 9. This will mark the record with ID 101 in the output of Application Instance eatl_profiler_04 as uploaded, so subsequent queries by the same Data Upload Application Instance will not return this record.


```
{
"NewData": [
    {"eatl_profiler_04": [101]}
  ]
}
```


Figure 9 - Example uploaded payload for a single record

8.6 Data Storage API

The Data Storage API provides a means to access the data persistently stored by the Data Storage Application. This API provides a mechanism for Applications to access data previously stored by Applications, e.g. where a system history is required.

This API operates on a pattern of separate topics for requests and response messages. When using this API, Applications should always subscribe to the response topic before publishing a request. This avoids a race between the response and the subscription which may cause the container to miss response messages.

All methods in this API work with the per-Application Instance database tables described in Section 7. The SubTopic and Data columns are set from the received message. The other columns in the table will be set automatically by the Data Storage Application.

Table 19 – Data Storage Container

Topic	QoS	Description	Sender	Receiver	Payload	Notes
storage/data/<IID>/	2	Insert data into the Data Storage Application, in the <IID> table.	Any Application	Data Storage Application	{ "Key": Data, "KeyN": DataN }	Data messages on this topic can hold anything the sender wishes. The message payload will be stored unaltered as a BLOB. If messages are sent on a sub-topic below storage/data/<IID>/ then the sub-topic will be stored in the SubTopic column of the table. This is equivalent to publishing data on algorithm/data/<IID> with the ToStore flag true.
storage/request/<IID>	1	Request data by the Application Instance <IID>.	Any Application	Data Storage Application	{ "MaxLength": <Integer>, "StartTime": <Int64>, "EndTime": <Int64>, "PreferOldest": <Boolean>, "InstanceID": <IID> "SubTopic": <String>, "MinPriority": <Int>, "MaxPriority": <Int>, "Timestamp": <Int64>, "Signature": {} }	The Data Storage Application will return the requested data on the storage response topic below. The request Payload keys are documented in Table 16

storage/response/<IID>	1	The response from the data storage container after a get request by Application Instance <IID>.	Data Storage Application	The <IID> Application which requested the table	<pre>{
					"Status": <Integer>,
					"Response": [
					{
					"TableName": <IID>,
					"TableRows": [
					{
					"ID":<Integer>,
					"Timestamp": <Int64>,
					"SubTopic": <string>,
					"Data": <JSON Object>,
					},
					{more rows}
					]
					}
					]
					}</pre>
						The response Payload keys are documented in Table 17.

Because of the automatic storing of Algorithm output described in Section 8.4, it will be unusual to need to explicitly store data using the "storage/data/" topic. Publishing on the "storage/data/<IID>" topic has exactly the same results as publishing on the "algorithm/data/<IID>" topic with the ToStore flag true.

The fields of the message on the request topic are documented in Table 16, and those of the response message on the response topic in Table 17. These objects are deliberately the same as those used by the Data Upload API. The same rules for InstanceID and SubTopic apply. Similarly, "storage/response/newdata/<IID>" and "storage/response/<IID>" use the same response format, although in this API there will only ever be data from one table and so only one element in the Response array. Note that the Instance ID <IID> in the topic names refers to the Application making the requests and receiving the data, not the table being accessed (except in the first topic documented, where they are the same).

9. JSON Object Structures

All messages passed through the Data Marketplace, and all Application Configuration data, is serialised as JSON objects. Whilst for some purposes bespoke JSON object structures are necessary, wherever possible use should be made of the standard JSON object structures defined in this section.

Using standard object structures ensures that data can be passed from any Application to any other Application without the need for bespoke software development. It minimises the need for Applications to cope with data from different sources in different formats. Applications which output in standard formats will be best placed to take advantage of facilities provided by LV-CAP and other Applications.

In applying these Object formats consideration should also be given to the general principals set out in Section 4.

9.1 Scalar Object Format

The default choice of JSON Object for almost all sensor readings and many algorithm outputs will be the Scalar Object. It represents a single value at a single point in time, for instance a temperature or a power flow. To provide more metadata about the value and how it was arrived at, a separate Data Series Metadata Object should be used (see Section 9.4).

```
{
    "Timestamp": <Int64>,
    "Value": < >,
    "Valid": <Boolean>,
    "ToStore": <Boolean>,
    "Priority": <Int>,
    "Signature": {}
}
```

Figure 10 - Scalar Object Format Structure

Table 20 - Scalar Object Format Keys

Key	Status	Description
`Timestamp`	Required	Standard LV-CAP timestamp (see Section 4.5) when the reading was made.
`Value`	Required	The reading, converted to base engineering units. The reading can be of any scalar type (Boolean, Integer or Floating Point).
`Valid`	Required	A logical value, showing if the Value is within the expected range (configured).
`ToStore`	Optional	Flag used historically to indicate whether the data should be stored by the Data Storage Application or not. May be over-ridden by the Data Storage Application configuration.
`Priority`	Optional	A priority indicator as in section 4.9, which allows the upload of certain messages to be prioritised by Data Upload Applications.
`Signature`	Reserved	See Section 8.1.2**Error! Reference source not found.**

The Value is always given in the base SI unit for the quantity being measured or calculated, as in Section 4.5. The units can be given explicitly in the optional Data Series Metadata Object (Section 9.4).

9.2 Series Object Format

Where a series of closely related values are to be sent as a set then a Series Object provides a way to package the complete set of values in a single JSON Object. It can represent a time series of values (anything from a fault waveform recorder (sampling many times per mains cycle) to a load profile (hourly load values), or a frequency spectrum. The object contains fields to record what range of source data was used to produce series. To provide more metadata about the value and how it was arrived at, a separate Data Series Metadata Object should be used (see Section 9.4).

```
{
    "Timestamp": <Int64>,
    "StartPoint": <Int64 or float>,
    "Interval" : <float>,
    "Value": [< >],
    "Confidence": [< >],
    "Valid": <Boolean>,
    "TimestampStart": <Int64>,
    "TimestampEnd": <Int64>,
    "ToStore": <Boolean>,
    "Priority": <Int>,
    "Signature": {}
}
```

Figure 11 - Series Object Format Structure

Table 21 - Scalar Object Format Keys

Key	Status	Description
`Timestamp`	Required	Standard LV-CAP timestamp (see Section 4.5) when the series was produced.
`Interval`	Required	Interval between values in the series. For time series, this is the time between samples in seconds (or decimals of them), for frequency spectrums Hertz and so on.
`StartPoint`	Required	The x-axis co-ordinate of the first value in the series. For time series this is the timestamp of the first value, for frequency spectrums the frequency of the first bin and so on.
`Value`	Required	An array of series values in base engineering units. The values can be of any scalar type (Boolean, Integer or Floating Point).
`Confidence`	Optional	An array, the same size as the Value array, of values giving the confidence in the values. This may be used to represent the uncertainties caused by missing input data or inadequate system history.
`Valid`	Required	A logical value, showing if the Series as a whole is thought to be valid for further use.
`TimestampStart`	Optional	The standard LV-CAP timestamp of the earliest source data used to produce this series.
`TimestampEnd`	Optional	The standard LV-CAP timestamp of the latest source data used to produce this series.
`ToStore`	Optional	Flag used historically to indicate whether the data should be stored by the Data Storage Container or not. May be over-ridden by the Data Storage Container configuration.
`Priority`	Optional	A priority indicator as in section 4.9, which allows the upload of certain messages to be prioritised by Data Upload Applications.
`Signature`	Reserved	See Section 8.1.2

This is deliberately an abstract data format designed to be capable of accommodating a wide range of different data types. The Value are always given in the base SI unit for the quantity being measured or calculated, as in Section 4.5**Error! Reference source not found.**. The units can be given e xplicitly in the optional Data Series Metadata Object (Section 9.4). Some practical examples of its use are given in Figure 12 and Figure 13.

Figure 12 shows the Series Object Format used for a predicted load profile.

- It was calculated and published at 14:30:05 UTC on 8th March 2017.
- The first predicted load segment in the prediction starts at 14:30:00 UTC on 8th March 2017
- The prediction is composed of half-hourly (1800 seconds) load current values.
- The prediction is for 4 hours, so has 8 values of load current in amps.
- The predictor is confident in the prediction for the first two hours and the last one, but is aware of limitations in the data for the third hour (e.g. because there are problems with missing data in that hour). These reduce the confidence in those predictions.
- Overall the predictor thinks that this data is valid for use.

- The prediction is built on the previous 4 weeks of data, so the oldest data used was from 14:30:00 UTC on 8th March 2017.
- The most recent data used was from 15:00:00 UTC on 1st March 2017, the end of the half hour period one week ago.
- The prediction is not signed.

The predictor does not stipulate whether this data is to be stored or not


```
{
    "Timestamp": 1488983405,
    "StartPoint": 1488983405,
    "Interval" : 1800.0,
    "Value": [120.0, 122.5, 125.7, 130.9, 124.8, 121.4, 118.6, 115.3],
    "Confidence": [1.0, 1.0, 1.0, 1.0, 0.85, 0.75, 1.0, 1.0],
    "Valid": true,
    "TimestampStart": 1486564200,
    "TimestampEnd": 1488380400
}
```


Figure 12 – Example of a Scalar Object used for a load prediction

Figure 13 shows the Series Object Format used for a harmonic spectrum.

- It was calculated and published at 11:00:00 UTC on 5th March 2017.
- The first harmonic in the spectrum is 50Hz
- The spectrum is composed of values for each harmonic, so every 50 Hz.
- The spectrum is for the first 5 harmonics only.
- The spectrum is calculated, so no confidence values are given.
- Overall the calculation succeeded, so the data is valid for use.
- The spectrum was calculated from the previous 10 minutes of data, so the oldest data used was from 10:50:00 UTC on 5th March 2017.
- The most recent data used was from 10:59:59 UTC on 5th March 2017, the end of the 10-minute period.
- The publishing container thinks that this data should be stored in the Data Storage Container.
- The publishing container has assigned this data the lowest available priority.
- The spectrum is not signed.


```
{
    "Timestamp": 1488711600,
    "StartPoint": 50,
    "Interval" : 50,
    "Value": [76423.0, 122.5, 86.7, 57.9, 12.4],
    "Valid": true,
    "TimestampStart": 1488711000,
    "TimestampEnd": 1488711599,
    "ToStore": true,
    "Priority": 5
}
```


Figure 13 – Example of a Scalar Object used for a harmonic spectrum

9.3 Co-ordinate Object Format

Where a group of co-ordinates are produced then a Co-ordinate Object provides a way to package them in a single JSON Object. The co-ordinates may be in a real space (e.g. Latitude and Longitude for geographic position) or a conceptual one (real and imaginary power in a power flow vector

diagram). To provide more metadata about the value and how it was arrived at, a separate Data Series Metadata Object should be used (see Section 9.4).

```
{
    "Timestamp": <Int64>,
    "Coordinates": [< >],
    "System": <String>,
    "Valid": <Boolean>,
    "ToStore": <Boolean>,
    "Priority": <Int>,
    "Signature": {}
}
```

Figure 14 - Co-ordinate Object Format Structure

Table 22 - Co-ordinate Object Format Keys

Key	Status	Description
`Timestamp`	Required	Standard LV-CAP timestamp (see Section 4.5**Error! Reference source not found.**) when the reading was made.
`Coordinates`	Required	Array of co-ordinate values, in base engineering units. The co-ordinate values will be Integer or Floating-Point values.
`System`	Optional	The co-ordinate system being used, e.g. Cartesian (for x-y plots) or WGS84 latitude and longitude.
`Valid`	Required	A logical value, showing if the Value is within the expected range (configured).
`ToStore`	Optional	Flag used historically to indicate whether the data should be stored by the Data Storage Container or not. May be over-ridden by the Data Storage Container configuration.
`Priority`	Optional	A priority indicator as in section 4.9, which allows the upload of certain messages to be prioritised by Data Upload Applications.
`Signature`	Reserved	See Section 8.1.2

No implementation of this Object Format yet exists.

9.4 Data Series Metadata Object Format

There will be various pieces of metadata (that is, information about the data) associated with the data published on a given topic. It may be desirable to transmit these in a machine-readable format, so that consuming Applications can make use of them. However, this metadata does not change from reading to reading, so it would be inefficient to transmit (and especially store) them alongside the readings themselves. Instead a separate /meta/ sub-topic is used for metadata, which is transmitted as Data Series Metadata Objects.

The metadata objects are sent only when an Application starts, or there is a change to the metadata. In order that subscribing Applications always receive this information, the messages are published with the Retained flag set to true. This means that the Data Marketplace will automatically send a copy of the latest metadata to any new client which subscribes the /meta/ sub-topic, without any effort from the publishing Application.

```
{
    "Name": <String>,
    "Units": <String>,
    "DisplayUnits": <String>,
    "SigFigs": <Integer>,
}
```

Figure 15 - Data Series Metadata Object Format Structure

9.5 Application Configuration Format

All configuration files must be valid JSON objects. Application configuration data will differ significantly from Application to Application depending on their structure, and only be of use to the Application it is intended for. To accommodate this, the structure of the configuration file for each container is largely up to the Application author, but a standard top-level structure is required in order to deliver updated configuration information to the correct container. Application authors are free to structure the ContainerConfig object within their configuration in whatever way suits their application, provided that it is a valid JSON object.

```
{
      "ContainerName": "<IID>",
      "ContainerConfig":
      {
            "<examplekey1>": <configvalue1>,
            "<examplekey2>": <configvalue2>,
            "<examplekeyN>": <configvalueN>
      }
}
```

Figure 16 - Third Party Container Configuration File Example

Table 23 - Third Party Configuration File Keys

Key	Status	Description
`ContainerName`	Required	IID of the Application the configuration is for.
`ContainerConfig`	Required	JSON object containing the Application Instance configuration. This object's contents and structure will change from Application to Application.

10. References

The following external resources provide more information to support this specification:

1. The MQTT Standard, version 3.1.1: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
2. ECMA Standard 404, "The JSON Data Interchange Format" http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
3. Blog Post "MQTT Topics & Best Practices" http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices
4. Docker Documentation for docker tag command: https://docs.docker.com/v1.12/engine/reference/commandline/tag/ and https://docs.docker.com/v17.03/engine/reference/commandline/tag/