

Sparkplug™ Specification



Sparkplug™

MQTT Topic &
Payload Definition

Version 2.2



Copyright © 2019 Eclipse Foundation, Inc. <https://www.eclipse.org/legal/efsl.php>

Sparkplug™ and the Sparkplug™ logo are trademarks of the Eclipse Foundation

Revision Number	Date	Author	Description
1.0	5/26/16	Cirrus Link	Initial Release
2.1	12/10/16	Cirrus Link	Payload B Addition
2.2	10/11/19	Cirrus Link	Re-branding for Eclipse foundation added TM to Sparkplug

Table of Contents

Table of Figures.....	5
1. Introduction	6
1.1. Define an MQTT Topic Namespace	6
1.2. Define MQTT State Management.....	6
1.3. Define the MQTT Payload.....	6
2. Background	7
3. Infrastructure Components	8
3.1. MQTT Server(s)	8
3.2. MQTT Edge of Network (EoN) Node	8
3.3. Device/Sensor	8
3.4. MQTT Enabled Device(Sparkplug™)	8
3.5. SCADA/IIoT Host	9
3.6. MQTT Application Node	9
3.7. Security	9
3.7.1. Authentication	9
3.7.2. Authorization	9
3.7.3. Encryption.....	9
4. Leveraging Standards and Open Source	10
4.1. OASIS MQTT V3.1.1 Specification	10
4.2. Eclipse Foundation IoT Resources	10
4.2.1. Paho	10
4.3. Google Protocol Buffers.....	10
4.3.1. Kura Google Protocol Buffer Schema	10
4.4. Raspberry Pi Hardware	10
5. General Message Flow.....	11
5.1. MQTT Session State Awareness.....	11
6. Sparkplug™ MQTT Topic Namespace	12
6.1. Sparkplug™ Topic Namespace Elements	12
6.1.1. namespace Element.....	12
6.1.2. group_id Element	12
6.1.3. message_type Element.....	13
6.1.4. edge_node_id Element.....	13
6.1.5. device_id Element.....	13

7.	Sparkplug™ MQTT Message Types	14
7.1.	MQTT EoN Birth and Death Certificate.....	14
7.1.1.	EoN Death Certificate (NDEATH)	15
7.1.2.	EoN Birth Certificate (NBIRTH).....	15
7.2.	MQTT EoN node Data (NDATA)	15
7.3.	MQTT Device Birth and Death Certificate	15
7.3.1.	MQTT Device Birth Certificate (DBIRTH).....	16
7.3.2.	MQTT Device Death Certificate (DDEATH)	16
7.4.	MQTT Device Data Messages (DDATA).....	16
7.5.	SCADA/IIoT Host Birth and Death Certificates.....	16
7.5.1.	SCADA/IIoT Host Death Certificate Payload (STATE)	17
7.5.1.	SCADA/IIoT Birth Certificate Payload (STATE)	17
7.6.	EoN node Command (NCMD)	17
7.7.	Device Command (DCMD)	18
8.	Sparkplug™ MQTT Session Management and Message Flow	19
8.1.	Primary Application Session Establishment.....	20
8.2.	EoN node Session Establishment.....	22
8.3.	MQTT Device Session Establishment.....	24
8.4.	General MQTT applications and non-primary Applications.	26
9.	Sparkplug™ MQTT Data and Command Messages.....	27
9.1.	EoN NDATA and NCMD Messages	28
10.	Primary Application STATE in Multiple MQTT Server Topologies	30
11.	Sparkplug™ Persistent versus Non-Persistent Connections	32
12.	Contact Information	33
Appendix 1	Sparkplug™ B Payload Definition.....	34

Table of Figures

Figure 1 - MQTT SCADA Infrastructure	8
Figure 2 - Simple MQTT Infrastructure	11
Figure 3 - Host Session Establishment	20
Figure 4 - EoN node MQTT Session Establishment	22
Figure 5 - MQTT Device Session Establishment	24
Figure 6 - EoN node NDATA and NCMD Message Flow	29
Figure 7 – Primary Application STATE flow diagram.....	30

1. Introduction

Sparkplug™ provides an open and freely available specification for how Edge of Network (EoN) gateways or native MQTT enabled end devices and MQTT Applications communicate bi-directionally within an MQTT Infrastructure. This document details the structure and implementation requirements for Sparkplug™ compliant MQTT Client implementations on both devices and applications.

It is recognized that MQTT is used across a wide spectrum of application solution use cases, and an almost indefinable variation of network topologies. To that end the Sparkplug™ specification strives to accomplish the three following goals.

1.1. Define an MQTT Topic Namespace

As noted many times in this document one of the many attractive features of MQTT is that it does not specify any required Topic Namespace within its implementation. This fact has meant that MQTT has taken a dominant position across a wide spectrum of IoT solutions. The intent of the Sparkplug™ specification is to identify and document a Topic Namespace that is well thought out and optimized for the SCADA/IIoT solution sector.

1.2. Define MQTT State Management

One of the unique aspects of MQTT is that it was originally designed for real time SCADA systems to help reduce data latency over bandwidth limited and often unreliable network infrastructure. In many implementations though the full benefit of this “Continuous Session Awareness” is not well understood, or not even implemented. The intent of the Sparkplug™ specification is to take full advantage of MQTT’s native Continuous Session Awareness capability as it applies to real time SCADA/IIoT solutions.

1.3. Define the MQTT Payload

Just as the MQTT specification does not dictate any particular Topic Namespace, nor does it dictate any particular payload data encoding. The intent of the Sparkplug™ specification is to strive to define payload encoding architectures that remain true to the original, lightweight, bandwidth efficient, low latency features of MQTT while adding modern encoding schemes targeting the SCADA/IIoT solution space.

Sparkplug™ has defined an approach where the Topic Namespace can aid in the determination of the encoding scheme of any particular payload. Currently there are two (2) Sparkplug™ defined encoding schemes that this specification supports. The first one is the Sparkplug™ A encoding scheme based on the very popular Kura open source Google Protocol Buffer definition. The second one is the Sparkplug™ B encoding scheme that provides a richer data model developed with the feedback of many system integrators and end user customers using MQTT.

2. Background

MQTT was originally designed as a message transport for real-time SCADA systems. The MQTT message transport specification does **not** specify the Topic Namespace to use nor does it define the Payload representation of the data being published and/or subscribed to. In addition to this, since the original use case for MQTT was targeting real-time SCADA, there are mechanisms defined to provide the **state** of an MQTT session such that SCADA/Control HMI application can monitor the current state of any MQTT device in the infrastructure. As with the Topic Namespace and Payload the way state information is implemented and managed within the MQTT infrastructure is not defined. All of this was intentional within the original specification to provide maximum flexibility across any solution sector that might choose to use MQTT infrastructures.

But at some point, for MQTT based solutions to be interoperable within a given market sector, the Topic Namespace, Payload representation and session state must be defined. The intent and purpose of the Sparkplug™ specification is to define an MQTT Topic Namespace, payload, and session state management that can be applied generically to the overall IIoT market sector, but specifically meets the requirements of real-time SCADA/Control HMI solutions. Meeting the operational requirements for these systems will enable MQTT based infrastructures to provide more valuable real-time information to Line of Business and MES solution requirements as well.

The purpose of the Sparkplug™ specification is to remain true to the original notion of keeping the Topic Namespace and message sizes to a minimum while still making the overall message transactions and session state management between MQTT devices and MQTT SCADA/IIoT applications simple, efficient and easy to understand and implement.

3. Infrastructure Components

This section details the infrastructure components implemented.

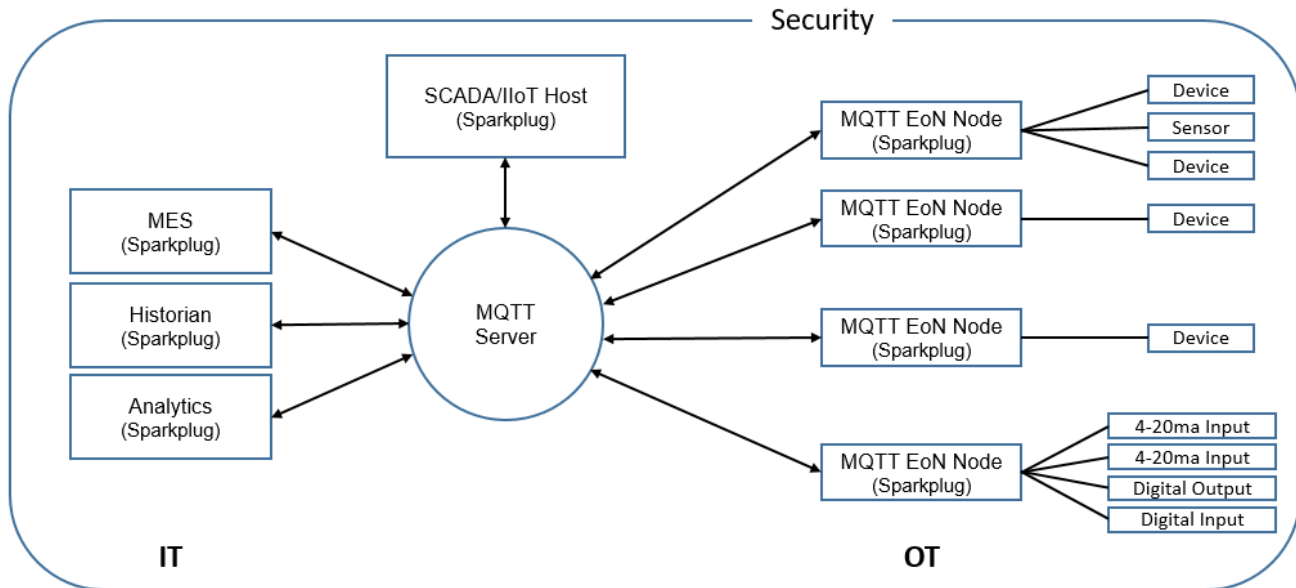


Figure 1 - MQTT SCADA Infrastructure

3.1. MQTT Server(s)

MQTT enabled infrastructure requires that one or more MQTT Servers are present in the infrastructure. The only requirement that the Sparkplug™ specification places on the selection of an MQTT Server component in the architecture is it is required to be compliant with the latest MQTT V3.1.1 specification and is sized to properly manage all MQTT message traffic.

One can implement the use (if required) of multiple MQTT servers for redundancy, high availability, and scalability within any given infrastructure.

3.2. MQTT Edge of Network (EoN) Node

In the context of this specification, an MQTT Edge of Network (EoN) Node is any V3.1.1 compliant MQTT Client application that manages an MQTT Session and provides the physical and/or logical gateway functions required to participate in the Topic Namespace and Payload definitions described in this document. The EoN node is responsible for any local protocol interface to existing legacy devices (PLCs, RTUs, Flow Computers, Sensors, etc.) and/or any local discrete I/O, and/or any logical internal process variables(PVs).

3.3. Device/Sensor

The Device/Sensor represents any physical or logical device connected to the MQTT EoN node providing any data, process variables or metrics.

3.4. MQTT Enabled Device(Sparkplug™)

This represents any device, sensor, or hardware that directly connects to MQTT infrastructure using a compliant MQTT 3.1.1 connection with the payload and topic notation as outlined in this Sparkplug™ specification. Note that it will be represented as an EoN node in the Sparkplug™ topic payload.

3.5. SCADA/IIoT Host

The SCADA/IIoT Host Node is any MQTT Client application that subscribes to and publishes messages defined in this document. In typical SCADA/IIoT infrastructure implementations, there will be only one **Primary** SCADA/IIoT Host Node responsible for the monitoring and control of a given group of MQTT EoN nodes. Sparkplug™ does support the notion of multiple critical Host applications. This does not preclude any number of additional MQTT SCADA/IIoT Nodes participating in the infrastructure that are in either a pure monitoring mode, or in the role of a hot standby should the Primary MQTT SCADA/IIoT Host go offline.

3.6. MQTT Application Node



An MQTT Application Node is any non-primary MQTT SCADA/IIoT Client application that consumes the real-time messages or any other data being published with proper permission and security.

3.7. Security

3.7.1. Authentication

There are several levels of security and access control configured within an MQTT infrastructure. From a pure MQTT client perspective, the client does need to provide a unique Client ID, and an optional Username and Password.

3.7.2. Authorization

Although access control is not mandated in the MQTT specification for use in MQTT Server implementations, Access Control List (ACL) functionality is available for most MQTT Server implementations. The ACL of an MQTT Server implementation is used to specify which Topic Namespace any MQTT Client can subscribe to and publish on. Examples are provided on how to setup and manage MQTT Client credentials and some considerations on setting up proper ACL's on the MQTT Servers.

3.7.3. Encryption

The MQTT specification does not specify any TCP/IP security scheme as it was envisaged that TCP/IP security would (and did) change over time. Although this document will not specify any TCP/IP security schema it will provide examples on how to secure an MQTT infrastructure using TLS security.


4. Leveraging Standards and Open Source

In addition to leveraging the latest MQTT V3.1.1 standards, the Sparkplug™ specification leverages as much open source development tooling and data encoding as possible.

4.1. OASIS MQTT V3.1.1 Specification

The Sparkplug™ specification specifies that MQTT Server/Clients in the infrastructure adhere to the MQTT V3.1.1 specification. The specification documentation refers to “*mqtt-v3.1.1-os.doc*”:

<http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>

Also referred is an addendum document to the MQTT V3.1.1 specification document that discusses best practices for implementing security on MQTT TCP/IP networks: 

<http://docs.oasis-open.org/mqtt/mqtt-nist-cybersecurity/v1.0/mqtt-nist-cybersecurity-v1.0.doc>

4.2. Eclipse Foundation IoT Resources

The Eclipse Foundation is an excellent resource for open source software supporting industry standards. Within the Eclipse Foundation is an Internet of Things (IoT) working group providing a wealth of information.

<http://iot.eclipse.org/>

4.2.1. Paho

Paho is an Eclipse Foundation project that offers excellent resources for mature, compliant MQTT Client and MQTT Server implementations and well as additional resources for all things MQTT.

<http://www.eclipse.org/paho/>

4.3. Google Protocol Buffers

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. Google Protocol Buffers are used to encode the Sparkplug™ payload in both versions A and B of the Sparkplug™ payload specification.

<https://developers.google.com/protocol-buffers/>

4.3.1. Kura Google Protocol Buffer Schema

Kura is another Eclipse Foundation project under IoT resources. Kura provides open source resources for the Google Protocol Buffer representation of MQTT payloads as defined in the Sparkplug™ A payload definition:

<https://github.com/eclipse/kura/blob/develop/kura/org.eclipse.kura.core.cloud/src/main/protobuf/kurapayload.proto>

4.4. Raspberry Pi Hardware

For the sake of keeping the Sparkplug™ specification as real world as possible, a reference implementation of an EoN node and associated Device is provided for the examples and screen shots in this document. All of this was implemented on Raspberry Pi hardware representing the EoN node with a Pibrella I/O board representing the Device.

5. General Message Flow

This section discusses the generic topology shown in Figure 3 – Simple MQTT Infrastructure identifying how each of the components of the infrastructure interacts.

At the simplest level, there are only two components required as shown below. An MQTT Client and an MQTT Server. With proper credentials, any MQTT Client can connect to the MQTT Server without any notion of other MQTT Client applications that are connected, and can issue subscriptions to any MQTT messages that it might be interested in as well as start publishing any message containing data that it has. This is one of the principal notions of IIoT, that is the decoupling of intelligent devices from any direct connection to any one consumer application.

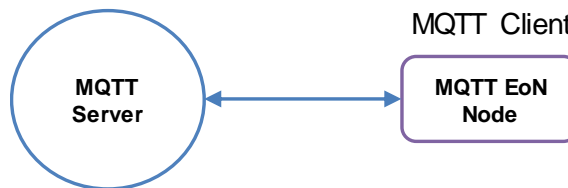


Figure 2 - Simple MQTT Infrastructure

5.1. MQTT Session State Awareness

In any network architecture, network connection **State** is important. In SCADA/IIoT, connection **State** is extremely important. **State** is the session awareness of the MQTT EoN and the MQTT Server. The very reason that most SCADA Host systems in this market sector are still using legacy poll/response protocols to maintain a notion of the **State** of the connection between the SCADA application and the connected devices. *“I poll, I get a response, I know the **State** of all the I/O points, but now I must poll again because that **State** may have changed.”*

Many implementations of solutions using MQTT treat it as a simple, stateless, pub/sub state machine. This is quite viable for IoT and some IIoT applications, however it is not taking advantage of the full capability of MQTT based infrastructures.

One of the primary applications for MQTT as it was originally designed was to provide reliable SCADA communications over VSAT topologies. Due to propagation delay and cost, it was not feasible to use a poll/response protocol. Instead of a poll/response protocol where all the data was sent in response to every poll, MQTT was used to “publish” information from remote sites only when the data changed. This technique is sometimes called Report by Exception or RBE. But for RBE to work properly in real-time SCADA, the “state” of the end device needs to be always known. In other words, SCADA/IIoT host could only rely on RBE data arriving reliably if it could be assured of the state of the MQTT session.

The Sparkplug™ specification defines the use of the MQTT V3.1.1 “Last Will and Testament” feature to provide MQTT session state information to any other interested MQTT client in the infrastructure. The session state awareness is implemented around a set of defined “Birth” and “Death” Topic Namespace and Payload definitions in conjunction with the MQTT connection “Keep Alive” timer.

6. Sparkplug™ MQTT Topic Namespace

To get a working Message Oriented Middleware (MOM) based SCADA system using MQTT, the first thing that must be defined is a Topic Namespace to work within. The beauty of MQTT is the fact that you can just come up with an arbitrary topic like “Portland/Temperature”, connect to an MQTT Server, and start publishing the temperature value. For this data to be useful to other MQTT Client applications that want to consume the temperature values, the Topic Namespace needs to be understood by everyone participating in the data exchange.

Every MQTT message published consist of a **topic** and a **payload** component. These components are the “overhead” of an MQTT message as measured in bytes on the wire. The Sparkplug™ specification is designed to keep these components meaningful and easy to understand, but not to get so verbose as to negatively impact bandwidth/time sensitive data exchange.

6.1. Sparkplug™ Topic Namespace Elements

All MQTT clients using the Sparkplug™ specification will use the following Topic Namespace structure:

namespace/group_id/message_type/edge_node_id/[device_id]

6.1.1. namespace Element

The **namespace** element of the Topic Namespace is the root element that will define both the structure of the remaining namespace elements as well as the encoding used for the associated payload data. The current Sparkplug™ specification defines two (2) namespaces. One is for Sparkplug™ payload definition A, and another one of for the Sparkplug™ payload definition B.

For the Sparkplug™ A version of the payload definition, the UTF-8 string constant for the **namespace** element will be:

“spAv1.0”

For the Sparkplug™ B version of the specification, the UTF-8 string constant for the **namespace** element will be:

“spBv1.0”

Note that for the remainder of this document, the version of the Sparkplug™ Payload definition does not affect the Topic Namespace or session state management as they will remain the same. There are separate appendices that defines the encoding used for both the A and B versions of Sparkplug™ MQTT message payload.

6.1.2. group_id Element

The **group_id** element of the Topic Namespace provides for a logical grouping of MQTT EoN nodes into the MQTT Server and back out to the consuming MQTT Clients. The format of the **group_id** element is not dictated in that it can be any valid UTF-8 alphanumeric string except for the reserved characters of ‘+’ (plus), ‘/’ (forward slash), and ‘#’ (number sign). In most use cases to minimize bandwidth, it should be descriptive but as small as possible. Examples of where the [group_id] might be used include Oil/Gas applications where MQTT EoN nodes on a physical pipeline segment all have the same [group_id]. Plant floor applications may group MQTT EoN nodes based on logical cell or manufacturing line requirements.

6.1.3. *message_type* Element

The ***message_type*** element of the Topic Namespace provides an indication as to how to handle the MQTT payload of the message. Note that the actual encoding of the payload will vary depending on the version of the Sparkplug™ implementation as indicated by the ***namespace*** element.

The following ***message_type*** elements are defined for the Sparkplug™ Topic Namespace:

- **NBIRTH** – Birth certificate for MQTT EoN nodes.
- **NDEATH** – Death certificate for MQTT EoN nodes.
- **DBIRTH** – Birth certificate for Devices.
- **DDEATH** – Death certificate for Devices.
- **NDATA** – Node data message.
- **DDATA** – Device data message.
- **NCMD** – Node command message.
- **DCMD** – Device command message.
- **STATE** – Critical application state message.

The specification for each of these *message_type* elements are detailed later in this document.

6.1.4. *edge_node_id* Element

The ***edge_node_id*** element of the Sparkplug™ Topic Namespace uniquely identifies the MQTT EoN node within the infrastructure. The ***group_id*** combined with the ***edge_node_id*** element must be unique from any other ***group_id/edge_node_id*** assigned in the MQTT infrastructure. The format of the ***edge_node_id*** can be valid UTF-8 alphanumeric String with the exception of the reserved characters of '+' (plus), '/' (forward slash), and '#' (number sign). The topic element ***edge_node_id*** travels with every message published and should be as short as possible.

6.1.5. *device_id* Element

The ***device_id*** element of the Sparkplug™ Topic Namespace identifies a device attached (physically or logically) to the MQTT EoN node. Note that the ***device_id*** is an optional element within the Topic Namespace as some messages will be either originating or destined to the ***edge_node_id*** and the ***device_id*** would not be required. The format of the ***device_id*** is a valid UTF-8 alphanumeric String except for the reserved characters of '+' (plus), '/' (forward slash), and '#' (number sign). The ***device_id*** must be unique from other devices connected to the same EoN node, but can be duplicated from EoN node to other EoN nodes. The ***device_id*** element travels with every message published and should be as short as possible.

7. Sparkplug™ MQTT Message Types

Sparkplug™ defines the Topic Namespace for set of MQTT messages that are used to manage connection state as well as bidirectional metric information exchange that would apply to many typical real-time SCADA/IIoT, monitoring, and data collection system use cases. The defined message types include:

- **NBIRTH** – Birth certificate for MQTT EoN nodes.
- **NDEATH** – Death certificate for MQTT EoN nodes.
- **DBIRTH** – Birth certificate for Devices.
- **DDEATH** – Death certificate for Devices.
- **NDATA** – Node data message.
- **DDATA** – Device data message.
- **NCMD** – Node command message.
- **DCMD** – Device command message.
- **STATE** – Critical application state message.

Using these defined messages host SCADA/IIoT applications can:

- Discover all metadata and monitor state of any EoN/Device connected to the MQTT infrastructure.
- Discover all metrics which include all diagnostics, properties, metadata, and current state values.
- Issue write/command messages to any EoN/Device metric.

This section of the document defines the Topic Namespace and how each of the associated messages types can be used.

7.1. MQTT EoN Node Birth and Death Certificate

A critical aspect for MQTT in a real-time SCADA/IIoT application is making sure that the primary MQTT SCADA/IIoT Host Node can know the “STATE” of any EoN node in the infrastructure within the MQTT Keep Alive period (*refer to section 3.1.2.10 in the MQTT Specification*). To implement the state a known **Will Topic** and **Will Message** is defined and specified. The Will Topic and Will Message registered in the MQTT CONNECT session establishment, collectively make up what we are calling the Death Certificate. Note that the delivery of the Death Certificate upon any MQTT client going offline unexpectedly is part of the MQTT protocol specification, not part of this Sparkplug™ specification (*refer to section 3.1 CONNECT in the MQTT Specification for further details on how an MQTT Session is established and maintained*).

Unlike the Death Certificate mechanism which is part of the MQTT transport specification, the Birth Certificate is a Sparkplug™ definition. The Birth Certificate is a logical reciprocal of the Death Certificate that is used to convey the fact that the associate MQTT EoN node now has an MQTT session established and can now start providing real-time metrics.

The first MQTT message that an EoN node MUST publish upon the successful establishment of an MQTT Session is an EoN BIRTH Certificate.

7.1.1. EoN Node Death Certificate (NDEATH)

The Death Certificate topic for an MQTT EoN node is:

namespace/group_id/NDEATH/edge_node_id

The Death Certificate topic and payload described here are not “published” as an MQTT message by a client, but provided as parameters within the MQTT CONNECT control packet when this MQTT EoN node first establishes the MQTT Client session.

Immediately upon reception of an EoN Death Certificate, any MQTT client subscribed to this EoN node should set the data quality of all metrics to **STALE** and should note the time stamp when the NDEATH message was received.

The MQTT payload typically associated with this topic can include a Birth/Death sequence number used to track and synchronize Birth and Death sequences across the MQTT infrastructure. Since this payload will be defined in advance, and held in the MQTT server and only delivered on the termination of an MQTT session, not a lot of additional diagnostic information can be pre-populated into the payload.

7.1.2. EoN Node Birth Certificate (NBIRTH)

The Birth Certificate topic for an MQTT EoN node is:

namespace/group_id/NBIRTH/edge_node_id

The EoN Birth Certificate payload contains everything required to build out a data structure for all metrics for this EoN node. The ONLINE state of this EoN node should be set to TRUE along with the associated ONLINE Date Time parameter. Note that the EoN Birth Certificate ONLY indicates the node itself is online and in an MQTT Session, but any devices that have previously published a DBIRTH will still have “**STALE**” metric quality until those devices come online with their associated DBIRTH.

7.2. MQTT EoN Node Data (NDATA)

Once an MQTT EoN node is online with a proper NBIRTH it is in a mode of quiescent Report by Exception (RBE) or time based reporting of metric information that changes. This enables the advantages of the native Continuous Session Awareness of MQTT to monitor the STATE of all connected MQTT EoN node and to rely on Report by Exception (RBE) messages for metric state changes over the MQTT session connection.

The Data Topic for an MQTT EoN node is:

namespace/group_id/NDATA/edge_node_id

The payload of NDATA messages will contain any RBE or time based metric EoN node values that need to be reported to any subscribing MQTT clients.

7.3. Device Birth and Death Certificate

Sparkplug™ specifies how a MQTT EoN node uses the MQTT transport layer Death Certificate along with the Sparkplug™ defined NBIRTH. Device Birth and Death certificates are defined and managed by the Sparkplug™ specification e.g. they are application level payloads published by the EoN node and are not part of the MQTT transport specification.

7.3.1. Device Birth Certificate (DBIRTH)

The Topic Namespace for a Birth Certificate for a device is:

namespace/group_id/DBIRTH/edge_node_id/device_id

The DBIRTH payload contains everything required to build out a data structure for all metrics for this device. The ONLINE state of this device should be set to TRUE along with the associated ONLINE date time this message was received.

The MQTT EoN node is responsible for the management of all attached physical and/or logical devices. Once the EoN node has published its NBIRTH, any consumer application ensures that the metric structure has the EoN node in an ONLINE state. But each physical and/or logical device connected to this node will still need to provide this DBIRTH before consumer applications create/update the metric structure (if this is the first time this device has been seen) and set any associated metrics in the application to a “GOOD” state.

7.3.2. Device Death Certificate (DDEATH)

The Sparkplug™ Topic Namespace for a device Death Certificate is:

namespace/group_id/DDEATH/edge_node_id/device_id

It is the responsibility of the MQTT EoN node to indicate the real-time state of either physical legacy device using poll/response protocols and/or local logical devices. If the device becomes unavailable for any reason (no response, CRC error, etc.) it is the responsibility of the EoN node to publish a DDEATH on behalf of the end device.

Immediately upon reception of a DDEATH, any MQTT client subscribed to this device should set the data quality of all metrics to “STALE” and should note the time stamp when the DDEATH message was received.

7.4. Device Data Messages (DDATA)

Once an MQTT EoN node and associated devices are all online with proper Birth Certificates it is in a mode of quiescent Report by Exception (RBE) reporting of any metric that changes. This takes advantage of the native Continuous Session Awareness of MQTT to monitor the STATE of all connected devices and can rely on Report by Exception (RBE) messages for any metric value change over the MQTT session connection.

As defined above, the Data Topic for an MQTT device is:

namespace/group_id/DDATA/edge_node_id/device_id

The payload of DDATA messages can contain one or more metric values that need to be reported.

7.5. SCADA/IIoT Host Birth and Death Certificates

In infrastructures where multiple MQTT Servers provide redundancy and scalability, the MQTT EoN nodes need to be aware of the “state” of the primary SCADA/IIoT Host application(s). This is accomplished with a unique set of Birth/Death Certificates that the SCADA/IIoT Host MQTT Client **MUST** publish when a new MQTT session is established. In the same manner used for the MQTT EoN node NBIRTH/NDEATH certificate generation, the

SCADA/IIoT Host Birth/Death certificates will use a combination of the built in MQTT Will Topic and Will Payload Death Certificate in conjunction with an application level Birth Certificate that is published as an MQTT message.

The topic that a SCADA/IIoT Host node will use is identical for both the Birth and the Death certificate. The Topic Namespace will be:

STATE/scada_host_id

It uses an aspect of the MQTT transport called a “RETAINED” publish to maintain the current state of the Primary Host MQTT Client session state to all available MQTT Servers. The usage of the Host state is explained in detail in the section on Sparkplug™ MQTT Session Management.

The format of the **scada_host_id** can be valid String with the exception of the reserved characters of ‘+’ (plus), ‘/’ (forward slash), and ‘#’ (number sign).

7.5.1. SCADA/IIoT Host Death Certificate Payload (STATE)

When the SCADA/IIoT Host MQTT client establishes an MQTT session to the MQTT Server(s), the Death Certificate will be part of the Will Topic and Will Payload registered in the MQTT CONNECT transaction. The **Will Topic** as defined above will be:

STATE/scada_host_id

The Will Payload will be the UTF-8 STRING “**OFFLINE**”.

The Will RETAIN flag will be set to TRUE, and the Will QoS will be set to 1.

7.5.1. SCADA/IIoT Birth Certificate Payload (STATE)

The first message a SCADA/IIoT MQTT Host MUST publish is a Birth Certificate. The SCADA/IIoT Host Death Certificate is registered above within the actual establishment of the MQTT session and is published as a part of the native MQTT transport if the MQTT session terminates for any reason.

The Birth Certificate that is defined here is an application level message published by the Host SCADA/IIoT MQTT Client applications.

The topic used for the Host Birth Certificate is identical to the topic used for the Death Certificate:

STATE/scada_host_id

The Birth Certificate Payload is the UTF-8 STRING “**ONLINE**”.

The RETAIN flag for the Birth Certificate is set to **TRUE**, and the Quality of Service (QoS) is set to 1.

7.6. MQTT EoN Node Command (NCMD)

The NCMD command topic provides the Topic Namespace used to send commands to any connected EoN nodes. This means sending an updated metric value to an associated metric included in the NBIRTH metric list.

namespace/group_id/NCMD/edge_node_id

7.7. Device Command (DCMD)

The DCMD topic provides the Topic Namespace used to publish metrics to any connected device. This means sending a new metric value to an associated metric included in the DBIRTH metric list.

namespace/group_id/DCMD/edge_node_id/device_id

8. Sparkplug™ MQTT Session Management and Message Flow

An MQTT based SCADA system is unique in that the Host node is NOT responsible for establishing and maintaining connections to the devices as is the case in most existing legacy poll/response device protocols. With an MQTT based architecture, both the Host application as well as the devices establish MQTT Sessions with a central MQTT Server(s). This is the desired functionality as it provides the necessary decoupling from any one application and any given device. Additional MQTT clients can connect and subscribe to any of the real time data without impacting the primary SCADA Host application(s).

Due to the nature of real time SCADA solutions, it is very important for the primary SCADA Host and all connected MQTT EoN nodes to have the MQTT Session STATE information for each other. In order to accomplish this the Sparkplug™ Topic Namespace definitions for Birth/Death certificates along with the defined payloads provide both state and context between the SCADA Host MQTT client and the associated node side MQTT Clients. In most use cases and solution scenarios there are two primary reasons for this “designation” of a primary SCADA Host:

1. Only the Host *Primary Application(s)* should have the permission to issue commands to end devices.
2. In high availability and redundancy use cases where multiple MQTT Servers are used, MQTT EoN nodes need to be aware of whether *Primary Application* has network connectivity to each MQTT Server in the infrastructure. If the *Primary Application* STATE shows that an EoN node is connected to an MQTT Server that the *Primary Application* is **NOT** connected to, then the EoN node should walk to the next available MQTT Server where STATE for the *Primary Application* is ‘ONLINE’.

8.1. Primary Application Session Establishment

The *Primary Application* upon startup or reconnect will immediately try to create a Host MQTT Session with the configured *MQTT Server infrastructure*. Note that the establishment of an MQTT Host session is asynchronous of any other MQTT Client session. If EoN nodes are already connected to the *MQTT Server infrastructure*, the *Primary Application* will synchronize with them. If associated EoN nodes are not connected, *Primary Application* will register them when they publish their Birth Certificate.

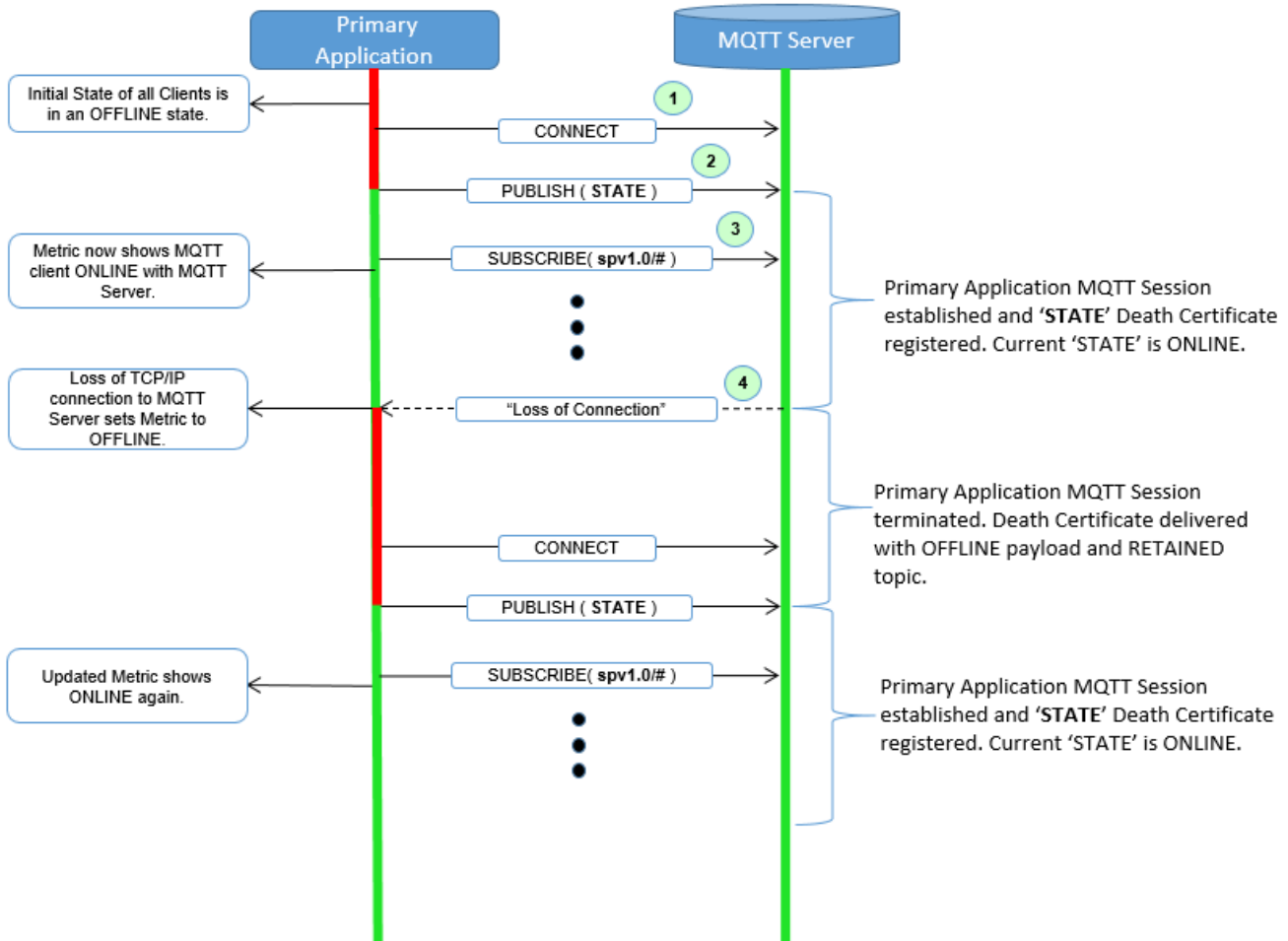



Figure 3 - Host Session Establishment

The session diagram in Figure 3 - Host Session Establishment shows a very simple topology with a single MQTT Server. The steps outlined in the session diagram are defined as follows:

1. *Primary Application* will try to create an MQTT Session using the MQTT CONNECT Control Packet (refer to section 3.1 in the MQTT V3.1.1 specification). A Death Certificate is constructed into the Will Topic and Will Payload of the Connect Control Packet with a Will QoS = 1 and Will Retain = true. The MQTT CONNECT Control Packet is acknowledged as successful with a valid CONNACK Control Packet. From this point forward in time, the MQTT Server is ready to deliver a Host Death Certificate any time the *Primary Application* MQTT Client loses connectivity to the MQTT Server.
2. Once an MQTT Session has been established, *Primary Application* will publish a new STATE message as defined in section 7.5.1, SCADA/IIoT Birth Certificate Payload. At this point, *Primary Application* can update the MQTT Client metrics in the *Primary Application* with a current state of ONLINE.

3. With the MQTT Session established, and a STATE Birth Certificate published, the *Primary Application* will issue an MQTT subscription for the defined Sparkplug™ Topic Namespace. The *Primary Application* is now ready to start receiving MQTT messages from any connected EoN node within the infrastructure. Since the *Primary Application* is also relying on the MQTT Session to the MQTT Server(s), the availability of Servers to the *Primary Application* is also being monitored and reflected in the MQTT Client metrics in the *Primary Application*.
4. If at any point in time *Primary Application* loses connectivity with the defined MQTT Server(s), the ONLINE state of the Server is immediately reflected in the MQTT Client metrics in the *Primary Application*. All metric data associated with any MQTT EoN node that was connected to that MQTT Server will be updated to a “**STALE**” data quality. 

8.2. EoN node Session Establishment

Any EoN node in the MQTT infrastructure must establish an MQTT Session prior to providing information for connected devices. Most implementations of an MQTT EoN node for real time SCADA will try to maintain a persistent MQTT Session with the *MQTT Server infrastructure*. But there are use cases where the MQTT Session does not need to be persistent. In either case, an EoN node can try to establish an MQTT session at any time and is completely asynchronous from any other MQTT Client in the infrastructure. The only exception to this rule is the use case where there are multiple MQTT Servers and a Primary Host application.

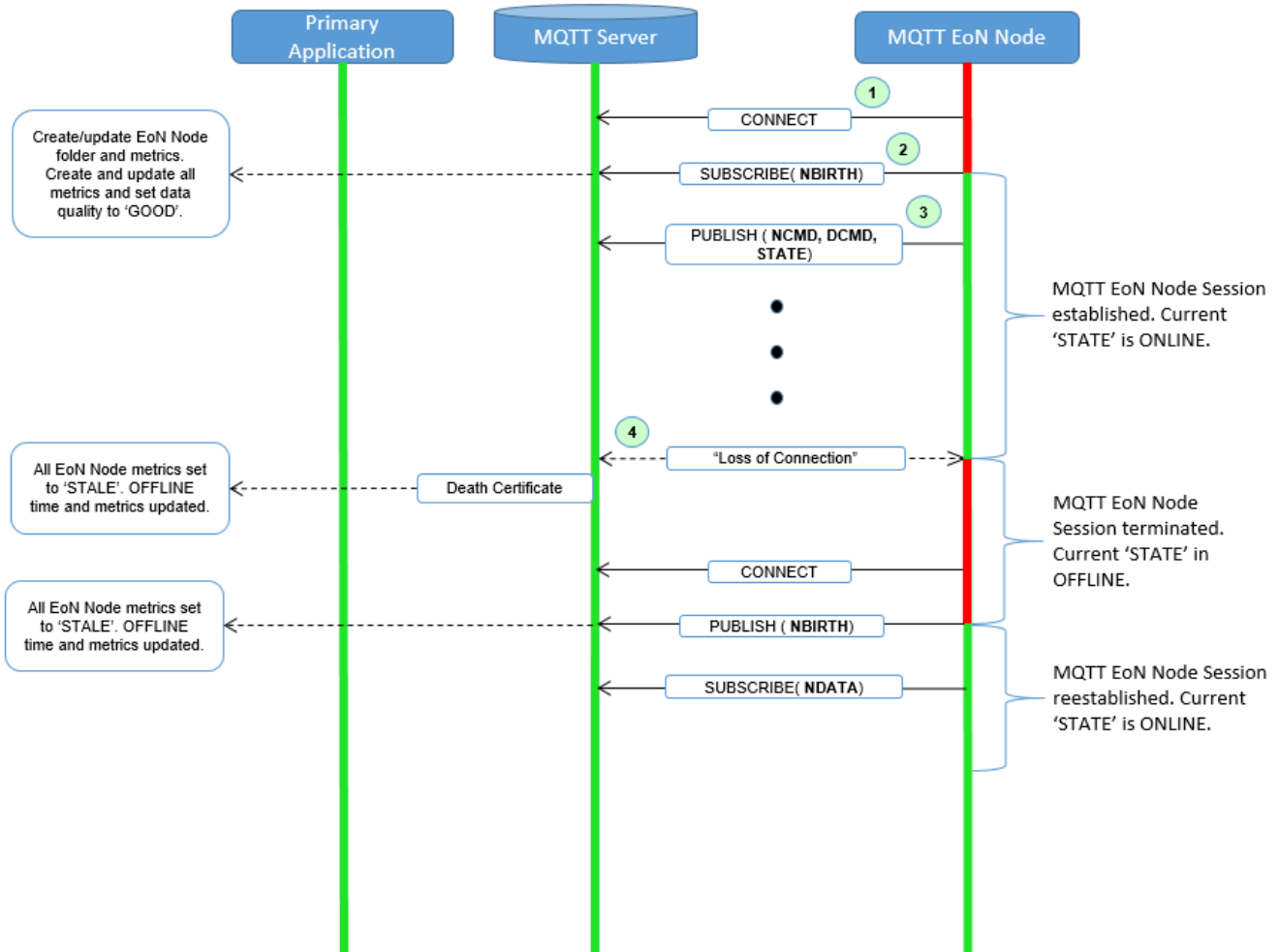


Figure 4 - EoN node MQTT Session Establishment

The session diagram in Figure 4 - EoN node MQTT Session Establishment shows a very simple topology with a single MQTT Server. The steps outlined in the session diagram are defined as follows:

1. The EoN node MQTT client will attempt to create an MQTT session to the available MQTT Server(s) using the MQTT **CONNECT** Control Packet (*refer to section 3.1 in the MQTT V3.1.1 specification*). The Death Certificate constructed into the Will Topic and Will Payload follows the format defined in section 0,

2. EoN Node Death Certificate (NDEATH). The MQTT CONNECT Control Packet is acknowledged as successful with a valid CONNACK Control Packet. From this point forward in time, the MQTT Server is ready to deliver an EoN node Death Certificate to any subscribing MQTT Client any time connectivity is lost.
3. The subscription to NCMD level topics ensures that EoN targeted messages from the *Primary Application* are delivered. The subscription to DCMD ensures that device targeted messages from the *Primary Application* are delivered. In applications with multiple MQTT Servers and designated Primary Host applications, the subscription to STATE informs the EoN node the current state of the Primary SCADA/IIoT Host. At this point the EoN node has fully completed the steps required for establishing a valid MQTT Session with the *Primary Application*.
4. Once an MQTT Session has been established, the EoN node MQTT client will publish an application level NBIRTH as defined in section 7.1.2, EoN Node Birth Certificate (NBIRTH). At this point, the *Primary Application* will have all the information required to build out the EoN node metric structure and show the EoN node in an “ONLINE” state.
5. If at any point in time the EoN node MQTT Client loses connectivity to the defined MQTT Server(s), a Death Certificate is issue by the MQTT Server on behalf of the EoN node. Upon receipt of the Death Certificate, the *Primary Application* will set the state of the EoN node to ‘OFFLINE’ and update all timestamp metrics concerning the connection. Any defined metrics will be set to a “STALE” data quality.

8.3. MQTT Device Session Establishment

The Sparkplug™ specification is provided to get real time process variable information from existing and new end devices measuring, monitoring and controlling a physical process into an MQTT MOM infrastructure and the *Primary Application* Industrial Internet of Things application platform. In the context of this document an MQTT Device can represent anything from existing legacy poll/response driven PLCs, RTUs, HART Smart Transmitter, etc., to new generation automation and instrumentation devices that can implement a conformant MQTT client natively.

The preceding sections in this document detail how the *Primary Application* interacts with the *MQTT Server infrastructure* and how that infrastructure interacts with the notion of an MQTT EoN node. But to a large extent the technical requirements of those pieces of the infrastructure have already been provided. For most use cases in this market sector the primary focus will be on the implementation of the Sparkplug™ specification between the native device and the EoN node API's.

In order to expose and populate the metrics from any intelligent device, the following simple session diagram outlines the requirements:

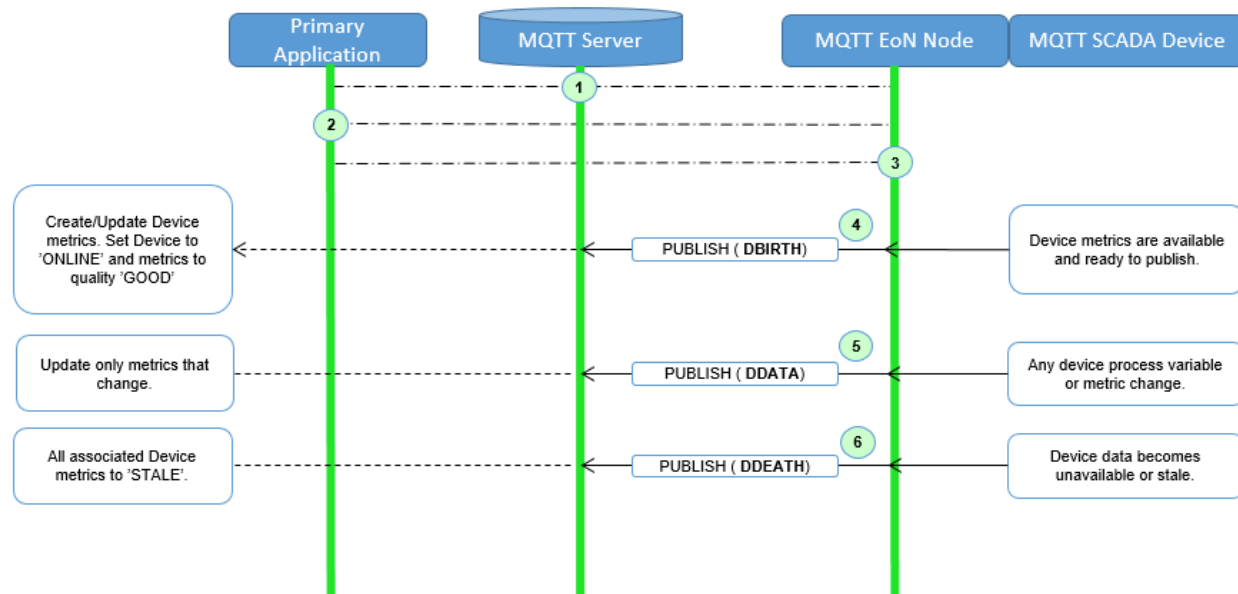


Figure 5 - MQTT Device Session Establishment

The session diagram in Figure 5 - MQTT Device Session Establishment shows a simple topology with all the Sparkplug™ elements in place i.e. *Primary Application*, MQTT Server(s), MQTT EoN node and this element, the device element. The steps outlined in the session diagram are defined as follows:

This flow diagram assumes that at least one MQTT Server is available and operational within the infrastructure. Without at least a single MQTT Server the remainder of the infrastructure is unavailable.

1. Assuming MQTT Server is available.
2. Assuming the *Primary Application* established MQTT Session with the MQTT Server(s).
3. The Session Establishment of the associated MQTT EoN node is described in section 8.2, EoN node Session Establishment. This flow diagram assumes that the EoN node session has already been established with the *Primary Application*. Depending on the target platform, the EoN node may be a physical “Edge of Network” gateway device polling physical legacy devices via Modbus, AB, DNP3.0, HART, etc., a MQTT enabled sensor or device, or it might be a logical implementation of one of the Eclipse Tahu reference

implementations for prototype EoN nodes running on the Raspberry PI platform. Regardless of the implementation, at some point the device interface will need to provide a state and associated metrics to publish to the MQTT infrastructure.

4. State #4 in the session diagram represents the state at which the device is ready to report all of its metric information to the MQTT EoN node as defined in Sparkplug™. It is the responsibility of the EoN node (logical or physical) to put this information in a form defined in 0,
- 5.
6. Device Birth Certificate (DBIRTH). Upon receiving the DBIRTH message, the *Primary Application* can build out the proper metric structure.
7. Following the Sparkplug™ specification in section 7.4, Device Data Messages (DDATA), all subsequent metrics are published to the *Primary Application* on a Report by Exception (RBE) basis using the DDATA message format.
8. In at any time the device (logical or physical) cannot provide real time information, the MQTT EoN node specification requires that an DDEATH be published. This will inform the *Primary Application* that all metric information be set to a “**STALE**” data quality.

8.4. General MQTT applications and non-primary Applications.

As noted above, there is the notion of a *Primary Application* instance in the infrastructure that has the required permissions to send command to nodes and devices and the fact that all EoN nodes need to know the *Primary Application* is connected to the same MQTT Server its connected to or it needs to walk to another one in the infrastructure. Both are known requirements of a mission critical SCADA system.

But unlike legacy SCADA system implementations, all real time process variable information being published thru the MQTT infrastructure is available to any number of additional MQTT Clients in the business that might be interested in subsets if not all of the real time data.

The **ONLY** difference between a *Primary Application* MQTT client and all other clients that *non-primary* Client do **NOT** issue the STATE Birth/Death certificates.

9. Sparkplug™ MQTT Data and Command Messages

Looking back in this document we've described the following components:

- Primary Application
- MQTT Server(s)
- Edge of Network (EoN) nodes
- Devices
- Topic Namespace
- Birth Certificates
- Death Certificates
- STATE Messages
- Primary Application, EoN node, and Device Session Establishment

All of these specifications and definitions get to the primary goal of Sparkplug™, that is to deliver a rich set of real time device metric data extremely efficiently to many data consumers within the Enterprise while still providing a best in class Command/Control SCADA/IIoT system.

The disruptive notion of the emerging IIoT mindset is that intelligent devices should be smart enough to deliver metric information to the infrastructure when it is required. But the fact of the matter is that the existing population of 100's of millions of the smart devices need to be "asked" if something has changed using poll/response protocols. This is why we're seeing the emergence of edge devices throughout the industrial sector. For the decade or more that it will take for device manufactures to embed IIoT technology natively, the solution being employed today is to place this capability in small embedded devices closer to the data producers themselves. So within the Sparkplug™ specification these devices called Edge of Network Nodes (EoN) represent this new class of Gateway, Edge Controller, Edge of Network Node, Protocol Gateway, and many more acronyms for the same class of devices. The capabilities of these devices are in an extreme range of low power microcontrollers to multicore Intel and ARM based processors. The operating systems range from full embedded Linux kernels and Windows embedded to small bare metal RTOS's. Regardless of the category these gateway devices fall into the simplicity of MQTT and the Sparkplug™ specification should be applicable across the board.

This section of the Sparkplug™ specification goes into detail on how metrics are published/subscribed to within an MQTT infrastructure in real time and the resulting metric information that the *Primary Application* can read/write to.

9.1. EoN NDATA and NCMD Messages

We'll start this section with a description of how metric information is published to the *Primary Application* from an EoN node in the MQTT infrastructure. The definition of an EoN node is generic in that it can represent both physical "Edge of Network Gateway" devices that are interfacing with existing legacy equipment and a logical MQTT endpoint for devices that natively implement the Sparkplug™ specification. Section 7.4.1 above defines the Birth Certificate MQTT Payload and the fact that it can provide any number of metrics that will be exposed in the *Primary Application*. Some of these will be "read only" such as:

- EoN Manufacture ID
- EoN Device Type
- EoN Serial Number
- EoN Software Version Number
- EoN Configuration Change Count
- EoN Position (if GPS device is available)
- EoN Cellular RSSI value (if cellular is being used)
- EoN Power Supply voltage level
- EoN Temperature

Other metrics may be dynamic and "read/write" such as:

- EoN Rebirth command to republish all EoN and Device Birth Certificates.
- EoN Next server command to move to next available MQTT Server.
- EoN Reboot command to reboot the EoN node.
- EoN Primary Network (PRI_NETWORK) where 1 = Cellular, 2 = Ethernet

The important point to realize is that the metrics exposed in the *Primary Application* for use in the design of applications are completely determined by what metric information is published in the NBIRTH. Each specific EoN node can best determine what data to expose, and how to expose it, and it will automatically appear in the *Primary Application* metric structure. Metrics can even be added dynamically at runtime and with a new NBIRTH. These metrics will automatically be added to the *Primary Application* metric structure.

The other very important distinction to make here is that EoN node NDATA and NCMD messages are decoupled from the device level data and command messages of DDATA and DCMD. This decoupling in the Topic Namespace is important because it allows interaction from all MQTT Clients in the system (to the level of permission and application) with the EoN nodes, but NOT to the level of sending device commands. The *Primary Application* could provide a configuration parameter that would BLOCK output DDATA and DCMD messages but still allow NDATA and NCMD messages to flow. In this manner, multiple application systems can be connected to the same MQTT infrastructure, but only the ones with DCMD enabled can publish Device commands.

The following simple message flow diagram demonstrates the messages used to update a changing cellular RSSI value in the *Primary Application* and sending a command from the *Primary Application* to the EoN node to use a different primary network path.

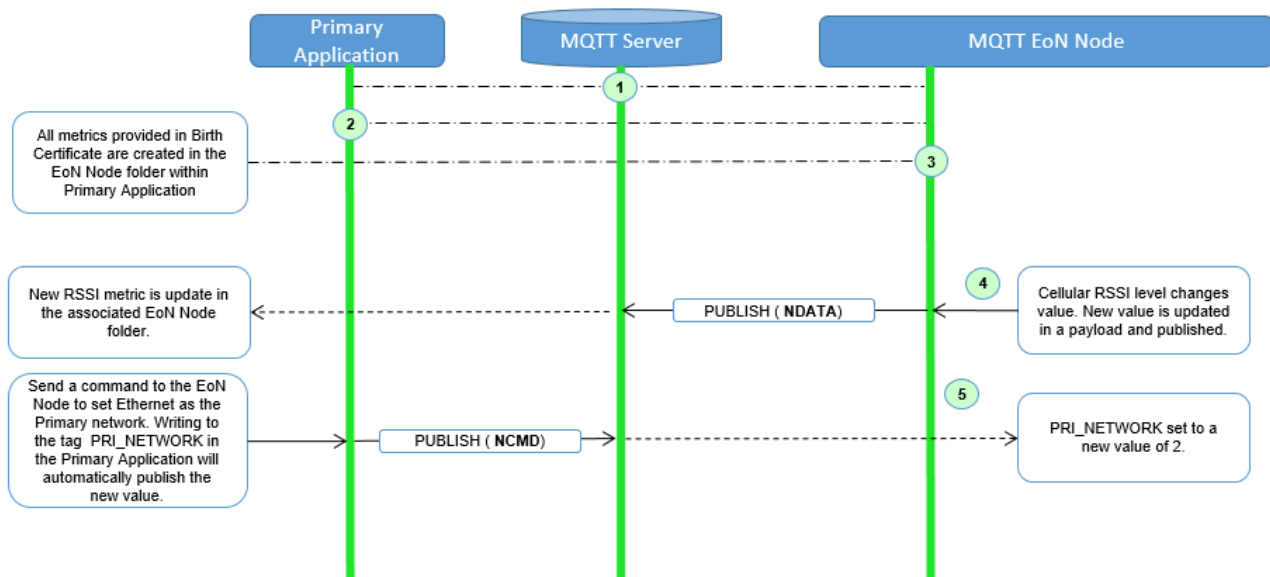


Figure 6 - EoN node NDATA and NCMD Message Flow

1. Assuming MQTT Server is available.
2. Assuming the *Primary Application* established MQTT Session with the MQTT Server(s).
3. The EoN node has an established MQTT Session and the NBIRTH has been published. *Primary Application* now has all defined metrics and their current value.
4. The EoN node is monitoring its local cellular RSSI level. The level has changed and now the EoN node wants to publish the new value to the associated metric in *Primary Application*.
 - The EoN node publishes a message (PUBLISH (NDATA)) to the MQTT Server.
 - The MQTT Server receives the message and updates the associated EoN Node folder in the Primary Application.
5. From an operational requirement, the EoN node needs to be told to switch its primary network interface from cellular to Ethernet. From the *Primary Application*, the new value is written to the metric and will automatically publish the new value to the EoN node parameters.
 - The Primary Application publishes a message (PUBLISH (NCMD)) to the MQTT Server.
 - The MQTT Server receives the message and sets the PRI_NETWORK to a new value of 2 in the EoN node.

10. Primary Application STATE in Multiple MQTT Server Topologies

For implementations with multiple MQTT Servers, there is one additional aspect that needs to be understood and managed properly. When multiple MQTT Servers are available there is the possibility of “stranding” and EoN node if the Primary command/control of the *Primary Application* loses network connectivity to one of the MQTT Servers. In this instance the EoN node would stay properly connected to the MQTT Server publishing information not knowing that *Primary Application* was not able to receive the messages. When using multiple MQTT Servers, the *Primary Application* instance must be configured to publish a STATE Birth Certificate and all EoN nodes need to subscribe to this STATE message.

The *Primary Application* will need to specify whether it is a “Primary” command/control instance or not. If it is a primary instance then every time it establishes a new MQTT Session with an MQTT Server, the STATE Birth Certificate defined in section above is the first message that is published after a successful MQTT Session is established.

EoN node devices in an infrastructure that provides multiple MQTT Servers can establish a session to any one of the MQTT Servers. Upon establishing a session, the EoN node should issue a subscription to the STATE message published by *Primary Application*. Since the STATE message is published with the RETAIN message flag set, MQTT will guarantee that the last STATE message is always available. The EoN node should examine the payload of this message to ensure that it is a value of “ONLINE”. If the value is “OFFLINE”, this indicates the Primary Application has lost its MQTT Session to this particular MQTT Server. This should cause the EoN node to terminate its session with this MQTT Server and move to the next available MQTT Server that is available. This use of the STATE message in this manner ensures that any loss of connectivity to an MQTT Server to the *Primary Application* does not result in EoN nodes being “stranded” on an MQTT server because of network issues. The following message flow diagram outlines how the STATE message is used when three (3) MQTT Servers are available in the infrastructure:

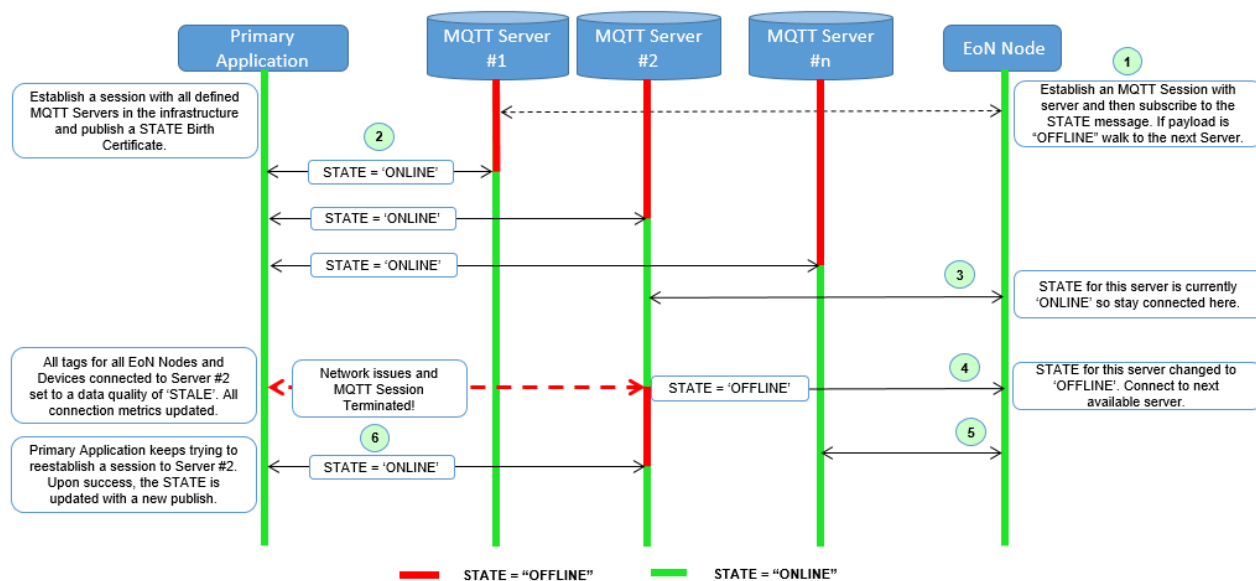


Figure 7 – Primary Application STATE flow diagram

1. When an EoN node is configured with multiple available MQTT Servers in the infrastructure it should issue a subscription to the *Primary Application* STATE message. The EoN nodes are free to establish an MQTT Session to any of the available servers over any available network at any time and examine the current

STATE value. If the STATE message payload is 'OFFLINE' then the EoN node should disconnect and walk to the next available server.

2. Upon startup, the configured Primary Application, the MQTT Session will be configured to register the *Primary Application* DEATH Certificate that indicates STATE is 'OFFLINE' with the message RETAIN flag set to true. Then the *Primary Application* BIRTH Certificate will be published with a STATE payload of 'ONLINE'.
3. As the EoN node walks its available MQTT Server table, it will establish an MQTT Session with a server that has a STATE message with a payload of 'ONLINE'. The EoN node can stay connected to this server if its MQTT Session stays intact and it does not receive the *Primary Application* DEATH Certificate.
4. Having a subscription registered to the MQTT Server on the STATE topic will result in any change to the current the *Primary Application* STATE being received immediately. In this case, a network disruption causes the *Primary Application* MQTT Session to server #2 to be terminated. This will cause the MQTT Server, on behalf of the now terminated the *Primary Application* MQTT Client to publish the DEATH certificate to anyone that is currently subscribed to it. Upon receipt of the *Primary Application* DEATH Certificate this EoN node will move to the next MQTT Server in its table.
5. The EoN node moved to the next available MQTT Server and since the current STATE on this server is 'ONLINE', it can stay connected.
6. In the meantime, the network disruption between *Primary Application* and MQTT Server #2 has been corrected. The *Primary Application* has a new MQTT Session established to server #2 with an update Birth Certificate of 'ONLINE'. Now MQTT Server #2 is ready to accept new EoN node session requests.

11. Sparkplug™ Persistent versus Non-Persistent Connections

Persistent connections are intended to remain connected to the MQTT infrastructure at all times. They never send an MQTT DISCONNECT message during normal operation. This fact lets the *Primary Application* provide the real-time state of every persistent node in the infrastructure within the configured MQTT Keep Alive period using the Birth/Death mechanisms defined above.

But in some use cases, such as sending GPS coordinates for asset tracking or other IOT applications with periodic data from sensors, MQTT enabled devices do not need to remain connected to the MQTT infrastructure. In these use cases, all the Device needs to do is to issue an MQTT DISCONNECT control packet prior to going offline to leave the MQTT infrastructure “gracefully”. In this case an MQTT Device or associated Device DEATH certificate will most normally not be seen. System designers just need to be aware that the metric in *Primary Application* in this case will represent “Last Known Good” values with a time stamp of this data where the current state of the of the MQTT Device is not a real-time indication. The *Primary Application* metric time stamp values can be used to determine when the values from this node were last updated.

Non-persistent MQTT Enabled Devices should still register a proper DEATH Certificate upon the establishment of an MQTT session. In this manner, the *Primary Application* can still have a good representation of Last Known Good process variable versus the fact that the MQTT session was terminated prior to the EoN node being able to complete its transaction.

12. Contact Information

The Eclipse Foundation appreciates any and all feedback on this specification. It is only from the feedback of end users that Sparkplug™ can a viable and vibrant reference implementation for MQTT based SCADA and IIoT solutions.

For any questions regarding this Sparkplug™ specification or for more information, please use the following details:

Eclipse Foundation

Website: www.eclipse.org

Phone: +1.613.224.9461

APPENDIX 1 SPARKPLUG™ B PAYLOAD DEFINITION

Sparkplug™ Specification

Sparkplug™ B Payload Definition Version v1.0



Revision Number	Date	Author	Description
1.0	1/16/2017	Cirrus Link	Initial Release

Table of Contents

Table of Figures.....	39
13. Introduction.....	40
14. Sparkplug™ B MQTT Payload Definition.....	40
14.1. Google Protocol Buffers.....	41
14.2. Sparkplug™ B Google Protocol Buffer Schema.....	41
14.3. Payload Metric Naming Convention.....	44
15. Sparkplug™ Bv1.0 Payload Components.....	46
15.1. Payload Component Definitions.....	46
15.1.1. Payload.....	46
15.1.2. Metric.....	46
15.1.3. MetaData.....	48
15.1.4. PropertySet.....	48
15.1.5. PropertyValue.....	49
15.1.6. PropertySetList.....	49
15.1.7. DataSet.....	49
15.1.8. DataSet.Row.....	50
15.1.9. DataSet.DataSetValue.....	50
15.1.10. Template.....	50
15.1.11. Template.Parameter.....	51
15.2. Sparkplug™ Bv1.0 Payload Datatypes.....	51
15.2.1. Metric Datatypes.....	51
15.2.2. PropertyValue Datatypes.....	53
15.2.3. DataSetValue DataTypes.....	55
15.2.4. Template.Parameter DataTypes.....	56
16. Payloads by Message Type.....	58
16.1. NBIRTH.....	58
16.2. DBIRTH.....	59
16.3. NDATA.....	59
16.4. DDATA.....	59
16.5. NCMD.....	60
16.6. DCMD.....	60
16.7. DDEATH.....	60

16.8.	NDEATH.....	60
16.9.	STATE	60
17.	Payload Representation on Backend Applications	61
17.1.	NBIRTH.....	61
17.2.	DBIRTH	62
17.3.	NDATA.....	64
17.4.	DDATA.....	65
17.5.	NCMD.....	66
17.6.	DCMD.....	66
17.7.	NDEATH.....	67
17.8.	DDEATH.....	67
17.9.	STATE	68

Table of Figures

Figure 1 – Payload Metric Folder Structure.....45

Figure 2 – Sparkplug™ B Metric Structure 162

Figure 3 – Sparkplug™ B Metric Structure 264

13. Introduction

The MQTT message transport specification does not define any required data payload format. From an MQTT infrastructure standpoint, the payload is treated as an agnostic binary array of bytes that can be anything from no payload at all, to a maximum of 256MB. But for applications within a known solution space to work using MQTT the payload representation does need to be defined.

This section of the Sparkplug™ specification defines how an MQTT Sparkplug™ B payload is encoded and the data that is required. Note that Sparkplug™ supports multiple payloads encoding definitions. For more detailed information on the payload formats and encoding associated with each Sparkplug™ release see the appendices.

The majority of devices connecting into next generation IIoT infrastructure are legacy equipment using poll/response protocols. This means we must take in account register based data from devices that talk protocols like Modbus. The existing legacy equipment needs to work in concert with emerging IIoT equipment that is able to leverage message transports like MQTT natively.

14. Sparkplug™ B MQTT Payload Definition

The goal of the Sparkplug™ is to provide a specification that both OEM device manufactures and application developers can use to create rich and interoperable SCADA/IIoT solutions using MQTT as a base messaging technology. In Sparkplug™ B message payload definition, the goal was to create a simple and straightforward binary message encoding that could be used primarily for legacy register based process variables (Modbus register value for example).

The Sparkplug™ B MQTT payload specification has come about based on the feedback from many system integrators and end user customers who wanted to be able to natively support a much richer data model within the MQTT infrastructures that they were designing and deploying. Using the feedback from the user community Sparkplug™ B provides support for:

- Complex data types using templates.
- Datasets.
- Richer metrics with the ability to add property metadata for each metric.
- Metric alias support to maintain rich metric naming while keeping bandwidth usage to a minimum.
- Historical data.
- File data.

Sparkplug™ B definition creates a bandwidth efficient data transport for real time device data. For WAN based SCADA/IIoT infrastructures this equates to lower latency data updates while minimizing the amount of traffic and therefore cellular and/or VSAT bandwidth required. In situations where bandwidth savings is not the primary concern, the efficient use enables higher throughput of more and interesting data eliminating sensor data that have been left stranded in the field. It is also ideal for LAN based SCADA infrastructures equating to higher throughput of real time data to consumer applications without requiring extreme networking topologies and/or equipment.

There are many data encoding technologies available that can all be used in conjunction with MQTT. Sparkplug™ B selected an existing, open, and highly available encoding scheme that efficiently encodes register based process variables. The encoding technology selected for Sparkplug™ B is Google Protocol Buffers also referred to as **Google Protobufs**.

14.1. Google Protocol Buffers

“Protocol Buffers are a way of encoding structured data in an efficient yet extensible format.”

Google Protocol Buffers, sometimes referred to as **“Google Protobufs”**, provide the efficiency of packed binary data encoding while providing the structure required to make it easy to create, transmit, and parse register based process variables using a standard set of tools while enabling emerging IIoT requirements around richer metadata. Google Protocol Buffers development tools are available for:

- C
- C++
- C#
- Java
- Python
- GO
- JavaScript

Additional information on Google Protocol Buffers can be found at:

<https://developers.google.com/protocol-buffers/>

14.2. Sparkplug™ B Google Protocol Buffer Schema

Using lessons learned on the feedback from the Sparkplug™ A implementation a new Google Protocol Buffer schema was developed that could be used to represent and encode the more complex data models being requested. The entire Google Protocol Buffers definition is below.

```
syntax = "proto2";

//
// To compile:
// cd client_libraries/java
// protoc --proto_path=../../ --java_out=src/main/java ../../Sparkplug™_b.proto
//
package org.eclipse.tahu.protobuf;

option java_package      = "org.eclipse.tahu.protobuf";
option java_outer_classname = "Sparkplug™BProto";

message Payload {
  /*
    // Indexes of Data Types

    // Unknown placeholder for future expansion.
    Unknown          = 0;

    // Basic Types
    Int8              = 1;
    Int16             = 2;
    Int32             = 3;
    Int64             = 4;
    UInt8             = 5;
    UInt16            = 6;
    UInt32            = 7;
    UInt64            = 8;
    Float             = 9;
    Double            = 10;
    Boolean            = 11;
    String             = 12;
    DateTime           = 13;
    Text               = 14;
```

```

// Additional Metric Types
UUID          = 15;
DataSet       = 16;
Bytes         = 17;
File          = 18;
Template      = 19;

// Additional PropertyValue Types
PropertySet   = 20;
PropertySetList = 21;

*/

message Template {

    message Parameter {
        optional string name          = 1;
        optional uint32 type          = 2;

        oneof value {
            uint32 int_value           = 3;
            uint64 long_value          = 4;
            float  float_value         = 5;
            double double_value        = 6;
            bool   boolean_value       = 7;
            string string_value        = 8;
            ParameterValueExtension extension_value = 9;
        }

        message ParameterValueExtension {
            extensions 1 to max;
        }
    }

    optional string version          = 1;           // The version of the Template to
                                                    // prevent mismatches
    repeated Metric metrics          = 2;           // Each metric is the name of the
                                                    // metric and the datatype of the
                                                    // member but does not contain a
                                                    // value
    repeated Parameter parameters   = 3;
    optional string template_ref     = 4;           // Reference to a template if this is
                                                    // extending a Template or an
                                                    // instance - must exist if an
                                                    // instance
    optional bool is_definition      = 5;
    extensions 6 to max;
}

message DataSet {

    message DataSetValue {

        oneof value {
            uint32 int_value           = 1;
            uint64 long_value          = 2;
            float  float_value         = 3;
            double double_value        = 4;
            bool   boolean_value       = 5;
            string string_value        = 6;
            DataSetValueExtension extension_value = 7;
        }

        message DataSetValueExtension {
            extensions 1 to max;
        }
    }

    message Row {
        repeated DataSetValue elements = 1;
    }
}

```

```

        extensions                2 to max;    // For third party extensions
    }

    optional uint64    num_of_columns    = 1;
    repeated string    columns           = 2;
    repeated uint32    types             = 3;
    repeated Row       rows              = 4;
    extensions         5 to max;    // For third party extensions
}

message PropertyValue {

    optional uint32    type              = 1;
    optional bool      is_null           = 2;

    oneof value {
        uint32         int_value         = 3;
        uint64         long_value        = 4;
        float          float_value       = 5;
        double         double_value      = 6;
        bool           boolean_value     = 7;
        string         string_value      = 8;
        PropertySet    propertyset_value = 9;
        PropertySetList propertysets_value = 10;    // List of Property Values
        PropertyValueExtension extension_value = 11;
    }

    message PropertyValueExtension {
        extensions                1 to max;
    }
}

message PropertySet {
    repeated string    keys              = 1;    // Names of the properties
    repeated PropertyValue values        = 2;
    extensions         3 to max;
}

message PropertySetList {
    repeated PropertySet propertyset = 1;
    extensions                2 to max;
}

message MetaData {
    // Bytes specific metadata
    optional bool    is_multi_part    = 1;

    // General metadata
    optional string content_type      = 2;    // Content/Media type
    optional uint64 size              = 3;    // File size, String size, Multi-part
                                           // size, etc.
    optional uint64 seq              = 4;    // Sequence number for multi-part
                                           // messages

    // File metadata
    optional string file_name         = 5;    // File name
    optional string file_type         = 6;    // File type (i.e. xml, json, txt, cpp,
                                           // etc.)
    optional string md5               = 7;    // md5 of data

    // Catchalls and future expansion
    optional string description       = 8;    // Could be anything such as json or
                                           // xml of custom properties

    extensions                9 to max;
}

message Metric {

    optional string    name            = 1;    // Metric name - should only be
                                           // included on birth

```

```

optional uint64    alias          = 2;          // Metric alias - tied to name on birth
// and included in all later DATA
// messages
optional uint64    timestamp      = 3;          // Timestamp associated with data
// acquisition time
optional uint32     datatype       = 4;          // DataType of the metric/tag value
optional bool       is_historical  = 5;          // If this is historical data and
// should not update real time tag
optional bool       is_transient  = 6;          // Tells consuming clients such as MQTT
// Engine to not store this as a tag
optional bool       is_null       = 7;          // If this is null - explicitly say so
// rather than using -1, false, etc. for
// some datatypes.
optional Metadata metadata        = 8;          // Metadata for the payload
optional PropertySet properties = 9;

oneof value {
  uint32    int_value              = 10;
  uint64    long_value             = 11;
  float     float_value            = 12;
  double    double_value           = 13;
  bool      boolean_value          = 14;
  string    string_value           = 15;
  bytes     bytes_value            = 16;          // Bytes, File
  DataSet   dataset_value          = 17;
  Template  template_value         = 18;
  MetricValueExtension extension_value = 19;
}

message MetricValueExtension {
  extensions 1 to max;
}

optional uint64    timestamp      = 1;          // Timestamp at message sending time
repeated Metric    metrics        = 2;          // Repeated forever - no limit in Google
// Protobufs
optional uint64    seq            = 3;          // Sequence number
optional string     uuid          = 4;          // UUID to track message type in terms of
// schema definitions
optional bytes     body           = 5;          // To optionally bypass the whole
// definition above
extensions         6 to max;      // For third party extensions
}

```

14.3. Payload Metric Naming Convention

For the remainder of this document JSON will be used to represent components of a Sparkplug™ B payload. It is important to note that the payload is a binary encoding and is not actually JSON. However, JSON representation is used in this document to represent the payloads in a way that is easy to read. For example, a simple Sparkplug™ B payload with a single metric can be represented in JSON as follows:

```

{
  "timestamp": <timestamp>,
  "metrics": [{
    "name": <metric_name>,
    "alias": <alias>,
    "timestamp": <timestamp>,
    "dataType": <datatype>,
    "value": <value>
  }],
  "seq": <sequence_number>
}

```

A simple Sparkplug™ B payload with values would be represented as follows:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "My Metric",
    "alias": 1,
    "timestamp": 1479123452194,
    "dataType": "String",
    "value": "Test"
  }],
  "seq": 2
}
```

Note that the ‘name’ of a metric may be hierarchical to build out proper folder structures for applications consuming the metric values. For example, in an application where an EoN node is connected to several devices or data sources, the ‘name’ could represent discrete folder structures of:

‘Metric Level 1/Metric Level 2/Metric Name’

Using this convention in conjunction with the **group_id**, **edge_node_id** and **device_id** already defined in the Topic Namespace, consuming applications can organize metrics in the same hierarchical fashion:

Metric	Value	Data Type
	value	type

Figure 8 – Payload Metric Folder Structure

15. Sparkplug™ Bv1.0 Payload Components

The Sparkplug™ specification document “*MQTT Topic Namespace and State Management*” document defines the Topic Namespace that Sparkplug™ uses to publish and subscribe between EoN nodes and applications within the MQTT infrastructure. Using that Topic Namespace, this section of the specification defines the actual payload contents of each message type in Sparkplug™ Bv1.0.

15.1. Payload Component Definitions

Sparkplug™ B consists of a series of one or more metrics with metadata surrounding those metrics. The following definitions explain the components that make up a payload.

15.1.1. Payload

A Sparkplug™ B payload is the top-level component that is encoded and used in an MQTT message. It contains some basic information such as a timestamp and a sequence number as well as an array of metrics which contain key/value pairs of data. A Sparkplug™ B payload includes the following components.

- **payload**
 - *timestamp*
 - This is the timestamp in the form of an unsigned 64-bit integer representing the number of milliseconds since epoch (Jan 1, 1970). It is highly recommended that this time is in UTC. This timestamp is meant to represent the time at which the message was published.
 - *metrics*
 - This is an array of metrics representing key/value/datatype values. Metrics are further defined in section 3.1.2.
 - *seq*
 - This is the sequence number which is an unsigned 64-bit integer. A sequence number must be included in the payload of every Sparkplug™ MQTT message. A NBIRTH message must always contain a sequence number of zero. All subsequent messages must contain a sequence number that is continually increasing by one in each message until a value of 255 is reached. At that point, the sequence number of the following message must be zero.
 - *uuid*
 - This is a field which can be used to represent a schema or some other specific form of the message. Example usage would be to supply a UUID which represents an encoding mechanism of the optional array of bytes associated with a payload.
 - *body*
 - This is an array of bytes which can be used for any custom binary encoded data.

15.1.2. Metric

A Sparkplug™ B metric is a core component of data in the payload. It represents a key/value/datatype along with metadata used to describe the information it contains. It includes the following components.

- **name**
 - This is the friendly name of a metric. It should be represented as a slash delimited UTF-8 string. The slashes in the string represent folders of the metric to represent hierarchical data structures. For example, ‘outputs/A’ would be a metric with a unique identifier of ‘A’ in the ‘outputs’ folder. There is no limit to the number of folders. However, across the infrastructure of MQTT publishers a defined folder should always remain a folder.
- **alias**

- This is an unsigned 64-bit integer representing an optional alias for a Sparkplug™ B payload. If supplied in an NBIRTH or DBIRTH it must be a unique number across this EoN nodes entire set of metrics. In other words, no two metrics for the same EoN node can have the same alias. Upon being defined in the NBIRTH or DBIRTH, subsequent messages can supply only the alias instead of the metric friendly name to reduce overall message size.
- **timestamp**
 - This is the timestamp in the form of an unsigned 64-bit integer representing the number of milliseconds since epoch (Jan 1, 1970). It is highly recommended that this time is in UTC. This timestamp is meant to represent the time at which the value of a metric was captured.
- **datatype**
 - This is an unsigned 32-bit integer representing the datatype. Datatypes are not explicitly defined in the Sparkplug™ B Protobuf definition. Instead they are defined in section 4 of this document.
- **is_historical**
 - This is a Boolean flag which denotes whether this metric represents a historical value. In some cases, it may be desirable to send metrics after they were acquired on a device or EoN node. This can be done for batching, store and forward, or sending local backup data during network communication loses. This flag denotes that the message should not be considered a real time/current value.
- **is_transient**
 - This is a Boolean flag which denotes whether this metric should be considered transient. Transient metrics can be considered those that are of interest to a back-end application(s) but shouldn't be stored in a historian on the backend.
- **is_null**
 - This is a Boolean flag which denotes whether this metric has a null value. This is Sparkplug™ B's mechanism of explicitly denoting a metric's value is actually null.
- **metadata**
 - This is a MetaData object associated with the metric for dealing with more complex datatypes. This is covered in section 3.1.3 of this document.
- **properties**
 - This is a PropertySet object associated with the metric for including custom key/value pairs of metadata associated with a metric. This is covered in section 3.1.4 of this document.
- **value**
 - The value of a metric utilizes the 'oneof' mechanism of Google Protocol Buffers. The value supplied with a metric must be one of the following types. Note if the metrics is_null flag is set to true the value can be omitted altogether.
 - *uint32*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *uint64*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *float*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *double*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *bool*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *string*

- Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
- *bytes*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
- *DataSet*
 - Defined in section 3.1.7 of this document.
- *Template*
 - Defined in section 3.1.10 of this document.

15.1.3. *MetaData*

A Sparkplug™ B MetaData object is used to describe different types of binary data. It includes the following components.

- **is_multi_part**
 - A Boolean representing whether this metric contains part of a multi-part message. Breaking up large quantities of data can be useful for keeping the flow of MQTT messages flowing through the system. Because MQTT requires in order delivery publishing very large messages can result in messages being blocked while delivery of large messages takes place.
- **content_type**
 - This is a UTF-8 string which represents the content type of a given metric value.
- **size**
 - This is an unsigned 64-bit integer representing the size of the metric value
- **seq**
 - If this is a multipart metric, this is an unsigned 64-bit integer representing the sequence number of this part of a multipart metric.
- **file_name**
 - If this is a file metric, this is a UTF-8 string representing the filename of the file.
- **file_type**
 - If this is a file metric, this is a UTF-8 string representing the type of the file.
- **md5**
 - If this is a byte array metric that can have a md5sum, this field can be used as a UTF-8 string to represent it.
- **description**
 - This is a freeform field with a UTF-8 string to represent any other pertinent metadata for this metric. It can contain JSON, XML, text, or anything else that can be understood by both the publisher and the subscriber.

15.1.4. *PropertySet*

A Sparkplug™ B PropertySet object is used with a metric to add custom properties to the object. The PropertySet is a map expressed as two arrays of equal size, one containing the keys and one containing the values. It includes the following components.

- **keys**
 - This is an array of UTF-8 strings representing the names of the properties in this PropertySet. It must contain the same number of values included in the array of PropertyValue objects.
- **values**
 - This is an array of PropertyValue objects representing the values of the properties in the PropertySet. It must contain the same number of items that are in the keys array.

15.1.5. PropertyValue

A Sparkplug™ B PropertyValue object is used to encode the value and datatype of the value of a property in a PropertySet. It includes the following components.

- **type**
 - This is an unsigned 32-bit integer representing the datatype of the value. Datatypes are not explicitly defined in the Sparkplug™ B Protobuf definition. Instead they are defined in section 4 of this document.
- **is_null**
 - This is a Boolean flag which denotes whether this property has a null value. This is Sparkplug™ B's mechanism of explicitly denoting a property's value is actually null.
- **value**
 - The value of a property utilizes the 'oneof' mechanism of Google Protocol Buffers. The value supplied with a metric must be one of the following types. Note if the metrics is_null flag is set to true the value can be omitted altogether.
 - *uint32*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *uint64*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *float*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *double*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *bool*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *string*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *PropertySet*
 - Defined in section 3.1.4 of this document.
 - *PropertySetList*
 - Defined in section 3.1.6 of this document

15.1.6. PropertySetList

A Sparkplug™ B PropertySetList object is an array of PropertySet objects. It includes the following components.

- **propertyset**
 - This is an array of PropertySet objects

15.1.7. DataSet

A Sparkplug™ B DataSet object is used to encode matrices of data. It includes the following components.

- **num_of_columns**
 - This is an unsigned 64-bit integer representing the number of columns in this DataSet.
- **columns**
 - This is an array of strings representing the column headers of this DataSet. It must have the same number of elements that the types array contains.
- **types**

- This is an array of unsigned 32 bit integers representing the datatypes of the columns. It must have the same number of elements that the columns array contains. Datatypes are not explicitly defined in the Sparkplug™ B Protobuf definition. Instead they are defined in section 4 of this document.
- **rows**
 - This is an array of DataSet.Row objects. It contains the data that makes up the data rows of this DataSet.

15.1.8. DataSet.Row

A Sparkplug™ B DataSet.Row object represents a row of data in a DataSet. It includes the following components.

- **elements**
 - This is an array of DataSet.DataSetValue objects. It represents the data contained within a row of a DataSet.

15.1.9. DataSet.DataSetValue

- **value**
 - The value of a DataSet.DataSetValue utilizes the ‘oneof’ mechanism of Google Protocol Buffers. The value supplied with a DataSet.DataSetValue must be one of the following types.
 - *uint32*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *uint64*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *float*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *double*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *bool*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *string*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>

15.1.10. Template

A Sparkplug™ B Template is used for encoding complex datatypes in a payload. It is a type of metric and can be used to create custom datatype definitions and instances. It includes the following components.

- **version**
 - This is a UTF-8 string representing the version of the Template.
- **metrics**
 - This is an array of metrics representing the members of the Template. These can be primitive datatypes or other complex datatypes as required for the Template.
- **parameters**
 - This is an array of Parameter objects representing parameters associated with the Template.
- **template_ref**
 - This is a UTF-8 string representing a reference to a Template name if this is a Template instance. If this is a Template definition this field must be null.

- **is_definition**

- This is a Boolean representing whether this is a Template definition or a Template instance. If true, this is a definition. If false, this is an instance.

15.1.11. Template.Parameter

A Sparkplug™ B Template.Parameter is a metadata field for a Template. This can be used to represent parameters that are common across a Template but the values are unique to the Template instances. It includes the following components.

- **name**

- This is a UTF-8 string representing the name of the Template parameter.

- **type**

- This is an unsigned 32-bit integer representing the datatype of the template parameter. Datatypes are not explicitly defined in the Sparkplug™ B Protobuf definition. Instead they are defined in section 4 of this document.

- **value**

- The value of a template parameter utilizes the ‘oneof’ mechanism of Google Protocol Buffers. The value supplied must be one of the following types. For a template definition, this is the default value of the parameter. For a template instance, this is the value unique to that instance.
 - *uint32*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *uint64*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *float*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *double*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *bool*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>
 - *string*
 - Defined here: <https://developers.google.com/protocol-buffers/docs/proto#scalar>

15.2. Sparkplug™ Bv1.0 Payload Datatypes

The Sparkplug™ B Google Protocol Buffers definition intentionally excludes datatypes in the definition. Different applications and systems have a wide variety of datatypes. As a result, Sparkplug™ B left them out and instead defines them in the client libraries. This allows consuming applications to be more dynamic in terms of adding new datatypes or even defining custom datatypes.

15.2.1. Metric Datatypes

- **Basic Types**

- *Unknown*
 - Sparkplug™ enum value: 0
- *Int8*
 - Signed 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 1
- *Int16*

- Signed 16-bit integer
- Google Protocol Buffer Type: uint32
- Sparkplug™ enum value: 2
- *Int32*
 - Signed 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 3
- *Int64*
 - Signed 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 4
- *UInt8*
 - Unsigned 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 5
- *UInt16*
 - Unsigned 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 6
- *UInt32*
 - Unsigned 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 7
- *UInt64*
 - Unsigned 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 8
- *Float*
 - 32-bit floating point number
 - Google Protocol Buffer Type: float
 - Sparkplug™ enum value: 9
- *Double*
 - 64-bit floating point number
 - Google Protocol Buffer Type: double
 - Sparkplug™ enum value: 10
- *Boolean*
 - Boolean value
 - Google Protocol Buffer Type: bool
 - Sparkplug™ enum value: 11
- *String*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 12

- *DateTime*
 - Date time value as uint64 value representing milliseconds since epoch (Jan 1, 1970)
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 13
- *Text*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 14
- **Custom Types**
 - *UUID*
 - UUID value as a UTF-8 string
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 15
 - *DataSet*
 - DataSet as defined in section 3.1.7
 - Google Protocol Buffer Type: none – defined in Sparkplug™
 - Sparkplug™ enum value: 16
 - *Bytes*
 - Array of bytes
 - Google Protocol Buffer Type: bytes
 - Sparkplug™ enum value: 17
 - *File*
 - Array of bytes representing a file
 - Google Protocol Buffer Type: bytes
 - Sparkplug™ enum value: 18
 - *Template*
 - Template as defined in section 3.1.10
 - Google Protocol Buffer Type: none – defined in Sparkplug™
 - Sparkplug™ enum value: 19

15.2.2. [Property Value Datatypes](#)

- **Basic Types**
 - *Unknown*
 - Sparkplug™ enum value: 0
 - *Int8*
 - Signed 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 1
 - *Int16*
 - Signed 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 2
 - *Int32*
 - Signed 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 3
 - *Int64*

- Signed 64-bit integer
- Google Protocol Buffer Type: uint64
- Sparkplug™ enum value: 4
- *UInt8*
 - Unsigned 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 5
- *UInt16*
 - Unsigned 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 6
- *UInt32*
 - Unsigned 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 7
- *UInt64*
 - Unsigned 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 8
- *Float*
 - 32-bit floating point number
 - Google Protocol Buffer Type: float
 - Sparkplug™ enum value: 9
- *Double*
 - 64-bit floating point number
 - Google Protocol Buffer Type: double
 - Sparkplug™ enum value: 10
- *Boolean*
 - Boolean value
 - Google Protocol Buffer Type: bool
 - Sparkplug™ enum value: 11
- *String*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 12
- *DateTime*
 - Date time value as uint64 value representing milliseconds since epoch (Jan 1, 1970)
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 13
- *Text*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 14

- **Custom Types**

- *PropertySet*
 - PropertySet as defined in section 3.1.4
 - Google Protocol Buffer Type: none – defined in Sparkplug™
 - Sparkplug™ enum value: 20
- *PropertySetList*
 - Template as defined in section 3.1.6
 - Google Protocol Buffer Type: none – defined in Sparkplug™
 - Sparkplug™ enum value: 21

15.2.3. [DataSetValue Data Types](#)

- **Basic Types**

- *Unknown*
 - Sparkplug™ enum value: 0
- *Int8*
 - Signed 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 1
- *Int16*
 - Signed 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 2
- *Int32*
 - Signed 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 3
- *Int64*
 - Signed 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 4
- *UInt8*
 - Unsigned 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 5
- *UInt16*
 - Unsigned 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 6
- *UInt32*
 - Unsigned 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 7
- *UInt64*
 - Unsigned 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 8

- *Float*
 - 32-bit floating point number
 - Google Protocol Buffer Type: float
 - Sparkplug™ enum value: 9
- *Double*
 - 64-bit floating point number
 - Google Protocol Buffer Type: double
 - Sparkplug™ enum value: 10
- *Boolean*
 - Boolean value
 - Google Protocol Buffer Type: bool
 - Sparkplug™ enum value: 11
- *String*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 12
- *DateTime*
 - Date time value as uint64 value representing milliseconds since epoch (Jan 1, 1970)
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 13
- *Text*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 14

15.2.4. [Template.Parameter Data Types](#)

- **Basic Types**

- *Unknown*
 - Sparkplug™ enum value: 0
- *Int8*
 - Signed 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 1
- *Int16*
 - Signed 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 2
- *Int32*
 - Signed 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 3
- *Int64*
 - Signed 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 4

- *UInt8*
 - Unsigned 8-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 5
- *UInt16*
 - Unsigned 16-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 6
- *UInt32*
 - Unsigned 32-bit integer
 - Google Protocol Buffer Type: uint32
 - Sparkplug™ enum value: 7
- *UInt64*
 - Unsigned 64-bit integer
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 8
- *Float*
 - 32-bit floating point number
 - Google Protocol Buffer Type: float
 - Sparkplug™ enum value: 9
- *Double*
 - 64-bit floating point number
 - Google Protocol Buffer Type: double
 - Sparkplug™ enum value: 10
- *Boolean*
 - Boolean value
 - Google Protocol Buffer Type: bool
 - Sparkplug™ enum value: 11
- *String*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 12
- *DateTime*
 - Date time value as uint64 value representing milliseconds since epoch (Jan 1, 1970)
 - Google Protocol Buffer Type: uint64
 - Sparkplug™ enum value: 13
- *Text*
 - String value (UTF-8)
 - Google Protocol Buffer Type: string
 - Sparkplug™ enum value: 14

16. Payloads by Message Type

16.1. NBIRTH

The NBIRTH message requires the following payload components.

- The NBIRTH must include the a seq number in the payload and it must have a value of 0.
- The NBIRTH must include a timestamp denoting the DateTime the message was sent from the EoN node.
- The NBIRTH must include every metric the EoN node will ever report on. At a minimum these metrics must include:
 - The metric name
 - The metric datatype
 - The current value
- If Template instances will be published by this EoN or any devices, all Template definitions must be published in the NBIRTH.
- A bdSeq number as a metric should be included in the payload. This should match the bdSeq number provided in the MQTT CONNECT packet's LW&T payload. This allows backend applications to correlate NBIRTHs to NDEATHs. The bdSeq number should start at zero and increment by one on every new MQTT CONNECT.

The NBIRTH message can also include optional 'Node Control' payload components. These are used by a backend application to control aspects of the EoN node. The following are examples of Node Control metrics.

- Metric name: 'Node Control/Reboot'
 - Used by backend application(s) to reboot an EoN node.
- Metric name: 'Node Control/Rebirth'
 - Used by backend application(s) to request a new NBIRTH and DBIRTH(s) from an EoN node.
- Metric name: 'Node Control/Next Server'
 - Used by backend application(s) to request an EoN node to walk to the next MQTT Server in its list in multi-MQTT Server environments.
- Metric name: 'Node Control/Scan rate'
 - Used by backed application(s) to modify a poll rate on an EoN node.

The NBIRTH message can also include optional 'Properties' of an EoN node. The following are examples of Property metrics.

- Metric name: 'Properties/Hardware Make'
 - Used to transmit the hardware manufacturer of the EoN node
- Metric name: 'Properties/Hardware Model'
 - Used to transmit the hardware model of the EoN node
- Metric name: 'Properties/OS'
 - Used to transmit the operating system of the EoN node
- Metric name: 'Properties/OS Version'
 - Used to transmit the OS version of the EoN node

16.2. DBIRTH

The DBIRTH message requires the following payload components.

- The DBIRTH must include the a seq number in the payload and it must have a value of one greater than the previous MQTT message from the EoN node contained unless the previous MQTT message contained a value of 255. In this case the seq number must be 0.
- The DBIRTH must include a timestamp denoting the DateTime the message was sent from the EoN node.
- The DBIRTH must include every metric the device will ever report on. At a minimum these metrics must include:
 - The metric name
 - The metric datatype
 - The current value

The DBIRTH message can also include optional 'Device Control' payload components. These are used by a backend application to control aspects of a device. The following are examples of Device Control metrics.

- Metric name: 'Device Control/Reboot'
 - Used by backend application(s) to reboot a device.
- Metric name: 'Device Control/Rebirth'
 - Used by backend application(s) to request a new DBIRTH from a device.
- Metric name: 'Device Control/Scan rate'
 - Used by backed application(s) to modify a poll rate on a device.

The DBIRTH message can also include optional 'Properties' of a device. The following are examples of Property metrics.

- Metric name: 'Properties/Hardware Make'
 - Used to transmit the hardware manufacturer of the device
- Metric name: 'Properties/Hardware Model'
 - Used to transmit the hardware model of the device
- Metric name: 'Properties/FW'
 - Used to transmit the firmware version of the device

16.3. NDATA

The NDATA message requires the following payload components.

- The NDATA must include the a seq number in the payload and it must have a value of one greater than the previous MQTT message from the EoN node contained unless the previous MQTT message contained a value of 255. In this case the seq number must be 0.
- The NDATA must include a timestamp denoting the DateTime the message was sent from the EoN node.
- The NDATA must include the EoN node's metrics that have changed since the last NBIRTH or NDATA message.

16.4. DDATA

The DDATA message requires the following payload components.

- The DDATA must include the a seq number in the payload and it must have a value of one greater than the previous MQTT message from the EoN node contained unless the previous MQTT message contained a value of 255. In this case the seq number must be 0.

- The DDATA must include a timestamp denoting the DateTime the message was sent from the EoN node.
- The DDATA must include the device's metrics that have changed since the last DBIRTH or DDATA message.

16.5. NCMD

The NCMD message requires the following payload components.

- The NCMD must include a timestamp denoting the DateTime the message was sent from the backend application's MQTT client.
- The NCMD must include the metrics that need to be written to on the EoN node.

16.6. DCMD

The DCMD message requires the following payload components.

- The DCMD must include a timestamp denoting the DateTime the message was sent from the backend application's MQTT client.
- The DCMD must include the metrics that need to be written to on the device.

16.7. DDEATH

The DDEATH message requires the following payload components.

- The DDEATH must include the a seq number in the payload and it must have a value of one greater than the previous MQTT message from the EoN node contained unless the previous MQTT message contained a value of 255. In this case the seq number must be 0.

16.8. NDEATH

The NDEATH message contains a very simple payload that only includes a single metric, the bdSeq number, so that the NDEATH event can be associated with the NBIRTH. Since this is typically published by the MQTT Server on behalf of the EoN node, information about the current state of the EoN node and its devices is not and cannot be known.

16.9. STATE

The STATE messages from the critical application must include a payload that is a UTF-8 string that is one of the following:

- OFFLINE
 - If the application is not connected
- ONLINE
 - If the application is connected

Sparkplug™ B payloads are not used for encoding in this payload. This allows critical/backend application(s) to work across Sparkplug™ payload types.

17. Payload Representation on Backend Applications

Sparkplug™ B payloads in conjunction with the Sparkplug™ topic namespace result in hierarchical data structures that can be represented in folder structures with metrics which are often called tags.

17.1. NBIRTH

The NBIRTH is responsible for informing the backend system of all of the information about the EoN node. This includes every metric it will publish data for in the future.

The following is a representation of a simple NBIRTH message on the topic:

spBv1.0/Sparkplug™ B Devices/NBIRTH/Raspberry Pi

In the topic above the following information is known based on the Sparkplug™ topic definition:

- The 'Group ID' of this EoN node is: Sparkplug™ B Devices
- The 'EoN node ID' of this EoN node is: Raspberry Pi
- * This is an NBIRTH message from the EoN node

Consider the following Sparkplug™ B payload in the NBIRTH message shown above:

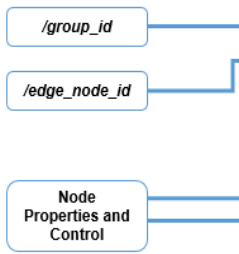
```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "bdSeq",
    "timestamp": 1486144502122,
    "dataType": "UInt64",
    "value": 0
  }, {
    "name": "Node Control/Reboot",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Node Control/Rebirth",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Node Control/Next Server",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Node Control/Scan Rate",
    "timestamp": 1486144502122,
    "dataType": "Int64",
    "value": 3000
  }, {
    "name": "Properties/Hardware Make",
    "timestamp": 1486144502122,
    "dataType": "String",
    "value": "Raspberry Pi"
  }, {
    "name": "Properties/Hardware Model",
    "timestamp": 1486144502122,
    "dataType": "String",
    "value": "Pi 3 Model B"
  }, {
    "name": "Properties/OS",
    "timestamp": 1486144502122,
    "dataType": "String",
    "value": "Raspbian"
  }, {
```

```

    "name": "Properties/OS Version",
    "timestamp": 1486144502122,
    "dataType": "String",
    "value": "Jessie with PIXEL/11.01.2017"
  }, {
    "name": "Supply Voltage (V)",
    "timestamp": 1486144502122,
    "dataType": "Float",
    "value": 12.1
  }],
  "seq": 0
}

```

This would result in a structure as follows on the backend system.



Metric	Value	Data Type
Sparkplug B Devices		
Raspberry Pi		
Node Control		
Reboot	FALSE	Boolean
Rebirth	FALSE	Boolean
Next Server	FALSE	Boolean
Scan Rate	3000	Int64
Properties		
Hardware Make	Raspberry Pi	String
Hardware Model	Pi 3 Model B	String
OS Version	Raspbian	String
OS Version	Jessie with PIXEL/11.01.2017	String
Supply Voltage (V)	12.1	Float

Figure 9 – Sparkplug™ B Metric Structure 1

17.2. DBIRTH

The DBIRTH is responsible for informing the backend system of all of the information about the device. This includes every metric it will publish data for in the future.

The following is a representation of a simple DBIRTH message on the topic:

spBv1.0/Sparkplug™ B Devices/DBIRTH/Raspberry Pi/Pibrella

In the topic above the following information is known based on the Sparkplug™ topic definition:

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this device is: Raspberry Pi
- The 'Device ID' is: Pibrella
- This is an DBIRTH message from the device

Consider the following Sparkplug™ B payload in the DBIRTH message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "Inputs/A",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Inputs/B",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Inputs/C",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Inputs/D",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Inputs/Button",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/E",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/F",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/G",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/H",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/LEDs/Green",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/LEDs/Red",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/LEDs/Yellow",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }, {
    "name": "Outputs/Buzzer",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": false
  }
]}
```

```

    }, {
      "name": "Properties/Hardware Make",
      "timestamp": 1486144502122,
      "dataType": "String",
      "value": "Pibrella"
    }
  ],
  "seq": 0
}

```

This would result in a structure as follows on the backend system.

Metric	Value	Data Type
Sparkplug B Devices		
Raspberry Pi		
Pibrella		
Inputs		
A	FALSE	Boolean
B	FALSE	Boolean
C	FALSE	Boolean
D	FALSE	Boolean
Outputs		
LEDs		
Green	FALSE	Boolean
Red	FALSE	Boolean
Yellow	FALSE	Boolean
E	FALSE	Boolean
F	FALSE	Boolean
G	FALSE	Boolean
H	FALSE	Boolean
Buzzer	FALSE	Boolean
Properties		
Hardware Make	Pibrella	String

Figure 10 – Sparkplug™ B Metric Structure 2

17.3. NDATA

NDATA messages are used to update the values of any EoN node metrics that were originally published in the NBIRTH message. Any time an input changes on the EoN node, a NDATA message should be generated and published to the MQTT Server. If multiple metrics on the EoN node change, they can all be included in a single NDATA message.

The following is a representation of a simple NDATA message on the topic:

spBv1.0/Sparkplug™ B Devices/NDATA/Raspberry Pi

In the topic above the following information is known based on the Sparkplug™ topic definition:

- The 'Group ID' of this EoN node is: Sparkplug™ B Devices
- The 'EoN node ID' of this EoN node is: Raspberry Pi
- * This is an NBIRTH message from the EoN node

Consider the following Sparkplug™ B payload in the NDATA message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "Supply Voltage (V)",
    "timestamp": 1486144502122,
    "dataType": "Float",
    "value": 12.3
  }],
  "seq": 2
}
```

This would result in the backend application updating the value of the Supply Voltage metric.

17.4. DDATA

DDATA messages are used to update the values of any device metrics that were originally published in the DBIRTH message. Any time an input changes on the device, a DDATA message should be generated and published to the MQTT Server. If multiple metrics on the device change, they can all be included in a single DDATA message.

The following is a representation of a simple DDATA message on the topic:

spBv1.0/Sparkplug™ B Devices/DDATA/Raspberry Pi/Pibrella

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this device is: Raspberry Pi
- The 'Device ID' is: Pibrella
- This is an DDATA message from the device

Consider the following Sparkplug™ B payload in the DDATA message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "Inputs/A",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": true
  }, {
    "name": "Inputs/C",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": true
  }],
  "seq": 0
}
```

This would result in the backend application updating the value of the 'Inputs/A' metric and 'Inputs/C' metric.

17.5. NCMD

NCMD messages are used by backend applications to write to EoN node outputs and send Node Control commands to EoN nodes. Multiple metrics can be supplied in a single NCMD message.

The following is a representation of a simple NCMD message on the topic:

spBv1.0/Sparkplug™ B Devices/NCMD/Raspberry Pi

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this EoN node is: Raspberry Pi
- This is an NCMD message to an EoN node

Consider the following Sparkplug™ B payload in the NCMD message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "Node Control/Rebirth",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": true
  }]
}
```

This NCMD payload tells the EoN node to republish its NBIRTH and DBIRTH(s) messages. This can be requested if a backend application gets an out of order seq number or if a metric arrives in an NDATA or DDATA message that was not provided in the original NBIRTH or DBIRTH messages.

17.6. DCMD

DCMD messages are used by backend applications to write to device outputs and send Device Control commands to devices. Multiple metrics can be supplied in a single DCMD message.

The following is a representation of a simple DCMD message on the topic:

spBv1.0/Sparkplug™ B Devices/DCMD/Raspberry Pi/Pibrella

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this device is: Raspberry Pi
- The 'Device ID' is: Pibrella
- This is an DCMD message from the device

Consider the following Sparkplug™ B payload in the DCMD message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "Outputs/LEDs/Green",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": true
  }, {
    "name": "Outputs/LEDs/Yellow",
    "timestamp": 1486144502122,
    "dataType": "Boolean",
    "value": true
  }]
}
```

The DCMD payload tells the EoN node to write true to the attached device's green and yellow LEDs. As a result, the LEDs should turn on and result in a DDATA message back to the MQTT Server after the LEDs are successfully turned on.

17.7. NDEATH

The NDEATH messages are registered with the MQTT Server in the MQTT CONNECT packet as the LW&T. This is used by backend applications to know when an EoN node has lost its MQTT connection with the MQTT Server.

The following is a representation of a NDEATH message on the topic:

spBv1.0/Sparkplug™ B Devices/NDEATH/Raspberry Pi

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this EoN node is: Raspberry Pi
- This is an NDEATH message from the MQTT Server on behalf of an EoN node

Consider the following Sparkplug™ B payload in the NDEATH message shown above:

```
{
  "timestamp": 1486144502122,
  "metrics": [{
    "name": "bdSeq",
    "timestamp": 1486144502122,
    "dataType": "UInt64",
    "value": 0
  }]
}
```

The payload metric of bdSeq allows a backend application to reconcile this NDEATH with the NBIRTH that occurred previously.

17.8. DDEATH

The DDEATH messages are published by an EoN node on behalf of an attached device. If the EoN node determines that a device is no longer accessible (i.e. it has turned off, stopped responding, etc.) the EoN node should publish a DDEATH to denote that device connectivity has been lost.

The following is a representation of a simple DDEATH message on the topic:

spBv1.0/Sparkplug™ B Devices/DDEATH/Raspberry Pi/Pibrella

- The 'Group ID' of this device is: Sparkplug™ B Devices
- The host 'EoN node ID' of this device is: Raspberry Pi
- The 'Device ID' is: Pibrella
- This is an DDEATH message from the EoN node on behalf of the device

Consider the following Sparkplug™ B payload in the DDEATH message shown above:

```
{
  "timestamp": 1486144502122,
  "seq": 123
}
```

A sequence number must be included with the DDEATH messages so the backend application can ensure order of messages and maintain the state of the data.

17.9. STATE

As noted previously, the STATE messages published by backend application(s) do not use Sparkplug™ B payloads.