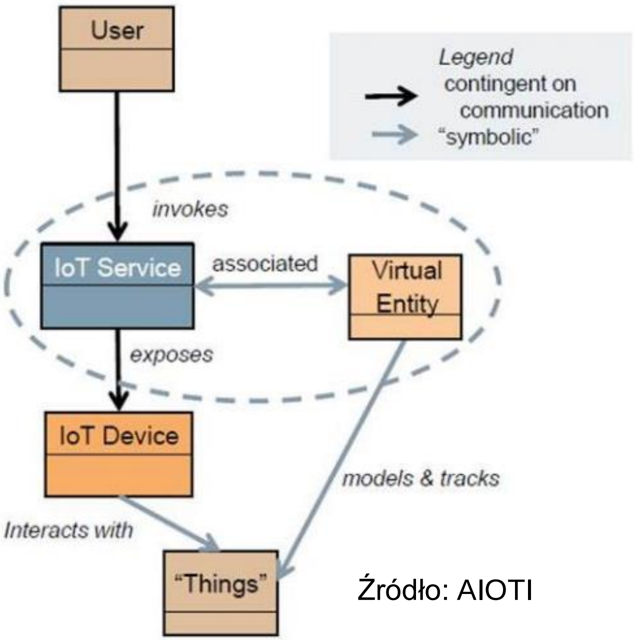
PBL5

MQTT Sparkplug

Jarosław Domaszewicz
Instytut Telekomunikacji Politechniki Warszawskiej

1



Źródło: AIOTI

Sparkplug™ Specification

Sparkplug™

MQTT Topic &
Payload Definition

Version 2.2

ECLIPSE
FOUNDATION

SPARKPLUG OBJECTIVES

SCADA/IIOT

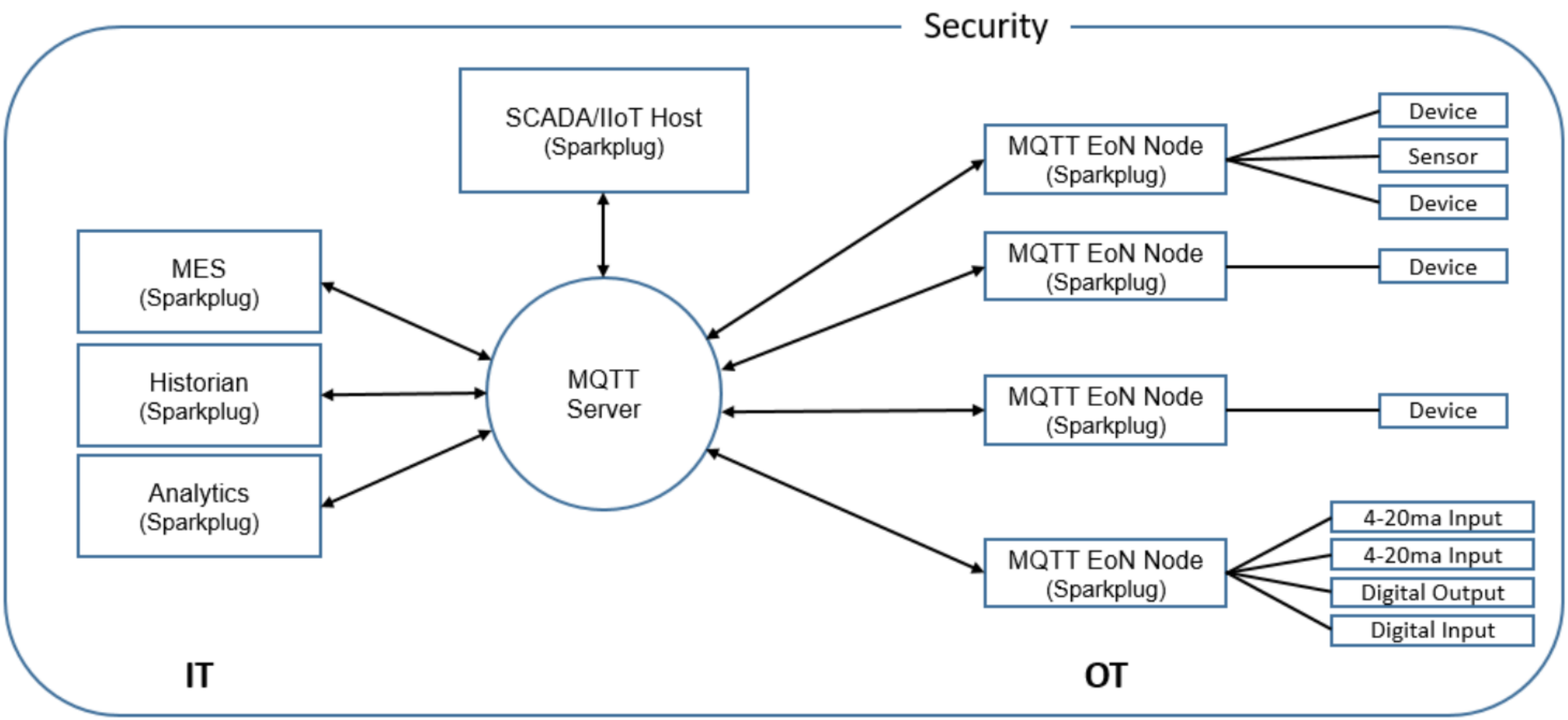
- Define an MQTT topic namespace
  - MQTT does not define any
  - the intent of the Sparkplug™ specification is to identify and document a topic namespace that is well thought out and optimized for the SCADA/IIoT solution sector

- Define MQTT State Management
  - the way state information is implemented and managed within the MQTT infrastructure is not defined.
  - 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

- Define the MQTT Payload
  - MQTT specification does not dictate any particular payload data encoding
  - Sparkplug A and Sparkplug B

Overall MQTT Sparkplug architecture

INFRASTRUCTURE COMPONENTS



INFRASTRUCTURE COMPONENTS (1/5)

- MQTT server (broker)
  - MQTT V3.1.1 compliant
  - sized to properly manage all MQTT message traffic

Infrastructure components (2/5)

- Edge of Network Node (EoN)
  - MQTT 3.1.1 and Sparkplug compliant (an MQTT client speaking Sparkplug)
- EoN as a gateway
  - 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)
- EoN as a device
  - any device, sensor, or hardware that natively implements MQTT/Sparkplug

Infrastructure components (3/5)

- Device/sensor
  - represents any physical or logical ”legacy” device providing any data, process variables or metric
  - does not implement MQTT/Sparkplug
  - connected to an EoN node that plays the role of a gateway

INFRASTRUCTURE COMPONENTS (4/5)

- SCADA/IIoT/host Node
  - any MQTT and Sparplug compliant cllient application
  - only one primary SCADA/IIoT/host application
  - any numer of non-primary SCADA/IIoT application

INFRASTRUCTURE COMPONENTS (5/5)

- primary SCADA/IIoT/host application (= Primary Application)
  - responsible for the monitoring and control of a given group of MQTT EoN nodes
  - control: only the Primary Application(s) should have the permission to issue commands
  - all EoN nodes needs to make sure they are talking to the same MQTT server as the Primary Application does
  - maintains a Primary Application metric structure
  - metrics are fully determined by NBIRTH messages (EoN and deviice produced)

- non-primary SCADA/IIoT application
  - works in pure monitoring mode, or in the role of a hot standby should the Primary MQTT SCADA/IIoT Host go offline
  - does not issue Birth/Death Certificates

Sparkplug topics

SPARKPLUG TOPICS

- In MQTT, almost anything goes as to topics.
  - recall: levels, level separators ("/"), wildcards ("+", "#")
  - but publishing and subscribing clients need to agree on the topics used

- Spakplug topic structure

namespace/group_id/message_type/edge_node_id/[device_id]

spBv1.0
(Sparkplug B)

uniquely identifies a logical grouping of EoN nodes

E.g., EoN nodes on a physical pipeline segment may all have the same group_id.

NBIRTH
NDEATH
DBIRTH
DDEATH
NDATA
DDATA
STATE
NCMD
DCMD

uniquely identifies an EON node within its group

uniquely identifies an EON node within its group

the topic identifies the EoN publisher

these messsages are subscribed to by the identified EoN

group_id/edge_node_id must be unique in the entire infrastructure.

Session state management:

NBIRTH,NDEATH

DBIRTH,DDEATH

STATE

Session state management

- The primary Application should be able to monitor the current state of any MQTT device in the infrastructure.
  - is the device online or offline
  - actually, it is about the state of the connection of the device
- Why is this monitoring needed?
  - recall Report by Exception (RBE)
  - for RBE to work properly in real-time SCADA, the "state" of the end device needs to be always known
  - SCADA/IIoT host could only rely on RBE data arriving reliably if it could be assured of the state of the MQTT session.
- What's included in MQTT?
  - Last Will and Testament (LWT)
- Sparkplug adds birth messages.

SESSION STATE MANEGEMENT IN MQTT: LWT

Last Will and Testament (LWT)

Control Packet	Payload	
CONNECT	Required	ClientID, Will Topic, Will Message, User Name, Password
CONNACK	None	
PUBLISH	Optional	Application Message (note: MQTT is agnostic as to the format)
PUBACK	None	
PUBREC	None	
PUBREL	None	
PUBCOMP	None	
SUBSCRIBE	Required	(Topic_Filter_1,QoS_1), (Topic_Filter_2,QoS_2),…// requested QoS
SUBACK	Required	QoS_1, QoS_2, … // granted QoS
UNSUBSCRIBE	Required	Topic_Filter_1, Topic_Filter_2, …
UNSUBACK	None	
PINGREQ	None	
PINGRESP	None	
DISCONNECT	None	

underlined payload elements are mandatory

Source: *MQTT Version 3.1.1*
OASIS Standard , October 2014

Session state management in MQTT: LWT

Source: *MQTT Version 3.1.1* OASIS Standard , October 2014

Figure 3.4 - Connect Flag bits

Bit	7	6	5	4	3	2	1	0
	User Name Flag	Password Flag	Will Retain	Will QoS		Will Flag	Clean Session	Reserved
byte 8	X	X	X	X	X	X	X	0

- recall CONNECT
  - variable header: Protocol Name and Level, Keep Alive time, Connect Flags
  - payload: ClientID, Will Topic, Will Message, User Name, Password
- graceful (clean) disconnection
  - via DISCONNECT
  - upon a graceful disconnection, the broker discards the stored LWT message
    - the LWT message not sent
- ungraceful disconnection
  - the broker has not heard from the client for 1.5*keepalive_time
  - the client closes the network connection without DISCONNECT
  - the broker closes the network connection because of a protocol error
  - the broker detects a network error
  - upon an ungraceful disconnection, the broker sends the LWT Message to all clients that subscribed to the Will Topic

EoN Death Certificate (NDEATH)

- EoN Death Certificate: Will Topic + Will Message
  - a part of MQTT (LWT)
  - included in CONNECT
  - delivered by the broker upon ungraceful disconnection

- Topic: `namespace/group_id/NDEATH/edge_node_id`

- 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
  - note the time stamp when the NDEATH message was received

- Payload
  - a single metric: Birth/Death sequence numer `bdSeq` (see below)
  - not much diagnostic information as to why EoN ”died” (why???)

EoN Birth Certificate (NBIRTH)

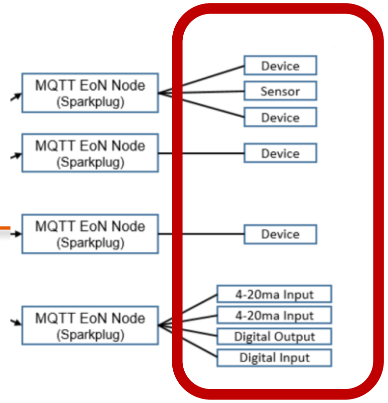
- EoN Birth Certificate
  - added by Sparkplug (not MQTT)
  - the first MQTT message that an EoN node MUST publish upon the successful establishment of an MQTT Session is an EoN BIRTH Certificate

- Topic: `namespace/group_id/NBIRTH/edge_node_id`

- Upon reception, a client should
  - set the ONLINE state of this EoN node to TRUE along with the associated ONLINE Date Time parameter

- Payload
  - metrics that will be published by this EoN node
  - commands that will be accepted by this EoN node (formally, these are also metrics)
  - properties of this EoN node (formally, these are also metrics)

DEVICE BIRTH CERTIFICATE (DBIRTH)



- Device Birth Certificate
  - Added by Sparkplug (not MQTT)
- Topic: `namespace/group_id/DBIRTH/edge_node_id/device_id`
- Sent by EoN node on behalf of "its" device.
- Upon reception, a client should
  - set the ONLINE state of this device to TRUE along with the associated ONLINE date time this message was received
- Payload:
  - contains everything required to build out a data structure for all metrics for this device.

DEVICE DEATH CERTIFICATE (DDEATH)

- Device Death Certificate
  - added by Sparkplug (not MQTT)
  - note: the EoN Death Certificate uses an MQTT mechanism (LWT)
- Topic: `namespace/group_id/DDEATH/edge_node_id/device_id`
- Sent by EoN on behalf of its device.
- Upon reception, a client should
  - set the data quality of all metrics to "STALE" and should note the time stamp when the DDEATH message was received

SCADA/IIoT Host birth and death (STATE)

- Birth Certificate:
  - added by Sparkplug (not MQTT)
  - the first MQTT message that a SCADA/IIoT node must publish upon the successful establishment of an MQTT Session is a SCADA/IIoT host BIRTH Certificate
- Death Certificate:
  - a part of MQTT (LWT)
  - included in CONNECT
  - delivered by the broker upon ungraceful disconnection

- Topic: `STATE/scada_host_id` (both birth and death)

- In both certificates, the retained flag is set.

- Birth payload: ”`ONLINE`”
- Death payload: ”`OFFLINE`”

Reporting metrics:

NDATA

DDATA

EoN Node Data (NDATA)

- After NBIRTH, an EoN node reports
  - Report by Exception (RBE)
  - note: Continuous Session Awareness makes it possible to use RBE (no need to worry, even if nothing changes for a long time)

- Topic: `namespace/group_id/NDATA/edge_node_id`

- Payload
  - RBE metrics

DEVICE DATA

- Topic: `namespace/group_id/DDATA/edge_node_id/device_id`
- Payload:
  - the payload of DDATA messages can contain one or more metric values

Issuing commands:

NCMD

DCMD

EoN Node Command (NCMD)

- Used to send commands to any connected EoN node.
- Topic: `namespace/group_id/NCMD/edge_node_id`
- Payload: updated metric.

Device Command (DCMD)

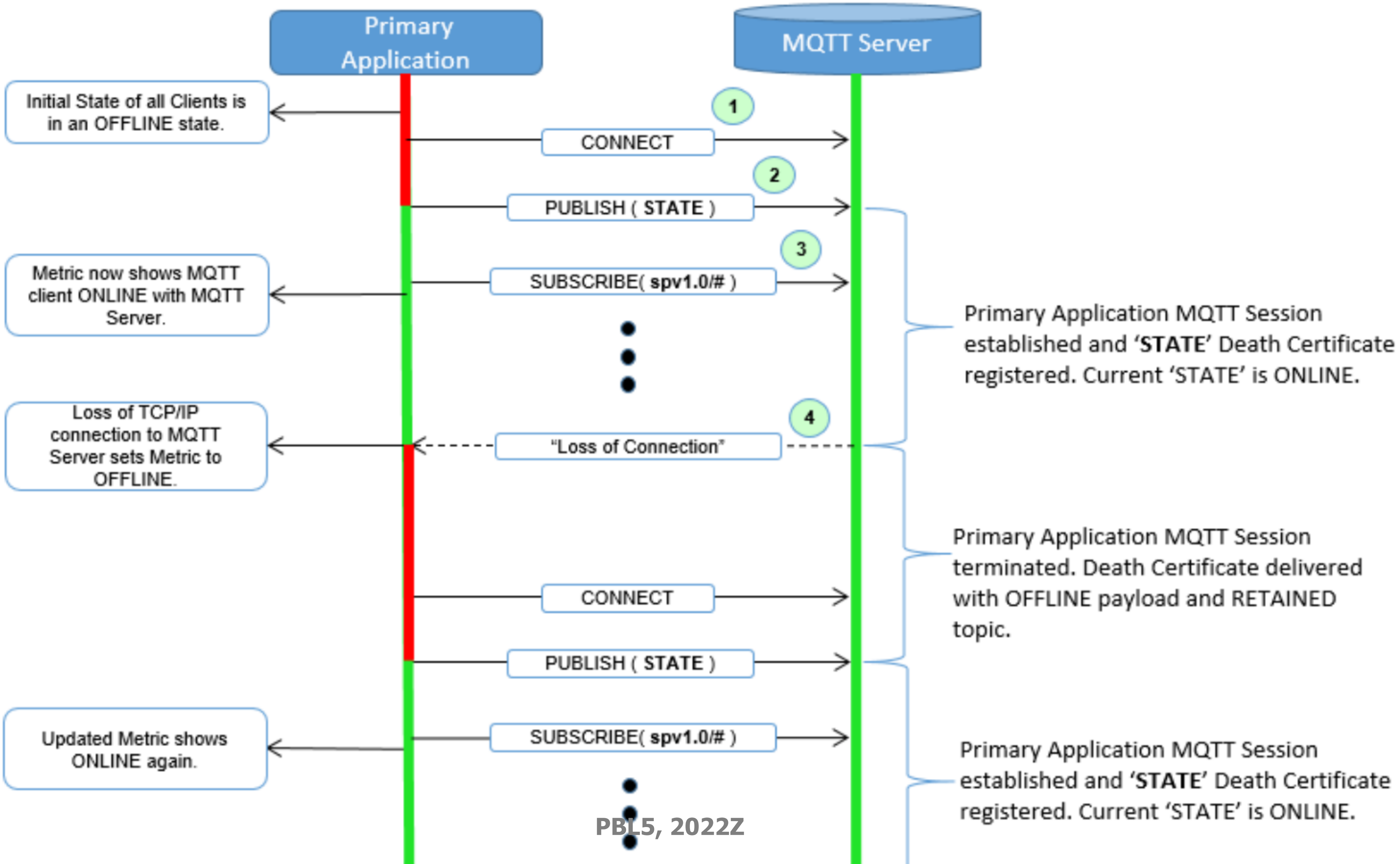
- Used to send commands to any connected device.
- Topic: `namespace/group_id/DCMD/edge_node_id/device_id`
- Payload: updated metric.

Typical message sequences

Źródło http://www.steves-internet-guide.com/sparkplug-payloads-and-messages/

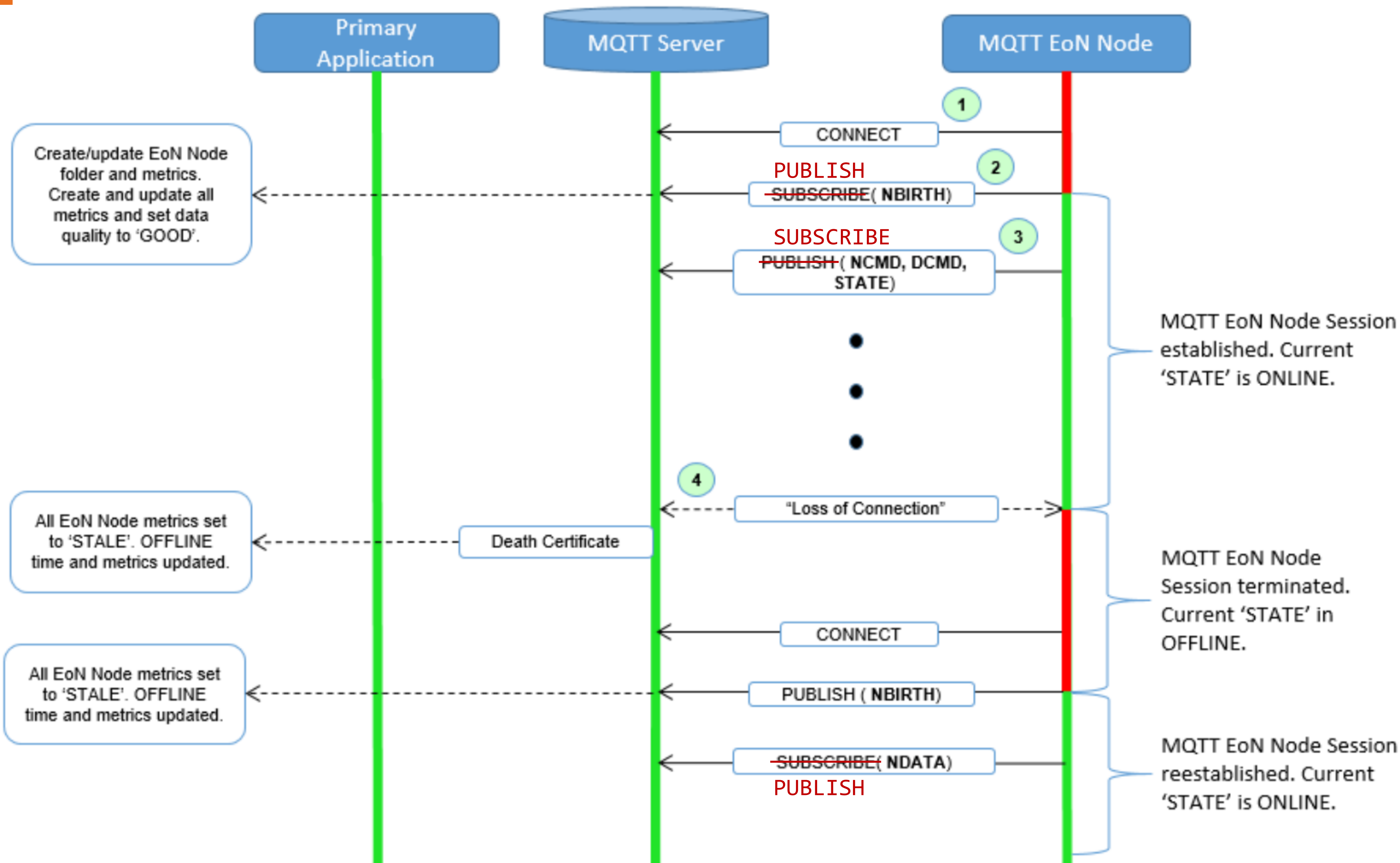
- A typical sequence for the Primary Application:
  - connect to the broker (CONNECT with LWT: STATE with ”OFFLINE”)
  - publish STATE (with ”ONLINE”), topic `STATE/<scada_host_id>`
  - subscribe to
    - `spBv1.0/#`
- A typical sequence for an EoN node:
  - connect to the MQTT broker (CONNECT with LWT: NDEATH)
  - publish a birth message NBIRTH, topic `spBv1.0/group/NBIRTH/<edge_node_id>`
  - subscribe to
    - `STATE/<scada_host_id>`
    - `spBv1.0/group/NCMD/<edge_node_id>`
    - `spBv1.0/group/DCMD/<edge_node_id>/#`
  - if you represent a device, publish a birth message DBIRTH, topic `spBv1.0/group/DBIRTH/edge_node_id>/<device_id>`

EXAMPLE: PRIMARY APP. SESSION ESTABLISHMENT

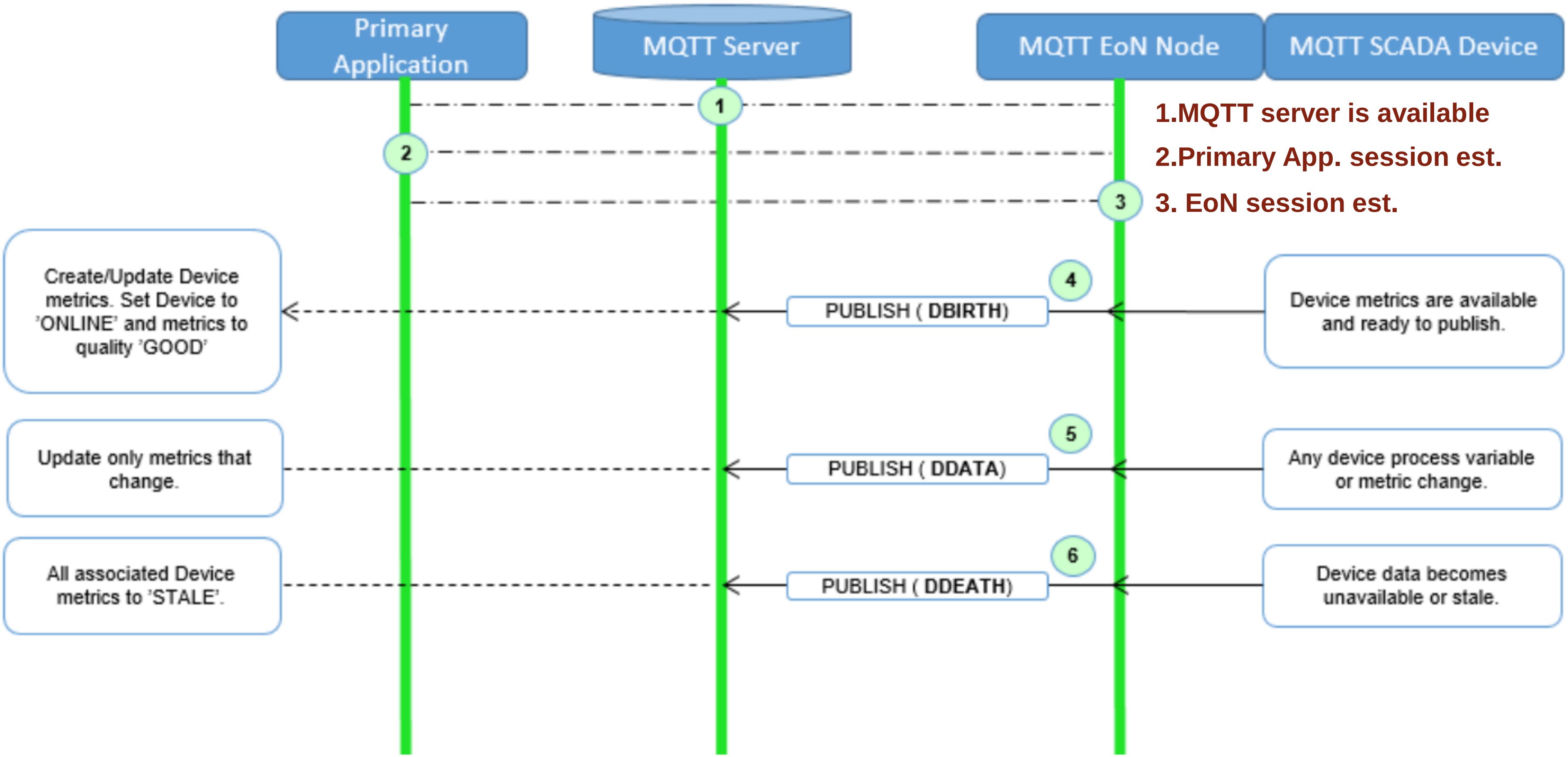


https://github.com/eclipse/sparkplug/issues/78

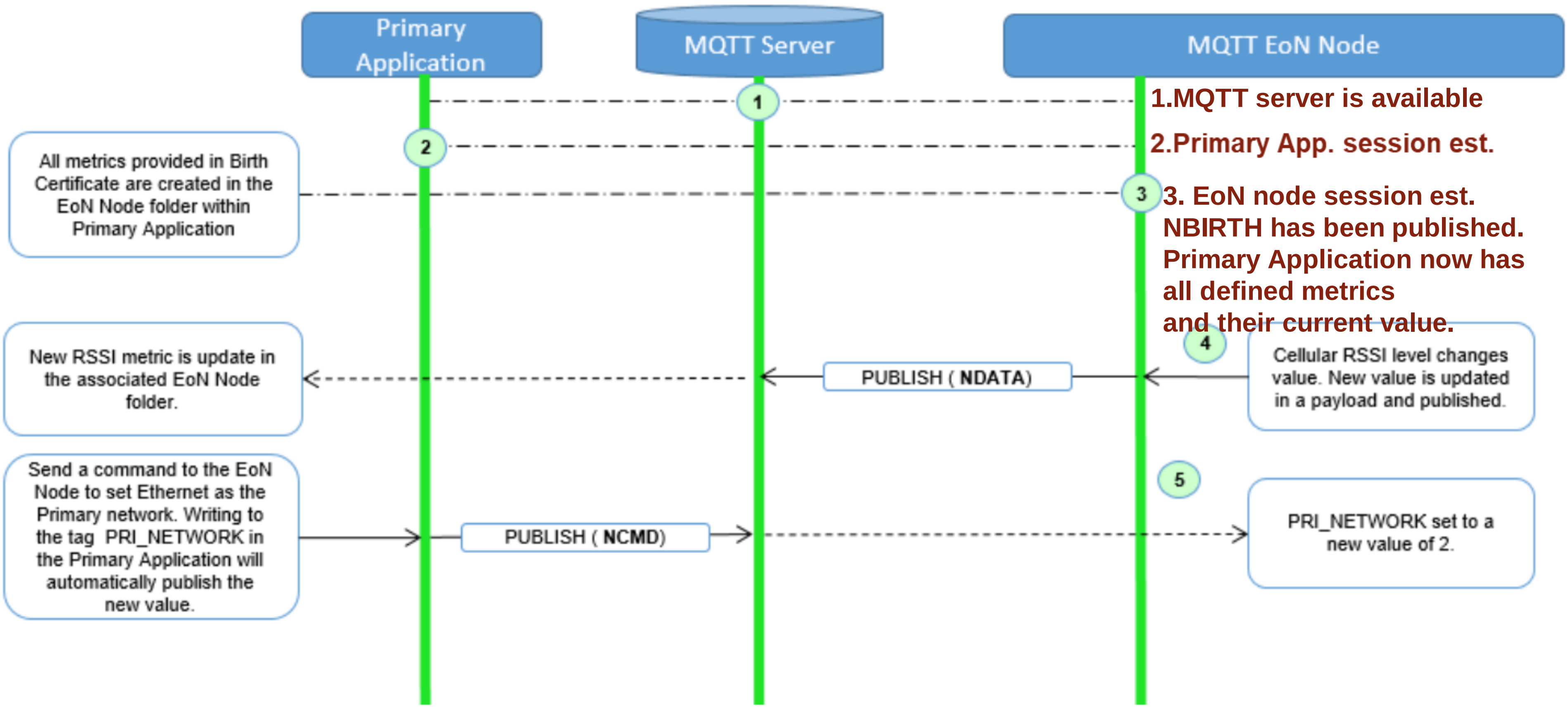
EXAMPLE: EON NODE SESSION ESTABLISHMENT



EXAMPLE: DEVICE SESSION ESTABLISHMENT



EXAMPLE: NDATA AND NCMD MESSAGES



Sparkplug payload

SPARKPLUG_B.PROTO

- Look at `sparkplug_b.proto`
  - Sparkplug™ B Google Protocol Buffer Schema
- Quite a few nested message types there.
- Let’s skip complex (and probably not so ferquently occuring) payload structures.

PAYLOAD

- Essentially, a payload consists of metrics (sensor readings).

sparkplug_b.proto

```
message Payload {
   …

   optional uint64   timestamp    = 1; // message sending (publishing) time,
                                       // ms since epoch
   repeated Metric   metrics      = 2;
   optional uint64   seq          = 3; // sequence number of this message
   optional string   uuid         = 4; // the id may, e.g., help decode bytes(below)
   optional bytes    body         = 5; // custom binary data
   extensions  6 to max;               // for third party extensions
}
```

METRIC

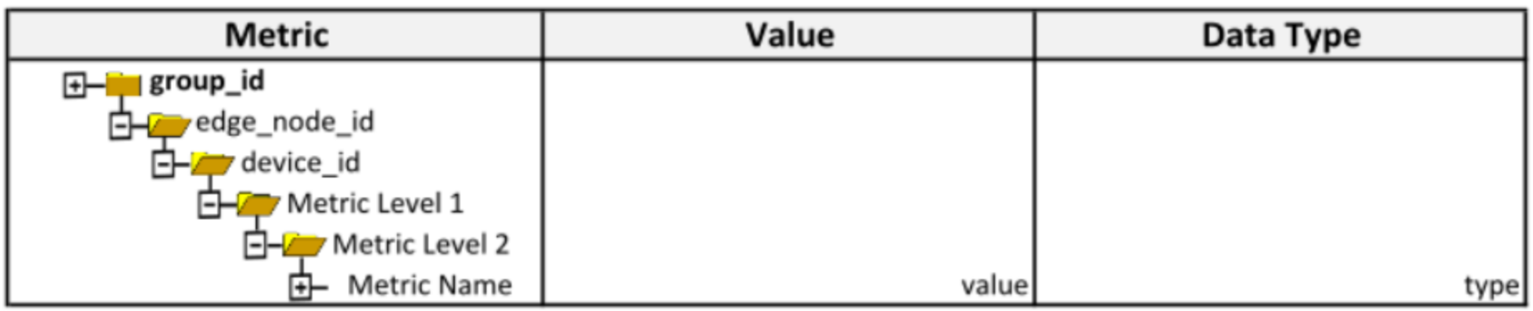
- `Metric` is a nested message type (inside `Payload`).
- A metric is essentially a key/value/datatype tuple along with metadata

sparkplug_b.proto

```
message Metric {
  optional string    name          = 1;  // should only be included on birth
  optional uint64    alias     key = 2;  // tied on birth, used in DATA messages
  optional uint64    timestamp     = 3;  // the capturing time of the mertic value
  optional uint32    datatype      = 4;  // DataType of the value
  optional bool      is_historical = 5;  // if historical, not a current value
  optional bool      is_transient  = 6;  // not to be stored by a historian
  optional bool      is_null       = 7;  // there is no value
  optional MetaData metadata       = 8;  // metadata
  optional PropertySet properties = 9;
  oneof value {                 value
    uint32    int_value            = 10; // one of protobufs data types
    uint64    long_value           = 11;
    float     float_value          = 12;
    double    double_value         = 13;
    bool      boolean_value        = 14;
    string    string_value         = 15;
    bytes     bytes_value          = 16;
    …
  }
}
```

NAMING METRICS

- Name
  - friendly (human-readable)
  - a slash delimited UTF-8 string
  - the hierarchical name represents folders in a hierarchical metric data structure in metric consuming applications
  - example:
    - `Metric Level 1/Metric Level 2/Metric Name`



- Alias
  - optional
  - unique within the metric's EoN node
  - if defined in the NBIRTH or DBIRTH, it may be used in subsequent messages to reduce message size

Example: simple payload

- Presented as JSON for readability. ← **recall: actual protobufs encoding is binary**

```
{
    "timestamp": <timestamp>,
    "metrics": [{ // just one metric
      „name": <metric_name>,
      "alias": <alias>,
      "timestamp": <timestamp>,
      "dataType": <datatype>,
      "value": <value>
     }],
    "seq": <sequence_number>
}
```

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

PropertySet, PropertyValue

- An array of custom properties for each metric, such as engineering units or scaling limits.

sparkplug_b.proto

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

sparkplug_b.proto

```
message PropertyValue {
  optional uint32     type                    = 1; // the type of the value
  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;
  }
}
```

Payloads by message types

NBIRTH PAYLOAD

- NBIRTH payload must include …
  - a sequence numer, `seq` (=0,this is the first message in the sequence by this EoN node)
  - a timestamp of the time this message was sent by this EoN node
  - every metric the EoN node will ever report on, at least …
    - the metric name
    - the metric datatype
    - the current value
  - `bdSeq` (a metric)
    - this should match the bdSeq numberc provided in the MQTT CONNECT packet's LW&T payload.
- NBIRTH payload may include optional 'Node Control' metrics.
  - used by the Primary Application to control the EoN node
  - examples:
    - `Node Control/Reboot`
    - `Node Control/Rebirth`
    - `Node Control/Next Server` (go to the next MQTT Server in a multi-MQTT Server environment)
    - `Node Control/Scan rate`
- NBIRTH payload may include optional 'Properties' metrics.
  - used to provide properties of this EoN node
  - examples:
    - `Properties/Hardware Model`, `Properties/OS`

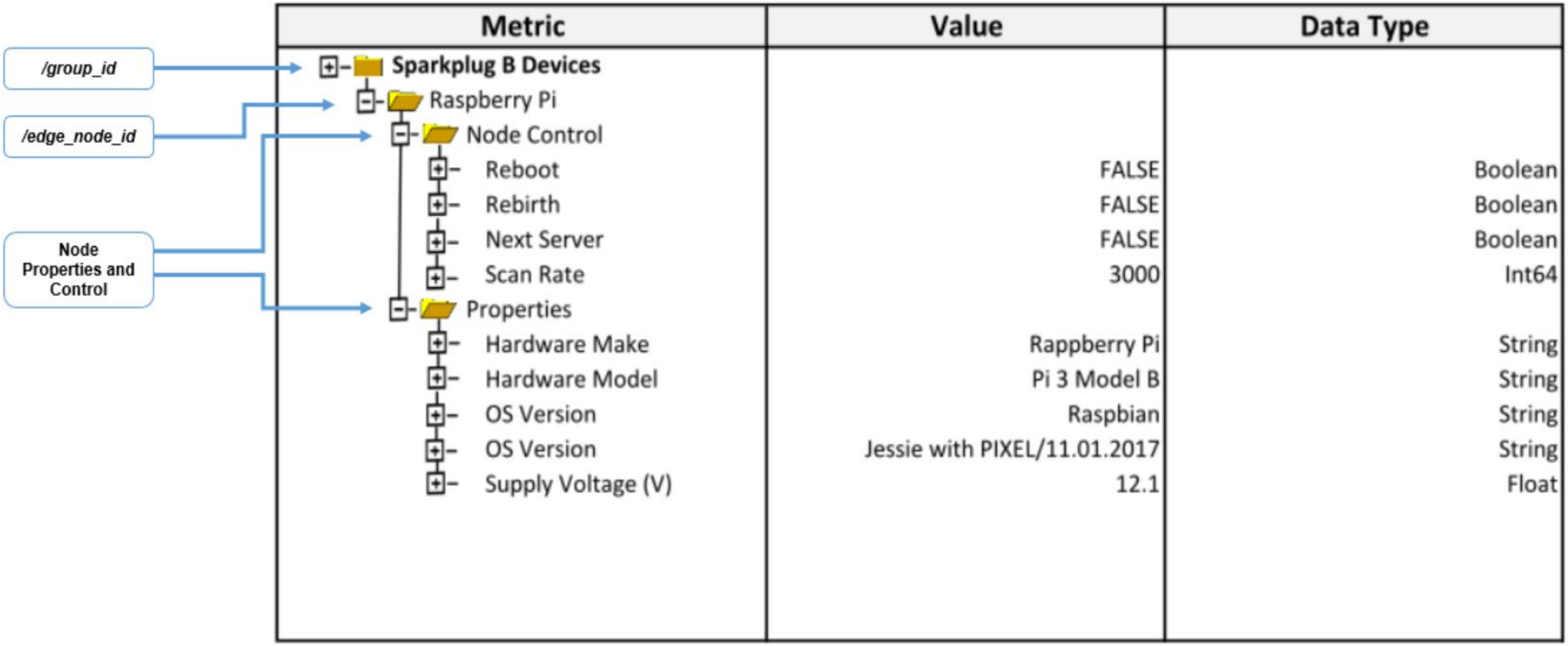
spBv1.0/Sparkplug™ B Devices/NBIRTH/Raspberry Pi

NBIRTH PAYLOAD: EXAMPLE (1/2)


```
{
  "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
}
```

NBIRTH PAYLOAD: EXAMPLE (2/2)

- A representation at the Primary Application:



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	Rappberry 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

NDEATH PAYLOAD

recall: NDEATH is an MQTT-level LWT message

- NDEATH payload must include …
  - a single metric, the `bdSeq` number

- This way the NDEATH event can be associated with the corresponding NBIRTH.

DBIRTH PAYLOAD

- DBIRTH payload must include …
  - a sequence number in the sequence by this EoN
  - a timestamp of the time this message was sent by this EoN
  - every metric the device will ever report on, at least …
    - the metric name
    - the metric datatype
    - the current value

recall: DBIRTH is not sent by a device; it is sent by the device's EoN node

- DBIRTH payload may include optional 'Device Control' metrics.
  - used by the Primary Application to control the device
  - examples:
    - `Device Control/Reboot`
    - `Device Control/Rebirth`
    - `Device Control/Scan rate`
- DBIRTH payload may include optional 'Properties' metrics.
  - use to provide properties of the device
  - examples:
    - `Properties/Hardware Make`
    - `Properties/Hardware Model`

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

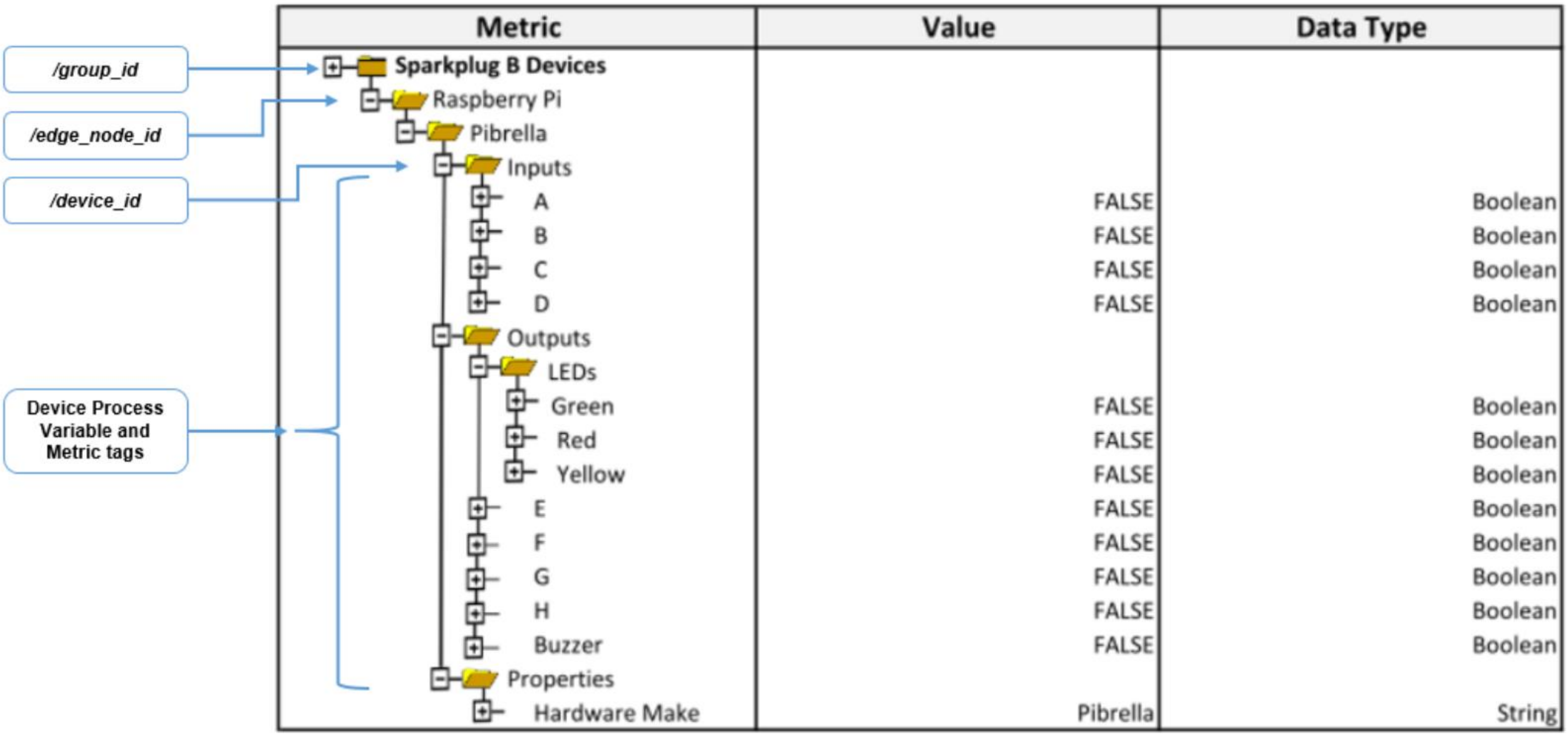
DBIRTH PAYLOAD: EXAMPLE (1/2)


```json
{
  "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
}
```

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

DBIRTH PAYLOAD: EXAMPLE (2/2)

- A representation at the Primary Application:



DDEATH PAYLOAD

- DDEATH payload must include …
  - a sequence number, `seq`, in the sequence by this EoN

STATE PAYLOAD

- STATE payload must include …
  - a UTF-8 string ”OFFLINE” or …
  - a UTF-8 string ”ONLINE” or …

recall: STATE with OFFLINE is an MQTT-level LWT message

- Sparkplug B is not used to allow the Primary Application to work with different clients (possibly not talking Sparkplug B).

NDATA PAYLOAD

- NDATA payload must include …
  - a sequence number, seq, in the sequence by this EoN
  - a timestamp of the time this message was sent by this EoN node
  - the EoN node's metrics that have changed since the last NBIRTH or NDATA message

Report by Exception (RBE)

DDATA PAYLOAD

- DDATA payload must include …
  - a sequence number, seq, in the sequence by this EoN
  - a timestamp of the time this message was sent by thid EoN node
  - the device’s metrics that have changed since the last DBIRTH or DDATA message

Report by Exception (RBE)

NCMD PAYLOAD

- NCMD payload must include …
  - a timestamp denoting the DateTime the message was sent by the Primary Application
  - the metrics that need to be written to on the EoN node

DCMD PAYLOAD

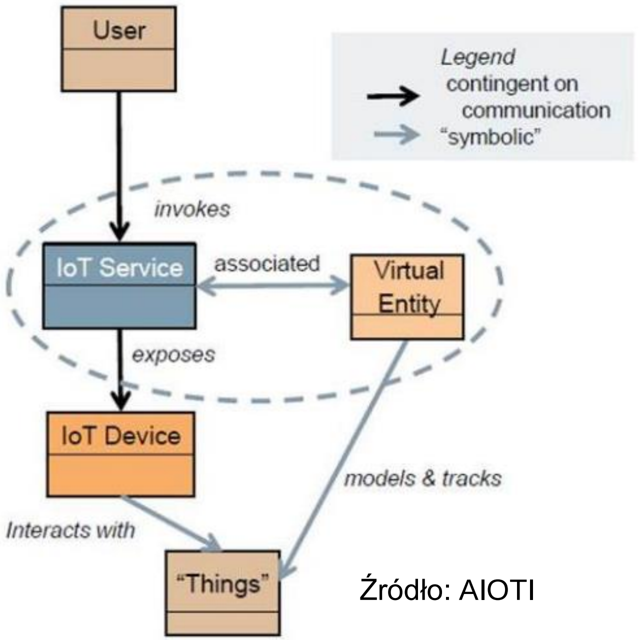
- NCMD payload must include …
  - a timestamp denoting the DateTime the message was sent by the Primary Application
  - the metrics that need to be written to on the device

PBL5

Google Protocol Buffers (Protobuf)

Jarosław Domaszewicz

Instytut Telekomunikacji Politechniki Warszawskiej



Źródło: AIOTI

Protobuf language

Źródło https://developers.google.com/protocol-buffers/

Message types

- Example: the definition of a simple *message type*

```
syntax = "proto2";

message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

field = (rule, type, name, number (tag))

  - note: a field's number is NOT the field's value

- Message types (possibly multiple ones) are included in a `*.proto` file.

Field types

- Scalar types

- Composite types
  - enumerations
  - message types
    - a message type can be the type of a field

SCALAR TYPES (1/)

- `double`
- `float`
- `int32`, `int64`
  - variable-length encoding
  - inefficient for encoding negative numbers
- `uint32`, `uint64`
  - variable-length encoding.
- `sint32`, `sint64`
  - variable-length encoding.
  - these more efficiently encode negative numbers than the int32/int64 types
- `fixed32`, `fixed64`
  - always four bytes.
  - more efficient than uint32/uint64 if values are often greater than 2↑28/2↑56
- `sfixed32`, `sfixed64`
  - always four/eight bytes.

Scalar types (1/)

- `bool`
  - boolean
- `string`
  - must always contain UTF-8 encoded text
- `bytes`
  - may contain any arbitrary sequence of bytes

COMPOSITE TYPES: ENUMERATIONS (ENUM)

- A composite data type.

```
enum Corpus {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}

message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  optional Corpus corpus = 4 [default = CORPUS_UNIVERSAL];
}
```

Composite types: message types

- You can use other message types as field types.
  - also note nested types (below)

```
message SearchResponse {
  repeated Result result = 1;
}

message Result {
  required string url = 1;
  optional string title = 2;
  repeated string snippets = 3;
}
```

Nested types (1/)

- One can define and use message types inside other message types.

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

NESTED TYPES (2/)

- You can reuse a nested message type outside its parent message type.
  - you refer to it as _Parent_._Type_

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}

message SomeOtherMessage {
  optional SearchResponse.Result result = 1;
}
```

Nested types (3/)

- You can nest messages as deeply as you like.

```
message Outer {       // Level 0
  message MiddleAA {  // Level 1
    message Inner {   // Level 2
      optional int64 ival = 1;
      optional bool  booly = 2;
    }
  }
  message MiddleBB {  // Level 1
    message Inner {   // Level 2
      optional string name = 1;
      optional bool   flag = 2;
    }
  }
}
```

the two nested types named `Inner` are entirely independent (they are defined within different messages)

Field numbers

- Field numbers are used to identify your fields in the message binary format.
- They should not be changed once your message type is in use.
- Encoding:
  - field numbers in the range 1 through 15 take one byte to encode, including the field number and the field's type
  - field numbers in the range 16 through 2047 take two bytes

EXTENSIONS AND EXTEND

- Extensions let you declare that a range of field numbers in a message are available for third-party extensions.

```
message Foo {
  // ...
  extensions 100 to 199;
}
```

these field numbers are reserved for extensions

```
message Foo {
  extensions 1000 to max;
}
```

max – the maximum possible field number

- Other users can now add new fields in their own `.proto` files that import your `.proto`.

```
extend Foo {
  optional int32 bar = 126;
}
```

FIELD RULES: `REQUIRED`

- `required`
  - in proto2
  - a well-formed message must have exactly one of this field

FIELD RULES: OPTIONAL

- `optional`
  - a well-formed message may or may not contain an optional element
  - when a message is parsed, if it does not contain an optional element, accessing the corresponding field in the parsed object returns the default value for that field

```
1.  optional int32 result_per_page = 3 [default = 10];
```

field number **field default value**

  - if the default value is not specified for an optional element, a type-specific default value is used instead
    - for strings, the default value is the empty string
    - for bytes, the default value is the empty byte string
    - for bools, the default value is false
    - for numeric types, the default value is zero
    - for enums, the default value is the first value listed in the enum's type definition.

FIELD RULES: `REPEATED`

- `repeated`
  - this field can be repeated any number of times (including zero) in a well-formed message.
  - the order of the repeated values will be preserved

- If you used the `packed` option, repeated items are encoded more efficiently.

```
message Test {
  repeated int32 f = 6 [packed=true];
}
```

  - only repeated fields of primitive numeric types can be declared "packed"

ONEOF

- Case: many optional fields and where at most one field will be set at one time.
- Solution: `oneof`

```
message SampleMessage {
  oneof test_oneof {
     string name = 4;
     SubMessage sub_message = 9;
  }
}
```

- You cannot use `required`, `optional`, and `repeated` with `oneof` fields.
- You can check which value in a oneof is set (if any).

WORKING WITH MULTIPLE PROTO FILES

```
import "myproject/other_protos.proto";
```

Protobuf wire format

Źródło https://developers.google.com/protocol-buffers/docs/encoding

VARINTS

**Recall: variable-length encoding in data compression (e.g., Huffman code),
There, codeword lengths depend on a letter's probability.
More probable letters get shorter codewords.**

- Variable length encoding for integers.
- For 64-bit integers: one to ten bytes.
- Small numbers are encoded with fewer bytes.
  - in your software, you are probably more likely to use the number 2 than, say, 3452813

- Each byte: continuation bit (MSB)+7-bit payload.
- Examples:
  - 1 is encoded as `00000001`
  - 150 is encoded as `10010110 00000001`

continuation bit highlighted

```
1.  10010110 00000001        // Original inputs.
2.   0010110  0000001        // Drop continuation bits.
3.   0000001  0010110        // Put into little-endian order.
4.   10010110                // Concatenate.
5.   128 + 16 + 4 + 2 = 150  // Interpret as integer.
```

- Why up to <u>ten</u> bytes are needed for a 64-bit integer?

Two’s complement (reminder)

- What happens for negative integers?
- As a rule, in computing we use two's complement representation.
- Example
  - one byte two’s complement representation

```
1.  0                00000000
2.  1                00000001
3.  127              01111111
4.  -128             10000000
5.  -127 =-128+1     10000001
6.  -1 =-128+127     11111111
```

The weight of the most significant bit is -128.
The weights of other bits are ”as usual” (positive).
For negative numbers, the most significant bit is always set.
All ones always represent -1.

VARINTS AND TWO'S COMPLEMENT

- `int32` and `int64`:
  - they use two's complement representation
  - negative numbers have the most significant bit set
  - a varint representing a negative int32 or int64 always uses the maximum number of bytes

- Example.
  - The varint for -2 (assuming it's an `int64`) is as follows:

```
1.  11111110 11111111 11111111 11111111 11111111
2.  11111111 11111111 11111111 11111111 00000001
```

  - explain why this is so

ZIGZAG ENCODING

- `sint32` and `sint64` use ZigZag encoding.

Signed Original *n*	Encoded As 2**n* if *n* >= 0 and as 2*(-*n*)-1 if *n*<0
0	0
-1	1
1	2
-2	3
…	…
0x7fffffff	0xfffffffe
-0x80000000	0xffffffff

- The values of `sint32` and `sint64` are first ZigZag encoded, and then represented as a varint.

RECORDS

- A protocol buffer message is a series of key-value pairs.
- Each key-value pair is turned into a record.
  - encoded using a kind of the TLV scheme (tag-length-value)
- A record contains:
  - a field number
  - a wire type (allows one to determine the size of the payload)
  - a payload (the value for that field)

Wire types

ID	Name	Used For
0	VARINT	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	I64	fixed64, sfixed64, double
2	LEN	string, bytes, embedded messages, packed repeated fields
5	I32	fixed32, sfixed32, float

TAG: ENCODING FIELD NUMBER AND WIRE TYPE

- Tag
  - form `(field_number<<3)|wire_type`
  - then encode this as a varint.

- BTW, why is wire type needed?
  - recall: wire type allows one to determine the size of the payload
  - with wire type old parsers can skip over new fields they don't understand (fields with unknown filed numbers)
  - examples:
    - if wire type == 0 (VARINT), skip a varint
    - If wire type == 1 (I64), skip 8 bytes
    - …

WIRE TYPE LEN (ID==2)

- Consider strings, bytes, …
- The wire type alone is not enough to determine the size of the payload.
- A varint for the length of the payload is placed immediately after the tag.
- The payload length varint is followed by the payload as usual.
- Example:

```
1.  message Test2 {
2.    optional string b = 2;
3.  }

4.  Let the string b be ”testing”

5.  Encoding: 12 07 [74 65 73 74 69 6e 67]
```

tag (a varint): 12 = 00010 010
010=2, wire type is LEN
the tag is followed by the length of the payload (a varint), 7

00010 = 2, the numer of the field

SUMMARY

A kind of BNF notation.
Are you familiar with BNF?

```
message    := (tag value)*
```

* - zero or more occurrences, (and) belong to BNF

zero occurrences of (tag, value)? all fields may be optional

```
tag        := (field << 3) bit-or wire_type;
                encoded as varint

value      := varint      for wire_type == VARINT,
              i32         for wire_type == I32,
              i64         for wire_type == I64,
              len-prefix  for wire_type == LEN,

varint     := int32 | int64 | uint32 | uint64 | bool | enum | sint32 | sint64;
                encoded as varints (sintN are ZigZag-encoded first)
i32        := sfixed32 | fixed32 | float;
                encoded as 4-byte little-endian;
i64        := sfixed64 | fixed64 | double;
                encoded as 8-byte little-endian;
```

recall: a message type can be the type of a field (note the recursion)

```
len-prefix := size (message | string | bytes | packed);
                size encoded as varint
string     := valid UTF-8 string (e.g. ASCII);
                max 2GB of bytes
bytes      := any sequence of 8-bit bytes;
                max 2GB of bytes
packed     := varint* | i32* | i64*;
                consecutive values of the type specified in `.proto
```

recall repeated ”packed” fields:
only one tag per a sequence of packed values

EXAMPLE (1/5)

- The *.proto file:

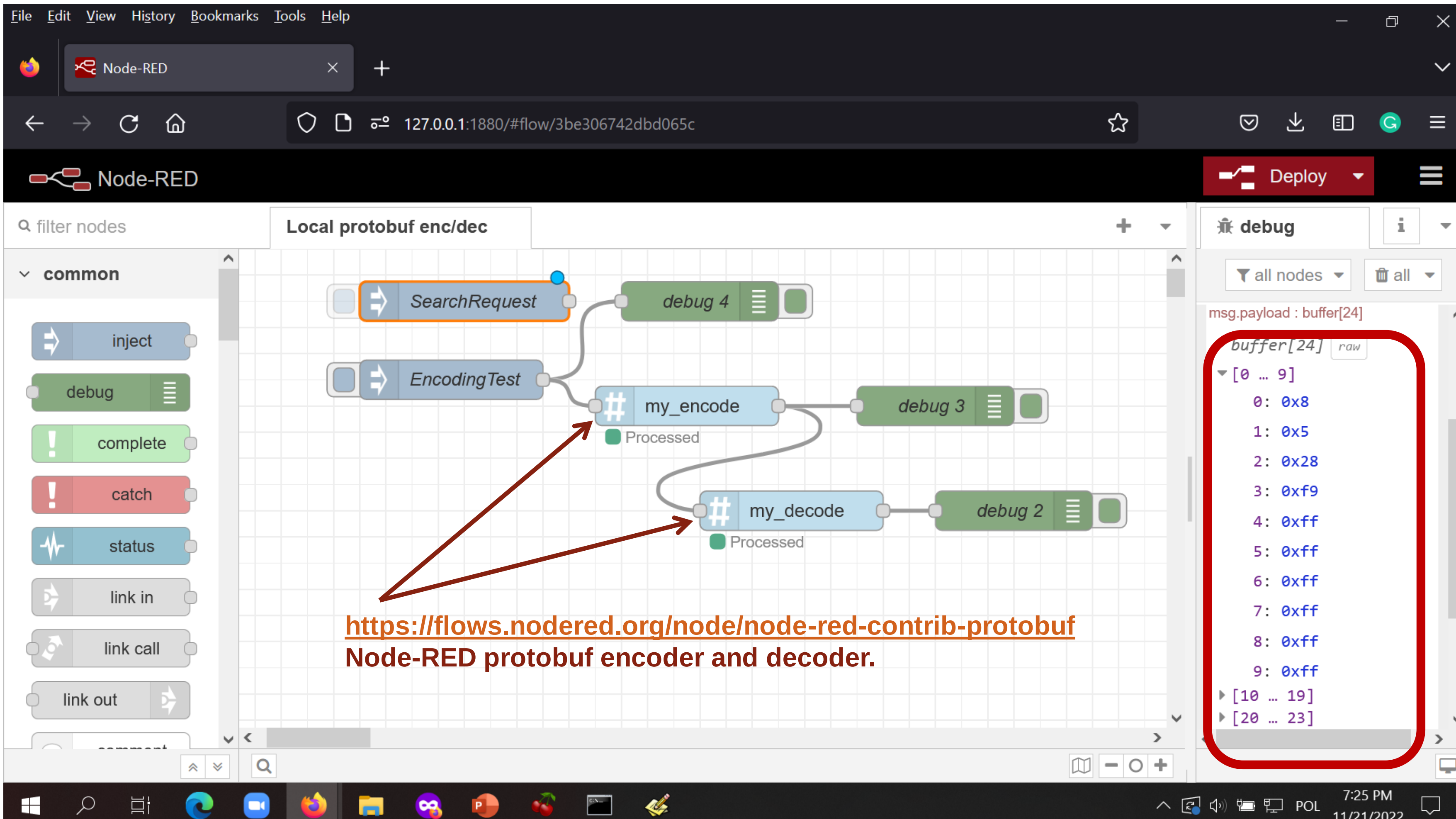
```
message EncodingTest {
  required int32 value1 = 1;
  required int32 value2 = 31;
  required int64 value3 = 5;
  required sint64 value4 = 6;
  required string value5 = 7;
}
```

- The message:

```
{
    "value1": 5,
    "value2": 3,
    "value3": -7,
    "value4": -7,
    "value5": "abcd"
}
```

EXAMPLE (2/5)

- Encoding/decoding the message according to the *.proto file:



Example (3/5)

- Wire representation (24 bytes):

```
0x08                         // tag 00001 000, field number==1, wire type==0 (VARINT)
0x05                         // value 00000101==5
0x28                         // tag 00101 000, field numer==5, wire type==0 (VARINT)
0xf9 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x01 //value
0x30                         // tag 00110 000, field numer==6, wire type==0 (VARINT)
0x0d                         // value 00001101==13 (==2*(-(-7)-1, ZigZag encoded -7
0x3a                         // tag 00111 010, field number==3, wire type==2 (LEN)
0x04                         // size==4
0x61 0x62 0x63 0x64          // value ”abcd”
0xf8 0x01                    // tag
0x03                         // value
```

```
message EncodingTest {
  required int32 value1 = 1;
  required int32 value2 = 31;
  required int64 value3 = 5;
  required sint64 value4 = 6;
  required string value5 = 7;
}
```

```
{
    "value1": 5,
    "value2": 3,
    "value3": -7,    10 byte value
    "value4": -7,    1 byte value
    "value5": "abcd"
}
```

EXAMPLE (4/5)

- Wire representation (24 bytes):

continuation bit==0

```
0xf9 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x01 //value (VARINT)
```

```
0xf9 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x01
```

VARINT: discard continuation bits, arrange in the little endian order, concatenate
64=9*7+1 bits highlighted (the leading zeros do not play any role)

most significant bit of 64-bit two’s complement representation

```
0000001 1111111 1111111 1111111 1111111
1111111 1111111 1111111 1111111 1111001     // -1-2-4=-7
```

With all ones, we would have -1.
The two zeros make it -1-2-4 = -7

EXAMPLE (5/5)

- Wire representation (24 bytes):

continuation bit==0

```
10. 0xf8 0x01                      // tag (VARINT)
11.
```

```
1.  0xf8 0x01
```

VARINT: discard continuation bits, arrange in the little endian order, concatenate

wire type == 0

```
1.  0000001 1111000 = 11111 000
```

field number == 31

Your task:
experiment with Protobuf
in Node-RED

See also https://flows.nodered.org/node/node-red-contrib-protobuf

Protobuf in Node-RED (1/2)

- Start Node-RED.
  - done on your VM
- Connect to Node-RED with your browser.
  - 127.0.0.1:1880
- Install node-red-contrib-protobuf nodes in your Node-RED.
  - done on your machine
- Check if Protobuf encode and decode nodes are available in the Node-RED palette.
- Create a Node-RED Protobuf encoding/decoding testbed flow.
  - an informal notation for the flow may look as follows

```
1. inject
2.          -> debug
3.          -> encode
4.                         -> debug
5.                         -> decode
6.                                        -> debug
```

See also https://flows.nodered.org/node/node-red-contrib-protobuf

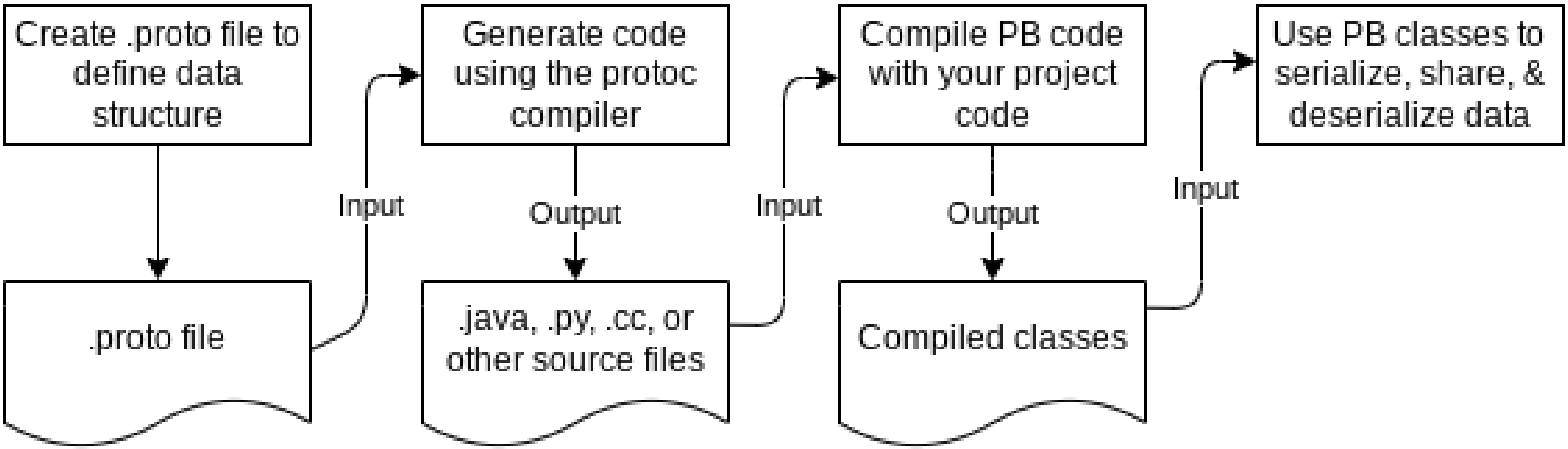
Protobuf in Node-RED (2/2)

- Create a *.proto file with your own, ”unique” message type.
  - the file should be on your VM (where Node-RED runs)
- Configure the protobuf encode and decode nodes by providing these properties:
  - a proto File (the path to your *.proto file)
  - a protobuf message type (a message type in your *.proto file)
- Configure the Inject node with a message compliant with one of the types defined in the *.proto file.
- Inject a message to be encoded and decoded.
- Using the Debug sidebar
  - check if the decoded message is the same as the original one
  - write down the sequence of bytes produced by the encoder
- Manually decode (parse) the sequence of bytes.
  - check if you’ve got the right message

Programming with Protobuf

Źródło https://developers.google.com/protocol-buffers/

PROTOCOL BUFFERS PROGRAMMING WORKFLOW



See also https://developers.google.com/protocol-buffers/docs/pythontutorial

Protobuf in Python (1/5)

- Create a *.proto file to define a data structure.
- Example:
  - my_message.proto

```
syntax="proto2";

message SearchRequest {
  required string query = 1;
  required int32 page_number = 2;
  required int32 counter = 3;
}
```

version 2 of the Protocol Buffers language (there is also version 3)

Protobuf in Python (2/5)

- Generate code using the protoc compiler.
- Example:
  - compiling a *. proto file

```
1. protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/my_message.proto
```

could select another language

Protobuf in Python (3/5)

- Example:
  - for my_message.proto, I got the file my_message_pb2.py (some whitespace deleted):

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: my_message.proto
"""Generated protocol buffer code."",,
from google.protobuf.internal import builder as _builder
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)
_sym_db = _symbol_database.Default()
DESCRIPTOR = 
_descriptor_pool.Default().AddSerializedFile(b'\n\x10my_message.proto\"D\n\rSearch
Request\x12\r\n\x05query\x18\x01 \x02(\t\x12\x13\n\x0bpage_number\x18\x02
\x02(\x05\x12\x0f\n\x07\x63ounter\x18\x03 \x02(\x05’)
_builder.BuildMessageAndEnumDescriptors(DESCRIPTOR, globals())
_builder.BuildTopDescriptorsAndMessages(DESCRIPTOR, 'my_message_pb2', globals())
if _descriptor._USE_C_DESCRIPTORS == False:
  DESCRIPTOR._options = None
  _SEARCHREQUEST._serialized_start=20
  _SEARCHREQUEST._serialized_end=88#
@@protoc_insertion_point(module_scope)my_message_pb2.py
```

Protobuf in Python (4/5)

- Invoke the package manager to install a Python protobuf package.
- Example:

```
1.  pip install protobuf
```

PROTOBUF IN PYTHON (5/5)

- Combine protobuf code with your project code.
- Example:

a message type defined in my_message.proto

```
import array
import my_message_pb2

# create a message
request = my_message_pb2.SearchRequest()
request.query = "laptops"
request.page_number = 23
request.counter = 10
print("I created a request.")

# encode to protobuf
binary_request=request.SerializeToString()
print("I encoded the request.")
print(array.array('b', binary_request))

# decode the message
decoded_request = my_message_pb2.SearchRequest()
decoded_request.ParseFromString(binary_request)
print("I decoded the request.")
print(decoded_request.query, decoded_request.page_number, decoded_request.counter)
```

output:

```
I created a request.
I encoded the request.
array('b', [10, 7, 108, 97, 112, 116, 111, 112, 115, 16, 23, 24, 10])
I decoded the request.
laptops 23 10
```

Your task: experiment with Protobuf in Python

PROTOBUF IN PYTHON

- Create a *.proto file with your own, ”unique” message type.
- Compile you message type with the protoc compiler.
- Write a simple Python script to
  - create an instance of your message type
  - initialize your message
  - encode (serialize) your message
  - decode (parse) your message
  - check if the decoded instance is the same as the original one

Your task:
use MQTT
to send Protobuf-encoded messages with Python
and receive them with Node-RED

Publishing Protobuf messages (Python) 1/2

- Continue with the Python script developed in the previous task.
- Add code to publish your Protobuf-serialized message.
  - use the Eclipse Paho MQTT Python client library
  - pick any topic
  - assume there is a local Mosquitto MQTT broker (127.0.0.1:1883)
- You may check with mosquitto_sub if the message gets delivered.

Publishing Protobuf messages (Python) 2/2

- The Paho code might look as follows.

```
…
# publish via MQTT (Paho)
broker="127.0.0.1"
port=1883
topic="products"
payload=binary_request

def on_connect(client, userdata, flags, rc):
      print("Connected with result code "+str(rc))
      client.publish(topic,payload)      # publish after the connection established
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))
def on_publish(client, userdata, mid):
      print("Published the message with mid", mid)
      client.disconnect()
      quit()

client= paho.Client("control1")            # create a client object
client.on_connect = on_connect
client.on_message = on_message
client.on_publish = on_publish
client.connect(broker,port)                # establish a connection
client.loop_forever()
```

Receiving Protobuf messages (Node-RED)

- Create a Node-RED flow to subscribe to and decode Protobuf messages.
  - an informal notation for the flow may look as follows

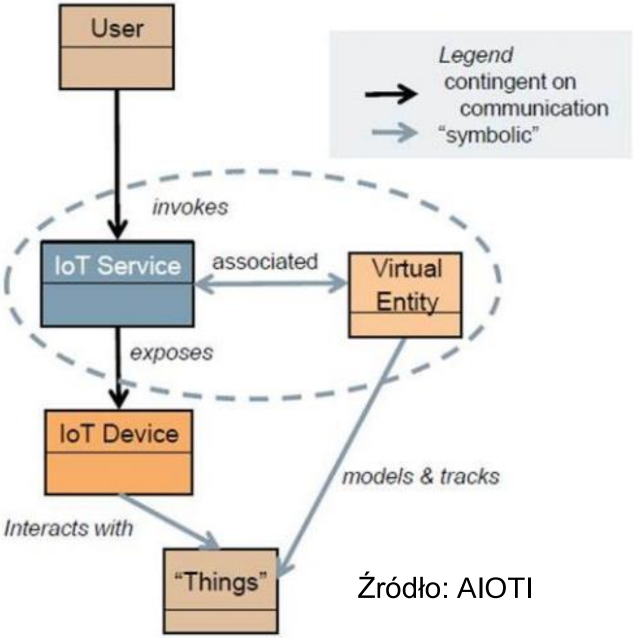
```
1. mqtt_in
2.        -> decode
3.                 -> debug
```

- Configure the subscribing client (mqtt_in) by providing these properties:
  - the MQTT server (localhost:1883)
  - the topic
- Configure the protobuf decode node by providing these properties:
  - a proto file (the path to your *.proto file used by the Python script)
  - a protobuf message type (the message type you used in your Python script)

Sending & Receiving Protobuf messages

- Now, run the Python script and check if the message encoded and published by the script is the same as the message received and decoded in Node-RED.

Dziękujemy za uwagę!



Źródło: AIOTI