OCPP Implementation Guide

Protocol Buffers & MQTT

OCPP 应用指南

基于 Protocol Buffer 和 MQTT 技术

Draft v0.3

2016/11/29

copyright © Chargerlink, Inc. 2014, 2015, 2016

1. Contributors

The following companies & people have contributed to the OCPP Implementation Guide - Protocol Buffer & MQTT.

Company (In alphabetic order)	**Name**
Chargerlink, Inc.	Jianping (Japy) Yuan, Sibo Li

2. Introduction

2.1. Purpose of this documents

The purpose of this document is to give reader the information required to create a correct interoperable OCPP Protocol Buffer and MQTT implementation.

2.2. Intended Audience

This document is intended for developers looking to understand and/or implement OCPP Protocol Buffers & MQTT in a correct and interoperable way. Rudimentary knowledge of implementing web services on a server or embedded device is assumed.

2.3. Protocol Buffers and MQTT

Protobuff stands for Protocol Buffers. Protocol Buffers is the Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. It has been used in internal projects since 2001 and went public in 2008.

MQTT stands for MQ Telemetry Transport. It was invented by Dr Andy Stanford-Clark of IBM and Arlen Nipper of Arcom in 1999. MQTT is “A light weight event and message oriented protocol allowing devices to asynchronously communicate efficiently across constrained networks to remote systems.” In this document, the implementation of MQTT is based on version 3.1 specifications.

2.4. Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

2.5. Definition & Abbreviations

PROTOBUF	Protocol Buffers
MQTT	MQ Telemetry Transport
RFC	Request for Comments
RPC	Remote procedure call
CP	Charge Point
CPO	Charge Point Operator
CS	Central System
OCPP	Open Charge Point Protocol

2.6. Reference

[RFC2119] “Key words for use in RFCs to Indicate Requirement Levels”. S. Bradner. March 1997. http://www.ietf.org/rfc/rfc2119.txt

[MQTT 3.1.1] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[Protobuf] https://developers.google.com/protocol-buffers/

3. Benefits & Issues

Protocol Buffer is used to serialize messages and reduce the bandwidth cost. And utilizing technology like MQTT allows OCPP more focus on the service-oriented logics and leaves the work for networks communication to MQTT.

3.1. Protocol Buffer

There are several benefits Protocol Buffer provides:

- **Data Compression Rate**: In general because the data structure was determined and stored in separate proto file. The raw data will reduce normally enormously from original data comparing with JSON implementation.
- **Backward Compatibility**: Numbered fields in proto files obviate the requirement for version checks, which avoids complicated logics and code for versions. You can easily maintain different versions of schema at the same time, as far as you don't reuse the same number for value fields.

3.2. MQTT

MQTT has many features and advantages including:

- Push-only communications: Compare with HTTP, this mechanism generates less latency time in processing.
- Bitwise header and low bandwidth cost comparing with HTTP
- Publish / Subscribe messaging principals, which allows Charge Points to multiple backend systems easily.
- It defines different level of QoS (Quality of Service) to ensure the delivery of messages.
- The feature of "Retained Messages" and "Last Will Testament" lowers the power consumption for maintaining connection sessions for IoT networks.

4. Connection

We use MQTT for network connection, which is based on TCP/IP. It is recommended to choose broker software supporting MQTT later than 3.1. After set up the MQTT server, we need to configure both Charge Points and Central System as MQTT clients. MQTT implementation is based on Publisher/Subscriber model. All OCPP messages are posted into MQTT topics, and message receivers need to subscribe to the appropriate topics.

4.1. Network Connection

To successfully establish network connection, the MQTT clients need to provide several parameters to server:

Table 1 Connection Parameters

Parameter	Type	Description
CLEAN_SESSION	Bool	To keep session on cloud
CLIENT_ID	String	Id of Client, including CPs and CS
USERNAME	String	Username need to be unique within same network
PASSWORD	String	Token for login verification

4.2. CONNECT and CONNACK

After a Network Connection is established by a MQTT Client to a Server, the first Packet sent from Client to Server MUST be a CONNECT Packet. For different MQTT client, the response might different. But all of them will return CONNACK. The variable header of CONNACK package contains Connect Return Code as following:

Table 2 Connect Return Code Values

Value	Return Code Response	Description
0	0x00 Connection Accepted	Connection accepted
1	0x01 Connection Refused, unacceptable protocol version	The Server does not support the level of the MQTT protocol requested by the Client
2	0x02 Connection Refused, identifier rejected	The Client identifier is correct UTF-8 but not allowed by the Server
3	0x03 Connection Refused, Server unavailable	The Network Connection has been made but the MQTT service is unavailable
4	0x04 Connection Refused, bad user name or password	The data in the user name or password is malformed
5	0x05 Connection Refused, not authorized	The Client is not authorized to connect
6-255		Reserved for future use

4.3. KeepAlive

MQTT is based on TCP/IP. MQTT provides KeepAlive function to solve sync problem between Server and Client. In MQTT 3.1 specification, it is the responsibility of the Client to ensure that the interval between Control Packets being sent does not exceed the Keep Alive value. In the absence of sending any other Control Packets, the Client MUST send a PINGREQ Packet. After receiving a PINGREQ, the broker must reply with a PINGRESP package. Both PINGREQ and PINGRESP don't contain any payload. If a client doesn't send PINGREQ or any other message in one and half time of the KeepAlive interval, the broker must disconnect it.

4.4. MQTT Topics

Based on the message direction, there are two types of OCPP topics: the message

sent from Charge Point to Central System, and the message sent from Central System to Charge Point. To support this feature, we defined two types of MQTT topics. It is recommended to define 3 levels of topics, though different CPOs may choose more sophisticated structure based on their network topology.

Table 3 MQTT Topics

Publisher	Subscriber	Topic
CP	CS	/OCPP/CP/${ChargePoint_ID}
CS	CP	/OCPP/CS/${ChargePoint_ID}

- **Message from Charge Point**

The message will be posted into topic such as "/OCPP/CP/CP0001", the first level of the path denote the type of sender as Charge Point. The second level includes the unique identification for Charge Point. Meanwhile the Central System is subscribed to the same topic, and processes the relevant message after it was published.

- **Message from Central System**

The message will be posted into topic such as "/OCPP/CS/CP0001", the first level of the path denote the type of sender as Central System. The second level includes the unique identification for Charge Point, the receiver. Meanwhile the Charge Point is subscribed to the same topic, and processes the relevant message after it was published.

4.5. Security

Since MQTT is based on TCP/IP, TLS/SSL is used to authenticate servers and clients and then used to encrypt messages between the authenticated parties.

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), both frequently referred to as "SSL", are cryptographic protocols that provide communications security over a computer network. The Transport Layer Security protocol aims primarily to provide privacy and data integrity between two communicating computer applications.

5. RPC Framework

5.1. Syntax File

To support OCPP protobuf Implementation work, Charge points and central systems should agree on consistent .proto files, which defined the syntax of messages and data. Consistent means all definition should not have any conflicted name or sequence. The syntax file shall include details for supporting messages and DataTypes. It is also recommended to include corresponding OCPP version information in the protocol buffer file.

Based on consistent syntax files, the sender of the message can serialize message information from a structured data into protocol butter message flow, and then the receiver would de-serialize it afterwards. The data format before serialization and after de-serialization depends on the local operating system.

In general, the syntax file will include a base class for message, the MessagesType enumeration for locally supported messages types by Charge Point or Central System, and then the definitions for locally supported Messages and their data types.

5.2. Base Class

Message Class is the parent class for all OCPP message class. It specifies data fields that may be included: Message Type ID, Unique ID, Error Code, Error Description and Payload information, The Payload fields include the actual OCPP information itself. For protocol buffer implementation, the sequence for each data field is required and need to be consistent through out all CPs and CS.

Table 4 Class Definition for Message

Field	Type	Seq	Required/ Optional	Description
MessageTypeId	MessageType	1	R	Define the type of Message, whether it is Call, CallResult or CallError.
UniqueId	String	2	R	This must be the exact same id that is in the call request so that the recipient can match request and result.
Action	String	3	O	The Message Name of OCPP. E.g. Authorize
ErrorCode	ErrorType	4	O	The string must contain one from ErrorType Table.
ErrorDescription	String	5	O	Detailed Error information
Payload	Bytes	6	O	Payload field contains the serialized strings of bytes for protobuf format of OCPP message.

5.3. Messages Type and Synchronicity

Basically what we need is very simple: we need to send a message (CALL) and receive a reply (CALLRESULT) or an explanation why the message could not be handled properly (CALLERROR). For possible future compatibility we will keep the numbering of these message in sync with WAMP. Our actual OCPP message will be put into a wrapper that at least contains the type of message, a unique message ID and the payload, the OCPP message itself.

Table 5 MessageType Definition

MessageType	MessageTypeId	Direction
CALL	2	Client-to-Server
CALLRESULT	3	Server-to-Client
CALLERROR	4	Server-to-Client

A Charge Point or Central System SHOULD NOT send a CALL message to the other party unless all the CALL messages it sent before have been responded to or have timed out. A CALL message has been responded to when a CALLERROR or CALLRESULT message has been received with the message ID of the CALL message.
A CALL message has timed out when:

• It has not been responded to, and

• an implementation-dependent timeout interval has elapsed since the message was sent.

Implementations are free to choose this timeout interval. It is RECOMMENDED that they take into account the kind of network used to communicate with the other party. Mobile networks typically have much longer worst-case round-trip times than fixed lines.
The above requirements do not rule out that a Charge Point or Central System will receive a CALL message from the other party while it is waiting for a CALLERROR or CALLRESULT. Such a situation is difficult to prevent because CALL messages from both sides can always cross each other.

Table 6 Fields for Different MessageTypes

Field	CALL	CALLRESULT	CALLERROR
MessageTypeId	Yes	Yes	Yes
UniqueId	Yes	Yes	Yes
Action	Yes	No	No
ErrorCode	No	No	Yes
ErrorDescription	No	No	Yes
Payload	Yes	Yes	No

The req message of OCPP is wrapped in the payload of CALL, and the conf message of OCPP is wrapped in the payload of CALLRESULT after proceed successfully.

Table 7 Example of Different MessageTypes

MessageTypes	Example
CALL	MessageTypeId=2 UniqueId=”12312313” Action=“StartTransaction“ Payload = “…”
CALLRESULT	MessageTypeId=3 UniqueId=”12312313” Payload = “…”
CALLERROR	MessageTypeId=4 UniqueId=”12312313” ErrorCode = ProtocError ErrorDescription = “…”

5.4. CallError

We only use CallError in two situations:

1) An error occurred during the transport of the message. This can be a network issue, an availability of service issue, etc.

2) The call is received but the content of the call does not meet the requirements for a proper message. This could be incorrect syntax, missing mandatory fields, an existing call with the same unique identifier is being handled already, unique identifier too long, etc.

Table 8 Error Codes

Error Code	Description
NotImplemented	Requested Action is not known by receiver
NotSupported	Requested Action is recognized but not supported by the receiver
InternalError	An internal error occurred and the receiver was not able to process the requested Action successfully
ProtocolError	Payload for Action is incomplete
SecurityError	During the processing of Action a security issue occurred preventing receiver from completing the Action successfully
FormationViolation	Payload for Action is syntactically incorrect or not conform the PDU structure for Action
PropertyConstraintViolation	Payload is syntactically correct but at least one field contains an invalid value
OccurrenceConstraintViolation	Payload for Action is syntactically correct but at least one of the fields violates occurence constraints
TypeConstraintViolation	Payload for Action is syntactically correct but at least one of the fields violates data type constraints (e.g. “somestring”: 12)
GenericError	Any other error not covered by the previous ones

6. Examples

The Following examples are excerpted from protocol buffer syntax file.

6.1. Base Class – Message

```
message Messages
{
    required MessageType MessageTypeId = 1;
    required string UniqueId = 2;
    optional string Action    = 3;
    optional ErrorType ErrorCode = 4;
    optional string ErrorDescription = 5
    optional bytes Payload = 6;
}
```

6.2. OCPP Message – BootNotification

```
message BootNotification_req
{
    optional string chargeBoxSerialNumber = 1;
    optional string chargePointModel = 2;
    optional string chargePointSerialNumber = 3;
    optional string chargePointVendor = 4;
    optional string firmwareVersion = 5;
    optional string iccid = 6;
    optional string imsi = 7;
    optional string meterSerialNumber = 8;
    optional string meterType = 9;
}
message BootNotification_conf
{
    required dateTime currentTime;
}
```