



OCPP 2.1
Part 4 - JSON over WebSockets implementation guide

Edition 2, 2025-12-03

Table of Contents

Disclaimer	1
Version History	2
1. Introduction	3
1.1. Purpose of this document	3
1.2. Intended audience	3
1.3. OCPP-S and OCPP-J	3
1.4. Conventions	3
1.5. Definitions & Abbreviations	3
1.6. References	4
2. Benefits & Issues	5
3. WebSockets	6
3.1. Client request	6
3.2. Maintaining support for OCPP 2.0.1	7
3.3. Server response	7
3.4. WebSocket Compression	8
4. RPC framework	9
4.1. Introduction	9
4.2. Message structures for different message types	11
4.3. RPC Framework Error Codes	16
4.4. Extension fallback mechanism	16
5. Connection	17
5.1. Data integrity	17
5.2. TLS fragment length	17
5.3. WebSocket Ping in relation to OCPP Heartbeat	17
5.4. Reconnecting	19
5.5. Network node hierarchy	19
6. OCPP Routing	20
6.1. Local Controller	20
6.2. Connections	20
6.3. Connection loss	21
6.4. Local Controller initiated messages	21
6.5. Local Controller Security	21
7. Signed Messages	23
7.1. Signed Message Format	23
7.2. Handling Signed Messages	23
7.3. Allowed Algorithms	23
7.4. Key Management	24
8. Configuration	25
8.1. RetryBackOffRepeatTimes	25
8.2. RetryBackOffRandomRange	25
8.3. RetryBackOffWaitMinimum	25
8.4. WebSocketPingInterval	25
9. CustomData Extension	27

Disclaimer

Copyright © 2010 – 2025 Open Charge Alliance. All rights reserved.

This document is made available under the **Creative Commons Attribution-NoDerivatives 4.0 International Public License** (<https://creativecommons.org/licenses/by-nd/4.0/legalcode>).

Version History

Version	Date	Description
2.1 Edition 2	2025-12-03	OCPP 2.1 Edition 2. All errata from OCPP 2.1 Part 4 until and including Errata 2025-11 have been merged into this version of the specification.
2.1 Edition 1	2025-01-23	OCPP 2.1 Edition 1

Chapter 1. Introduction

1.1. Purpose of this document

The purpose of this document is to give the reader the information required to create a correct interoperable OCPP [JSON](#) implementation (OCPP-J). We will try to explain what is mandatory, what is considered good practice and what one should not do, based on our own experience. Undoubtedly misunderstandings or ambiguities will remain but by means of this document we aim to prevent them as much as possible.

1.2. Intended audience

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

1.3. OCPP-S and OCPP-J

With the introduction of OCPP 1.6, there were two different flavors of OCPP; SOAP and JSON. To avoid confusion in communication on the type of implementation we recommend using the distinct suffixes -J and -S to indicate JSON or SOAP. In generic terms this would be OCPP-J for JSON and OCPP-S for SOAP. Version specific terminology would be OCPP1.6J or OCPP1.2S. If no suffix is specified for

OCPP 1.2 or 1.5 then a SOAP implementation must be assumed. For release 1.6 this is no longer implicit and should always be made clear. If a system supports both the JSON and SOAP variant it is considered good practice to label this OCPP1.6JS instead of just OCPP1.6.

As of OCPP 2.0.1 the SOAP transport is no longer supported.

1.4. 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 [\[RFC2119\]](#).

1.5. Definitions & Abbreviations

Abbreviation	Description
IANA	Internet Assigned Numbers Authority (www.iana.org).
OCPP-J	OCPP communication over WebSocket using JSON. Specific OCPP versions should be indicated with the J extension. OCPP1.6J means we are talking about a JSON/WebSocket implementation of 1.6.
OCPP-S	OCPP communication over SOAP and HTTP(S). As of version 1.6 this should explicitly mentioned. Older versions are assumed to be S unless clearly specified otherwise, e.g. OCPP1.5 is the same as OCPP1.5S
RPC	Remote procedure call
WAMP	WAMP is an open WebSocket subprotocol that provides messaging patterns to handle asynchronous data.

1.6. References

Reference	Description
[ISO15118-2]	Road vehicles – Vehicle to grid communication interface – Part 2: Technical protocol description and Open Systems Interconnection (OSI) layer requirements, Document Identifier: 69/216/CDV. https://webstore.iec.ch/publication/9273
[OCPP2.0-PART2]	"OCPP 2.0.1: Part 2 - Specification". http://www.openchargealliance.org/downloads/
[OCPP2.1-PART2]	"OCPP 2.1: Part 2 - Specification". http://www.openchargealliance.org/downloads/
[RFC1951]	"DEFLATE Compressed Data Format Specification version 1.3". https://www.ietf.org/rfc/rfc1951
[RFC2119]	"Key words for use in RFCs to Indicate Requirement Levels". S. Bradner. March 1997. http://www.ietf.org/rfc/rfc2119.txt
[RFC2616]	"Hypertext Transfer Protocol – HTTP/1.1". http://tools.ietf.org/html/rfc2616
[RFC3629]	"UTF-8, a transformation format of ISO 10646". http://tools.ietf.org/html/rfc3629
[RFC3986]	"Uniform Resource Identifier (URI): Generic Syntax". http://tools.ietf.org/html/rfc3986
[RFC5246]	"The Transport Layer Security (TLS) Protocol; Version 1.2". http://tools.ietf.org/html/rfc5246
[RFC6455]	"The WebSocket Protocol". http://tools.ietf.org/html/rfc6455
[RFC7515]	"JSON Web Signatures (JWS)". https://tools.ietf.org/html/rfc7515
[RFC7518]	"JSON Web Algorithms (JWA)". https://tools.ietf.org/html/rfc7518
[RFC7617]	"The 'Basic' HTTP Authentication Scheme". https://tools.ietf.org/html/rfc7617
[RFC7692]	"Compression Extensions for WebSocket". https://tools.ietf.org/html/rfc7692
[RFC8259]	"The JavaScript Object Notation (JSON) Data Interchange Format". T. Bray. December 2017. https://tools.ietf.org/html/rfc8259
[RFC8265]	Describes methods for handling Unicode strings representing usernames and passwords. https://datatracker.ietf.org/doc/html/rfc8265
[WAMP]	http://wamp-proto.org/

Chapter 2. Benefits & Issues

The WebSocket protocol is defined in [\[RFC6455\]](#). Working implementations of earlier draft WebSocket specifications exist, but OCPP-J implementations SHOULD use the protocol described in [\[RFC6455\]](#).

Be aware that WebSocket defines its own message structure on top of TCP. Data sent over a WebSocket, on a TCP level, is wrapped in a WebSocket frame with a header. When using a framework this is completely transparent. When working for an embedded system however, WebSocket libraries may not be available and then one has to frame messages correctly according to [\[RFC6455\]](#) him/herself.

Chapter 3. WebSockets

For the connection between a Charging Station and a Charging Station Management System (CSMS) using OCPP-J, the CSMS acts as a WebSocket server and the Charging Station acts as a WebSocket client.

3.1. Client request

To set up a connection, the Charging Station initiates a WebSocket connection as described in [\[RFC6455\]](#) section 4, "Opening Handshake".

OCPP-J imposes extra constraints on the URL and the WebSocket subprotocol, detailed in the following two sections 3.1.1 and 3.1.2.

The Client (Charging Station) SHALL keep this WebSocket connection open all the time.

3.1.1. The connection URL

To initiate a WebSocket connection, the Charging Station needs a URL ([\[RFC3986\]](#)) to connect to. This URL is henceforth called the "connection URL". This connection URL is specific to a Charging Station. The Charging Station's connection URL contains the Charging Station identity so that the CSMS knows which Charging Station a WebSocket connection belongs to. However it is RECOMMENDED to let the CSMS NOT solely rely on the connection URL to identify a Charging Station, but to double-check the Charging Station's identity against their authentication credentials.

A CSMS supporting OCPP-J MUST provide at least one OCPP-J endpoint URL, from which the Charging Station SHOULD derive its connection URL. This OCPP-J endpoint URL can be any URL with a "ws" or "wss" scheme. How the Charging Station obtains an OCPP-J endpoint URL is outside of the scope of this document.

To derive its connection URL, the Charging Station modifies the OCPP-J endpoint URL by appending to the path first a '/' (U+002F SOLIDUS) and then a string uniquely identifying the Charging Station. This uniquely identifying string has to be percent-encoded / URL encoded as necessary as described in [\[RFC3986\]](#).

Example 1: for a Charging Station with identity "CS001" connecting to a CSMS with OCPP-J endpoint URL "ws://csms.example.com/ocpp" this would give the following connection URL:

`ws://csms.example.com/ocpp/CS001`

Example 2: for a Charging Station with identity "RDAM123" connecting to a CSMS with OCPP-J endpoint URL "wss://csms.example.com/ocppj" this would give the following URL:

`wss://csms.example.com/ocppj/RDAM%7C123`

The Charging Station identity datatype is *identifierString* (For definition see [\[OCPP2.1-PART2\]](#) . In addition, the colon ":" character SHALL NOT be used, because the unique identifier is also used for the basic authentication username. The colon ":" character is used to separate the basic authentication username and the password. The maximum length of the Charging Station identity is: 48 characters. (Note: Maximum length was chosen to ensure compatibility with EVSE ID from [\[ISO15118-2\]](#).)

3.1.2. OCPP version negotiation

(Updated in OCPP 2.1)

The OCPP version(s) MUST be specified in the Sec-Websocket-Protocol field. The client lists the OCPP version(s) it supports in order of preference. They SHOULD be one or more of the following values:

Table 1. OCPP Versions

OCPP version	WebSocket subprotocol name
1.2	ocpp1.2
1.5	ocpp1.5
1.6	ocpp1.6
2.0	ocpp2.0
2.0.1	ocpp2.0.1
2.1	ocpp2.1

The ones for OCPP 1.2, 1.5, 1.6, 2.0, 2.0.1 and 2.1 are official WebSocket subprotocol name values. They are registered as such with IANA. These are official WebSocket subprotocol name values, that are registered as such with IANA.

Note that OCPP 1.2 and 1.5 are in the list. Since the JSON over WebSocket solution is independent of the actual message content the solution can be used for older OCPP versions as well. Please keep in mind that in these cases the implementation should preferably also maintain support for the SOAP based solution to be interoperable.

If the OCPP version is part of the OCPP-J endpoint URL it SHALL NOT determine the OCPP version to use, because the OCPP version is selected via the websocket protocol negotiation mechanism, as explained in section 3.3 [Server response](#).

3.1.3. Example of an opening HTTP request

(Updated in OCPP 2.1)

The following is an example of an opening HTTP request of an OCPP-J connection handshake:

```
GET /webServices/ocpp/CS3211 HTTP/1.1
Host: some.server.com:33033
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMBDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: ocpp2.1, ocpp2.0.1, ocpp1.6
Sec-WebSocket-Version: 13
```

The bold parts are found as such in every WebSocket handshake request, the other parts are specific to this example.

In this example, the CSMS's OCPP-J endpoint URL is "ws://some.server.com:33033/webServices/ocpp". The Charging Station's unique identifier is "CS3211", so the path to request becomes "webServices/ocpp/CS3211".

With the Sec-WebSocket-Protocol header, the Charging Station indicates here that it can use OCPP 2.1, OCPP 2.0.1 and OCPP 1.6J, with a preference for the first in the list, in this example OCPP 2.1.

The other headers in this example are part of the HTTP and WebSocket protocols and are not relevant to those implementing OCPP-J on top of third-party WebSocket libraries. The roles of these headers are explained in [\[RFC2616\]](#) and [\[RFC6455\]](#).

3.2. Maintaining support for OCPP 2.0.1

A CSMS that supports OCPP 2.1 IS RECOMMENDED to also support OCPP 2.0.1, such that Charging Stations in the network that use OCPP 2.0.1 can still connect.

A Charging Station that supports OCPP 2.1 IS RECOMMENDED to also support OCPP 2.0.1, such that it can still connect to CSMSs that use OCPP 2.0.1.

3.3. Server response

Upon receiving the Charging Station's handshake request, the CSMS has to finish the handshake with a response as described in [\[RFC6455\]](#).

The following OCPP-J-specific conditions apply:

- If the CSMS does not recognize the Charging Station identifier in the URL path, it SHOULD send an HTTP response with status 404 and abort the WebSocket connection as described in [\[RFC6455\]](#).
- If the CSMS does not agree to using one of the subprotocols offered by the client, it MUST complete the WebSocket handshake with a response without a Sec-WebSocket-Protocol header and then immediately close the WebSocket connection.

If the CSMS accepts the above example request and agrees to using OCPP 2.1 with the Charging Station, the CSMS's response will look as follows:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRBK+xOo=

Sec-WebSocket-Protocol: ocpp2.1

The bold parts are found as such in every WebSocket handshake response, the other parts are specific to this example.

The role of the Sec-WebSocket-Accept header is explained in [\[RFC6455\]](#).

The Sec-WebSocket-Protocol header indicates that the server will be using OCPP2.1 on this connection.

For a definition on how a server SHALL report a 'failure to process the message', see: [CALLERROR](#)

3.4. WebSocket Compression

OCPP 2.1 supports RFC 7692: Compression Extensions for WebSocket see: [\[RFC6455\]](#)

Table 2. WebSocket compression support requirement for devices

Device	WebSocket Compression Support
Charging Station	Optional
CSMS	Required
Local Controller	Required

The CSMS (and Local Controller) SHALL support [RFC 7692](#), WebSocket compression, which is a relative simple way to reduce mobile data usage. For a Charging Station this is not a hard requirement, as this might be more complex to implement on an embedded platform. It is RECOMMENDED to be implemented on a Charging Station, because it is an efficient solution to reduce mobile data usage.

[RFC 7692](#) allows the Charging Station and the CSMS to do a negotiation during the connection setup. When both parties support the Compression Extension they will then use DEFLATE compression ([\[RFC1951\]](#)) when sending data over the line. When one of the parties doesn't support it, the JSON will be sent uncompressed (like in OCPP 1.6J).

When the Charging Station detects that compression is not used, it is RECOMMENDED not to close the connection, as turning off compression can be very useful during development, testing and debugging.

For more detailed information read the [RFC 7692](#).

Chapter 4. RPC framework

4.1. Introduction

A websocket is a full-duplex connection, simply put a pipe where data goes in and data can come out and without a clear relation between in and out. The WebSocket protocol by itself provides no way to relate messages as requests and responses. To encode these request/response relations we need a small protocol on top of WebSocket. This problem occurs in more use cases of WebSocket so there are existing schemes to solve it. The most widely-used is WAMP (see [WAMP]) but with the current version of that framework handling RPCs symmetrically is not WAMP compliant. Since the required framework is very simple we decided to define our own framework, inspired by WAMP, leaving out what we do not need and adding what we find missing.

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). Many CALLRESULT message have a simple content with a status describing whether the CALL was accepted or not, but for those cases where a CALLRESULT contains complex data elements that the receiver might not be able to process, a receiver can respond to the CALLRESULT with a CALLRESULTTERROR message to notify the sender that its response could not be processed.

Besides the request-response pair of CALL-CALLRESULT the framework also offers a SEND message, which only sends a message and does not expect a response.

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.

4.1.1. Synchronicity

A Charging Station or CSMS SHALL NOT send a CALL message to the other party unless the CALL message it sent before has been responded to or has timed out. This does not mean that the CSMS cannot send a message to another Charging Station, while waiting for a response of a first Charging Station, this rule is per OCPP-J connection. 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.

NOTE

The above requirements do not rule out that a Charging Station or CSMS 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.

NOTE

A Charging Station or CSMS is allowed to send a SEND message to the other party before the CALL message it has sent has been responded to or has timed out.

4.1.2. Message validity and Character encoding

The whole message consisting of wrapper and payload MUST be valid JSON encoded with the UTF-8 (see [RFC3629]) character encoding. Furthermore, the Charging Station and CSMS have the right to reject messages which are not conform the JSON schemas.

Note that all valid US-ASCII text is also valid UTF-8, so if a system sends only US-ASCII text, all messages it sends comply with the UTF-8 requirement. Non US-ASCII characters SHOULD only be used for sending natural-language text. An example of such natural-language text is the MessageType which contains the text of a DisplayMessage in OCPP 2.1

4.1.3. The message type

(Updated in OCPP 2.1)

To identify the type of message one of the following Message Type Numbers MUST be used.

Table 3. Message types

MessageType	Message Type Number	Description
CALL	2	Request message, i.e. messages ending in "Request"
CALLRESULT	3	Response message, i.e. message ending in "Response"
CALLERROR	4	Error response to a request message
CALLRESULTERROR	5	(2.1) Error response to a response message
SEND	6	(2.1) Unconfirmed message without response, i.e. messages not ending in "Request" or "Response"

When a system receives a message with a Message Type Number not in this list, it SHALL ignore the message payload. Each message type may have additional required fields.

4.1.4. Message ID

The message ID serves to identify a request. A message ID for any CALL or SEND message MUST be different from all message IDs previously used by the same sender for any other CALL or SEND message on any WebSocket connection using the same unique Charging Station identifier. The message ID for a retried message (e.g. when no response was received within timeout) MAY be identical to the message ID of the original message.

A message ID for a CALLRESULT or CALLERROR message MUST be equal to that of the CALL message that the CALLRESULT or CALLERROR message is a response to.

A message ID for a CALLRESULTERROR message MUST be equal to that of the CALLRESULT message that caused the CALLRESULTERROR message to be sent.

Table 4. Unique Message ID

Name	Datatype	Restrictions
messageld	string[36]	Unique message ID, maximum length of 36 characters, to allow for UUIDs/GUIDs

4.1.5. JSON Payload

The Payload of a message is a JSON object containing the arguments relevant to the Action.

If there is no payload JSON allows for two different notations: `null` or and empty object `{}`. Although it seems trivial, we consider it good practice to only use the empty object statement. Null usually represents something undefined, which is not the same as empty, and also `{}` is shorter.

When a field is optional in the OCPP action (0..1 or 0..*) and is left empty for a specific request/response, JSON allows for a couple of different ways to put this in a JSON string. But because OCPP is designed for wireless links, OCPP only allows 1 option: Do not put the field in the payload (so `null`, `{}` or `[]` are not allowed for an empty field).

When a field has a cardinality of zero/one to many (0..* or 1..*) and it has been given one entity, than it will still remain a list, but of size 1.

4.1.6. Action

The **Action** field in the [CALL](#) message MUST be the OCPP message name without the "Request" suffix.

For example: For a "BootNotificationRequest" the action field will be set to "BootNotification".

BTW: The [CALLRESULT](#) does not contain the action field. A client can match the Response ([CALLRESULT](#)) with the Request ([CALL](#)) via the **MessageId** field.

4.1.7. Message Validity

An message is only valid when:

- **Action** is a known [Action](#).
- The JSON payload is valid JSON
- All the required field for the [Action](#) are present
- All data is of the correct data type.

When a message is not valid, the server SHALL respond with a [CALLERROR](#)

4.2. Message structures for different message types

NOTE

You may find the Charging Station identity missing in the following paragraphs. The identity is exchanged during the WebSocket connection handshake and is a property of the connection. Every message is sent by or directed at this identity. There is therefore no need to repeat it in each message.

4.2.1. CALL

A CALL always consists of 4 elements: The standard elements **MessageTypeId** and **MessageId**, a specific **Action** that is required on the other side and a payload, the arguments to the **Action**. The syntax of a CALL looks like this:

[<MessageTypeId>, <MessageId>, <Action>, {<Payload>}]

Table 5. CALL Fields

Field	Datatype	Meaning
MessageTypeId	integer	This is a Message Type Number which is used to identify the type of the message.
MessageId	string[36]	This is a unique identifier that will be used to match request and result.
Action	string	The name of the remote procedure or action. This field SHALL contain a case-sensitive string. The field SHALL contain the OCPP Message name without the "Request" suffix. For example: For a "BootNotificationRequest", this field shall be set to "BootNotification".
Payload	JSON	JSON Payload of the action, see: JSON Payload for more information.

For example, a BootNotificationRequest could look like this:

```
[ 2,
  "19223201",
  "BootNotification",
  {
    "reason": "PowerUp",
    "chargingStation": {
      "model": "SingleSocketCharger",
      "vendorName": "VendorX"
    }
  }
]
```

4.2.2. CALLRESULT

If the call can be handled correctly the result will be a regular CALLRESULT. Error situations that are covered by the definition of the OCPP response definition are not considered errors in this context. They are regular results and as such will be treated as a normal CALLRESULT, even if the result is undesirable for the recipient.

A CALLRESULT always consists of 3 elements: The standard elements MessageTypeld and Messageld and a payload, containing the response to the *Action* in the original Call.

The syntax of a CALLRESULT looks like this:

[<MessageTypeld>, "<Messageld>", {<Payload>}]

Table 6. CALLRESULT Fields

Field	Datatype	Meaning
MessageTypeld	integer	This is a Message Type Number which is used to identify the type of the message.
Messageld	string[36]	This MUST be the exact same ID that is in the call request so that the recipient can match request and result.
Payload	JSON	JSON Payload of the action, see: JSON Payload for more information.

For example, a BootNotification response could look like this:

```
[ 3,
  "19223201",
  {
    "currentTime": "2013-02-01T20:53:32.486Z",
    "interval": 300,
    "status": "Accepted"
  }
]
```

4.2.3. CALLERROR and CALLRESULTERROR

(Updated in OCPP 2.1)

CALLERROR and CALLRESULTERROR are basically the same. A CALLERROR, rather than a CALLRESULT, is sent in response to a CALL message type, when it contains errors. A CALLRESULTERROR is sent back on receipt of a CALLRESULT that contains errors to notify the sender that the response could not be processed.

We only use CALLERROR or CALLRESULTERROR 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 or CALLRESULT is received but the content of the message does not meet the requirements for a proper message. This could be missing mandatory fields, an existing message with the same unique identifier is being handled already, unique identifier too long, invalid JSON or OCPP syntax etc.

When a server needs to report a 'failure to process the message', the server SHALL use a Message Type: CALLERROR (MessageTypeNumber = 4).

When a server receives a corrupt message, the CALLERROR SHALL NOT directly include syntactically invalid JSON (For example, without encoding it first). When also the Messageld cannot be read, the CALLERROR SHALL contain "-1" as Messageld.

When a message contains any invalid OCPP and/or it is not conform the JSON schema, the system is allowed to drop the message.

A CALLERROR and CALLRESULTERROR always consist of 5 elements: The standard elements MessageTypeId and Messageld, an errorCode string, an errorDescription string and an errorDetails object.

The syntax of a CALLERROR looks like this:

```
[<MessageTypeId>, "<Messageld>", "<errorCode>", "<errorDescription>", {<errorDetails>}]
```

Table 7. CALLERROR/CALLRESULTERROR Fields

Field	Datatype	Meaning
MessageTypeId	integer	This is a Message Type Number which is used to identify the type of the message.
Messageld	string[36]	This MUST be the exact same id that is in the call request so that the recipient can match request and result.
ErrorCode	string	This field MUST contain a string from the RPC Framework Error Codes table .
ErrorDescription	string[255]	Should be filled in if possible, otherwise a clear empty string "".
ErrorDetails	JSON	This JSON object describes error details in an undefined way. If there are no error details you MUST fill in an empty object {}.

For example, a CALLERROR could look like this:

```
[ 4,
  "162376037",
  "Not Supported",
  "SetDisplayMessageRequest not supported",
  {}
]
```

4.2.4. SEND

(New in OCPP 2.1)

A SEND message type is sent as an unconfirmed message: it does not expect a response. It is used, for example, for frequent periodic monitor values that can be processed faster if no response has to be returned. Since SEND messages use the same websocket connection as CALL/CALLRESULT messages, the frequent sending of large SEND messages may cause a delay for other messages.

The receiver SHALL NOT respond to a SEND message with a CALLRESULT or CALLERROR message.

The sender MAY send a SEND message at any time, i.e. there is no need to wait for the CALLRESULT of the previous CALL

message before sending it.

A SEND always consists of the same 4 elements as a CALL message: The standard elements MessageTypeld and Messageld, a specific Action that is required on the other side and a payload, the arguments to the Action. The syntax of a SEND looks like this:

[<MessageTypeld>, <Messageld>, <Action>, {<Payload>}]

Table 8. SEND Fields

Field	Datatype	Meaning
MessageTypeld	integer	This is a Message Type Number which is used to identify the type of the message.
Messageld	string[36]	This is a unique identifier that will be used to match request and result.
Action	string	The name of the remote procedure or action. This field SHALL contain a case-sensitive string. The field SHALL contain the OCPP Message name. For example: For a "NotifyPeriodicEventStream", this field shall be set to "NotifyPeriodicEventStream".
Payload	JSON	JSON Payload of the action, see: JSON Payload for more information.

For example, a NotifyPeriodicEventStream could look like this:

```
[ 6,
  "19223201",
  "NotifyPeriodicEventStream",
  {
    "id": 123,
    "pending": 0,
    "data": [ { "t": "2024-08-27T12:30:40Z", "v": "230.4" }, { "t": "2024-08-
  27T12:30:45Z", "v": "230.2" } ]
  }
]
```

4.2.5. Diagram showing message types

(Updated in OCPP 2.1)

The following sequence diagram is an illustration of the use of the message type described above.

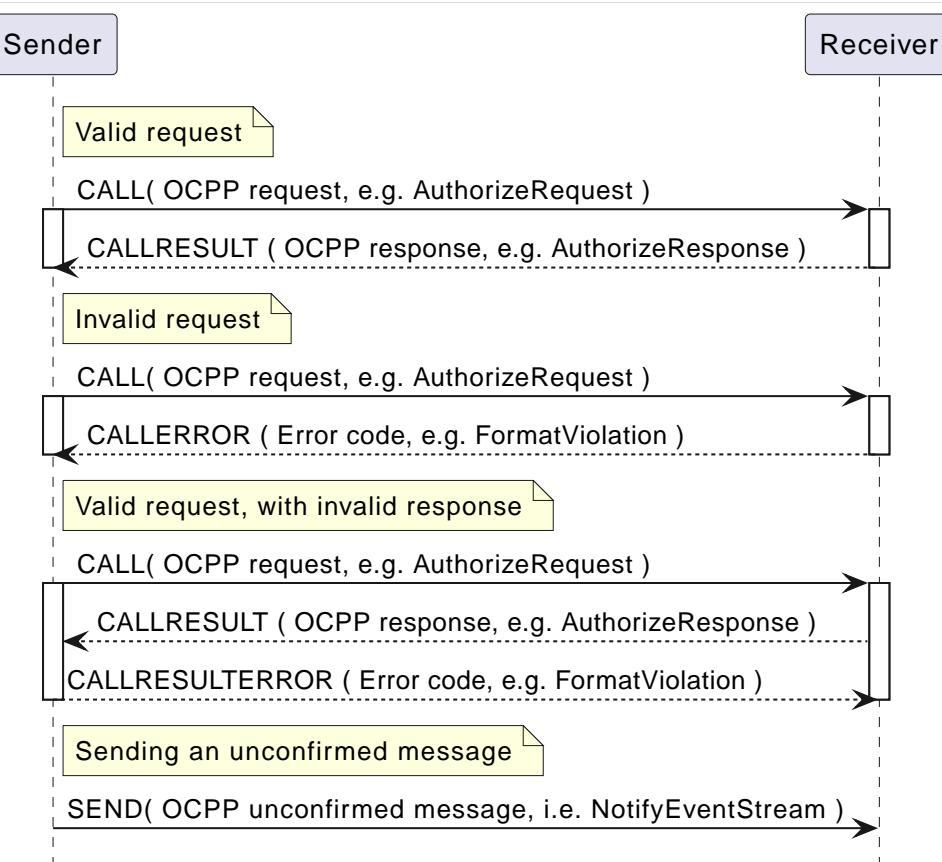


Figure 1. Sequence diagram showing message types

4.3. RPC Framework Error Codes

The following table contains all the allowed error codes for the OCPP RPC Framework.

Table 9. Valid Error Codes

Error Code	Description
FormatViolation	Payload for Action is syntactically incorrect
GenericError	Any other error not covered by the more specific error codes in this table
InternalError	An internal error occurred and the receiver was not able to process the requested Action successfully
MessageTypeNotSupported	A message with an Message Type Number received that is not supported by this implementation.
NotImplemented	Requested Action is not known by receiver
NotSupported	Requested Action is recognized but not supported by the receiver
OccurrenceConstraintViolation	Payload for Action is syntactically correct but at least one of the fields violates occurrence constraints
PropertyConstraintViolation	Payload is syntactically correct but at least one field contains an invalid value
ProtocolError	Payload for Action is not conform the PDU structure
RpcFrameworkError	Content of the call is not a valid RPC Request, for example: MessageId could not be read.
SecurityError	During the processing of Action a security issue occurred preventing receiver from completing the Action successfully
TypeConstraintViolation	Payload for Action is syntactically correct but at least one of the fields violates data type constraints (e.g. "somestring": 12)

4.4. Extension fallback mechanism

(Updated in OCPP 2.1)

Future versions of OCPP might add extra Message Types (other then [CALL](#), [CALLRESULT](#), [CALLERROR](#), [CALLRESULTERROR](#) and [SEND](#).)

When a system receives a message with a Message Type Number not in this list, it SHALL ignore the message payload. (See also paragraph [The message type](#)).

Chapter 5. Connection

5.1. Data integrity

For data integrity we rely on the underlying TCP/IP transport layer mechanisms.

5.2. TLS fragment length

TLS involves sending "Records" between peers. Records can be of type "Handshake", "Alert", "ChangeCipherSpec", "Heartbeat" or "Application". OCPP messages are sent in Application records. The payload contains a "fragment" of the application data. The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes (16kB) or less.

TLS peers need to maintain an input and an output buffer to store an entire fragment of 16 kB. For a low resource device it is a large cost to allocate 32 kB for the TLS connection.

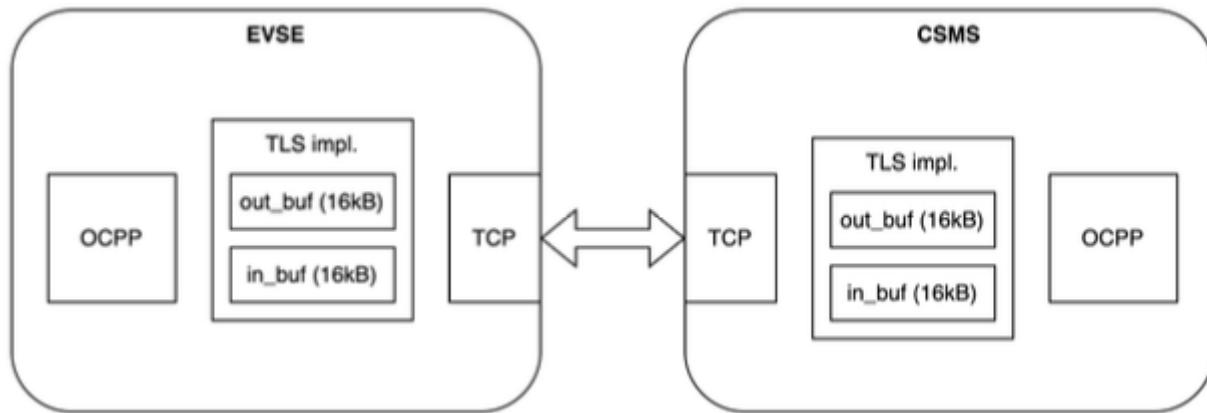


Figure 2. Peers allocating standard 16 kB TLS buffers

A TLS extension is defined in TLS Extensions RFC6066 Section 4, that allows the client to ask for a different maximum fragment length than the default 16kB. A client can ask for a maximum fragment length of 0.5 kB, 1 kB, 2 kB or 4 kB. This TLS extension is, however, not widely supported and native managed cloud TLS termination services typically don't support this.

A resource-constrained Charging Station SHOULD try to negotiate a smaller TLS maximum fragment size, and if that is not accepted by the peer, then Charging Station MAY unilaterally decide to allocate less memory to its TLS output buffer. A TLS maximum fragment length of 2 kB is suggested based on data collection during certification tests, which shows that 99% of the messages fit in a 2 kB buffer.

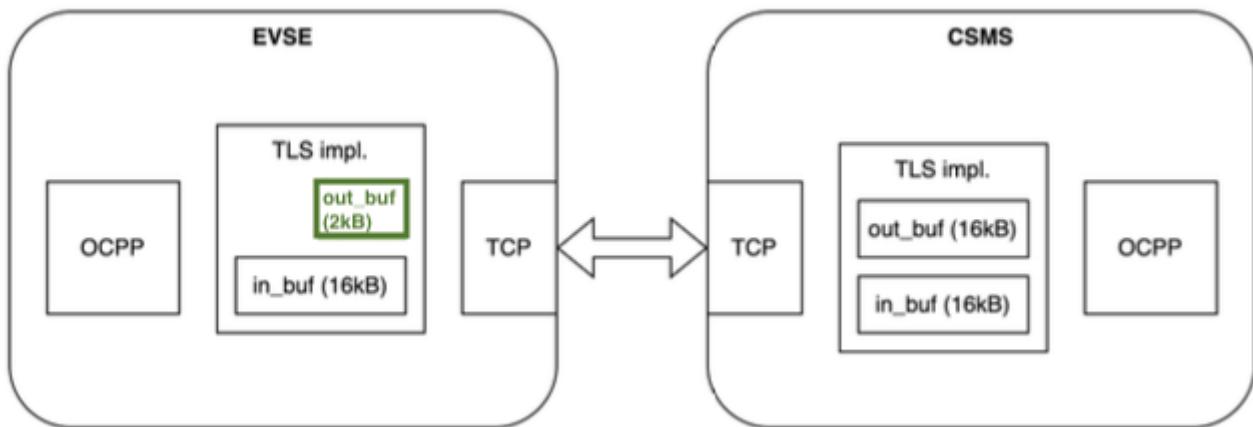


Figure 3. Charging Station allocating a 2 kB TLS output buffer

5.3. WebSocket Ping in relation to OCPP Heartbeat

The WebSocket specification defines Ping and Pong frames that are used to check if the remote endpoint is still responsive. In practice this mechanism is also used to prevent the network operator from quietly closing the underlying network connection after a certain period of inactivity. This websocket feature can be used as a substitute for most of the OCPP Heartbeat messages, but cannot replace all of its functionality. A Heartbeat message checks connectivity end-to-end, whereas a Websocket ping/pong only

checks from point-to-point. This makes a difference in an extended network topology with a Local Controller between Charging Station and CSMS.

Another important aspect of the Heartbeat response is time synchronisation. The Ping and Pong frames cannot be used for this so at least one original Heartbeat message a day is recommended to ensure a correct clock setting on the Charging Station.

5.4. Reconnecting

When the connection is lost, the Charging Station SHALL try to reconnect. When reconnecting, the Charging Station SHALL use an increasing back-off time with some randomization until it has successfully reconnected. This prevents an overload of the CSMS when all Charging Stations reconnect after a restart of the CSMS.

The first reconnection attempts SHALL be after a back-off time of: `RetryBackOffWaitMinimum` seconds, plus a random value with a maximum of `RetryBackOffRandomRange` seconds. After every failed reconnection attempt the Charging Station SHALL double the previous back-off time, with a maximum of `RetryBackOffRepeatTimes`, adding a new random value with a maximum of `RetryBackOffRandomRange` seconds to every reconnection attempt. After `RetryBackOffRepeatTimes` reconnection attempts, the Charging Station SHALL keep reconnecting with the last back-off time, not increasing it any further. After a successful connection the backoff wait timer SHALL be reset to `RetryBackOffWaitMinimum` seconds.

When reconnecting, a Charging Station SHOULD NOT send a BootNotification unless one or more of the elements in the BootNotification have changed since the last connection. For the previous SOAP based solutions this was considered good practice but when using WebSocket the server can already make the match between the identity and a communication channel at the moment the connection is established. There is no need for an additional message.

5.5. Network node hierarchy

The physical network topology is not influenced by a choice for JSON or SOAP. In case of JSON however the issues with Network Address Translation (NAT) have been resolved by letting the Charging Station open a TCP connection to the CSMS and keeping this connection open for communication initiated by the CSMS. It is therefore no longer necessary to have a smart device capable of interpreting and redirecting SOAP calls in between the CSMS and the Charging Station.

Chapter 6. OCPP Routing

For some topologies it is required to route OCPP-J messages. For example when implementing a Local Controller.

This section contains a solution for OCPP message routing that will work with any Charging Station and CSMS.

6.1. Local Controller

A Local controller is a device that sits between the CSMS and any number of Charging Stations, creating a local group. It is located near to the Charging Station (maybe even connected wired to the Charging Stations), so it does not have problem of losing the connection to the Charging Stations. This is practically useful for doing Local Smart Charging: load balancing between the Charging Stations on the same location. The Local Controller can see all the messages, ongoing transactions etc. It can send charging profiles to the Charging Station to influence the energy used by the Charging Stations, this way preventing the group to use more energy than available at the location at that time.

The Local Controller SHALL work so the Charging Station doesn't have to behave different when connected to the Local Controller, compared to a direct connection to a CSMS. A Local Controller SHALL work so that a Charging Station can work out of the box with the Local Controller, requiring only the parameters that are needed to connect to the Local Controller to be set. The Local Controller SHALL work so that the CSMS can not notice if the Charging Station is connecting to it directly, or via the Local Controller.

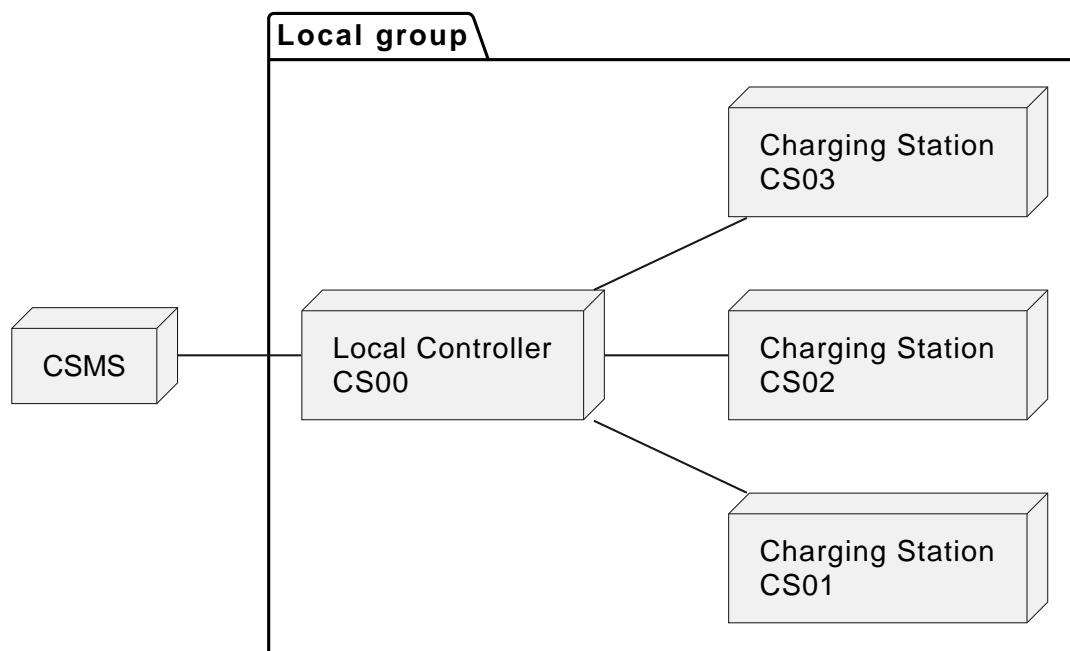


Figure 4. Local Controller Topology

6.2. Connections

For each Charging Station that connects to the Local Controller, two connections will be established:

1. A WebSocket connection from the Charging Station to the Local Controller (configured in the Charging Station)
2. A WebSocket connection from the Local Controller to the CSMS (configured in the Local Controller)

Both connections SHOULD use a similar connection URI with the same Charging Station identifier. To the CSMS, the connection from the Local Controller appears to be a regular Charging Station connection.

The Local Controller may open a separate WebSocket connection to the CSMS that allows the CSMS to address the Local Controller directly, which may be useful for changing settings or setting overall Charging Profiles.

When a Charging Station connects to the Local Controller, it SHALL connect to it like it would to a CSMS, using the same URI Path in the [connection URL](#) as it would use to connect to the CSMS. When the connection between Charging Station and the CSMS is successfully set up, the Local Controller SHALL set up a WebSocket connection to the CSMS with the same URI Path in the [connection URL](#) that was used by the Charging Station to setup the connection. The Local Controller SHALL open a WebSocket connection for every Charging Station that connects to it.

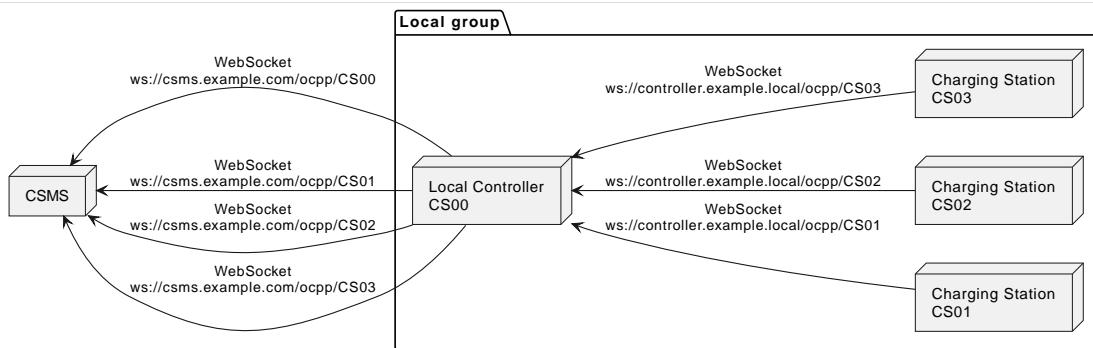


Figure 5. Local Controller WebSocket Connections

6.3. Connection loss

Whenever one or more WebSocket connections between CSMS and the Local Controller are lost, the Local Controller SHALL close all corresponding WebSockets to the Charging Stations that are connected to it, unless the Local Controller is capable of responding to Charging Station requests, and forwards transaction-related requests to the CSMS once the connection is restored. This is needed to force the Charging Station to queue messages as it would have done if it would have been connected to the CSMS directly and would have lost the connection to that CSMS.

Whenever the connection between a Charging Station and the Local Controller is lost, the Local Controller SHALL close the WebSocket connection it has for the Charging Station to the CSMS. This is needed to let the CSMS know the Charging Station is offline and no CSMS initiated messages can be sent to it.

6.4. Local Controller initiated messages

The Local Controller SHALL relay any Charging Station initiated message to the CSMS (and vice versa).

Since the Local Controller can also initiate its own messages to the Charging Station(s), a Local Controller SHALL take care of the following:

1. If a Local Controller sends its own messages to a Charging Station, it SHALL guarantee that its messages have IDs that do not collide with IDs used by the CSMS, now and in the future. This can be done by either assigning a range of numbers to the Local Controller to use (and the CSMS to skip), or by using UUIDs/GUIDs.
2. Replies to messages from the Charging Station to messages initiated by the Local Controller SHALL NOT be sent to the CSMS.

6.5. Local Controller Security

For the local controller, the normal OCPP security mechanisms will be used, as described in [\[OCPP2.1-PART2\]](#), part A. Security. All security profiles described there MAY be used when a Local Controller is deployed. The security section (part A) only describes the roles of the CSMS, and Charging Station. When a local controller is used, the security specification SHALL be interpreted as follows:

- In the connection from the Charging Station to the Local Controller, the Charging Station SHALL act as the Charging Station, and the Local Controller SHALL act as the CSMS. When TLS is used, the Local Controller SHALL be the TLS server, and the Charging Station SHALL be the TLS client.
- In the connection from the Local Controller to the CSMS, the Local Controller SHALL act as the Charging Station, and the CSMS SHALL act as the CSMS. When TLS is used, the CSMS SHALL be the TLS server, and the Local Controller SHALL be the TLS client.

When TLS with Client Side Certificates is used, the Local Controller SHALL have both a CSMS Certificate, and a Charging Station certificate (see [\[OCPP2.1-PART2\]](#) Part A - Keys used in OCPP), as it can function in both roles. These certificates SHALL be unique to the Local Controller. The Local Controller SHALL NOT store the Charging Station certificates of the attached Charging Stations. It SHALL also NOT store the CSMS Certificate of the CSMS. These certificates SHALL be kept private on their respective owners. The Local Controller SHALL only use its own certificates for setting up the TLS connections.

It SHALL be possible to distinguish the Local Controller from the CSMS based on the URL in the CSMS Certificate. Because the Local Controller is placed in the field, there is a risk that its certificates get stolen from it, e.g. by an attack on the hardware. In that case, it SHALL only be possible to use the CSMS Certificate on the Local Controller to communicate with the attached Charging Stations, not with any other Charging Stations in the infrastructure.

The TLS connections terminate on the Local Controller. So, the Local Controller can both read and manipulate data sent between

the CSMSSs and Charging Stations. If the security of the Local Controller is compromised, it will affect all attached Charging Stations. It is therefore RECOMMENDED to take sufficient security measures to protect the Local Controller. It is also RECOMMENDED to sign critical commands or replies with the mechanism described in [Signed Messages](#). In this way, it can be detected if the Local Controller tries to manipulate data.

Chapter 7. Signed Messages

For certain architectures it can be useful to use signed OCPP messages. This gives the Charging Station and the CSMS the ability to guarantee that messages are sent by the other party. For example when a Local Controller is involved, the Charging Station can know that a message received from the Local Controller is created and signed by the CSMS.

Message signing can also be used when forwarding data from the Charging Station or the CSMS to 3rd parties such as a DSO (Distribution System Operator).

Because message signing is not needed in all architectures and scenarios, it is not required for all OCPP implementations. It will depend on the security requirements if this is required.

This section defines a method to digitally sign any OCPP-J message. For each normal OCPP message an equivalent signed message is defined that encapsulates the normal message, and adds a digital signature.

7.1. Signed Message Format

For Signed OCPP Messages, JWS is used. For more information see: [\[RFC7515\]](#).

Suppose we have an OPC calls encoded in the OCPP-J message:

```
[<MessageTypeId>, "<MessageId>", {<Extension>}, "<Action>", {<Payload>}]
```

Then we define the equivalent signed message as follows.

```
[<MessageTypeId>, "<MessageId>", {<Extension>}, "<SignedAction>", {<SignedPayload>}]
```

The `MessageTypeId` and `MessageId` SHALL stay the same. The `<SignedAction>` field SHALL be the action name (`<Action>`) with the string "-Signed" appended. For instance, if "`<Action>`" is "BootNotification", then "`<SignedAction>`" is "BootNotification-Signed". The `<SignedPayload>` SHALL be the JWS encoding of the payload, computed according to the following settings:

- The JWS Payload SHALL be the `<Payload>` from the original message.
- The JWS Protected Header SHALL contain a field called "OCPPAction" containing the name (`<Action>`) of the OCPP action, and a field called "OCPPMessageTypedId" containing the message ID (`<MessageTypeId>`).
- The JWS Protected Header SHOULD contain the x5t#S256 field to identify the key used for signing, as specified in Section 7.4.
- The `<SignedPayload>` SHALL be encoded using the Flattened JWS JSON Serialization syntax.

7.2. Handling Signed Messages

When a Charging Station or CSMS receives a signed message, it SHALL extract the encapsulated normal message. It SHALL process it normally, following the OCPP 2.1 standard. It MAY perform additional actions that use the digital signature. This is optional, because a secure connection between the CSMS and the Charging Station is expected, hence a second process to validate a signature on message level is redundant. When a Charging Station receives a signed request, and it supports digital signing, it SHALL send a signed reply.

7.3. Allowed Algorithms

The algorithms allowed for use with JSON Web Signatures are defined in the JSON Web Algorithms standard [\[RFC7518\]](#). To limit the cryptographic algorithms that a Charging Station has to implement, for OCPP the same algorithms SHALL be used as for the TLS connection used to secure communications. This means that for generating the digital signatures, the Charging Station and CSMS SHALL use the following algorithms from the JSON Web Algorithms standard [\[RFC7518\]](#), section 3.1:

- ES256: ECDSA using P-256 and SHA-256
- RS256: RSASSA-PKCS1-v1_5 using SHA-256
- RS384: RSASSA-PKCS1-v1_5 using SHA-384

Note that RS256 and ES256 are the algorithms recommended by [\[RFC7518\]](#).

7.4. Key Management

This section does not prescribe specific keys to be used for digital signatures. The CSMS Certificate and Charging Station Certificate, used for setting up a secure TLS connection, MAY be used for signing. For many use cases, these will however not be the correct keys. For instance, if the use case is to provide non-repudiation of meter readings, the messages should be signed with a certificate stored in the calibrated measuring chip.

To be able to verify the digital signature, one needs to know which key was used to sign it. JSON Web Signatures supports several ways to store a key identifier with the signed message. As the certificates that can be used for signing are not specified, hash values will be used to identify them. Within the OCPP, the Charging Station and CSMS SHOULD include the field `x5t#S256` in the JWS Protected Header to identify the certificate. Following the JSON Web Signatures standard [RFC7515] the value of this field SHOULD be set to the SHA-256 hash of the DER encoding of the signing certificate. How stakeholders can look up the certificate based on the hash value is out of scope for this document.

NOTE

In the set up with a Local Controller, described in [Local Controller](#), the TLS client key that would be signing messages to the CSMS will, in fact, not be the TLS client key that the TLS connection is using, since the key in the Local Controller is different from that in the Charging Station. Similarly, the TLS server key signing messages will be that of the CSMS, not that of the local controller. Therefore, implementation of the protocol MUST NOT regard this mismatch as invalidating the signatures; in fact, it is an expected and desired property to provide end-to-end authenticity.

Chapter 8. Configuration

The following Configuration Variables are added to control JSON/WebSockets behaviour:

8.1. RetryBackOffRepeatTimes

Required	yes		
Component	componentName	OCPPCommCtrlr	
Variable	variableName	RetryBackOffRepeatTimes	
	variableAttributes	mutability	ReadWrite
	variableCharacteristics	dataType	integer
Description	When the Charging Station is reconnecting, after a connection loss, it will use this variable for the amount of times it will double the previous back-off time. When the maximum number of increments is reached, the Charging Station keeps connecting with the same back-off time.		

8.2. RetryBackOffRandomRange

Required	yes		
Component	componentName	OCPPCommCtrlr	
Variable	variableName	RetryBackOffRandomRange	
	variableAttributes	mutability	ReadWrite
	variableCharacteristics	unit	s
Description	When the Charging Station is reconnecting, after a connection loss, it will use this variable as the maximum value for the random part of the back-off time. It will add a new random value to every increasing back-off time, including the first connection attempt (with this maximum), for the amount of times it will double the previous back-off time. When the maximum number of increments is reached, the Charging Station will keep connecting with the same back-off time.		

8.3. RetryBackOffWaitMinimum

Required	yes		
Component	componentName	OCPPCommCtrlr	
Variable	variableName	RetryBackOffWaitMinimum	
	variableAttributes	mutability	ReadWrite
	variableCharacteristics	unit	s
Description	When the Charging Station is reconnecting, after a connection loss, it will use this variable as the minimum back-off time, the first time it tries to reconnect.		

8.4. WebSocketPingInterval

Required	yes		
Component	componentName	OCPPCommCtrlr	

Variable	variableName	WebSocketPingInterval	
	variableAttributes	mutability	ReadWrite
	variableCharacteristics	unit	s
Description	<p>A value of 0 disables client side websocket Ping / Pong. In this case there is either no ping / pong or the server initiates the ping and client responds with Pong.</p> <p>Positive values are interpreted as number of seconds between pings. Negative values are not allowed, SetConfiguration is then expected to return a <i>Rejected</i> result.</p> <p>It is recommended to configure WebSocketPingInterval smaller than: MessageAttemptsTransactionEvent * MessageAttemptIntervalTransactionEvent. This will limit the chance of the resend mechanism for transaction-related messages being triggered by connectivity issues.</p>		

Chapter 9. CustomData Extension

In the JSON schema files all classes have the attribute *additionalProperties* set to *false*, such that a JSON parser will not accept any other properties in the message. In order to allow for some flexibility to create non-standard extensions for experimentation purposes, every JSON class has been extended with a "customData" property. This property is of type "CustomDataType", which has only one required property: "vendorId", which is used to identify the kind of customization. However, since it does not have *additionalProperties* set to *false* it can be freely extended with new properties.

In the same way as is defined for the DataTransfer message, the "vendorId" SHOULD be a value from the reversed DNS namespace, where the top tiers of the name, when reversed, corresponds to the publicly registered primary DNS name of the Vendor organization.

The following example shows the "CustomDataType" definition and the (optional) "customData" property in the schema definition of HeartbeatRequest:

```
{
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "HeartbeatRequest",
    "definitions": {
        "CustomDataType": {
            "description": "This class does not get 'AdditionalProperties = false' in the schema generation, so it can be extended with arbitrary JSON properties to allow adding custom data.",
            "javaType": "CustomData",
            "type": "object",
            "properties": {
                "vendorId": {
                    "type": "string",
                    "maxLength": 255
                }
            },
            "required": [ "vendorId" ]
        }
    },
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "customData": {
            "$ref": "#/definitions/CustomDataType"
        }
    }
}
```

Whereas the standard HeartbeatRequest has an empty body, a customized version, that provides the value of the main meter and a count of all sessions to date, could look like this:

```
{
    "customData": {
        "vendorId": "com.mycompany.customheartbeat",
        "mainMeterValue": 12345,
        "sessionsToDate": 342
    }
}
```

A CSMS that has implemented this extension, identified by its "vendorId", will be able to process the data. Other CSMS

implementations will simply ignore these custom properties.

A CSMS can request a report of the *CustomizationCtrlr* component to get a list of all customizations that are supported by the Charging Station.