Open Charge Point Protocol JSON 1.6, OCPP-J 1.6 Specification

Table of Contents

Document Version	**1.6**
Document Status	FINAL
Document Release Date	2015-10-08

Version History

Version	Date	Author	Description
1.6	2015-10-08	Patrick Rademakers Reinier Lamers Robert de Leeuw	Updated to 1.6 Asciidoc formatting, remove JSON schema for 1.5 Some clarification Added 1.5 json schema

Contents

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 are two different flavours of OCPP; next to the SOAP based implementations, there is the possibility to use the much more compact JSON alternative. 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. As of release 1.6 this can no longer be 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.

This document describes OCPP-J, for OCPP-S see: [OCPP_IMP_S]

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

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. OCPP1.5J means we are talking about a JSON/WebSocket implementation of 1.5.
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

[JSON]	**http://www.json.org/**
[OCPP_IMP_S]	OCPP SOAP implementation specification
[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
[RFC2617]	“HTTP Authentication: Basic and Digest Access Authentication”. http://tools.ietf.org/html/rfc2617
[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
[WAMP]	http://wamp.ws/
[WIKIWS]	http://en.wikipedia.org/wiki/WebSocket
[WS]	http://www.websocket.org/

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.

3. Connection

For the connection between a Charge Point and a Central System using OCPP-J, the Central System acts as a WebSocket server and the Charge Point acts as a WebSocket client.

3.1. Client request

To set up a connection, the Charge Point 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 4.1.1 and 4.1.2.

3.1.1. The connection URL

To initiate a WebSocket connection, the Charge Point needs a URL ([RFC3986]) to connect to. This URL is henceforth called the "connection URL". This connection URL is specific to a charge point. The charge point's connection URL contains the charge point identity so that the Central System knows which charge point a WebSocket connection belongs to.

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

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

Example 1: for a charge point with identity “CP001” connecting to a Central System with OCPP-J endpoint URL "ws://centralsystem.example.com/ocpp" this would give the following connection URL:

ws://centralsystem.example.com/ocpp/CP001

Example 2: for a charge point with identity “RDAM 123” connecting to a Central System with OCPP-J endpoint URL "wss://centralsystem.example.com/ocppj" this would give the following URL:

wss://centralsystem.example.com/ocppj/RDAM%20123

3.1.2. OCPP version

The exact OCPP version MUST be specified in the Sec-Websocket-Protocol field. This SHOULD be one 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

The ones for OCPP 1.2, 1.5 and 2.0 are official WebSocket subprotocol name values. They 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.

It is considered good practice to include the OCPP version as part of the OCPP-J endpoint URL string. If you run a web service that can handle multiple protocol versions on the same OCPP-J endpoint URL this is not necessary of course.

3.1.3. Example of an opening HTTP request

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

GET /webServices/ocpp/CP3211 **HTTP/1.1**

Host: some.server.com:33033

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: ocpp1.6, ocpp1.5

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 Central System's OCPP-J endpoint URL is "ws://some.server.com:33033/webServices/ocpp". The Charge Point's unique identifier is "CP3211", so the path to request becomes "webServices/ocpp/CP3211".

With the Sec-WebSocket-Protocol header, the Charge Point indicates here that it can use OCPP1.6J and OCPP1.5J, with a preference for the former.

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. Server response

Upon receiving the Charge Point's request, the Central System has to finish the handshake with a response as described in [RFC6455].

The following OCPP-J-specific conditions apply:

- If the Central System does not recognize the charge point 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 Central System 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.

So if the Central System accepts the above example request and agrees to using OCPP 1.6J with the Charge Point, the Central System's response will look as follows:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: ocpp1.6

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

The roleof the Sec-WebSocket-Accept header is explained in [RFC6455].

The Sec-WebSocket-Protocol header indicates that the server will be using OCPP1.6J on this connection.

3.3. More information

For those doing their own implementation of the WebSocket handshake, [WS] and [WIKIWS] give more information on the WebSocket protocol.

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). 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.

4.1.1. Synchronicity

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.

NOTE	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.

4.1.2. Character encoding

The whole message consisting of wrapper and payload MUST be valid JSON encoded with the UTF-8 (see [RFC3629]) character encoding.

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. A Charge Point or Central System SHOULD only use characters not in US-ASCII for sending natural-language text. An example of such natural-language text is the text in the LocalizedText type in OCPP 2.0.

4.1.3. The message type

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

Table 2: Message types

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

When a server 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. The message ID

The message ID serves to identify a request. A message ID for a CALL message MUST be different from all message IDs previously used by the same sender for CALL messages on the same WebSocket connection. 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.

Table 3: Unique Message ID

Name	Datatype	Restrictions
messageId	string	Maximum of 36 characters, to allow for GUIDs

4.2. Message structures for different message types

NOTE You may find the charge point 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 UniqueId, 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>, "<UniqueId>", "<Action>", {<Payload>}]

Table 4: Call Fields

Field	Meaning
UniqueId	this is a unique identifier that will be used to match request and result.
Action	the name of the remote procedure or action. This will be a case-sensitive string containing the same value as the Action-field in SOAP-based messages, without the preceding slash.

Field	Meaning
Payload	Payload 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.

For example, a BootNotification request could look like this:

```
[2,
 "19223201",
 "BootNotification",
 {"chargePointVendor": "VendorX", "chargePointModel": "SingleSocketCharger"}
]
```

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 MessageTypeId and UniqueId and a payload, containing the response to the *Action* in the original Call. The syntax of a call looks like this:

[<MessageTypeId>, "<UniqueId>", {<Payload>}]

Table 5: CallResult Fields

Field	Meaning
UniqueId	This must be the exact same ID that is in the call request so that the recipient can match request and result.
Payload	Payload is a JSON object containing the results of the executed *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.

For example, a BootNotification response could look like this:

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

4.2.3. 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 missing mandatory fields, an existing call with the same unique identifier is being handled already, unique identifier too long, etc.

A CallError always consists of 5 elements: The standard elements MessageTypeId and UniqueId, an errorCode string, an errorDescription string and an errorDetails object. The syntax of a call looks like this:

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

Table 6: CallError Fields

Field	**Meaning**
UniqueId	This must be the exact same id that is in the call request so that the recipient can match request and result.
ErrorCode	This field must contain a string from the ErrorCode table below.
ErrorDescription	Should be filled in if possible, otherwise a clear empty string “”.
ErrorDetails	This JSON object describes error details in an undefined way. If there are no error details you MUST fill in an empty object *\{}*.

Table 7: Valid Error Codes

Error Code	**Description**
NotImplemented	Requested Action is not known by receiver

Error Code	Description
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
OccurenceConstraintViolation	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

5. Connection

5.1. Compression

Since JSON is very compact we recommend not to use compression in any other form than allowed as part of the WebSocket [RFC6455] specification. Otherwise it may compromise interoperability.

5.2. Data integrity

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

5.3. WebSocket Ping in relation to OCPP Heartbeat

The WebsSocket 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.

An 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 Charge Point.

5.4. Reconnecting

When reconnecting a charge point 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 WebsSocket the server can already make the match between the identity and a communciation 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 Charge Point open a TCP connection to the Central System and keeping this connection open for communication initiated by the Central System. It is therefore no longer necessary to have a smart device capable of interpreting and redirecting SOAP calls in between the Central System and the Charge Point.

6. Security

Two approaches exist for security with OCPP-J. Either one can rely on network-level security, or one uses OCPP-J over TLS. Both approaches are described below.

It is important that at all times, one of these approaches is used. Practically, this means that a Central System SHOULD NOT listen for incoming unencrypted OCPP-J connections from the internet.

6.1. Network-level security

For security one MAY rely on the security at a network level. This has historically been done with OCPP-S, and on networks that are set up appropriately one can also use OCPP-J without additional encryption or authentication measures.

6.2. OCPP-J over TLS

Sometimes however a secured network is not available between Charge Point and Central System. In that case one can use OCPP-J over TLS. This section explains how this is done.

The security needed for OCPP communication actually consists of two separate features: encryption and charge point authentication.

Encryption means that the OCPP messages are encrypted so no unauthorized third party can see the messages exchanged.

Charge point authentication means that Central System can verify the identity of a charge point, so that no unauthorized third party can pretend to be a charge point and send malicious messages to a central system.

6.2.1. Encryption

The industry standard for encryption on the internet is Transport Layer Security (TLS) [RFC5246]. Therefore OCPP is also adopting protocol for encrypting the connection between Central System and Charge Point. TLS with WebSocket is widely supported by libraries and for clients should be hardly more difficult than using unencrypted WebSocket.

When using TLS, the central system MAY also provide a signed certificate that a charge point can use to verify the central system's identity.

As some Charge Point implementations are using embedded systems with limited computing resources, we impose an additional restriction on the TLS configuration on the server side:

- The TLS certificate SHALL be an RSA certificate with a size no greater than 2048 bytes

6.2.2. Charge point authentication

For authentication, OCPP-J over TLS uses the HTTP Basic authentication scheme ([RFC2617]). The relatively simple HTTP Basic authentication can be used because the connection is already TLS-encrypted, so there is no need to encrypt the credentials a second time.

When using HTTP Basic authentication, the client, i.e. the Charge Point, has to provide a username and password with its request. The username is equal to the charge point identity, which is the identifying string of the charge point as it uses it in the OCPP-J connection URL. The password is a 20-byte key that is stored on the charge point.

Example

If we have a charge point with:

- charge point identity "AL1000"
- authorization key `000102030405060708090A0B0C0D0E0F10111213`

the HTTP authorization header should be:

```
Authorization: Basic QUwxMDAwOgABAgMEBQYH////////////////
```

A note on encryption

The authentication mechanism via HTTP Basic Authentication is meant to be used on TLS-encrypted

connections. Using this mechanism on an unencrypted connection means that anyone who can see the network traffic between Charge Point and Central System can see the charge point credentials, and can thus impersonate the Charge Point.

Setting the charge point's credentials

For this charge point authentication scheme, the charge point needs to have an authentication key. This authentication key has to be transferred onto the charge point in some way. What is a good way depends on the business model of the charge point manufacturer and central system operator.

Setting during or before installation

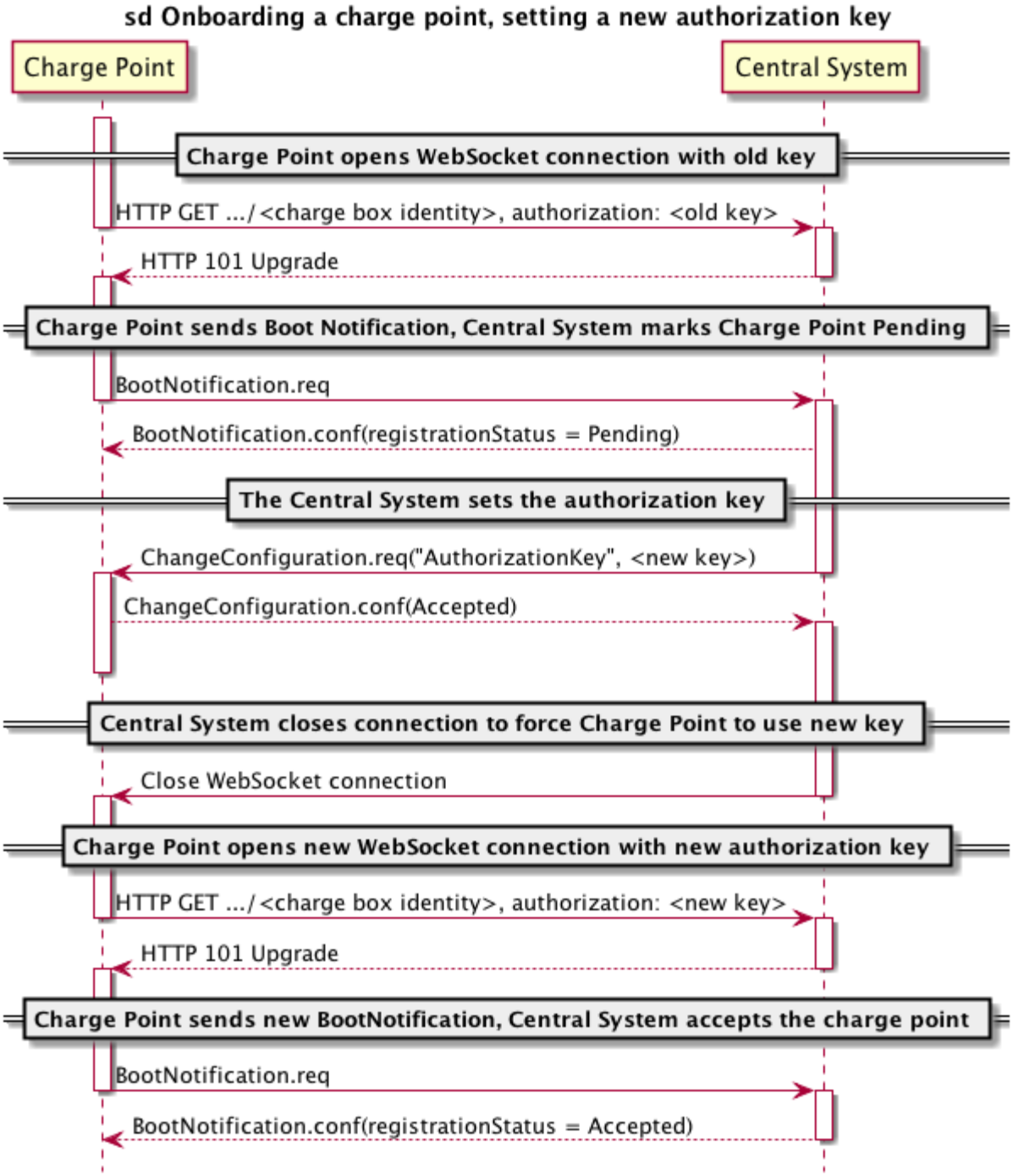
The desired, secure situation is that every charge point has its own, unique authorization key. If an authorization key is not unique, an attacker who discovers the authorization key of a single charge point can impersonate many or even all charge points in an operator's Central System.

The simplest way to achieve this is to install the authorization key on the charge point during manufacture or installation. In these cases, the key will be securely communicated between the central system operator and installer or manufacturer by communication channels outside of OCPP. This scenario is secure because the key is not sent over the channel it is meant to secure, so an attacker eavesdropping the connection between Charge Point and Central System cannot impersonate the Charge Point.

Setting the key over OCPP

If the processes of manufacturing, sale and installation of a charge point are not under the central system operator's control, there is no way to put a unique key on each individual charge point and also make sure the central system operator knows these keys and the charge points they belong to. For such scenarios, it is desirable for all charge points of a series to have the same "master" key when they leave the factory and are installed, or to have keys that are derived from the charge point identity by the same algorithm. Still the Central System operator will want to keep adversaries from impersonating all charge points of a series if the master key is leaked. For this use case, there is a possibility for the Central System to send a unique key to the charge point via OCPP after charge point installation.

To set a charge point's authorization key via OCPP, the Central System SHALL send the Charge Point a *ChangeConfiguration.req* message with the key AuthorizationKey and as the value a 40-character hexadecimal representation of the 20-byte authorization key. If the Charge Point responds to this *ChangeConfiguration.req* with a *ChangeConfiguration.conf* with status *Accepted*, the Central System SHALL assume that the authorization key change was successful, and no longer accept the credentials previously used by the charge point. If the Charge Point responds to the *ChangeConfiguration.req* with a *ChangeConfiguration.conf* with status *Rejected* or *NotSupported*, the Central System SHALL keep accepting the old credentials. While the Central System SHALL still accept an OCPP-J connection from the Charge Point in this case, it MAY treat the Charge Point's OCPP messages differently, e.g. by not accepting the Charge Point's boot notifications.



The charge point should not give back the authorization key in response to a GetConfiguration request. It can either not report the AuthorizationKey key at all or give back a value that is not related to the actual authorization key.

Note that while sending a key over the channel to be secured is normally considered a bad practice, we believe it is appropriate here to at least offer the possibility to do so. Typically the authorization key will be set when a charge point is first 'on-boarded' in the central system. If the charge point then later produces the key that was set during on-boarding, it at least means this is the same system that connected during the on-boarding. While it may be possible to successfully on-board a spoofed new charge point to an adversary who knows the single "master" key for all new charge points, it is not

possible to pretend to be an already-installed and operating charge point. This makes still makes a number of conceivable attacks impossible:

- "reservation" of a charge point by spoofing messages marking it as occupied
- marking your just-started session on a public charge point as stopped so you won't have to pay as much
- sending many spoofed transactions and/or errors from already on-boarded charge points to confuse a central system operator's operations
- send spoofed transactions with another person's token ID to the central system to incur financial damage to the token ID's owner

It is RECOMMENDED that the Central System operator makes setting the authorization key part of a charge point onboarding procedure, using the new OCPP 1.6 *Pending* value of the registration status in *BootNotification.conf*. A newly-connecting Charge Point will first get a *Pending* registration status on its first *BootNotification.conf*. The Central System will then set the Charge Point's unique authorization key with a *ChangeConfiguration.req*. Only when this *ChangeConfiguration.req* has been responded to with a *ChangeConfiguration.conf* with a status of *Accepted*, will the Central System respond to a boot notification with an *Accepted* registration status.

It is RECOMMENDED that the Central System operator checks for anomalies in the newly-connecting charge points. Thus he can try to detect if an attacker has managed to steal the master key or key derivation algorithm, and a list of registered charge point identities. For example, if the rate at which new charge points connect suddenly increases, this may indicate an attack.

Storing the credentials

It is important that the credentials are stored on the Charge Point in such a way that they are not easily lost or reset. If the credentials are lost, erased or changed unilaterally, the Charge Point can no longer connect to the Central System and requires on-site servicing to install new credentials.

On the Central System side, it is RECOMMENDED to store the authorization key hashed, with a unique salt, using a cryptographic hash algorithm designed for secure storage of passwords. This makes sure that if the database containing the charge points' authorization keys is leaked, the attackers still cannot authenticate as the charge points to the Central System.

6.2.3. What it does and does not secure

The scope of these security measures is limited to authentication and encryption of the connection between Charge Point and Central System. It does not address every current security issue in the EV Charging IT landscape.

It does provide the following things:

- authentication of the Charge Point to the Central System (using HTTP Basic Authentication)
- encryption of the connection between Charge Point and Central System

- authentication of the Central System to the Charge Point (with a TLS certificate)

It does not provide:

- A guarantee that the meter values are not tampered with between the meter and the Central System
- Authentication of the driver
- Protection against people physically tampering with a charge point

6.2.4. Applicability to OCPP-S

The approach of OCPP-J over TLS cannot be applied to OCPP-S. There are two reasons.

Firstly, in OCPP-S a new TCP connection is created for every request-response exchange. One would thus have to do a new TLS handshake for each such request-response exchange, incurring a great bandwidth overhead.

Secondly, in OCPP-S the Charge Point also acts a server, and would thus need a server certificate. It would be a great administrative burden to keep track of so many server certificates and the charge points they belong to.

7. Configuration

The following items in OCPP Get/ChangeConfiguration messages are added to control JSON/WebSockets behaviour:

Table 8: Additional OCPP Keys

Key	Value
WebSocketPingInterval	integer 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. Positive values are interpreted as number of seconds between pings. Negative values are not allowed. ChangeConfiguration is expected to return a REJECTED result.