INTERNATIONAL JOURNAL OF SCIENCE, TECHNOLOGY AND DESIGN

ULUSLARARASI
BİLİM, TEKNOLOJİ VE TASARIM DERGİSİ

Elektrikli Araç Şarj Ünitesi için OCCP ile Yazılım Uygulaması

Recep GÖZÜTOK

TOFAŞ AR&GE, BURSA, TURKEY

Corresponding Author: Recep GÖZÜTOK, recep.gozutok@tofas.com.tr

Özet

Elektrikli Araçlar tüm dünyada mobilite için yeni standart haline gelmektedir. Bu gelişme ancak şarj istasyonlarının geniş kullanım alanına sahip olması ile mümkündür.

Şarj altyapısının yaygınlaştırılmasını ilerletmek için, açık iletişim standartları kilit bir rol oynar: tüm Şarj İstasyonlarını değiştirmeden şarj ağından geçişi mümkün kılmak, yenilikçiliği ve maliyet etkinliğini teşvik etmek ve çok sayıda ve farklı oyuncunun bu yeni sektöre katılmasına izin vermek.

Ek olarak, elektrikli araç şarj altyapısı, aktörler, cihazlar ve protokollerden oluşan daha büyük ve hala gelişen bir ekosistem olan Akıllı Şebekenin bir parçasıdır. Bu Akıllı Şebeke ekosisteminde, açık iletişim standartları iki yönlü güç akışları, gerçek zamanlı bilgi alışverişi, talep kontrolü ve eMobilite hizmetleri için temel kolaylaştırıcılardır.

Açık Şarj Noktası Protokolü (OCPP), bir Şarj İstasyonu ile Şarj İstasyonu Yönetim Sistemi (CSMS) arasındaki iletişim için endüstri tarafından desteklenen fiili standarttır ve her türlü şarj tekniğini barındıracak şekilde tasarlanmıştır.

Bu yazıda, OCPP'nin sunduğu işlevlerin ve elektrikli araç şarj altyapısında nasıl kullanılabileceğinin gözden geçirilmesi amaçlanmaktadır.

Article Info

Research Article
Received:10/06/2020
Accepted:21/08/2020

Anahtar Kelimeler

WSDL - Web Hizmetleri Açıklama Dili, OCPP - Açık Şarj Noktası Protokolü, XML - Genişletilebilir İşaretleme Dili, SOAP - Basit Nesne Erişim Protokolü, OCA - Açık Şarj Birliği, HTTP - Köprü Metni Aktarım Protokolü.

Software application for electric vehicle charging unit with OCCP

Abstract

Keywords

Electric Vehicles (EVs) are becoming the new standard for mobility all over the world. This development is only possible with good coverage of Charging Stations.

To advance the roll out of charging infrastructure, open communication standards play a key role: to enable switching from charging network without necessarily replacing all the Charging Stations, to encourage innovation and cost effectiveness and to allow many and diverse players to participate in this new industry.

Additionally, the EV charging infrastructure is part of the Smart Grid, a larger and still evolving ecosystem of actors, devices and protocols. In this Smart Grid ecosystem, open communications standards are key enablers for two-way power flows, real-time information exchange, demand control and eMobility services.

The Open Charge Point Protocol (OCPP) is the industry-supported de facto standard for communication between a Charging Station and a Charging Station Management System (CSMS) and is designed to accommodate any type of charging technique. OCPP is an open standard with no cost or licensing barriers for adoption.

In this paper, providing a review of the functionalities OCPP offers and how it can be used in the electrical vehicle-charging infrastructure is aimed.

WSDL - Web Services Description Language, OCPP - Open Charge Point Protocol, XML - Extensible Markup Language, SOAP - Simple Object Access Protocol, OCA - Open Charge Alliance, HTTP - Hypertext Transfer Protocol.

1. Introduction

The Open Charge Point Protocol (OCPP) is an application protocol for communication between EV (Electric Vehicle) charging stations and a central management system, also known as a charging station network, like cell phones and cell phone networks (Pruthvi et.al 2019).

The protocol is an initiative of the E-Lad foundation in the Netherlands. It aimed to create an open application protocol which allows EV charging stations and central management systems from different vendors to communicate with each other.

It is in use by many vendors of EV charging stations and central management systems all over the world. Open Automated Demand Response (OpenADR) is an open and standardized way for electricity providers and system operators to communicate Demand Response (DR) signals with each other and with their customers using a common language over any existing IP-based communications network, such as the Internet (Hoekstra et.al. 2019).

Smart charging allows initiating and stopping the charge process with a high level of controllability. It also supports authentication mechanisms between the user or the vehicle and the charging station, as well as the exchange of contract data. This may be used for payment systems in public charging stations (Wellisch et al. 2015 and Zhao and You 2017).

Implementing smart charging in a liberalized context, calls for an interaction and corresponding information exchange between many actors: DSOs, charge stations, EVs, EV drivers, energy suppliers and possibly new market participants like CSPs and CSOs. Without standardized protocols for smart charging these information exchanges will be implemented on a project and ad-hoc basis resulting in extra costs, long implementation times and a system that is not interoperable (Ferwerda et. al 2018).

Table 1. Commonly used terms

Term	Meaning
Charging Station	The Charging Station is the physical system where an EV can be charged. A Charging Station has one or more EVSEs
Charging Station Management System (CSMS)	Charging Station Management System: manages Charging Stations and has the information for authorizing Users for using its Charging Stations.
Electric Vehicle Supply Equipment (EVSE)	An EVSE is considered as an independently operated and managed part of the Charging Station that can deliver energy to one EV at a time
CSO	Charging Station Operator
EV	Electric Vehicle
RFID	Radio-Frequency Identification

However, to my knowledge, none of the previously proposed studies examine working logic in real-time. As a result, the aim of this study is to show the real-time operation of Open Charge Point Protocol and to consider its results. And you can find the commonly used term are mentioned in Table 1.

2. Background

The Open Charge Point Protocol (OCPP) is an initiative led by the Open Charge Alliance (OCA). It is an open communication protocol that allows electric vehicle charging stations and central management software to communicate with each other. The protocol has been adopted by dozens of leading charging station providers and auto manufacturers around the world (Engelen 2019).

Open protocols are crucial for the budding EV charging market. They enable interoperability between charging stations, vehicles, and station management services. Open protocols promote innovation and collaboration, and they ensure the cost of EV charging remains competitive for business owners and EV drivers alike.

OCPP also makes it easier to create a large-scale, visible network that uses a range of different charging stations since there is a requirement for only one operating system.

Proponents of OCPP also cite a reduction in development costs since software designed to provide additional functionality would only need to be developed once and not several times to fit with each individual operating system.

Finally, OCPP will ease interoperability across the United States, and elsewhere, and minimize remedial work on systems.

SOAP (originally Simple Object Access Protocol) is a protocol specification for exchanging structured information in the implementation of web services in computer networks. Its purpose is to induce extensibility, neutrality and independence.

It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP allows processes running on disparate operating systems (such as Windows and Linux) to communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms (Box and Curbera 2004).

Element	Description	Required
Envelope	Identifies the XML document as a SOAP message.	Yes
Header	Contains header information.	No
Body	Contains call, and response information.	Yes
Fault	Provides information about errors that occurred while processing the message.	No

Figure 1: A SOAP message is an ordinary XML document containing the following elements

SOAP provides the Messaging Protocol layer of a web services protocol stack for web services. It is an XML-based protocol consisting of three parts:

- an envelope, which defines the message structure and how to process it
- a set of encoding rules for expressing instances of application-defined datatypes
- a convention for representing procedure calls and responses SOAP has three major characteristics:
- extensibility; (security and WS-Addressing are among the extensions under development)
- neutrality; (SOAP can operate over any protocol such as HTTP, SMTP, TCP, UDP, or JMS)
- independence; (SOAP allows for any programming model)

As an example of what SOAP procedures can do, an application can send a SOAP request to a server that has web services enabled such as a real-estate price database with the parameters for a search. The server then returns a SOAP response (an XML-formatted document with the resulting data), e.g., prices, location, features. Since the generated data comes in a standardized machine-parable format, the requesting application can then integrate it directly.

The SOAP architecture consists of several layers of specifications for:

- message format
- Message Exchange Patterns (MEP)
- underlying transport protocol bindings
- message processing models
- protocol extensibility

SOAP evolved as a successor of XML-RPC, though it borrows its transport and interaction neutrality from Web Service Addressing and the envelope/header/body from elsewhere (probably from WDDX) as shown Figure1.
XML Information Set was chosen as the standard message format because of its widespread use by major corporations and open source developing efforts. Typically, XML Information Set is serialized as XML. A wide variety of freely available tools significantly eases the transition to a SOAP-based implementation.

The somewhat lengthy syntax of XML can be both a benefit and a drawback. While it promotes readability for humans, facilitates error detection, and avoids interoperability problems such as byte-order, it can slow processing speed and can be cumbersome. For example, CORBA, GIOP, ICE, and DCOM use much shorter, binary message formats.

On the other hand, hardware appliances are available to accelerate processing of XML messages. Binary XML is also being explored as a means for streamlining the throughput requirements of XML. XML messages by their self-documenting nature usually have more 'overhead' (headers, footers, nested tags, delimiters) than actual data in contrast to earlier protocols where the overhead was usually a relatively small percentage of the overall message.

In this work, new SOAP type gives up to four times larger message than previous protocols FIX (Financial Information Exchange) and CDR (Common Data Representation).

The gSOAP tools provide an automated SOAP and XML data binding for C and C++ based on compiler technologies. The tools simplify the development of SOAP/XML Web services and XML application in C and C++ using auto code generation and advanced mapping methods. Most toolkits for Web services adopt a WSDL/SOAP-centric view and offer APIs that require the use of class libraries for XML-specific data structures. This forces a user to adapt the application logic to these libraries because users must write code to populate XML and extract data from XML using a vendor-specific API.

This often leads to fragile solutions with little or no assurances for data consistency, type safety, and XML validation. By contrast, gSOAP provides a type-safe and transparent solution using compiler technology that hides irrelevant WSDL-, SOAP-, REST-, and XML-specific protocol details from the user, while automatically ensuring XML validity checking, memory management, and type-safe serialization.

The gSOAP tools automatically map native and user-defined C and C++ data types to semantically equivalent XML data types and vice-versa. As a result, full SOAP/REST XML interoperability is achieved with a simple API relieving the user from the burden of WSDL/SOAP/XML details, thus enabling him or her to concentrate on the application-essential logic.

The gSOAP tools are also popular to implement XML data binding in C and C++. This means that application-native data structures can be encoded in XML automatically, without the need to write conversion code. The tools also produce XML schemas for the XML data binding, so external applications can consume the XML data based on the schemas.

The gSOAP tools support the integration of (legacy) C/C++ codes (and other programming languages when a C interface is available), embedded systems, and real-time software in SOAP/XML applications that share computational resources and information with other SOAP applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls.

3. Implementation and evaluation

3.1 Developing a web service client application

The gSOAP tools minimize application adaptation efforts for building Web Services by using a XML data binding for C and C++ implemented by advanced XML schema analyzers and source-to-source code generation tools.

The gSOAP wsdl2h tool imports one or more WSDLs and XML schemas and generates a gSOAP header file with familiar C/C++ syntax to define the Web service operations and the C/C++ data types.

The gSOAP soapcpp2 compiler then takes this header file and generates XML serializers for the data types (soapH.h and soapC.cpp), the client-side stubs (soapClient.cpp), and server-side skeletons (soapServer.cpp).

The gSOAP soapcpp2 compiler can also generate WSDL definitions for implementing a service from scratch, i.e. without defining a WSDL first. This "closes the loop" in that it enables Web services development from WSDL or directly from a set of C/C++ operations in a header file without the need for users to analyze Web service details.

3.2 Developing a Web Service Application

Start with a gSOAP header file, currentTime.h which contains the service definitions. Such a file can be obtained from a WSDL using wsdl2h when a WSDL is available. When

a WSDL is not available, you can define the service in C/C++ definitions in a newly created header file and let the gSOAP tools generate the source code and WSDL for us. Our currentTime service only has an output parameter, which is the current time defined in our currentTime.h gSOAP service specification:

It associates an XML namespace prefix ns and namespace name urn:currentTime with the service WSDL and SOAP/XML messages. The gSOAP tools use a special convention for identifier names that are part of a namespace: a namespace prefix (ns in this case) followed by a double underscore. This convention is used to resolve namespaces and to avoid name clashes. The ns namespace prefix is bound to the urn:currentTime namespace name with the //gsoap directive. The //gsoap directives are used to set the properties of the service, in this case the name, namespace, and location endpoint.

A more elegant server implementation in C++ can be obtained by using the soapcpp2 option - i (or -j) to generate C++ client-side proxy and server-side service objects as classes that you can use to build clients and services in C++. The option removes the generation of soapClient.cpp and soapServer.cpp, since these are no longer needed when that have classes that implement the client and server logic:

> soapcpp2 -i -S currentTime.h

This generates soapcurrentTimeService.h and soapcurrentTimeService.cpp files, as well as auxiliary files soapStub.h (included by default by all codes) and currentTime.nsmap and then compile with the Makefile file with "make" order. The soapcpp2 tool generated the WSDL definitions currentTime.wsdl. Therefore, it can be run the binary on the auto-generated example request XML file to test your server:

> /currentTime

WSDL can be used to advertise our service and not need to use this WSDL to develop a gSOAP client. currentTime.h file can be used with soapcpp2 option -C to generate client-side code:

> soapcpp2 -i -C currentTime.h

3.3 Developing a web service client application

3.3.1 Boot notification

After start-up, a Charge Point shall send a request to the Central System with information about its configuration (e.g. version, vendor, etc.). The Central System shall respond to indicate whether it will accept the Charge Point.

The Charge Point shall send a BootNotification.req PDU each time it boots or reboots. Between the physical power-on/reboot and the successful completion of a Boot Notification, where Central System returns Accepted or Pending, the Charge Point shall not send any other request to the Central System.

This includes cached messages that are still present in the Charge Point from before. When the Central System responds with a BootNotification.conf with a status Accepted, the Charge Point will adjust the heartbeat interval in accordance with the interval from

the response PDU and it is recommended to synchronize its internal clock with the supplied Central System's current time.

If the Central System returns something other than Accepted, the value of the interval field indicates the minimum wait time before sending a next Boot Notification request. If that interval value is zero, the Charge Point chooses a waiting interval on its own, in a way that avoids flooding the Central System with requests. awaiting interval on its own, in a way that avoids flooding the Central System with requests as shown Figure 2.

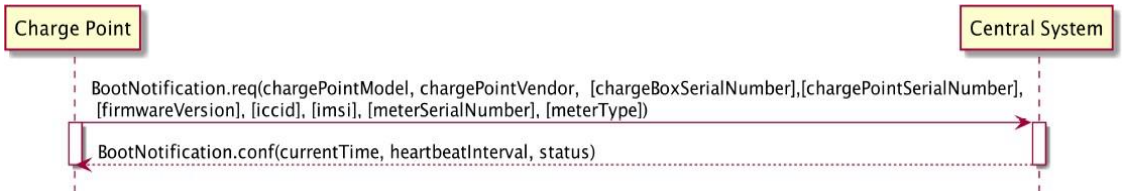

Figure 2: Sequence diagram of Boot Notification

3.3.2 Heartbeat

To let the Central System, know that a Charge Point is still connected, a Charge Point sends a heartbeat after a configurable time interval. The Charge Point shall send a Heartbeat.req PDU for ensuring that the Central System knows that a Charge Point is still alive.

Upon receipt of a Heartbeat.req PDU, the Central System shall respond with a Heartbeat.conf. The response PDU shall contain the current time of the Central System, which is recommended to be used by the Charge Point to synchronize its internal clock as referenced in Figure 3.

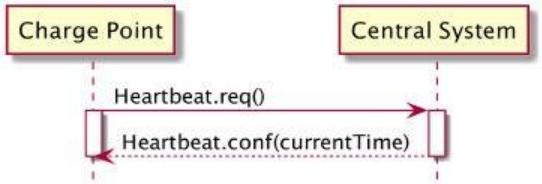

Figure 3: Sequence diagram of Heartbeat

The Charge Point may skip sending a Heartbeat.req PDU when another PDU has been sent to the Central System within the configured heartbeat interval. This implies that a Central System should assume availability of a Charge Point whenever a PDU has been received, the same way as it would have, when it received a Heartbeat.req PDU.

3.3.3 Authorize

Before the owner of an electric vehicle can start or stop charging, the Charge Point must authorize the operation. The Charge Point shall only supply energy after authorization.

When stopping a Transaction, the Charge Point shall only send an Authorize.req when the identifier used for stopping the transaction is different from the identifier that started the transaction.

Authorize.req should only be used for the authorization of an identifier for charging. A Charge Point may authorize identifier locally without involving the Central System, as

described in Local Authorization List. If an idTag presented by the user is not present in the Local Authorization List or Authorization Cache, then the Charge Point shall send an Authorize.req PDU to the Central System to request authorization.

If the idTag is present in the Local Authorization List or Authorization Cache, then the Charge Point may send an Authorize.req PDU to the Central System as shown in Figure 4.

Figure 4: Sequence diagram of Authorize

3.3.4 Thread

Multithreading is a specialized form of multitasking and multitasking is a feature that allows your computer to run two or more programs concurrently. There are two types of multitasking: process-based and thread based. Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

```
1  // thread example
2  #include <iostream>       // std::cout
3  #include <thread>         // std::thread
4
5  void foo()
6  {
7    // do stuff...
8  }
9
10 void bar(int x)
11 {
12   // do stuff...
13 }
14
15 int main()
16 {
17   std::thread first (foo);     // spawn new thread that calls foo()
18   std::thread second (bar,0);  // spawn new thread that calls bar(0)
19
20   std::cout << "main, foo and bar now execute concurrently...\n";
21
22   // synchronize threads:
23   first.join();                // pauses until first finishes
24   second.join();               // pauses until second finishes
25
26   std::cout << "foo and bar completed.\n";
27
28   return 0;
29 }
```

Figure 5: An example of the Thread

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature. Class to represent individual threads of execution.

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

An initialized thread object represents an active thread of execution; Such a thread object is joinable and has a unique thread id. A default-constructed (non-initialized) thread object is not joinable, and its thread id is common for all non-joinable threads. A joinable thread becomes not joinable if moved from, or if either join or detach are called on them.

Using the thread system in the software application of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

3.3.5 Reserve Now

A Central System can issue a ReserveNow.req to a Charge Point to reserve connector for use by a specific idTag. To request a reservation the Central System shall send a ReserveNow.req PDU to a Charge Point. The Central System may specify a connector to be reserved. Upon receipt of a ReserveNow.req PDU, the Charge Point shall respond with a ReserveNow.conf PDU.

If the reservationId in the request matches a reservation in the Charge Point, then the Charge Point shall replace that reservation with the new reservation in the request. If the reservationId does not match any reservation in the Charge Point, then the Charge Point shall return the status value ‘Accepted’ if it succeeds in reserving a connector as shown in Figure 6.

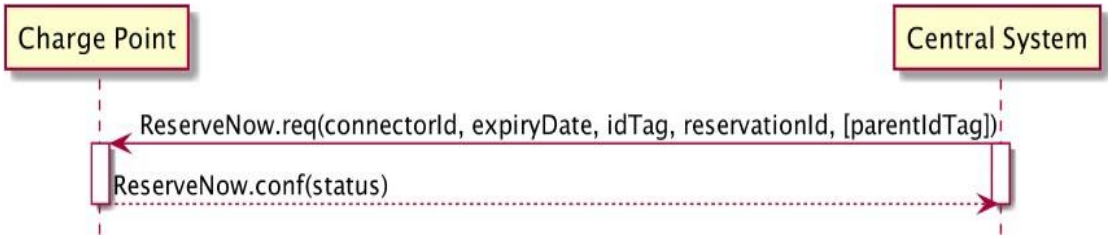

Figure 6: Sequence diagram of the Reserve Now

The Charge Point shall return ‘Occupied’ if the Charge Point or the specified connector are occupied. The Charge Point shall also return ‘Occupied’ when the Charge Point or connector has been reserved for the same or another idTag.

The Charge Point shall return ‘Faulted’ if the Charge Point or the connector are in the Faulted state. The Charge Point shall return ‘Unavailable’ if the Charge Point or connector are in the Unavailable state. The Charge Point shall return ‘Rejected’ if it is configured not to accept reservations.

If want to generate the ReserveNow.res in our Client Point, firstly we must structure the .h file of the Reserve Now operation. In addition, that if we want to structure the .h file we must know the field definitions of the ReserveNow.res and Reserve.req.

4. Conclusion

This paper reviews the Open Charge Point Protocol (OCPP), which can be adopted as a standard for the back-end communication between electric vehicle charging stations and the central management system. The various versions of the protocol were discussed along with the improvements in its functionalities and
features. Sample user interfaces of the possible implementation of the OCPP protocol were also shown for the central management system and the charging
station, both.

The OCPP protocol describes a large number of use cases and messages, which are not all needed to implement a basic Charging Station or CSMS. This software is written in C ++ code using the Gsoap library, based on XML, using the SOAP procedure according to the OCP procedure.

At the end of the study, the software is ready to run without error. Other functions included in the OCP procedure can be added to this software with the acquisition of hardware. The Table 1 below lists messages that are typically implemented to deliver basic functionality for an OCPP managed Charging Station. There are OCPP functionalities in Table 2.

Table 2. OCPP functionalities

Functionality	Messages
Booting a Charging Station	Boot Notification
Configuring a Charging Station	Set Variables, Get Variables and Get Report Base (respond correctly to requests with report Base = Configuration Inventory, Full Inventory, Summary Inventory).
Resetting a Charging Station	Reset
Authorization Options	Authorize
Transaction Mechanism	Transaction Event
Availability	Only Change Availability and Status Notification.
Monitoring Events	A basic implementation of the Notify Event message to be used to report operational state changes and problem/error conditions of the Charging Station, e.g. for Lock Failure. Also used for reporting built-in monitoring events.

Sending Transaction Related Meter Values	Transaction Event
Data Transfer	Any OCPP implementations should at least be able to reject any request for Data Transfer if no (special) functionality is implemented.
Resetting a Charging Station	Reset
Authorization Options	Authorize
Transaction Mechanism	Transaction Event

This software is written according to the OCP procedure to provide the commonization of the electric vehicle charging points. With this integration, all charging points can be controlled from a center.

5. References

Box, D., Curbera, F., 2004. Web Services Addressing (WS-Addressing). W3C Member Submission, https://www.w3.org/Submission/ws-addressing/

Engelen, R.V., 2019. gSOAP user guide. https://www.genivia.com/doc/guide/html/index.html

Ferwerda, R., Bayings, M., Kam, M.V, Bekkers, R. 2018. Advancing E-Roaming in Europe: Towards a Single "Language" for the European Charging Infrastructure, World Electr. Veh. J.

Hoekstra, A, Bienert, R., Wargers, A., Singh, H., Voskuilen, P., 2019. Using OpenADR with OCPP. https://openadr.memberclicks.net

Pruthvi, T.V., Dutta, N., Bobba, P.B., Sasudeva, S. 2019. Implementation of OCPP Protocol for Electric Vehicle. E3S Web of Conferences.

Wellisch, D., Lenz, J., Faschingbauer, A., Pöschl, R., Kunze, S. 2015. An Approach for Smart Charging Development. IFAC-PapersOnLine.

Zhao, C., You, X. 2017. Research and Implementation of OCPP 1.6 Protocol. 2nd International Conference on Machinery Electronics and Control Simulation.