OSMP Reference Implementation

Documentation of Open System Management Protocol Reference Implementation in C#

Protocol Version 1.0
Document Version: 2017.09.05
Author: Meinrad Recheis, Eqqon GmbH

Introduction

Open System Management Protocol is an open protocol developed by Eqqon in an effort to establish an open protocol that will allow better interoperability between products in (but not necessary limited to) the Public Address domain. The protocol is text based and uses the open standard JSON for data exchange over a secure WebSocket transport.

Legal Notice

Table of Contents

OsmpClient Usage

Getting started

To connect to an OSMP server, reference the OsmpClient.dll assembly from the Reference Implementation. Furthermore you'll need the nuget packages Json.DotNet and websocket-sharp.

Here is a simple example that will connect to the server, send a command and evaluate the response data:

```
var client = new OsmpClient() { Address = "wss://localhost:443/osmp/v1" };
await client.Connect();
var response=await client.SendCommand( "echo",
        new JObject {{"token", "Hello World!"}});
Debug.Assert( response != null);
Debug.Assert( response.Status == "OK");
Debug.Assert( response.Data.Value<string>("token") == "Hello World!");
```

The echo command just returns the token you pass to it unchanged.

Command Factory Methods

Even easier than crafting the message object yourself, is calling the existing factory methods to create the command messages:

```
var msg = Standard.Echo("What was was and never will be - Plato" ));
var response = await client.Send(msg);
```

The Standard.Echo factory method creates a message with command id "echo" and the data object containing the token.

Here is another simple example:

```
var response = await client.Send( Standard.Time());
```

See the Command Reference for a list of all factory methods.

Response Data Types

Now we want something similar for accessing the returned data from the response. The **DataAs<T>()** function takes the returned JSON object and tries to align it to the given type T. For every command that returns data, there should be such a datatype with the naming

convention <InstructionSet>.<Cmd>Result. In our example case the correct type is **Standard.EchoResult**.

```
Debug.Assert( response.DataAs<Standard.EchoResult>().Token ==
        "What was was and never will be - Plato");
```

Using DataAs<T> you can easily cast the response data into a specialized data type (here Standard.EchoData) that provides easy access to the data. Otherwise you'll have to access the fields of the response data as a JObject.

Another example for the Standard.Time command:

```
var server_time = response.DataAs<Standard.TimeResult>().DateTime;
```

Cancelling Commands

Normally the response to a command is delivered immediately. Some commands however can take longer to process or are designed to run indefinitely until cancelled, like the wait command.

Here is how do cancel a command by using a cancellation token:

```
var cancellation_source = new CancellationTokenSource();
var task = client.Send( Standard.Wait( cancellation_source.Token));
```

Obviously we don't want to await the response here, because it will never come if we don't cancel wait. Now we cancel the command:

```
cancellation_source.Cancel();
```

After cancellation, the server will finally send the result which we await here:

```
var response = await task;
Debug.Assert( response.IsOk);
```

If you don't care about selectively cancelling commands (i.e. because you don't have any commands running in the background which should not be cancelled there is also the possibility to cancel all active commands:

```
client.CancelAll();
```

Authentication

To authenticate use the methods OsmpClient.Login(...), OsmpClient.LoginWithPublicKey(...), and OsmpClient.Logout(). The authentication methods are designed to be highly secure

even against eavesdroppers who managed to breach the SSL encryption of the websocket connection (see the OSMP spec for details).

Here is an example of password authentication:

```
var client = new OsmpClient() { Address = "wss://localhost:443/osmp/v1" };
await client.Connect();
var response = await client.Login("JonSnow", "LordCommanderOfTheWatch");
Debug.Assert( response.Status ==  "OK");
Debug.Assert( response.Result == "Login successful!");
```

The second way of authentication is public-key authentication. The username and public key have to be configured in the server for this to work.

This code creates an RSA key pair of 1024 bit:

```
using (var rsa = new RSACryptoServiceProvider(1024))
{
        var public_key=rsa.ToXmlString( false);
        var private_key_pair=rsa.ToXmlString( true);
}
```

Note: Keep the private key pair secret and protect it against unauthorized access! The public key is used to configure user accounts on the server which can login without a password.

Here is an example of a public key:

<RSAKeyValue><Modulus>v+HYa8fTYEkodX/kqMAdaSlvuYj49i9I0kBDJh9vs7qlg0HYjUeOCRqG1FqzuAI1h22YjbSOo2DtMKMInnuDMakyQjG29Qh3uX0kl64zMtGEC7+bluM/VG4xYzUlXNs5G0AoJAaI1wMTsD9aPSaZ1llzlGjrZqqXZhI74ktUIV0=</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>

To load the secret private key into the OsmpClient use the method OsmpClient.LoadKeypair(string keypair). The client now has both the public and the private key and can use them to authenticate itself with the server.

Here is an example of public-key authentication with the keypair we generated earlier.

```
var client = new OsmpClient() { Address = "wss://localhost:443/osmp/v1" };
client.LoadKeypair( private_key_pair);
await client.Connect();
var response = await client.LoginWithPublicKey("JonSnow");
Debug.Assert( response.Status ==  "OK");
Debug.Assert( response.Result == "Login successful!");
```

Advanced Client Side Topics

If you do not intend to write your own factory methods or create your own types to deserialize JSON responses you can skip the advanced topics.

How to write a Cmd Factory Method

The static class Standard in OsmpClient.dll contains factory methods to easily create command messages without having to know their structure.

As an example here is how Standard.Echo is defined:

```
public static OsmpMessage Echo(object token = null)
{
        return new OsmpMessage() { Id = "echo", Type = "cmd", Data =
                JObject.FromObject(new EchoData { Token = token }) };
}
```

And here is how Standard.EchoData is defined:

```
[JsonObject(MemberSerialization.OptIn)]
public class EchoData
{
        [JsonProperty("token")]
        public object Token { get; set; }
}
```

As you can see, both definitions are pretty simple. But for more complicated commands such factory methods and data types make a lot of sense. They help reduce coding errors by allowing the consumer of the API to utilize Intellisense.

How to Map To Response Data Types

The static class Standard in OsmpClient.dll contains command result data types which can be used to easily access command results without having to know their structure.

Here is an example of accessing result data with response.DataAs<T>:

```
        // start a Standard.Wait command
        client.Send(Standard.Wait());
// get the active commands
        var response = await client.Send(Standard.ActiveCmds());
```

The response.Data contains the result data as JObject with the following content:

```
{
        "now": "2017-09-05T09:36:19.8793121+02:00",
        "cmds": [
                {
                        "name": "wait",
                        "cmd-nr": 1,
                        "start-time": "2017-09-05T09:35:36.2418162+02:00"
                }
        ]
 }
```

Now, instead of working with the raw JSON you can just use DataAs<T> to map it to the correct data model and access it through that:

```
var active_cmds=response.DataAs<Standard.ActiveCmdsResult>();
```

This utilizes Json.Net's powerful deserialization capabilities to align JSON to business objects. Here is how Standard.ActiveCmdsResult is defined:

```
[JsonObject(MemberSerialization.OptIn)]
public class ActiveCmdsResult
{
        [JsonProperty("now")]
        public DateTime Now { get; set; }

        [JsonProperty("cmds")]
        public IEnumerable<ActiveCommand> Commands { get; set; }
}

[JsonObject(MemberSerialization.OptIn)]
public class ActiveCommand
{
        [JsonProperty("name")]
        public string Name { get; set; }

        [JsonProperty("cmd-nr")]
        public int CmdNr { get; set; }

        [JsonProperty("start-time")]
        public DateTime StartTime { get; set; }
}
```

You'll see that the whole result data from our response above, namely a list of active commands, can be accessed through the data model:

```
Debug.Assert( active_cmds.Commands.Count() == 1);
Debug.Assert( active_cmds.Commands.First().Name == "wait");
```

Class Reference

Class OsmpClient

Public methods:

Public Method	Description
Cancel (OsmpMessage msg)	Cancel a long running command msg: The originally sent command to be cancelled
Cancel (int[] cmds)	Cancel the commands with given cmnd-nrs cmds: Array of command numbers
CancelAll ( )	Cancel all active commands
Connect ( ) returns: async Task<bool>	Connect to the OsmpServer through the configured Address and waits for the connection result
Disconnect ( )	Close the connection to the server
Dispose ( )	Destroy the client and release all resources
LoadKeypair (string key_pair_xml)	Load the given private and public key pair for authentication key_pair_xml: The public/private key pair
Login (string username, string plaintext_password) returns: async Task<OsmpResponse>	Login with username and plaintext password
LoginWithPublicKey (string username) returns: async Task<OsmpResponse>	Login with username and public-key
Logout ( ) returns: async Task<OsmpResponse>	Logout the currently authorized user
Send (OsmpMessage msg)	Send the given message to the server and wait for

returns: async Task<OsmpResponse>	the response msg: The message can be a command, response, event, stream or cancel message
SendCommand (string command, JObject data = null) returns: async Task<OsmpResponse>	Send command with optional parameters to server and wait for the response command: The name of the command data: The command parameters
SendText (string text)	Send the given text to the server. This is fire and forget, there is no way to wait for a response. Note: this method is used for testing the server's reaction to ill formed messages. text: the text message to send

Public properties:

Property	Description
Address { get; set; } = "ws://localhost:443/osmp/v1"; string	The server address that will be used by Connect
CertValidationCallback { get; set; } = (o, cert, chain, ssl_errors) => true; RemoteCertificateValidationCallback	Set this callback to validate the server certificate. If not set all certificates (even invalid ones) are accepted.
IsConnected { get; } bool	Connection status
LastReceived { get; } DateTime	Last time the client received a message from the server
PublicKey { get; } string	The client's public key. To change it use LoadKeypair

Events:

Event	Description
CmdResponseReceived	Raised on receiving a response

Action<OsmpResponse>	
CmdStreamReceived Action<OsmpStream>	Raised on receiving a stream package
ConnectionStateChanged Action	Raised when the connection to the server changes (call IsConnected to get current status)
Error Action<string, Exception>	Raised whenever an exception or other error occurs string: Error message Exception: Exception object (can be null)
EventsReceived Action<OsmpEvent[]>	Raised whenever events were received
MessageReceived Action<string>	Raised whenever the socket receives something. string: The raw message
MessageSent Action<string, bool>	Raised whenever the socket receives something. string: The raw message bool: True if successfully sended

Class OsmpMessage

The OsmpMessage is the base class of all messages that are sent back and forth between the client and the server.

Public methods:

Public Method	Description
SetData (object data)	Serialize the given object into the message data. This is the opposite of DataAs<T>. See chapter “How to Map To Response Data Types” for info about writing such data types. data: The object representing the message data.
DataAs<T> ( ) T	Deserialize the message data into a new object of given type T This is the opposite of SetData

Public properties:

Property	Description
Type { get; set; } string	The message type (cmd | response | stream | event | cancel)
Nr { get; set; } int	Incremental message number. Set by client on send automatically
Id { get; set; } string	Command/Event name
Data { get; set; } JObject	The message payload
TaskSource { get; set; } TaskCompletionSource OsmpResponse>	This is used to wait for the response. You can await the response like this: var response = await msg.TaskSource.Task;
CancellationToken { get; set; } CancellationToken	CancellationToken is used to react to cancellation of the command.

Class OsmpResponse : OsmpMessage

OsmpResponse represents a response that is sent to the client from the server as the result of a command. The original command is referenced. The result payload is stored in the inherited Data property.

Public properties:

Property	Description
CmdNr { get; set; } string	The message number of the command that caused this response
Status { get; set; } string	The response status (OK | ERROR | TIMEOUT)

Result { get; set; } string	Result message of this response or error message if not OK
Command { get; set; } OsmpMessage	The original command message that led to this response
IsOk { get; } bool	True if Status == “OK”
CancellationToken { get; set; } CancellationToken	CancellationToken is used to react to cancellation of the command.

Class OsmpStream : OsmpMessage

OsmpStream is a stream package that is sent from the server to the client. It references the command that started the stream. The payload is stored in the inherited Data field.

Public properties:

Property	Description
CmdNr { get; set; } string	The message number of the command that started this stream
Command { get; set; } OsmpMessage	The original command message that started this stream

Class OsmpEvent

OsmpEvent represents an event (i.e. status change) that is sent from the server to the client.

Public properties:

Property	Description
Id { get; set; } string	The event name

Data { get; set; } JObject	The event payload

Command Reference

Every factory method returns an OsmpMessage object that can be passed to the OsmpClient.Send(OsmpMessage msg) method to execute the command. For every command the result type is listed if the command has relevant result data. The result data types are to be used as type parameters for OsmpResponse.DataAs<T>().

Standard Instruction Set

The factory methods of the standard instruction set are all public static methods of public static class Standard in namespace Osmp. For every command the result type is listed if the command has result data. For documentation of the commands and their parameters see the OSMP documentation.

- ActiveCmds()
    - ActiveCmdsResult
- Echo(object token = null)
    - EchoResult
- EventList()
    - EventListResult
- EventSubscribe(string event_name, DateTime? timeout=null)
- EventSubscribeAll(DateTime? timeout = null)
- EventUnsubscribe(string event_name)
- EventUnsubscribeAll()
- Login(string username, string password_hash)
- Logout()
- Time()
    - TimeResult
- Wait(CancellationToken? token=null)
- Wait(double seconds, CancellationToken? token=null)

Vd1 Instruction Set

The factory methods of the Vd1 instruction set are all public static methods of public static class Vd1 in namespace Osmp. For documentation of the commands and their parameters see the OSMP specification document.

- VCallCreate(string call_id, string[] sources, string[] zones, int priority = 41, string owner = null, bool start = false)
    - VCallCreateResult

Note: The VCallEdit functions actually use the command vcall-create to change certain parameters of an existing call. This means, you'll get a VCallCreateResult.

- VCallEdit(string call_id, string[] sources=null, string[] zones=null)
- VCallEdit(string call_id, int priority)
- VCallEdit(string call_id, string owner)
    - VCallCreateResult
- VCallPlay(string call_id)
- VCallStop(string call_id)
- VCallStopAll()
- VCallList(string filter=null)
    - VCallListResult
- VCallStatus(string call_id)
    - VCallStatusResult
- VDeviceStatus(string device_id, bool ignore_not_installed=true, string include_filter = null, string exclude_filter=null)
- VDeviceStatusAll(bool ignore_not_installed = true, string include_filter = null, string exclude_filter = null)
    - VDeviceStatusResult
- VDeviceStatusFilter()
- VDeviceStatusFilter(bool ignore_not_installed, string include_filter, string exclude_filter)