Generic VRPC connector

by Mariano Simone

Prerequisites

- Be familiar with the VRPC project
- Medium Javascript knowledge
- Be familiar with npm
- Be familiar with Docker

Conventions

From now on I'll reference the protocol we are building as *newprotocol*. You should pick your own name.

Introduction

This article will guide you on how to build your own protocol connector using the *VRPC generic protocol*.
You will learn how to build an agent for your protocol and how to integrate it on a Connectware Service Commissioning File.
You will also learn how to package your project as a docker image for easy distribution.
For better understanding of all the topics discussed here you should take a look at generic VRPC protocol.

What is the VRPC generic protocol?

With the help of the VRPC project we can extend the capabilities of the Connectware by allowing users to implement their own protocols.
We do this by using a special protocol (genericVrpc) that will wrap around your own implementation.
All the communication between the Connectware and the new protocol will be managed by VRPC.

Creating the project

Before we begin laying out the code for our protocol, we need to create a repository for it.

So, go ahead and run the following commands:

```
mkdir newprotocol-connector
cd newprotocol-connector
git init .
echo node_modules > .gitignore
```

Alright, almost done! Now, we need to create a npm project so we can manage our project dependencies.
We can do this with the following command:
`npm init -y`

Installing libraries

Now we have our project base ready, lets install some basic libraries that we need to make it all work:

```
npm install config  # Node module to handle project configurations
npm install js-yaml # Allow loading yaml based config files
npm install vrpc    # The VRPC library
```

Additionally, you should install all the libraries you need to make your project work.

Configurations

Now, it is time to add the configuration files to our project.
So, go ahead and create the *config* directory in the project root by running:
`mkdir config`

Then create the following files inside:

default.yaml:

```
mqtt:
  scheme: mqtt
  host: broker
  port: 1883
vrpc:
  domain: 'newprotocol.vrpc'
  agent: 'newprotocol.connector'
```

NOTE
You should configure *vrpc.domain* and *vrpc.agent* properties according to your protocol.

custom-environment-variables.yaml:

```
mqtt:
  scheme: MQTT_SCHEME
  host: MQTT_HOST
  port: MQTT_PORT
 username: MQTT_USER
  password: MQTT_PASSWORD
vrpc:
  domain: VRPC_DOMAIN
  agent: VRPC_AGENT
```

Cool, configuration is done!
Now you can configure your project with additional variables you may need.

Let's add some code!

We can now proceed to add a class that will implement our protocol.
First, create the *src* directory:
`mkdir src`

Then, add the following class template inside:

NewProtocol.js:

```
const { EventEmitter } = require('events')

class NewProtocol extends EventEmitter {
  constructor (params) {
    super()

    const defaultOptions = {
        // Your protocol default options.
    }

    // Merge default options with the ones specified on the commissioning file.
    this._options = Object.assign(defaultOptions, params.options)
  }

  async connect () {
    // Connect to your protocol here

    // Emit 'connected' when you are done
    this.emit('connected')
 }

  async disconnect () {
    // Disconnect from your protocol here

    // Emit 'disconnected' when you are done
    this.emit('disconnected')
  }

  async subscribe (address, id) {
    // Subscribe to your protocol here

    // Emit an event when there is data on this subscription
    this.emit(id, { value: 'some data' })
  }

  async unsubscribe (id) {
    // Unsubscribe to your protocol here
  }

  async read (address) {
    // Implement the read operation of your protocol here

    // then return it
    return 'read data'
  }

  async write (address, data) {
    // Implement the write operation of your protocol here
  }
}

// Export the class
module.exports = NewProtocol
```

Now, you need to implement all the protocol main functions (connect, disconnect, write, read, subscribe, unsubscribe).

Once we have our implementation ready, we need to register the class on a VRPC agent and serve it to the Connectware.
To do that, create the following file in the project root, with this content:

index.js:

```
'use strict'
const { VrpcAdapter, VrpcAgent } = require('vrpc')
const config = require('config')

// Register our new protocol class
VrpcAdapter.register(require('./src/NewProtocol'))

// Create a VRPC agent
const agent = new VrpcAgent({
  domain: config.get('vrpc.domain'),
  agent: config.get('vrpc.agent'),
  username: config.get('mqtt.username'),
  password: config.get('mqtt.password'),
  broker: '${config.get('mqtt.scheme')}://${config.get(
    'mqtt.host')}:${config.get('mqtt.port')}'
})

// Serve all registered classes back to the Connectware
agent.serve()
```

Dockerfile

To be able to run this project we need to package it on a docker image.
So, lets create a Dockerfile in the project root with the following content:

```
FROM node:12.18.2-alpine@sha256:b48d5259d91e549e4941d5170870619d2e9c27de648e
6230625752481232a005

WORKDIR /app
COPY . /app

RUN npm install
```

```
ENTRYPOINT ["node", "."]
```

Building the image

To build the image run the following command in the project root:
`docker build -t newprotocol-connector:0.1 .`

Running the image

To run the image run the following command in the project root:

```
docker run -v /tmp/data:/data -d --name newprotocol-connector --rm -e MQTT_
USER=admin -e MQTT_PASSWORD=admin newprotocol-connector:0.1
```

Working example

You can find a working connector for the Atlas Copco OpenProtocol here.