DoIP-Software-Documentation

March 12, 2020

Contents

1 Introduction

This document describes the open source software components for DoIP (Diagnostics over IP) which are all hosted in Github.

They are all located at `https://github.com/doip`.

The implementation is distributed over several repositories.

`doip`: This is the general repository which contains overall content for the whole project. For example this document is located in this repository. All issues for the subprojects are also managed here.

`doip-library`: Contains source files for DoIP communication which can be used in a DoIP simulation as well as in an diagnostic tester for DoIP.

`doip-simulation`: Contains source files which are specific for a DoIP simulation. The project does not contain an executable. It is still a library which can be used in a custom DoIP simulation

`doip-custom-simulation`: This project finally produces an executable to start an DoIP simulation. It is supposed to create a copy of the source files and start your own implementation.

`doip-tester`: This project contains mainly unit tests for testing a real ECU or the simulation. The unit tests expect that somewhere a DoIP ECU or a simulation is running. The unit tests can be executed with gradle.

`doip-logging`: Contains just a wrapper around log4j2. The intention was to have the possibility to create a different impementation than log4j. It is also easier to have only one central place which refers to log4j2, so updates to a newer version are easier to manage.

`doip-junit`: It's a wrapper class for the class Assert in junit. The problem in junit is that asserts will be written to stderr. That makes it difficult to read logfiles created by the build system Gradle. The wrapper class here just logs a failed assert also to stdout, so they appear in a logfile at the right position.

2 Getting Started

2.1 Build and Install Example Simulation

An example to run the DoIP simulation is available at https://github.com/doip/doip-custom-simulation.git. To run the DoIP simulation follow the following steps:

- Clone the project `doip-custom-simulation`:

```
git clone https://github.com/doip/doip-custom-simulation.git
```

- Now the source files of this project are located in the folder `doip-custom-simulation`.
- Later there might be updates available at Github. To update the repository navigate into the folder `doip-custom-simulation` and run git to update the local repository

```
cd doip-custom-simulation
git pull origin master
```

- We want to have the possibility to update the example when there are new versions available but we also want to create our own simulation with our custom settings. Therefore we create a copy of the exampe repositoriy.
- Create a copy of the repository to a meaningful name (here it is `doip-myproject-simulation`).

```
cp doip-custom-simulation doip-myproject-simulation
```

- Delete the files for git from our copy

```
rm -rf doip-myproject-simulation/.git/
```

- Navigate into the folder of our copy

```
cd doip-myproject-simulation
```

- Change the name of the project in the file `settings.gradle` from `doip-custom-simulation` to `doip-myproject-simulation`
- Change version number in file `build.gradle`, for example to 1.0.0
- Modify the start script because name of JAR file has changed. It is located in `src/main/dist/start.sh`.
- Modify all configuration files in `src/main/dist`.
- Build the project

```
./gradlew build
```

- Install the distribution

```
./gradlew installDist
```

- Navigate to the installation folder

```
cd build/install/doip-myproject-simulation/
```

- Start the simulation

```
start.sh gateway.properties
```

2.2 Configuration of the DoIP Simulation

The main configuration file is given at a argument to the shell script "start.sh". This file is a normal Java property file. The following listing is the example configuration which is used in the repository "doip-custom-simulation".

Listing 1: gateway.properties

```
# Name of the gateway. It will be used for logging
name = GW

# Defines the local port on which the gateway is listening.
# For DoIP it is always 13400
local.port = 13400

# Defines the maximum number of bytes of a message which will be printed to log files
# If a message is longer it will be logged at the end with ...
# With this performance can be increased because logging very long messages takes time.
maxByteArraySize.logging = 64

# Defines the maximum number of bytes which will be used for lookup tables
# It takes long time to check regular expressions if messages are very long
maxByteArraySize.lookup = 16

# Entity ID of the gateway
eid = E1 E2 E3 E4 E5 E6

# Group ID of the gateway
gid = A1 A2 A3 A4 A5 A6

# VIN number as hex bytes
vin.hex = 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37

# Logical address of the gateway
logicalAddress = 1

# List of configuration files for the ECUs. The path to that files is relative from this
    file.
# Multiple files need to be separated by a semicolon.
ecu.files=EMS.properties;TCU.properties
```

In that file are references to the configuration files for the ECUs. The following listing shows the configuration file for an ECU:

Listing 2: EMS.properties

```
# Name of the ECU, will be used for logging
name=EMS

# Physical address of the ECU
address.physical=4711

# Functional address of the ECU (not used at the moment)
address.functional=0

# List of lookup files for UDS messages. Multiple files need to be separated by a
    semicolon.
uds.files=standard.uds;EMS.uds
```

Within the ECU property file there is a reference to one or more configuration files for the UDS messages. The following listing shows such a configuration file for UDS messages.

Listing 3: standard.uds

```
# This file defines request/response pairs for UDS messages
# Request and response are separated by a colon (:)
```

```
# Spaces can be added between the characters, they will
# be removed by the software before evaluating an expression.
# The request will be treated as a Java regular expression.
# The response must be a hex string, that means only characters
# from 0-9 and A-F are allowed.

################################
# Service 0x10                 #
################################

10 01 : 50 01 00 32 01 F4
50 02 : 50 02 00 32 01 F4
10 03 : 50 03 00 32 01 F4
10 \w\w : 7F 10 10

################################
# Service 0x3E                 #
################################

3E 00 : 7E 00
3E 80 :
```

In addition there can be so called modifiers added in the UDS configuration file. In case a request matches it will send the response and in addition it can modify other responses based on the specified request.

Listing 4: modifier.uds

```
# This file contains an exampe how to use the modifiers in the UDS files.
# After the first request/response pair there can added aadditional
# request/response pairs. If the reqnuest will match the request
# pattern the response will be send out as usual. In addition the
# patterns after the first request/response pair will modify other
# request/response pair.

# With service 0x22 F1 86 you typically can read the current
# active diagnostic session. The diagnostic session can be changed with
# servicer 0x10. The simulation data looks like the following example

# If request 0x10 01 will be received it sends out the response
# 0x50 01 00 32 01 F4. In addition it will change the response on request
# 0x22 F1 86 to 0x62 F1 86 01

10 01 : 50 01 00 32 01 F4 : 22 F1 86 : 62 F1 86 01

# Similar definition for service 0x10 03, just the session is
# a different one.

10 03 : 50 03 00 32 01 F4 : 22 F1 86 : 62 F1 86 03

# This is the response for request 0x22 F1 86. This response will
# be modified by the definitions before for service 0x10.

22 F1 86 : 62 F1 86 01
```

3 Customization of DoIP Simulation

The possibilities to configure the DoIP simulations might might not cover all use cases which are needed for a DoIP simulation. To cover this use case the DoIP simulation was designed to give you to possiblity to override the implementation of the predefined functions. This will be done by creation of a custom class which inherits from an existing class in the DoIP simulation and overrides the functions.

The problem is that the existing DoIP simulation can not create an instance of the inherited class, because it is not yet known at implementation time. The solution is to create a custom main function where th

first instance of a DoIP simulation will be created which will now be the inherited class.

This is already done in the project `doip-custom-simulation`. Create a copy of the source files and adjust the implementation according to your needs.

The following listing shows how to create an own main function and create a custom DoIP simulation.

Listing 5: doip/custom/simulation/Main.java

```
package doip.custom.simulation;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import doip.library.properties.EmptyPropertyValue;
import doip.library.properties.MissingProperty;
import doip.simulation.nodes.GatewayConfig;

public class Main {

    public static void main(String[] args) throws IOException, MissingProperty,
        EmptyPropertyValue {
        if (args.length != 1) {
            System.out.println("Invalid number of arguments.");
            System.out.println("Usage:");
            System.out.println("    ./start.sh <property-file>");
            System.out.println("Example:");
            System.out.println("    ./start.sh gateway.properties");
            System.exit(1);
        }
        System.out.println("Starting DoIP simulation ...");

        GatewayConfig config = new GatewayConfig();

        config.loadFromFile(args[0]);

        CustomGateway gateway = new CustomGateway(config);
        gateway.start();

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Press <Enter> to terminate simulation");
        reader.readLine();

        gateway.stop();
        System.out.println("... DoIP simulation finished.");

    }

}
```

Listing 6: doip/custom/simulation/CustomGateway.java

```
package doip.custom.simulation;

import doip.simulation.nodes.EcuConfig;
import doip.simulation.nodes.GatewayConfig;
import doip.simulation.standard.StandardEcu;
import doip.simulation.standard.StandardGateway;

public class CustomGateway extends StandardGateway {

    public CustomGateway(GatewayConfig config) {
        super(config);
    }

    @Override
    public StandardEcu createEcu(EcuConfig config) {
```

```
        return new CustomEcu(config);
    }

}
```

Listing 7: doip/custom/simulation/CustomEcu.java

```
package doip.custom.simulation;

import doip.library.message.UdsMessage;
import doip.simulation.nodes.EcuConfig;
import doip.simulation.standard.StandardEcu;

public class CustomEcu extends StandardEcu {

    public CustomEcu(EcuConfig config) {
        super(config);
    }

    @Override
    public boolean processRequestBeforeLookupTable(UdsMessage udsRequest) {
        // Here you can add some special handling for a UDS message
        // before the message will be handled by searching patterns
        // in the lookup table.

        // Return false means that message was not handled in this
        // function and that processing message shall be
        // continued by further functions (lookup table, normal message interpretation)

        return false;
    }

    @Override
    public boolean processRequestAfterLookupTable(UdsMessage request) {
        // If a message was not handled by the lookup table then message
        // processing will be handled within this function.

        // Default handling will be to send negative response with NRC 0x10
        return super.processRequestAfterLookupTable(request);
    }
}
```

4 Gradle Build Tool

All Java projects use Gradle as a build tool. It is not required that each user will install Gradle. All projects use the so called "Gradle wrapper" which makes it easy to run and ensure that all users are using the same Gradle version.

The project layout in a Gradle project is defined by convention.

`settings.gradle` : Settings for gradle, for example the project name

`build.gradle` : Build script where is defined what to build and how to build it

`src/main/java` : Source code of the project which will be build into one jar file

`src/test/java` : Unit tests which will be executed after compilation

`src/main/dist` : All these files and folders will be packed also into the distribution (only if distribution plugin is used).

To run a Gradle task just type `./gradlew` in a bash followed by the gradle task.

To build a project just run

```
./gradlew build
```

The build task will also execute the unit tests. The comiled jar will be located at `build/libs`. If distribution plugin is used then the distribution will be build into folder `build/distributions`.

In case of using the distribution plugin you can run the following command to extract the distribution.

```
./gradlew installDist
```

After that the extracted files can be found in the folder `build/install`. You can navigate in the subfolder and run the application. In our case there should be a shell script "start.sh" which can be called.