SEAMS ARTIFACT

DeltaIoT User Guide

March 21, 2017

Muhammad Usman Iftikhar
Gowri Sankar Ramachandran

Introduction

DeltaIoT is a real world artifact of Internet-of-Things (IoT) for research on self-adaptation. The artifact is deployed at the campus of the Computer Science department of KU Leuven, Belgium. It consists of 25 IoT devices (motes) and a gateway. DeltaIoT can be accessed via client software that connects remotely with the physical network to perform adaptations. The client can monitor and adapt the network using probes and effectors provided by the artifact.

In this document, we explain how to use DeltaIoT for experimentation with self-adaptation techniques. The guideline consists of four sections. In the first section, we discuss preparation steps needed to start an experiment. The second section explains the DeltaIoT package that can be downloaded from the DeltaIoT website. In the third section, we discuss the graphical user interface (GUI) of the client software and the simulator. Finally, the fourth section explains how to create the adaptation logic for DeltaIoT using a simple adaptation example.

1 Preparation Steps

DeltaIoT provides access to the real IoT network (DeltaIoT server) via a virtual private network (VPN). To access the DeltaIoT, a client should use the VPN with the right credentials that authorize access. On the DeltaIoT website, an **Installation Guide** is available that explains how a client machine can connect to the VPN.

A user can get credentials, by sending an email to contact persons provided on the DeltaIoT website.[1] The email should clearly include the purpose of using the DeltaIoT network and how long (days) access is required. In response, the user will get an authentication token and a URL. The authentication token will be used by the system to identify the user, and the URL provides the location of the DeltaIoT WSDL (Web service description language). Furtheron, we discuss how to use these credentials.

2 DeltaIoT Package

The DeltaIoT package is available for download from the `Download` section of the DeltaIoT website. The package is available as a zip file, containing the source and executable files of DeltaIoT. To perform experiments with DeltaIoT, two sub-packages are provided:

[1] https://people.cs.kuleuven.be/danny.weyns/software/DeltaIoT/

1. **DeltaIoTClient:** The `DeltaIoTClient` provides the software to implement your own adaptation logic and remotely connect with the physical IoT system using probes and effectors to perform adaptations.

2. **Simulator:** The `Simulator` as the name suggests enables simulation of the DeltaIoT network (on your local machine). Thus, the simulator enables offline experiments without connecting with the real DeltaIoT.

The `Simulator` and the `DeltaIoTClient` have the same interface, which allows seamless integration with the simulator and the physical IoT system. The simulator is useful for debugging, testing and performing offline experiments before using the real set up (which is more time consuming).

Both the DeltaIoTClient and the Simulator are included in the zip file of the DeltaIoT package. The zip file contains the following folders:

1. `DeltaIoT-Source`

2. `DeltaIoT-Executables`

We now briefly discuss the sources and how to use the executables.

Source files

The source folder is an `Eclipse` workspace that contains several Java projects. The source folder can easily be used by `Eclipse` software (`www.eclipse.org`). To use the source folder, after installing Eclipse, run `Eclipse` and select the source folder as workspace. The source folder contains the following projects:

1. **DelataIoTClient:** `DeltaIoTClient` is a framework that enables the adaptation logic to monitor and adapt the real IoT system.

2. **DelataIoTGUI:** A graphical user interface that allows visualising the current topology of the DeltaIoT.

3. **SimpleAdaptationWithDeltaIoTClient:** A simple example of adaptation logic that uses the `DeltaIoTClient` framework to demonstrate how the real IoT system can be adapted.

4. **Simulator:** `Simulator` software that enables offline experiments of the simulated DeltaIoT network on a local machine.

5. **SimpleAdaptationWithSimulation:** A simple example of adaptation logic that uses the `Simulator` for performing adaptation experiments in simulation.

6. **SimulatorGUI:** A Graphical user interface that allows to visualize simulation results with and without adaptation. `SimpleAdaptationWithSimulation` shows the adaptation results graphically.

The `DeltaIoTClient` folder contains a `config.properties` file. As we already discussed above, access to the DeltaIoT network requires credentials that can be obtained via email to contact persons of the DeltaIoT network. The credentials including the authorization token and URL should be saved in the `config.properties` file before using the DeltaIoT network.

Executable files

The `Executables` folder contains two subfolders `DeltaIoT` and `Simulator`, which separates the executables of both the `DeltaIoTClient` and the `Simulator`.

DeltaIoT folder:

The `DeltaIoT` folder contains the executable of `DeltaIoTGUI`. Before executing this file, the user should make sure that the credentials are saved in the `config.properties` file in the folder as explained above.

To execute the `DeltaIoTGUI`, open a terminal and go to the `DeltaIoT` folder inside the `DeltaIoT-Executables` folder using the following command:

```
cd DeltaIoT-Executables/DeltaIoT
```

After that run the following command to execute the `DeltaIoTGUI`.

```
java -jar DeltaIoTGUI.jar
```

Once the above command is executed, a graphical user interface (GUI) will appear. In the **User Guide** we discuss the graphical user interface of `DeltaIoTGUI` in detail.

Simulator folder:

The `Simulator` folder contains the executable of the `SimulatorGUI` software. The Simulator graphical user interface (GUI) enables visualising the results of the simulated DeltaIoT network with and without adaptation. To view the results of adaptation, we use the `SimpleAdaptationWithSimu` example.

To run the GUI, open the terminal and go to the `DeltaIoT-Executables/Simulator` folder using the cd command.

```
cd DeltaIoT-Executables/Simulator
```

After that run the following command in a terminal:

```
java -jar SimulatorGUI.jar
```

Once this command is executed a graphical user interface will appear which is discussed in the next section 3.

3 Graphical User Interface

This section explains how to use the graphical user interface of both the physical network (`DeltaIoTGUI`) and the simulator (`SimulatorGUI`).

DeltaIoTGUI

The graphical user interface of DeltaIoT (`DeltaIoTGUI`) shows the current topology of the physical DeltaIoT network, see Figure1. The GUI consists of two sections. The top section displays the current topology. For each link, we can see the current power and distribution settings. If the topology is changed after an adaptation, these changes are marked in green color on the respective links.

The second part shows the log of topology changes. The log shows the date and time of the change, link information (source and destination), the previous settings (power, distribution factor) and the new settings.

SimulatorGUI

The simulator provides a graphical user interface that enables users to run experiments on the simulated DeltaIoT network. The interface provides support for two kind of experiments: 1) run a simulation of the DeltaIoT network without adaptation logic, 2) and simulate the network with the simple adaptation logic. This simple adaptation logic can be replaced by a new developed adaption solution. We explain the simple adaptation example in section 4.

Figure2 shows the graphical user interface of the Simulator software. The interface also consists of two sections. The top section consists of two charts that show the energy

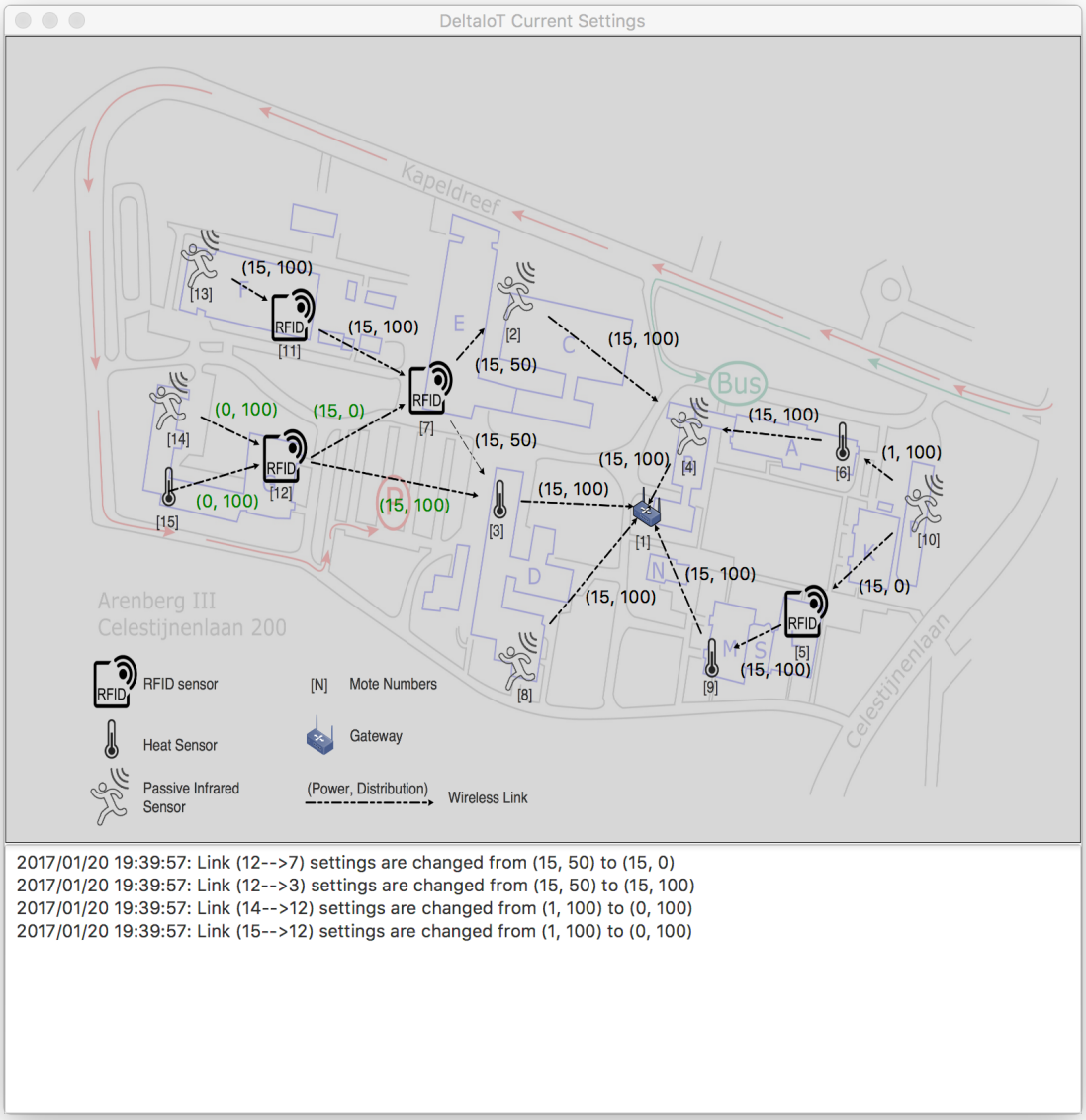

Figure 1: Graphical user interface of `DeltaIoTGUI`

consumption and packet loss of an experiment (a series of runs). Each dot in these charts shows the data collected by simulator while running.

The bottom section consists of multiple buttons that helps the user to select the required experiment and a progress bar that shows the progress of the experiment while it is running. The following buttons are available:

- **Run Simulator:** This button runs the simulator without any adaptation logic. The results are shown in the graphs at top section. In the example, we simulated the DeltaIoT

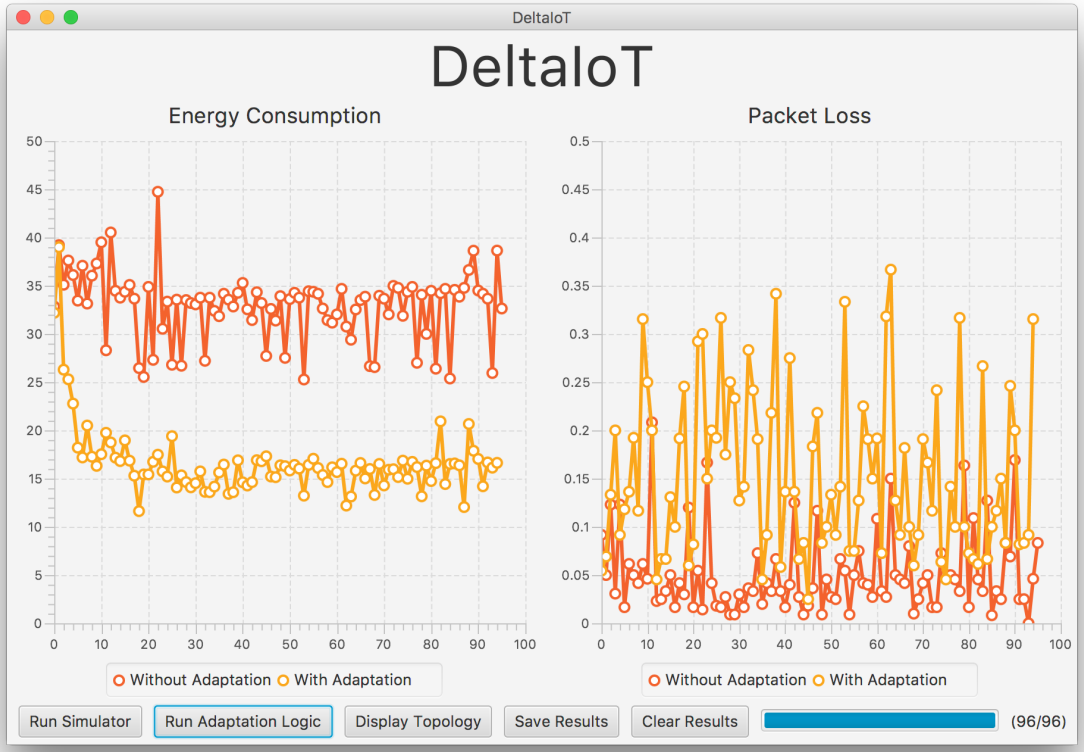

Figure 2: Graphical user interface of the Simulator

network for a period of time that corresponds with one day, where the simulator collects data after each 15 minutes.

- **Run Adaptation Logic:** This button runs the simulator with adaptation logic. We use the `SimpleAdaptationWithSimulator` example here with the results displayed in the top section.
- **Display Topology:** Clicking this button will open a new window that displays the current topology of the simulated DeltaIoT network. This new window is the same as the window shown in the `DeltaIoTGUI` interface discussed before, see Figure1.
- **Save Results:** This button enables saving the experiment results as a csv file.
- **Clear Results:** This button clears the experiment results.

4 Creating and Evaluating Your Own Adaptation Logic

To create your own self-adaptive solution (adaptation logic), both the `DeltaIoTClient` and the `Simulator` provides the same interface with a probe and effector. The probe enables monitoring data about motes and links, and the effector enables adapting the network settings, such as, transmission power over links, the distribution factor (percentages of messages sent to different parents) and the spreading factor (a higher spreading factor increases the communication range at the cost of more energy).

We now discuss the functions of both the probe and the effector. After that we discuss the simple adaptation example. We conclude with a step-by-step guide about how to use the probe and effector to create a self-adaptive solution, illustrated with the simple adaptation example

Probe

The Probe enables monitoring the status of DeltaIoT using the following functions:

- `ArrayList<Mote> getAllMotes()`: Returns a list of Mote objects, each object containing data about the mote setting and its links.
- `double getMoteEnergyLevel(int moteId)`: Returns the current energy level of a mote.
- `double getMoteTrafficLoad(int moteId)`: Returns the current traffic load generated by a mote.
- `int getLinkPowerSetting(int src, int dest)`: Returns the current power settings of a link.
- `int getLinkSpreadingFactor(int src, int dest)`: Returns the current spreading factor setting of a link.
- `double getLinkSignalNoise(int src, int dest)`: Returns the current signal to noise ratio of a link.
- `double getLinkDistributionFactor(int src, int dest)`: Returns the percentages of messages sent over a link.

- `ArrayList<QoS> getNetworkQoS(int period)`: Returns the result of the adaptations for the whole network over a period of time, including packet loss and energy consumption.[2]

Effector

The Effector enables adaptation of the parameters of the motes and links. The Effector provides the following functions:

- `void setMoteSettings(int moteId, List<LinkSettings> linkSettings)`: Adapt the network settings of a mote, including the transmission power, the distribution factor, and the spreading factor.
- `void setDefaultConfiguration()`: Set the DeltaIoT network to a default configuration. The default configuration (which corresponds to the reference approach for experiments) sets the transmission power for all the links of the network to maximum (i.e., 15), duplicates all traffic sent by a mote to all its parents (i.e., distribution factor 100% for each link), and sets the spreading factor for the whole network to a default value (i.e. 11).

Simple Adaptation Example

The simple adaptation example performs two basic adaptations on the DeltaIoT network:

1. `Optimize power settings` of each link based on the actual Signal to Noise Ratio (SNR) for that link. Concretely, the power settings is increased with 1 if the SNR is less than 0 and decreased with 1 if the SNR is higher than 0. With this adaptation strategy we keep the SNR on a level where no packets gets lost (SNR at least 0) on the link and power is optimal.
2. `Optimize distribution factor` of the links such that the messages are routed to the links that use less power. If the situation is not optimal, the distribution is adapted in steps of 10% in favour of the parent with the lowest power setting.

Now we will illustrate step-by-step how to the `DeltaIoTClient` and the `Simulator` was used to create the simple adaptation example with `Eclipse`. The source code of this example is available in the `DeltaIoT-source` folder.

[2] In the future, the latency of the network will be included in the quality of service results.

Step 1: Create a Java project. Open the Eclipse IDE and Click File → New → Java Project. This will open the "New Java Project" window. Provide a name for the project and click the Finish button.

If you don't see the "Java Project" in the menu then click on File → New → Other. A dialog will open where you can filter and select Java project as shown in Figure3.

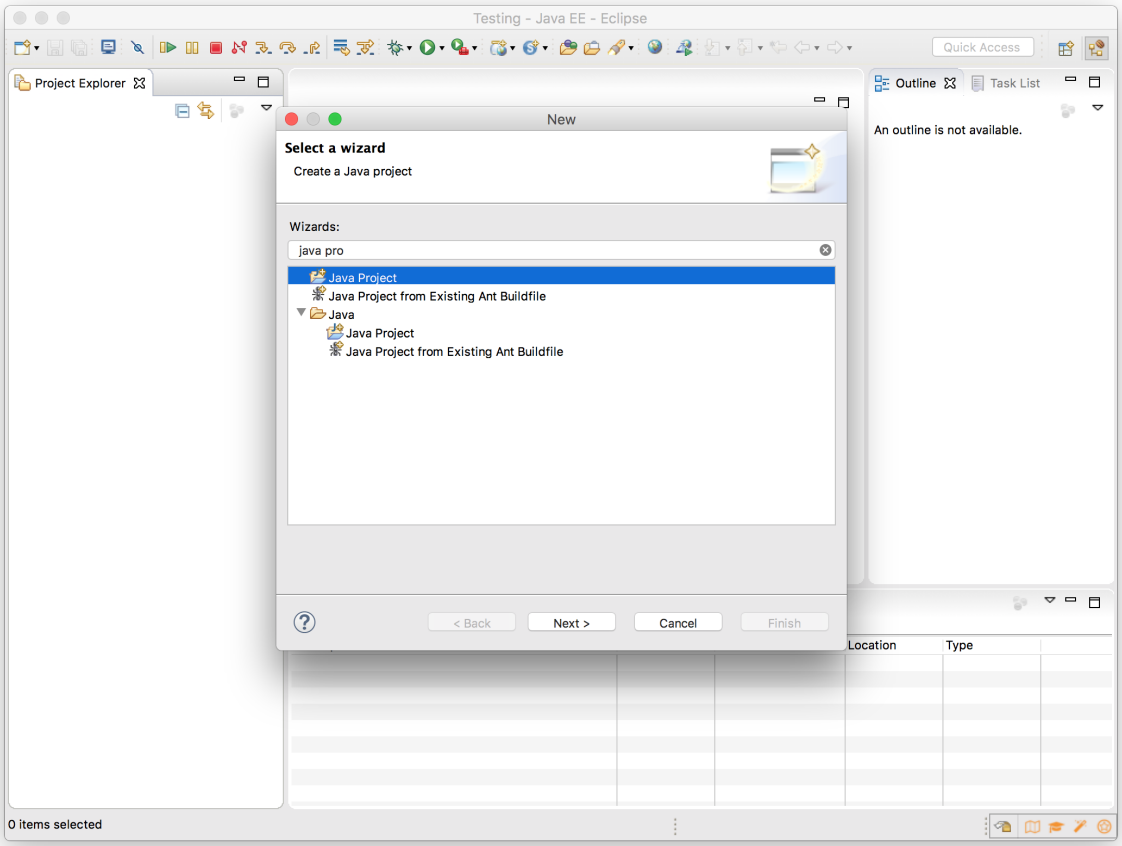

Figure 3: Create New Project

Step 2: Add DeltaIoTClient to the buildpath of the new project. Right Click on the project in the Project Explorer pane. A menu will appear. Select BuildPath → ConfigureBuildPath as shown in Figure4.

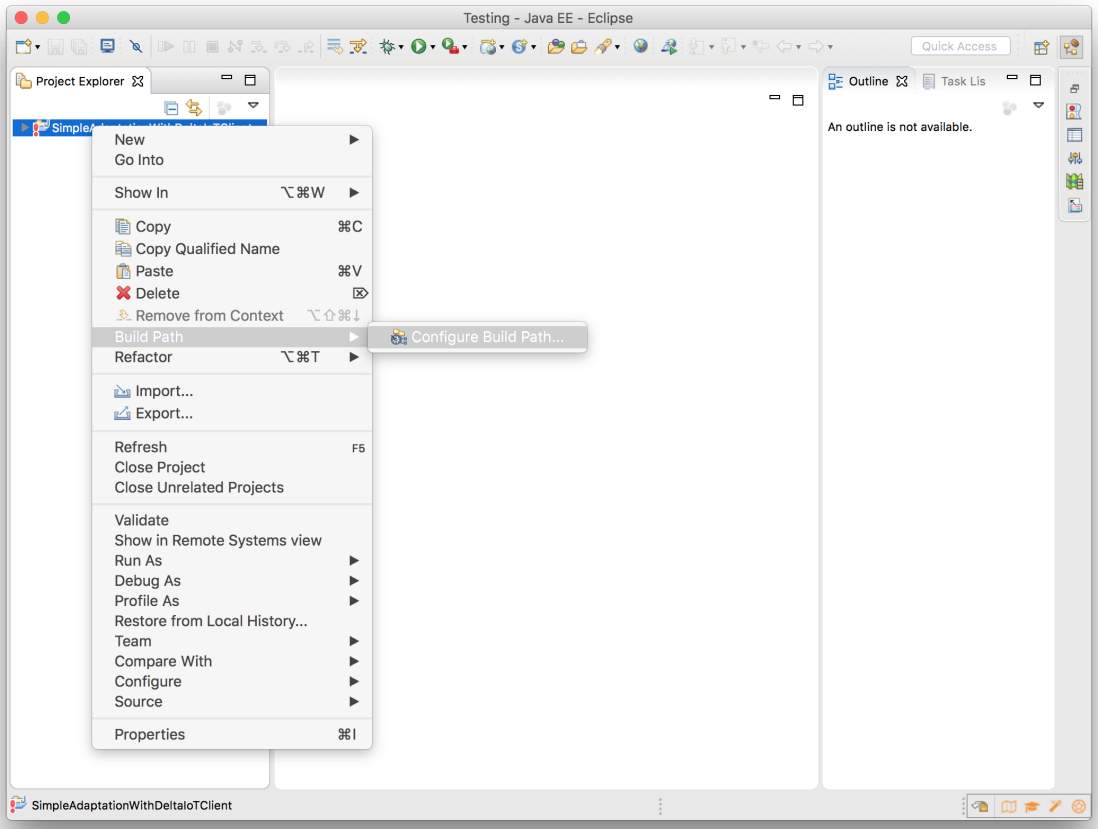

Figure 4: Configure Build Path of the project

In a similar way, the `Simulator` software can be added to the project, if you want to use the `Simulator`.

After that a dialog window will open as shown in Figure5. Then go to the Libraries tab and Click on Add External Jars. A file dialog will open to select the `DeltaIoTClient.jar`; click Open. This will add the `DeltaIoTClient.jar` to the libraries. After that click `OK` to return to the Eclipse workbench.

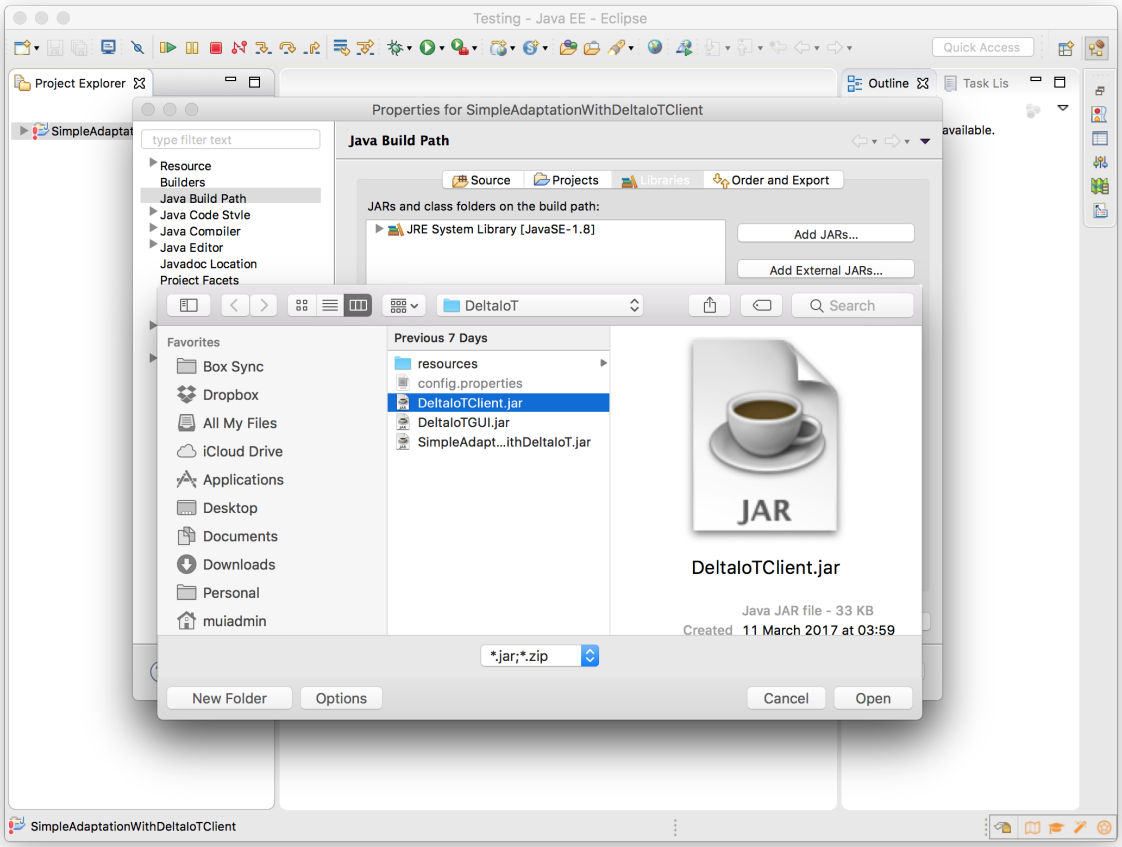

Figure 5: Add `DeltaIoTClient` as External Jar to the project

Step 3. Copy the config.properties file into the project and add the given token and URL Go to the DeltaIoT executables folder and copy the `config.properties` file, then right click on the root of the project in the Project Explorer pane and paste. This will add the config.properties file to the project. Double click on it to edit it in Eclipse and set the provided token and URL as shown in figure 6.

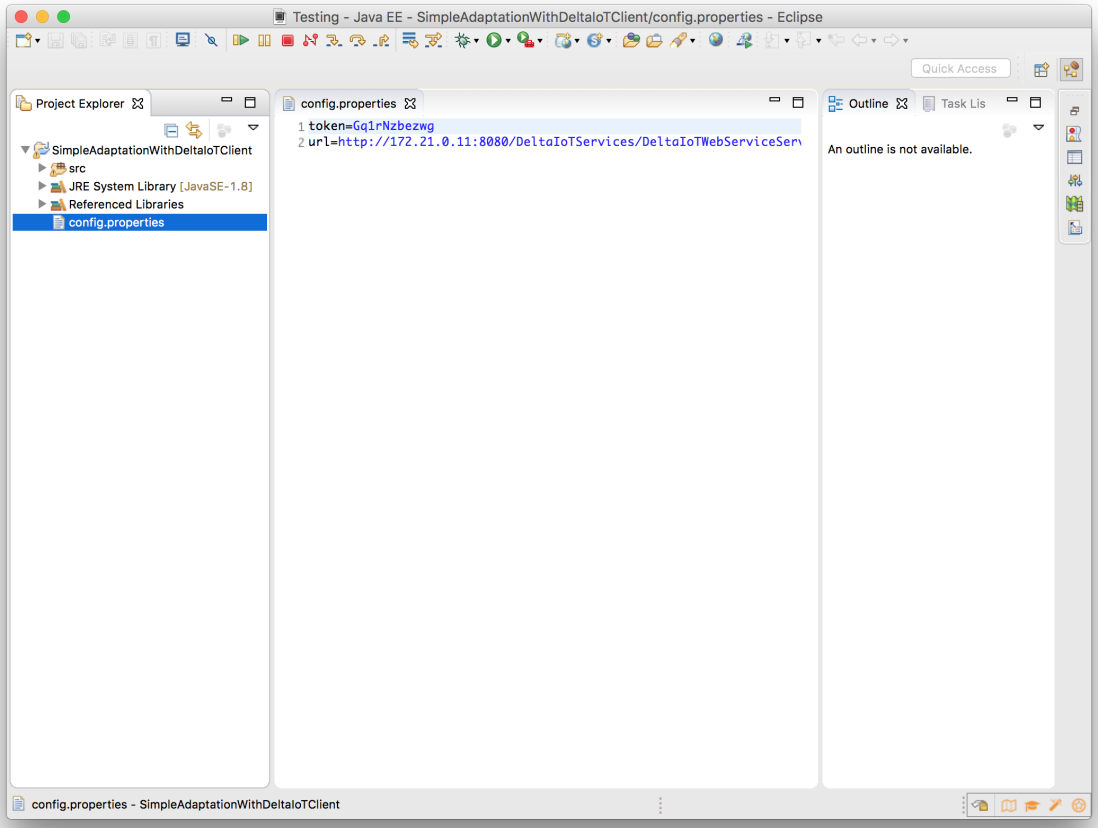

Figure 6: Add `config.properties` file to the project and set the token and URL

Step 4: Instantiate DeltaIoTClient Probe and Effectors Now it is time to instantiate the Probe and Effector classes that enable the communication between the client application and the IoT system. These classes are provided in the `deltaiot.client` package. Import these classes using the `import` statement as show in Figure 7 `(line 6)` and then instantiate the probe and effector using the `new` operator as shown in the `start` method `(line 18)`.

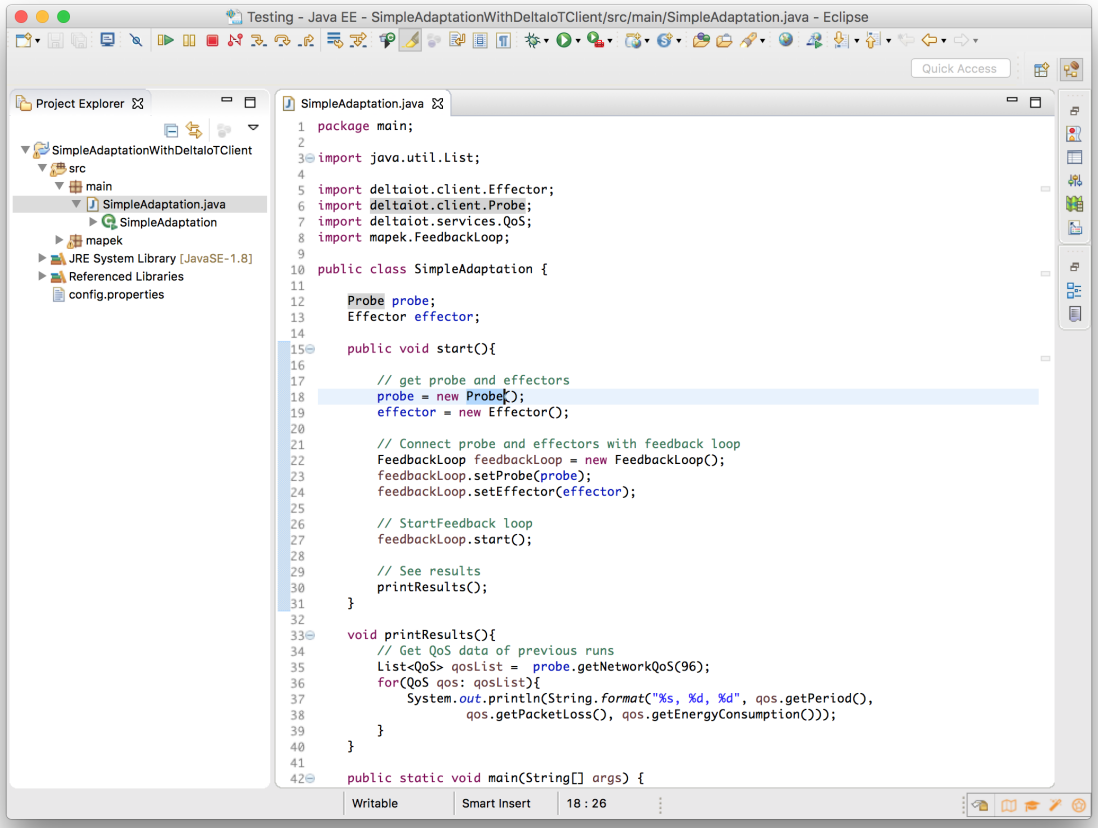

Figure 7: Instantiate `DeltaIoTClient` Probe and Effector classes

If you are using the simulator, then the probe and effector can be retrieved using the following lines:

```
SimulationClient simulationClient = new SimulationClient();
Probe probe = simulationClient.getProbe();
Effector effector = simulationClient.getEffector();
```

Step 5: Monitor the DeltaIoT network using the Probe functions As mentioned before, the Probe class of the `DeltaIoTClient` provides many functions that help monitoring the DeltaIoT network. For example, the `probe.getAllMotes()` function returns the data of all motes with all the links settings. A call to this function will return the current information of all motes that can be analyzed to check whether any adaptation is required.

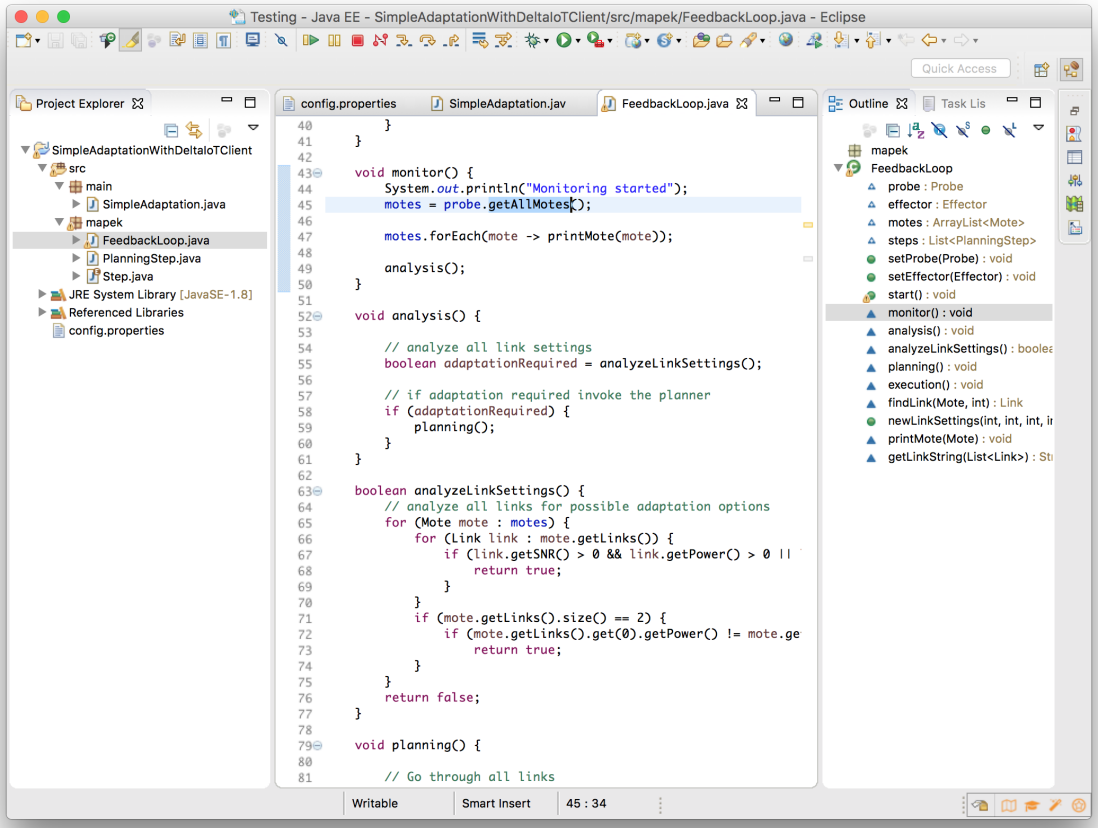

Figure 8: Monitor the DeltaIoT using Probe class functions

Figure 8 shows the `monitor()` method used in the simple adaptation example, where a function of the Probe class `probe.getAllMotes()` is called `(line 45)` to receive the network data. After that the method `analysis()` is called that uses the `analyzeLinkSettings()` function to traverse along each mote and checks all the links to find out whether an adaptation is required.

Step 6: Plan adaptations Figure 9 shows the `planning()` function of the simple adaptation example. When invoked, this function checks each link of the mote and decides whether there is a need to adapt the power setting. If there is a need, a planning step is created that is added to a list of plan steps (`line 88 & 91`). Similarly, if a mote has more than one parent, the planning function decides whether the distribution factor should be changed. If the distribution factor should be changed, then for each link a plan step to change the distribution is created that is added to the plan ((`line 106...110`)).

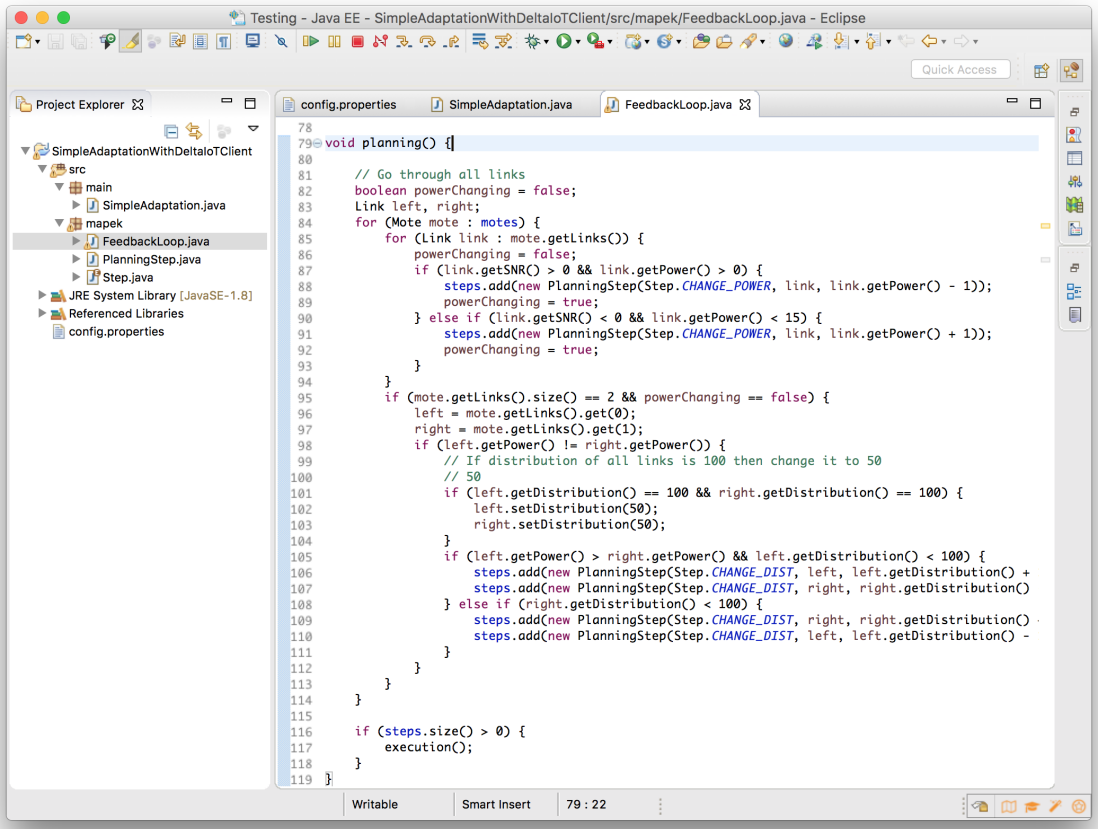

Figure 9: Planning function of the Simple adaptation example

Step 7: Execute adaptations using the Effector functions Once planning is completed, the functions of the Effector class can be used to execute the planning steps.

Figure 10 illustrates this for the simple adaptation example, where the `execution()` function uses the function of the effector `saveMoteSettings (line 148)` to adapt the network. The execution function first goes through all the adaptation steps (created by the planner) and creates a list of link settings for each mote. After that the list of link settings for each mote is sent to the DeltaIoT network using the `saveMoteSettings` function.

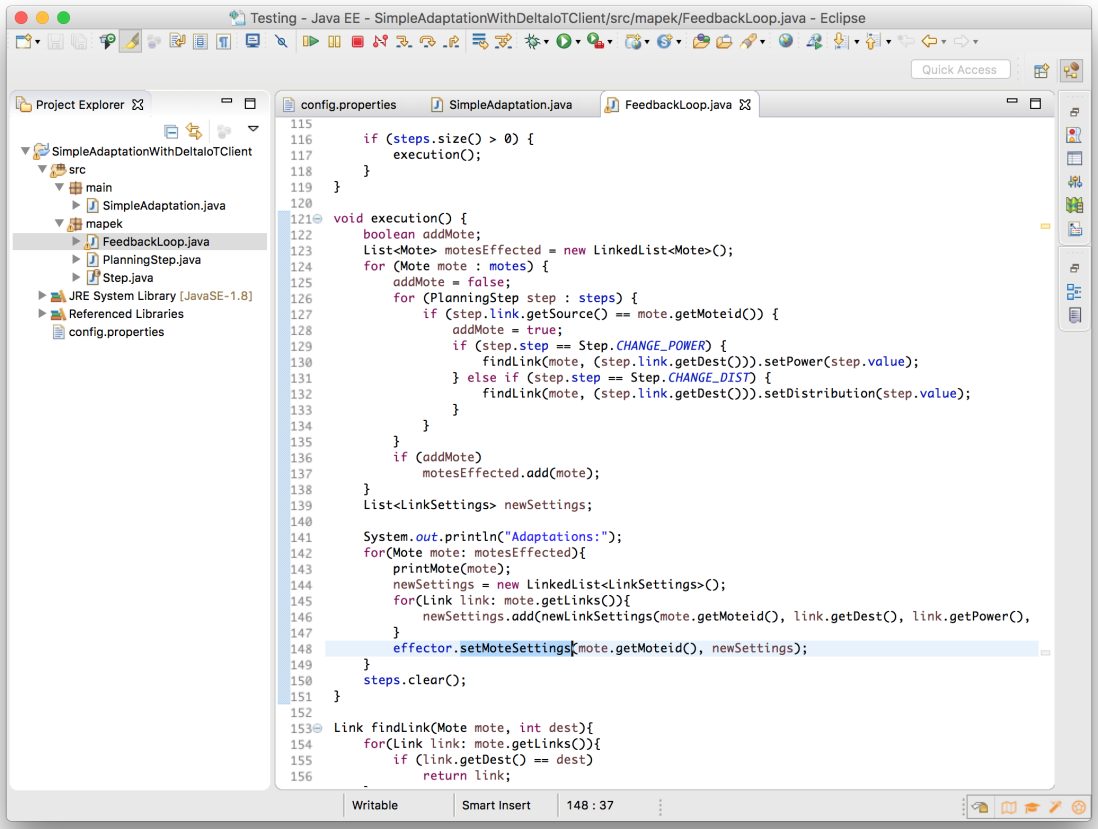

Figure 10: Execute adaptations using Effector class functions

Step 8: Monitor the results of adaptations After adaptation is applied, the results can be observed. To that end, the Probe class provides the function `getNetworkQoS(period)` that returns the quality of service (QoS) results of the DeltaIoT network. In the example one period consists of 15 minutes, during which each mote gets a turn to send data to the gateway. It is important to know that each mote adapts to the new settings at the start of each period. So any adaptation performed during a period will only be effected after that period. This implies that the results of an adaptation will only be available when the period is completed.

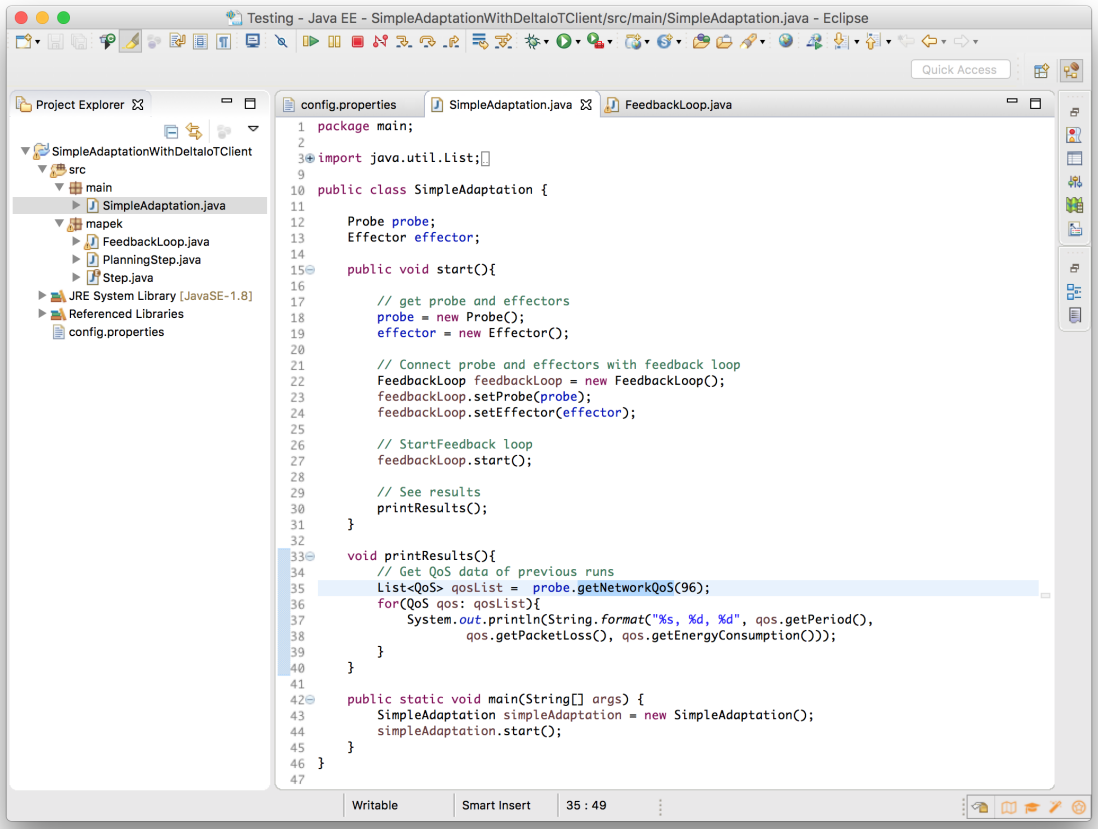

Figure 11: Monitor the QoS results

Figure 11 shows an excerpt of the code from the example, where we print the results of the last 96 periods, each period corresponding to 15 minutes; so 96 periods equal to 1 day). The function `getNetworkQoS` returns a list of QoS that contains the following information: 1) the time when the last period finished, 2) the packet loss of the network, and 3) the energy consumption of the network.