Lablink OPC UA client

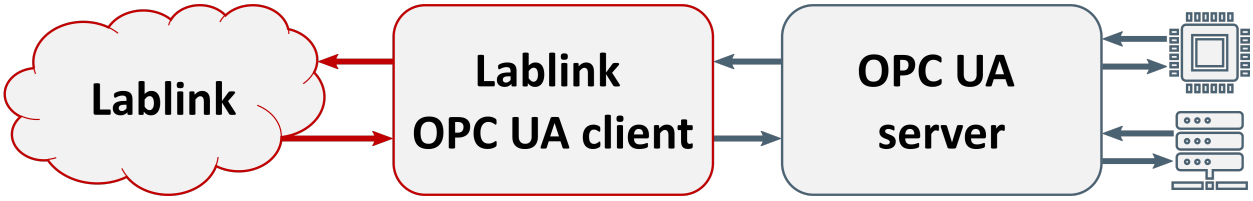
AIT Lablink Development Team

Feb 02, 2022

INSTALLATION

This package provides the base functionality for Lablink clients that use OPC UA for communication. The clients in this package rely on functionality provided by Eclipse Milo.

The functionality of the clients provided by this package only covers a subset of the large set of functionality defined by the OPC UA standard. For advanced use cases, where the functionality of this package's Lablink clients is not sufficient, this package provides are reasonable code base to start developing your own clients.



This package provides a basic clients called `BasicOpcUaClient`, which acts as an adapter between Lablink and an OPC UA server. Data received by this client will be written to the corresponding variables on the OPC UA server. Conversely, this client sends data whenever the corresponding variables on the OPC UA server change. This data exchange happens only on demand, i.e., whenever either a new input is sent to the client or in case a value changes on the OPC UA server.

This simple adapter only works for basic OPC UA data types (**Boolean**, **SByte**, **Byte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Float**, **Double**, **String**). The adapter takes care of casting these types to/from the data types supported by the Lablink data services (**Double**, **Long**, **Boolean**, **String**). It does not support other types, complex objects, function calls, etc.

CHAPTER

ONE

INSTALLATION

Find information about the installation of the Lablink OPC UA clients *here*.

1.1 Maven project dependency

The Lablink OPC UA client's compiled Java package is available on the Maven Central Repository. Use it in your local Maven setup by including the following dependency into your *pom.xml*:

```
<dependency>
  <groupId>at.ac.ait.lablink.clients</groupId>
  <artifactId>opcuaclient</artifactId>
  <version>0.0.2</version>
</dependency>
```

Note: You may have to adapt this snippet to use the latest version, please check the Maven Central Repository.

1.2 Installation from source

Installation from source requires a local **Java Development Kit** installation, for instance the Oracle Java SE Development Kit 13 or the OpenJDK.

Check out the project and compile it with Maven:

```
git clone https://github.com/ait-lablink/lablink-opcua-client
cd lablink-opcua-client
mvnw clean package
```

This will create JAR file *opcuaclient-<VERSION>-jar-with-dependencies.jar* in subdirectory *target/assembly*. Furthermore, all required JAR files for running the example will be copied to subdirectory *target/dependency/*.

1.3 Troubleshooting the installation

Nothing yet . . .

CHAPTER

TWO

RUNNING THE CLIENTS

Find basic instructions for running the clients *here*.

2.1 Invoking the clients from the command line

When running the clients, the use of the `-c` command line flag followed by the URI to the configuration (see *here*) is mandatory. For example, on Windows this could look something like this:

```
SET LLCONFIG=http://localhost:10101/get?id=
SET CONFIG_FILE_URI=%LLCONFIG%ait.test.opcuaclient.config

SET BASIC_OPCUAC_LIENT=at.ac.ait.lablink.clients.opcuaclient.BasicOpcUaClient
SET OPCUA_CLIENT_JAR_FILE=\path\to\lablink-opcua-client\target\assembly\opcuaclient-
↪<VERSION>-jar-with-dependencies.jar

"%JAVA_HOME%\bin\java.exe" -cp %OPCUA_CLIENT_JAR_FILE% %BASIC_OPCUAC_LIENT% -c %CONFIG_
↪FILE_URI%
```

CHAPTER

THREE

CONFIGURATION

Find the reference for writing a configuration for a Lablink OPC UA client *here*.

3.1 Overview

The configuration has to be JSON-formatted. It is divided into the following categories:

Client basic configuration of the Lablink client (JSON object)

Config-OPC-UA basic configuration related to the OPC UA client (JSON object)

Input configuration of the client's inputs, each associated to an OPC UA node (JSON array of JSON objects)

Output configuration of the client's outputs, each associated to an OPC UA node (JSON array of JSON objects)

In the following, the configuration parameters for these categories are listed.

See also:

See *below* for an example of a complete JSON configuration.

3.2 Basic Lablink Client Configuration

Required parameters

ClientName client name

GroupName group name

ScenarioName scenario name

labLinkPropertiesUrl URI to Lablink configuration

syncHostPropertiesUrl URI to sync host configuration (*currently not supported, use dummy value here*)

Optional parameters

ClientDescription description of the client

ClientShell activate Lablink shell (default: `false`).

3.3 OPC UA Client Configuration

Required parameters

EndpointURL URL of OPC UA server

NamespaceURI URI of OPC UA server namespace

ClientURI URI of Lablink OPC UA client

Optional parameters for BasicOpcUaClient

Username username for accessing the OPC UA server

Password password for accessing the OPC UA server

DefaulSamplingInterval_ms sampling interval for OPC UA server subscription (default: `1000`)

Note: In case no login credentials are provided (username *and* password), the client will attempt to connect as anonymous user.

3.4 Input and Output Configuration

Required configuration parameters for each input/output

Name: name of the client's input/output data service

DataType data type of the client's input/output data service; allowed values are `double`, `long`, `boolean` and `string`

***NodeIdString* or NodeIdStringNumeric** ID of associated OPC UA server node

Optional configuration parameters for each input/output

Unit unit associated to the client's input/output data service

3.5 Example Configuration

The following is an example configuration for a *BasicOpcUaClient* client:

```
{
   "Client": {
      "ClientDescription": "Lablink OPC UA client example.",
      "ClientName": "TestOPCUAClient",
      "ClientShell": true,
      "GroupName": "OPCUADemo",
      "ScenarioName": "OPCUAClientTest",
      "labLinkPropertiesUrl": "http://localhost:10101/get?id=ait.all.llproperties",
      "syncHostPropertiesUrl": "http://localhost:10101/get?id=ait.all.sync-host.
↪properties"
   },
   "Config-OPC-UA": {
      "EndpointURL": "opc.tcp://localhost:12345/lablink-test",
      "NamespaceURI": "urn:lablink:opcua-test",
      "ClientURI": "urn:lablink:clients:opcuaclient:test",
      "Username": "LablinkTestUser",
      "Password": "zQC37UiH6ou",
      "DefaulSamplingInterval_ms": 3000
   },
   "Input": [
      {
         "DataType": "double",
         "Name": "x",
         "NodeIdString": "LablinkTest/ScalarTypes/LlTestDouble",
         "Unit": "none"
      }
   ],
   "Output": [
      {
         "DataType": "integer",
         "Name": "y",
         "NodeIdString": "LablinkTest/ScalarTypes/LlTestUInt16",
         "Unit": "none"
      }
   ]
}
```

CHAPTER

FOUR

EXAMPLES

Find step-by-step instructions for running the examples *here*.

4.1 Prerequisites

4.1.1 Required Lablink resources

The following Lablink resources are required for running the examples:

- Configuration Server: * config-0.1.0-jar-with-dependencies.jar*

When *building from source*, the corresponding JAR files will be copied to directory *target/dependency*.

4.1.2 Starting the configuration server

Start the configuration server by executing script run_config.cmd in subdirectory examples/0_config/lablink. This will make the content of database file *example-lablink-config.db* available via http://localhost:10101.

Note: Once the server is running, you can view the available configurations in a web browser via http://localhost:10101.

See also:

A convenient tool for viewing the content of the database file (and editing it for experimenting with the examples) is DB Browser for SQLite.

4.1.3 MQTT broker

An **MQTT broker** is required for running the example, for instance Eclipse Mosquitto or EMQ.

4.1.4 Required OPC UA resources

The example uses a very basic OPC UA server, which is implemented with the help of the Python OPC-UA client and server library. Use `pip` in subdirectory examples/0_config/opcua to install all required Python packages:

```
pip install -r requirements.txt
```

Note: This setup has been tested with Python 3.8.5, you may have to adapt the package versions in file `requirements.txt` for other versions of Python.

4.1.5 Starting the OPC UA test server

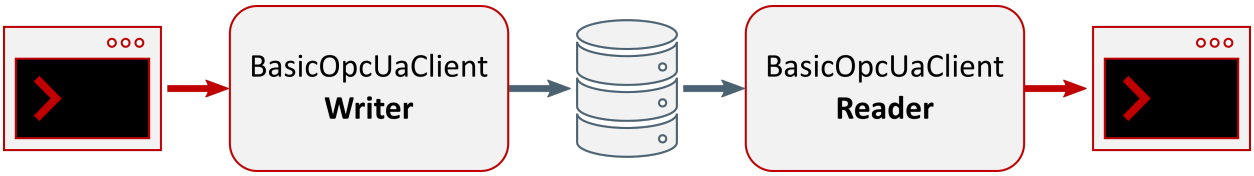
Start the OPC UA test server by executing script example-opcua-server.py in subdirectory examples/0_config/opcua:

```
python example-opcua-server.py
```

4.2 Example: Reading from and writing to an OPC UA server

In this example, two instances of class *BasicOpcUaClient* are used to write/read data from/to an OPC UA server:

- One client is configured to only write values to the server. Via the console, the user can set new values to the client's data services, which will be written to the associated variables on the OPC UA server. (In a realistic setup, the data services would receive data from other Lablink clients.)
- The other client is configured to only read values from the server. In the console, new values are displayed every time on of the associated variables changes on the OPC UA server. (In a realistic setup, the data services would be sent to other Lablink clients.)



All relevant scripts can be found in subdirectory examples/1_read_write. To run the example, execute all scripts either in separate command prompt windows or by double-clicking:

- writer.cmd: runs the client that writes values to the OPC UA server
- reader.cmd: runs the client that reads values from the OPC UA server

Note: The order in which the scripts are started is arbitrary.

Once the write-only client client starts up, the client shell can be used to interact with the OPC UA server. To start with, you can type `ls` to list all available data services:

```
llclient> ls
Name            DataType                State
xds     Double  0.0
xluis   Long    0
```

(continues on next page)

(continued from previous page)

```
xlis    Long    0
xbs     Boolean false
xluil   Long    0
xlil    Long    0
xlui    Long    0
xld     Long    0
xli     Long    0
xdb     Double  0.0
xdd     Double  0.0
xbb     Boolean false
xdf     Double  0.0
xbd     Boolean false
xdi     Double  0.0
```

You can use the console to change the values of these data services, which will cause the associated variable on the OPC UA server to be updated accordingly. For instance, data service `xdf` expect an input of type `Double` and will write this value to the OPC UA server variable with node ID `LablinkTest/ScalarTypes/LlTestFloat`. To update the value of this data service, use command `svd`:

```
llclient> svd xdf 12.34
Success
```

After a short delay, all the read-only client's data services subscribed to OPC UA server variable `LablinkTest/ScalarTypes/LlTestFloat` will receive the corresponding value. When this happens, you should see log outputs in the client's console similar to the following:

```
19:01:52.459 [milo-shared-thread-pool-0] INFO  OpcUaClientBase - subscription value␣
↪received: item=NodeId{ns=2, id=LablinkTest/ScalarTypes/LlTestFloat}, value=Variant
↪{value=12.34}, handle=1
19:01:52.464 [milo-shared-thread-pool-0] INFO  OpcUaClientBase - subscription value␣
↪received: item=NodeId{ns=2, id=LablinkTest/ScalarTypes/LlTestFloat}, value=Variant
↪{value=12.34}, handle=10
19:01:52.469 [milo-shared-thread-pool-0] INFO  OpcUaClientBase - subscription value␣
↪received: item=NodeId{ns=2, id=LablinkTest/ScalarTypes/LlTestFloat}, value=Variant
↪{value=12.34}, handle=14
```

To check the actual value of the data services, you can again type `ls`. You will see that the value of `12.34` has been received by several data services, with the value cast accordingly to the service's data type:

```
ysf     String  12.34
ylf     Long    12
ydf     Double  12.34000015258789
```

Note that the casting of the original value to data type `Long` and `Double` causes rounding errors! Hence, using the appropriate data type is always advisable ...

CHAPTER

FIVE

VERSION HISTORY

0.0.1 initial version

0.0.2 add login via username & password