OPC-Proxy

Release 0.0

Ale

Jan 17, 2020

CONCEPTS

The OPC-Proxy is a set of libraries aimed to facilitate communication between the industial autonomation world and **big data** tools. It is a gateway to put in comunication a network/server that speaks the OPC protocol with a network that supports a variety of other protocols. The supported protocols/services to date are: gRPC, Kafka and InfluxDB.

This Proxy was developed to allow full duplex comunication between an OPC-server and any of its clients, one of the main features is that it support an RPC-style type of communication, so that to allow reads and writes to and from an OPC-server.

This tool uses quite a few opensource libraries, we are gratefull to the opc-foundation, the lite-DB, the Confluent-platform, the gRPC framework, whitout which it would have been impossible to reach this result.

CHAPTER

ONE

JUMP TO CONCEPT

- The OPC-Proxy core functionality *Introduction*
- Description of supported *Connectors*
- How to *Write Your Own Connector*
- *Configuration* options

1.1 Introduction

1.1.1 What is OPC-Proxy?

The OPC-Proxy allows to build and deploy a customized IoT gateway to connect any OPC server with your network of microservices or cloud. This library is suitable for monitoring and control of devices, while tipically vendors of IoT gateways only offer monitoring. We focused on defining a protocol for bidirectional communication exposing the user to a simple API, so that one can read, but also write values to the OPC server without knowing details about OPC.

Features:

- Suitable for **monitoring** and **controlling** devices.
- Simple API.
- Reliable OPC client build with the OPC-foundation standard library.
- Load nodes from an XML file (Nodeset XML spec) or simply browsing the server
- Powerful Nodes loading selection options
- Modular design with external connectors that can be added, extended and customized.
- Supported connectors: *gRPC*, *Kafka*, *InfluxDB*.
- Written in C#.

Here we describe the features of the OPC-Proxy core library.

1.1.2 Basics of OPC

OPC is an opensource protocol used in industrial autonomation. It allows real time communication beween rugged industrial devices. The full specification is more than a 1000 pages and is quite complex (you can find it here), it describes several use cases and different comunication options, here we aim to support OPC UA. In this section we try to summarize briefly the important notions that are needed in the following.

OPC Data Structure

Data in an OPC server is organized in a tree structure, similar to a filesystem, there are *folders*, which contain *objects*, each object can contain *variables*. Each variable has an associated *data type* and various *references*. Now, one can access any item of the tree, each item is generalized as a **Node**. To access any node one needs to know its **node id** and the **namespace** under which it is stored. The OPC-proxy can interact only with nodes related to variables. Each node that is related to a variable has the following properties:

Variable Node properties

NodeID examples are *"ns=3;s=var_name"* for string identifier and *"ns=3;i=1234"* for integer identifier, where *ns* is the namespace. Weather the identifier is a string or a number depends on the server implementation.

BrowseName *Server given name*, it is usually related to the variable name, but it can contain additional caracters or can be truncated. It depends on the server implementation.

DisplayName *Human readable* variable name, this is usually the name that is displayed to the user.

Value Value of the variable.

Loading Nodes

A client does not in general know the node ids of the variables on which wants to operate on, usually they are strings related to the variable name, but in general these can be random integers, depending on the implementation of the server. There are two ways to make an opc-proxy aware of the node ids for the relevant variables, one can feed an XML file produced from the server with all nodes using the `OPC UA Nodeset XML schema`. Or one can simply browse each branch of the tree for variable nodes using consequent network calls, this last option is referred to as *browsing*. See configuration of the *Nodes Loader* for more info on the nodes selection criteria.

Warning: Browsing feature: The use of browsing feature for a server that contains many variables may lead to a large amout of network requests and can take long time (up to minutes). This of course depends on the server implementation.

Read, Write and Subscribe

An OPC client can *read*, *write* and *subscribe* to change of a server variable. Both read and write are single network calls, one can specify to read and write multiple variables at the same time. The OPC-proxy only support to read and write *values* on variable nodes, does not support any other attribute. Once the OPC-proxy is connected to the server and the session is open, one can monitor a list of variables for their change, these are called *monitored items*, the server will push those change when they occur.

Behaviours

Session Once the session with the opc-server is established, the OPC-proxy will the care of keeping the connection alive, in case of network interruption the OPC-proxy will keep trying to reconnect at user defined intervals.

Node Publishing once one subscribe for monitor a list of variables, the OPC-server will push their updated values and statuses at intervals to save network requests. The server will aggregate all the change for all the variables within a certain user defined time, to minimize the network requests it will send all the changes within the interval in a single network call. The OPC-proxy gives the user the possibility to set this time intervall.

Node Monitoring The OPC-proxy will automatically monitor (subscribe to change) all nodes specified in the configuration file.

Read and Write The OPC-proxy will only allow to read and write the nodes specified in the configuration file.

Memory Cache The OPC-proxy chaches the latest value for each variable in a in-memory database, so it will always return the value from cache for each read request made by the client and will not hit the OPC-server.

1.1.3 Connectivity Modules

The OPC-proxy can connect with only one OPC-server, on the other hand it can put the OPC-server in comunication with a large number of clients, each of these clients can use its own favourite protocol. The OPC-proxy has a modular design, to add new capabilities one simply needs to add the corresponding connector. Connectors are modules for the OPC-proxy that implement an endpoint for a communication protocol, they can leverage the OPC-proxy core library to interact with the OPC-server. To write your own connector see the *Extend Connectors* section.

The currently supported connectors are:

- **gRPC:** Implements an RPC type of comunication between a server and a client over HTTP. It uses the gRPC framework, see more details in the *gRPC Connector* section.
- **Kafka:** Implements a data stream to a Kafka topic trught the *Kafka producer* library. Implements an RPC type of comunication trough Kafka topics using the JSON-RPC protocol, it accepts write requests. More details in the *Kafka* connector section.
- **InfluxDB:** Submits a stream of metrics to InfulxDB on variables change. More details in the *InfluxDB* connector section.

1.2 Connectors

The OPC-proxy can connect with only one OPC-server, on the other hand it can put the OPC-server in comunication with a large number of clients, each of these clients can use its own favourite protocol. The OPC-proxy has a modular design, to add new capabilities one simply needs to add the corresponding connector. Connectors are modules for the OPC-proxy that implement an endpoint for a communication protocol, they can leverage the OPC-proxy core library to interact with the OPC-server. To write your own connector see the *Extend Connectors* section. In this section we describe the currenlty supported connectors: gRPC, Kafka and InfluxDB.

An OPC-Proxy **can run multiple connectors**, so for example one can push metrics to InfluxDB while serving a gRPC or Kafka (or both) endpoint.

1.2.1 gRPC Connector

gRPC is a modern open source high performance RPC framework, initially developed at Google. It is very flexible and userfriendly, it can easily put in communication different services independently on the programming language used. For more information visit grpc.io.

As a comunication layer gRPC uses HTTP2, while it uses protocol buffers as serialization/deserialization and Interface Definition Language.

The OPC-Proxy gRPC-Connector repository can be found at gitHub-GrpcConnector. You find all its configuration options in the *Configuration* section.

Protocol Definition

A client can initiate `Read` request and a `Write` request. In future also a subscritpion to server push on variable change will be available. The read/write request and response are defined in the proto-config-file that you can download from this gitHub-repo, you can use the proto-config to generate automatically the code needed for the comunication in almost any language.

A node is represented as:

```
NodeValue : {
    name : string,   // name of the node
    value : string,  // is always a string you need to deserialize it
    timestamp : string, // format ISO 8601 - UTC
    type : string      // int, bool, float, string
}
```

The value of the nodes is always serialized to `string`, is you responsability to deserialize it to the `type` specified.

A read request and its response is defined as:

```
ReadOpcNodes( ReadRequest ) returns ( ReadResponse )
// where:
ReadRequest : string[];  // list of node names to be read

ReadResponse : {
    nodes : NodeValue[], // list of read nodes
    isError : bool,
    errorMessage : string
}
```

The data exchanged in a write request is:

```
WriteRequest : {
    name : string,
    value : string
}

WriteResponse : {
    isError : bool,
    errorMessage : string
}
```

The value of the write request need to be serialized always to a `string`, the OPC-Proxy knows what is the right type and will take care of the conversion. In case of conversion error or OPC-server error it will return an error message (error message still in preparation see issue #1)in the response and the `isError` boolean will be true. A successful write request will have a `isError = false`.

gRPC Client Example

An example client based on NodeJs is provided in the gitHub-OPC-Node-Client-Example Follow the instruction reported there.

First run the OPC-Proxy configured with a gRPC endpont, this example assumes an OPC-Proxy running on `port:5051`, which is default, it also assume that the OPC-server is the Python-OPCUA, or in general that there will be an exposed variable called `MyVariable`.

The example will read and write a value to `MyVariable` of the python test server example. The value of MyVariable is always increasing by 0.1 every half a second. The client will read its value and reset it to 1.

Keep in mind that the OPC-server will push variables values (if they change) to the OPC-Proxy with rate of 1 sec, you can query the OPC-Proxy much faster than that, the write request will be forwared to the server immediately, but read request will read the latest value from the memory cache of the OPC-Proxy.

1.2.2 Kafka-Connector

Apache Kafka is an open-source stream-processing platform, it is the de facto standard for high-throughput, low-latency handling of real-time data feeds.

The Kafka-Connector add the ability to the opc-proxy to stream data to a kafka server. It supports:

- Sending a message on a topic when a node value changes (notification form opc-server)
- Bidirectional comunication, `read/write` and possibly more, with the PLC using an RPC protocol. The protocol supported is JSON-RPC-2.0.

This library uses the Avro serialization library, which allows great flexibility in defining the structure of the data exchanged. As storage engine for data schemas we are using the Confluent SchemaRegistry, which is necessary for this library. In the future a `JSON` based serialization option will be available and so the additional complexity of a schema registry will not be required anymore (see issue #4).

The OPC-Proxy Kafka-connector repository can be found at gitHub-KafkaConnectorLib. You find all its configuration options in the *Configuration* section.

Data Streams

The Kafka-Connector will by default define three **topics** the name of which depends on the configuration variable `opcSystemName`:

- The **Metrics-topic**, the one containing a stream of nodes value on change, is named as the `opcSystemName` configuration variable.
- The **RPC-request** topic, the one where all the (write) request are send, is named `opcSystemName` with appended suffix `-request`.
- The **RPC-response** topic, the one where all the RPC-style responses are served, is named `opcSystemName` with appended suffix `-response`.

Note: Keep in mind that your consumer clients need to have different `group ID` if you want all of them to receive updates. Also **Do Not** assign same `group ID` as the the OPC-Proxy to any other clients.

Serialization Deserialization

In Kafka messages are organized as a key:value pair. `Key` and `Value` can be serialized/deserialized with independent serializers, we choose to serialize `Keys` using the standard string-serializer, so the message key will always be a strings, while the values are serialized using the Avro framework.

In Avro one describes the data structure in a JSON-schema, the schema is stored in a schema-registry server, each message has an ID that refer to that schema, so each client can deserialize the messages dinamically. One can add a data type or modify an existing schema and the client will be able to properly deserialize the data without the need of additional code. So changing data structure (if backwords compatible) will not break clients. It also give flexibility, one can configure it to send messages with different data content on the same topic and the consumer will always be able to deserialize it correctly. Here we use this last property, independently on the Node value type, the consumer will always get the value properly deserialized (and already with the right type) depending on which language you are using.

For the **Metrics-topic**, the one where are streamed the nodes values on change, we use as Kafka **message Key** the node variable name, while as Kafka **message Value** we use the following Avro schema:

```
{
    type: 'record',
    name: datatype + 'Type',
    fields:[
        {
            name:'value',
            type: datatype
        }
    ]
}
```

Where `datatype` can be: `string`, `double`, `float`, `boolean`, `int`, `long`.

Kafka-RPC

For the RPC-style comunication we are using Kafka as a simple message broker, the default configuration of the producer and consumer of the RPC-topics are such that the comunication between the OPC-server and your client is preformed with latency of the order of `10 ms`. The protocol used used for this comunication is defined in the JSON-RPC spec. Even tough Kafka might seems inadequate for an RPC-style comunication, we find that it makes communication in a system (with many microservices) simpler and more flexible, it is a way to standardize comms, allowing for example a `kafka-stream` or a Storm-bolt to write easily to an OPC-server.

You can find the Avro schema used for the **RPC-request** and **RPC-response** topics at gitHub-RPC-Schemas. We tried to make it as close as possible to the original spec, but there are a few differences. The **RPC-request** kafka **message-key** is a string and is controlled by the client, the Kafka-Connector does nothing with it except making sure that the corresponding **RPC-response** will have the same message-key. A request message will look like:

```
// Request
{
    method : string,
    params : string[]  // list of strings
    id: long or null
}
```

Where the only `method` supported by now is `write` (but in the future might be more) and `params` is expected to contain a list with two strings, the first one representing the name of the node, the second one its value. The Kafka-Connector will take care of serializing the value from string to the correct data type expected by the OPC-server. The `id` if provided will be forwarded to the response, in case of `null` then the Kafka-Connector will return the Kafka

offset as `id` in the response, in this way you can let Kafka worry to generate system-wide unique ids for you (well, in this case topic-wide unique ids), you can collect the offset at request time, tide it to the topic and store it memory, then wait for the corresponding response.

A response message value in the **RPC-response** topics, would look like:

```
// Response
{
    result : string or null,

    error : {  // != null only if an error exist
        code : integer,
        message : string,
    }

    id: long
}
```

Given that the only supported method is "write", the `result` will be a string representing the written node value to the OPC-server, and will differ from `null` only in case of successful write operation. The `error` object will only be present in case of an error (when "result" is null). The `id` is either forwarded from the request or the Kafka-Offset of the related request-message in the RPC-request topic.

Note: If the RPC-request topic has more than one partition and the `id` is set to `null` in a request, the response `id` will be ambiguous. Please inform us if you have such a use case.

Kafka-Connector Client Example

You can find an example of kafka client for NodeJs in this repository gitHub-NodeKafka_client. Here we assume as opc test-server the *OPCUA-Python-server*. Using Node we show how to connect to a published data stream of nodes change on kafka, and how to interface to the kafka-RPC for writing nodes values.

1.2.3 InfluxDB

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics.

- Library used
- pushing metrics

1.3 Extend Connectors

Warning: This section is under construction.

1.4 Configuration

Configuration can be done via JSON file, the default file name is `proxy_config.json`. All the config keys that are not recognized will be ignored, if no configuration is provided for a parameter its default value will be loaded. An example of configuration file can be the following:

```
{
    "opcServerURL":"opc.tcp://localhost:4840/freeopcua/server/",

    "loggerConfig" :{
        "loglevel" :"info"
    },

    "nodesLoader" : {
        "targetIdentifier" : "browseName",
        "whiteList":["MyVariable"]
    },
    "gRPC" :{
            "port" : 5051
    }
}
```

1.4.1 OPC related Configs

These configs are related to core features and define how the opc client must behave. They must be placed at the root level of the json file.

Config Key	type	Default	Notes
opcServerURL	string	none	OPC server TCP URL endpoint
reconnectPeriod	int	10 [s]	Time interval [seconds] to wait before retry to reconnect to OPC server
publishingInterval	int	1000 [ms]	This is a subscription parameter, time intervall [millisecond] at which the OPC server will send node values updates.
opcSystemName	string	OPC	Name of the OPC system that will be used for identification

Nodes Loader

These are config related to nodes loading methods and selection rules.

Note: If no selection rules are defined then **ALL** nodes of type **Variable** are loaded automatically. If more selection rules are defined they Add/Excludes nodes (of type Variable) based on the following priority order: `whiteList`, `blackList`, `notContain`, `contains`, `matchRegEx`.

Config Key	type	Default	Notes
browseNodes	bool	true	If `true` load nodes via recursively drilling trough the server tree, it may use many network requests. If false will load nodes from an xml file, according to the Nodeset2 OPC specification.
targetIdentifier	string	DisplayName	Node attribute that undergoes selection rules, it can be: `displayname`, `browsename`, `nodeid`. In case of `nodeid` is necessary to specify also the prefix like `i=123` or `s=VarName`. It is case insensitive.
filename	string	nodeset.xml	Path to the xml file where the nodes are defined. Necessary if `browseNodes = false`.
whiteList	string[ ]	empty	Accept all nodes with `targetIdentifier` **exactly equal** to one of the string in the list. Runs first.
blackList	string[ ]	empty	Exclude nodes with `targetIdentifier` **exactly equal** to one of the string in the list. Runs second.
notContain	string[ ]	empty	Excludes nodes with `targetIdentifier` **containing** one of the string in the list. Runs before `contains` and `matchRegEx`.
contains	string[ ]	empty	Accept nodes with `targetIdentifier` **containing** one of the string in the list. Runs before `matchRegEx`.
matchRegEx	string[ ]	empty	Accept nodes with `targetIdentifier` **matching** one of the regular expression string in the list. Runs last.

1.4.2 gRPC-Connector Configs

These configs are related to the *gRPC Connector*, they must be placed under the key `gRPC` as follows:

```
{
    // Other config here

    "gRPC" :{
            "port" : 5051
    }
}
```

Config Key	type	Default	Notes
host	string	localhost	host name on the network.
port	int	5051	Port on which to listen for client requests

1.4.3 Kafka-Connector Configs

These configs are related to the *Kafka-Connector*, they must be placed under the keys `kafkaProducer` and `kafkaRPC`, there are also two root level configs: `KafkaSchemaRegistryURL` and `KafkaServers`, as in the example:

```
{
    // Other config here

    opcSystemName : "OPC",
```

(continues on next page)

(continued from previous page)

```
    KafkaSchemaRegistryURL : "localhost:8081",
    KafkaServers : "localhost:9092",

    kafkaProducer : {
        // Producer conf
    },
    kafkaRPC : {
        // RPC conf
    }
}
```

Root level cofigs:

Config Key	type	Default	Notes
opcSystemName	string	OPC	System name is a core variable, it will be used to evaluate the topic names for nodes publishing, see *Kafka-Connector*
KafkaSchemaRegistryURL	string	localhost:8081	Endpoint of the schema registry
KafkaServers	string	localhost:9092	Comma separated list of kafka brokers. These will be set for the producer and the consumer of the OPC-Proxy, this can be overidden, see below.

kafkaProducer:

Config Key	type	Default	Notes
BootstrapServers	string	localhost:9092	Comma separated list of Kafka brokers endpoints. If not set, this will be set to the value of **KafkaServers**.
BatchNumMessages	int	10000	See Confluent producer docs
LingerMs	int	100 [ms]	See Confluent producer docs
QueueBufferingMaxKbytes	int	1048576 [Kbytes]	See Confluent producer docs
QueueBufferingMaxMessages	int	100000	See Confluent producer docs
MessageTimeoutMs	int	300000	See Confluent producer docs
EnableIdempotence	bool	false	See Confluent producer docs
RetryBackoffMs	int	100 [ms]	See Confluent producer docs
MessageSendMaxRetries	int	2	See Confluent producer docs

kafkaRPC:

All the non reported kafka consumer configurations are set to default values.

Config Key	type	Default	Notes
BootstrapServers	string	localhost:9092	Comma separated list of Kafka brokers endpoints. If not set, this will be set to the value of **KafkaServers**.
GroupId	string	OPC	Group ID of the RPC kafka consumer. No other consumer can have this group ID in the whole system. If not set, default is to be set to `opcSystemName`.
enableKafkaRPC	bool	true	Enable the RPC-style comunication trough kafka topics.
EnableAutoCommit	bool	true	See Confluent consumer docs
EnableAutoOffsetStore	bool	true	See Confluent consumer docs
AutoCommitIntervalMs	int	5000 [ms]	See Confluent consumer docs
SessionTimeoutMs	int	10000 [ms]	See Confluent consumer docs
AutoOffsetReset	string	latest	See Confluent consumer docs
EnablePartitionEof	bool	false	See Confluent consumer docs
FetchWaitMaxMs	int	1000 [ms]	See Confluent consumer docs
FetchMinBytes	int	1	See Confluent consumer docs
HeartbeatIntervalMs	int	3000 [ms]	See Confluent consumer docs

1.4.4 InfluxDB-Connector Configs

1.5 Getting Started

- Read the *Requirements* before starting.
- Getting started with *Run With Docker* tutorial.
- Getting started with *Build Your Proxy* tutorial, if you want to compile your code.
- Customize your OPC-Proxy with the *Configuration* options.

1.5.1 Requirements

This library is written in C#, so you would need to install Microsoft .Net Core framework on your machine. This step is not necessary in case you use the provided Docker image of the standalone application. You do need it if you want to write a custom OPC-Proxy, in which case you would need to build the library into an executable.

Note: There are two ways to get up and running: using the docker image or compiling the code. A standalone OPC-Proxy is provided in both cases, this has already the basic functionality and just need to be configured.

In case you want to build your own custom application you would need the following:

- .NET Core >= 3.1 following the description here
- Your favourite OPC test server
- Git

1.5.2 OPC Test Server

This library is an OPC-client which is quite useless without an OPC-server to connect to. It is very usefull to have a test OPC-server that you can configure and run to quicly check and play arount with the OPC-Proxy library.

There are many opensource (and proprietary) library to write your own server, we reccomend two, that are decently complete and easy to use:

- The Python library (this is the default for all the examples)
- The NodeJS library

Setup an OPC-Server with Python

The Python based OPCUA library is maybe not the most complete but certainly one of the simplest to get up and running. Install the module:

```
pip install opcua
```

Copy and paste the server example from this repository into a file, call it for example `server-minimal.py`, for completeness we report the file below:

```
# server-minimal.py

import sys
sys.path.insert(0, "..")
import time


from opcua import ua, Server


if __name__ == "__main__":

    # setup our server
    server = Server()
    server.set_endpoint("opc.tcp://0.0.0.0:4840/freeopcua/server/")

    # setup our own namespace, not really necessary but should as spec
    uri = "http://examples.freeopcua.github.io"
    idx = server.register_namespace(uri)

    # get Objects node, this is where we should put our nodes
    objects = server.get_objects_node()

    # populating our address space
    myobj = objects.add_object(idx, "MyObject")
    myvar = myobj.add_variable(idx, "MyVariable", 6.7)
    myvar.set_writable()    # Set MyVariable to be writable by clients

    # starting!
    server.start()

    try:
        count = 0
        while True:
            time.sleep(1)
            count += 0.1
```

(continues on next page)

(continued from previous page)

```
            myvar.set_value(count)
    finally:
        #close connection, remove subcsriptions, etc
        server.stop()
```

Run it:

```
python server-minimal.py
```

This runs an OPC-server on port `4840` that expose an ever increasing variable of type Double called `MyVariable`. Now you need to run the OPC-Proxy to listen for that variable changes.

Setup the NodeJS OPC-Server

If you are familiar with NodeJS, a quite good library to check out is the Node-OPCUA. For this you need to install NodeJS and NPM, once you have done it is quite simple:

```
git clone git@github.com:node-opcua/node-opcua-sampleserver.git
cd node-opcua-sampleserver
npm install
node server.js
```

This will start a server on `port:  26543` and will expose two variables, one called `Temperature` and the other `MyVariable2`.

1.5.3 Run With Docker

Docker is a way to distribute self-contained applications easily. We provide a Docker image for the Community Edition that you can very easily install and upgrade on your servers. Your machine needs to have the **Docker Engine Communiti Edition (CE)** installed first. Refer to the docker installation page

Download

```
docker pull openscada/opc-proxy
```

This will pull an image with the .NET framework dependencies already installed and with a compiled executable. The image is built from this repository, it contains a standalone opc-proxy that can provide all supported connectors endpoint, *Kafka*, *InfluxDB*, *gRPC*.

Configure

Create a configuration file:

```
# host directory to share with the docker container
mkdir  opcProxyConfigs
# the configuration file must be called "proxy_config.json"
touch opcProxyConfigs/proxy_config.json
```

Edit the config:

```
/* proxy_config.json */
{
    "opcServerURL":"opc.tcp://localhost:4840/freeopcua/server/",

    "loggerConfig" :{
        "loglevel" :"debug"
    },

    "nodesLoader" : {
        "targetIdentifier" : "browseName",
        "whiteList":["MyVariable"]

    },
    "grpcConnector" :   false,
    "influxConnector" : false,
    "kafkaConnector":   false
}
```

This will tell the OPC-Proxy that:

- Needs to connect to an OPC server at the specified URL, we use a python test server as described in *Setup an OPC-Server with Python*, if you are using another test server you need to update that line.
- The nodesLoader here will match against a whitelist all nodes of the server, it will look for a Node with `BrowseName` attribute (see *OPC Data Structure*) equals to `MyVariable`, which is default for our test server.
- The log level is set to `DEBUG`, so that we will see the output of the variable changing.
- All connectors are set to `false`, meaning that this proxy will only connect to the opc-server and nothing more.

Setup a docker container:

```
cd opcProxyConfigs
# Env variable to make easier the next command
OPC_LOCAL_CONF=$(pwd)

docker create       \   # (1)
--name proxy_test   \   # (2)
--network="host"    \   # (3)
-v ${OPC_LOCAL_CONF}:/app/configs  \  # (4)
openscada/opc-proxy     # (5)

# below the same command as above but in one line (copy-paste friendly)
docker create --name proxy_test --network="host" -v ${OPC_LOCAL_CONF}:/app/configs␣
↪openscada/opc-proxy
```

This is quite a long command, let's brake it and see what it means:

- It creates a container of the image in `(5)` named as defined in `(2)`.
- In `(3)` set the `localhost` reference inside the container to point to the image host machine, so one can use in the config file `localhost` to reference to a service running on the host machine. If you would like to use the default docker networking option then you would need to find the IP of the docker `network bridge`, more details in the Docker guide Configure Networking.

- Line `(4)` is the most important, here we are mounting an external volume to the docker container, the syntax is simple: `-v absolute_path_to_host_dir : mirror_dir_in_container`, now all the content of the `host_dir` will be available to the docker container dynamically. Here we want to pass the directory we just created that contains the configuration file.

Warning: the volume path must be an absolute path from the `/`, even if the dir does not exist docker will not output an error.

Tip: Docker containers must have different names, so unless you remove the container (*docker rm*) you must change the name.

Run the Container

First you need to start your OPC test server (see *OPC Test Server*), then you can run the docker container:

```
# start the container and attach output to STDIN, close with Ctrl-C
docker start -i proxy_test
```

This should output something like this:

```
2020-01-08 17:05:53.5762|INFO|OPCclient|Creating Application Configuration.
2020-01-08 17:05:54.1004|WARN|OPCclient|Automatically accepting untrusted␣
↪certificates. Do not use in production. Change in 'OPC.Ua.SampleClient.Config.xml'.
2020-01-08 17:05:54.1004|INFO|OPCclient|Trying to connect to server endpoint:  opc.
↪tcp://localhost:4840/freeopcua/server/
2020-01-08 17:05:54.3017|INFO|OPCclient|Selected endpoint uses the following security␣
↪policy: None
2020-01-08 17:05:54.3017|INFO|OPCclient|Creating a session with OPC UA server.
2020-01-08 17:05:54.3495|INFO|serviceManager|Loading nodes via browsing the OPC␣
↪server...
2020-01-08 17:05:54.3765|INFO|OPCclient|Surfing recursively trough server tree....
2020-01-08 17:05:54.5011|INFO|cacheDB|Number of selected nodes: 1
```

Usefull Docker Commands

```
# start container in the background
docker start proxy_test

# stop container
docker stop proxy_test

# restart container (usefull when edit config)
docker restart proxy_test

# list running container
docker ps

# list all containers
docker ps -a
```

(continues on next page)

(continued from previous page)

```
# Remove container
docker rm __container_name__

# remove image
docker rmi  __image_name__
```

1.5.4 Build Your Proxy

In case you want to add your own feature to the proxy or not familiar with docker, here we explain all the steps needed to write and compile to executable your own proxy.

We are going to build from scratch a proxy very similar to the one in the standalone repository.

Install .NET

If you havn't done it yet, and especially if you are using a Linux machine. You can find instruction on how to do it on the Microsoft Website. You will need:

- .NET Core >= 3.1
- We suggest to install all three library: **.NET core SDK, .NET core Runtime, ASP .NET core runtime**.

Start with a clone Project

Probably the easiest is to clone the standalone repository and to take it as starting point.

```
git clone git@github.com:opc-proxy/opcProxy-Standalone.git
cd opcProxy-Standalone/
```

Now build it:

```
dotnet build
```

Now you can jump to *Add a Configuration file* and continue from there.

Create a project from scratch

Here we describe in steps how to create a .NET project that can be compiled to executable and runs a basic opc-proxy. You can get the same result by cloning the standalone repository described in *Start with a clone Project* section.

Crate a new blank project:

```
dotnet new console --name myproxy
cd myproxy
```

This is a scaffolding command, which created a directory with a file **Program.cs** and a configuration file **myproxy.csproj**. Now you can compile this "hello world" program and run it like this:

```
dotnet build
dotnet run
```

The console should output `Hello World!`

Install OPC-Proxy libs

The library can be installed using the package manager of .NET, nuget. In this demo we will use all the suppoted packages, of course you can install only the one you need, the following command will install their lastest version:

```
dotnet add package opcProxy.core
dotnet add package opcProxy.GrpcConnector
dotnet add package opcProxy.InfluxDBConnector
dotnet add package opcProxy.KafkaConnector
```

Modify the Program

Open the file `Program.cs` and edit it like below:

```
using System;
using OpcProxyCore;

namespace myproxy
{
    class Program
    {

        static int Main(string[] args)
        {
            // instantiaing the manager,
            // this will load configuration from file or args
            serviceManager manager = new serviceManager(args);

            // This runs the OPC-Proxy manager with all core
            // functionalities: connects to server, monitor items...
            manager.run();
            return 0;
        }

    }
}
```

Now build with `dotnet build`, there should be no error.

Add a Configuration file

Configuration can only be given via `JSON` file format. A configuration file is necessary, the program will stop otherwise. The default config file name is `proxy_config.json` and the program look for it in the directory where you run it. You can change the path or the name of the config file via the `--config path_to_file` flag at run time.

Create the following file (it is already there in case you are cloning the repo) in the main directory and name it `proxy_config.json`:

```
/* proxy_config.json */
{
    "opcServerURL":"opc.tcp://localhost:4840/freeopcua/server/",

    "loggerConfig" :{
        "loglevel" :"debug"
```

(continues on next page)

(continued from previous page)

```
    },

    "nodesLoader" : {
        "targetIdentifier" : "browseName",
        "whiteList":["MyVariable"]

    }
}
```

This will tell the OPC-Proxy that:

- Needs to connect to an OPC server at the specified URL, we use a python test server as described in *Setup an OPC-Server with Python*, if you are using another test server you need to update that line.
- The nodesLoader here will match against a whitelist all nodes of the server, it will look for a Node with `BrowseName` attribute (see *OPC Data Structure*) equals to `MyVariable`, which is default for our test server.
- The log level is set to `DEBUG`, so that we will see the output of the variable changing.

There are many configuration options and possibilities for loading nodes, they are described in detail in the *Configuration* section.

Run The Proxy

Before actually running the program we need two things:

- An additional config file. Copy this file from the standalone repository and place it in the main directory (not needed if you cloned it). The file name is important, so keep same naming `Opc.Ua.SampleClient.Config.xml`. Soon this will not be needed anymore, refer to issue #17.
- Run your favourite test opc-server. Remember the configuration we used will only work for the *Python test server*.

Now you can simply do:

```
dotnet run
# Or for a custom config file
dotnet run --config __path_to_file__
```

Press `Ctrl-C` to end the process.

Adding Connectors

Up to now the OPC-Proxy would only connect to the opc-server, browse its variable tree and subscribe to change of the variables that match the `nodeLoader` criteria. Now we will add **connectors** that will allow Read, Write, Subscribe acess to the external world.

If you followed the *Start with a clone Project* section you can add connectors via config file, by adding the following options:

```
{
    /* some config.... */


    "grpcConnector" :   false,
    "influxConnector" : false,
```

(continues on next page)

(continued from previous page)

```
    "kafkaConnector":   false
}
```

Turnig `true`/`false` those switches you can enable/disable the corresponding connector. You can find more details on each of these connectors and their configurations in the *Connectors* section.

If you followed *Create a project from scratch* instead, then to add connectors you need to modify the `Program.cs`. Any connector must implement the *OPC-Proxy interface* so independently of its implementation you can add any connector as follows:

```
// load the library at the beginning of the file
using OpcGrpcConnect;
using OpcInfluxConnect;
using opcKafkaConnect;

        .
        .
        .
// Initialize and add the connectors to the serviceManager
// before the call to "manager.run();"

HttpImpl opcHttpConnector = new HttpImpl();
manager.addConnector(opcHttpConnector);

InfluxImpl influx = new InfluxImpl();
manager.addConnector(influx);

KafkaConnect kafka = new KafkaConnect();
manager.addConnector(kafka);
```

You can find more details on each of these connectors and their configurations in the *Connectors* section.

Note: The `InfluxDB` and `Kafka` connectors will not work if the respective servers are not running.