Industrial IoT

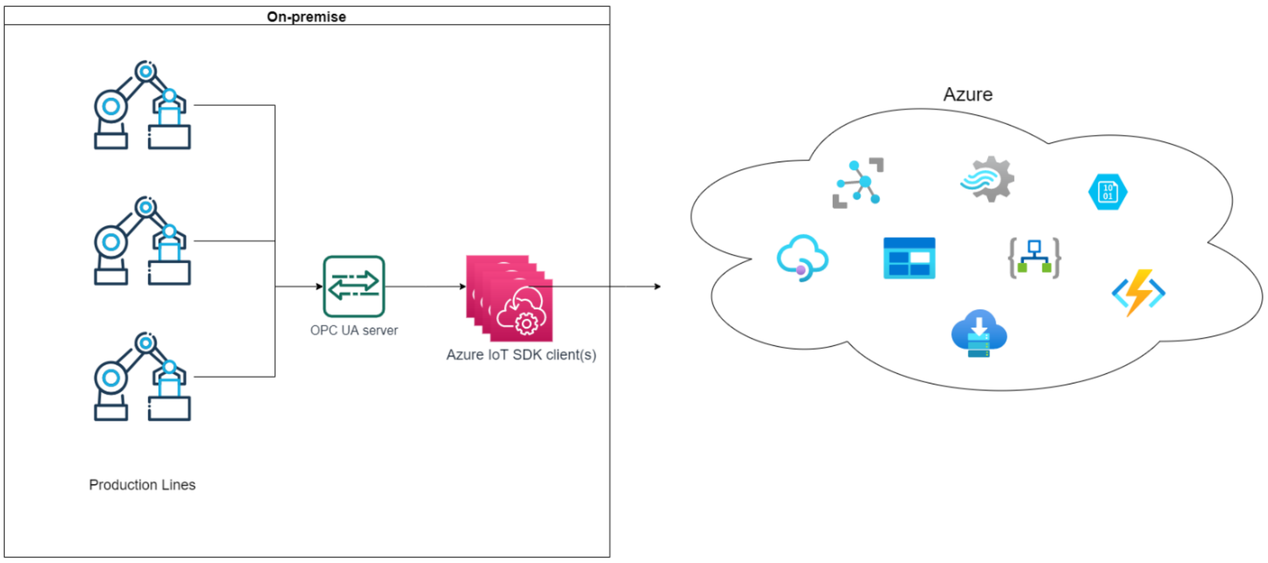
Case Study - Azure

1 Problem

Company X runs a factory with multiple production lines. Production is handled by machine operators and controlled using different, isolated systems. There's no system that allows for real-time monitoring of all production lines and most data is gathered using paper documentation.

The company wants to transform their production and as the first step they want to connect their production lines to an Industrial IoT platform. After a series of PoCs and workshops they chose Azure.

Your task is to design and develop a Production Monitoring System that will gather data from available production lines and perform basic analysis, calculations and business logic on that data.



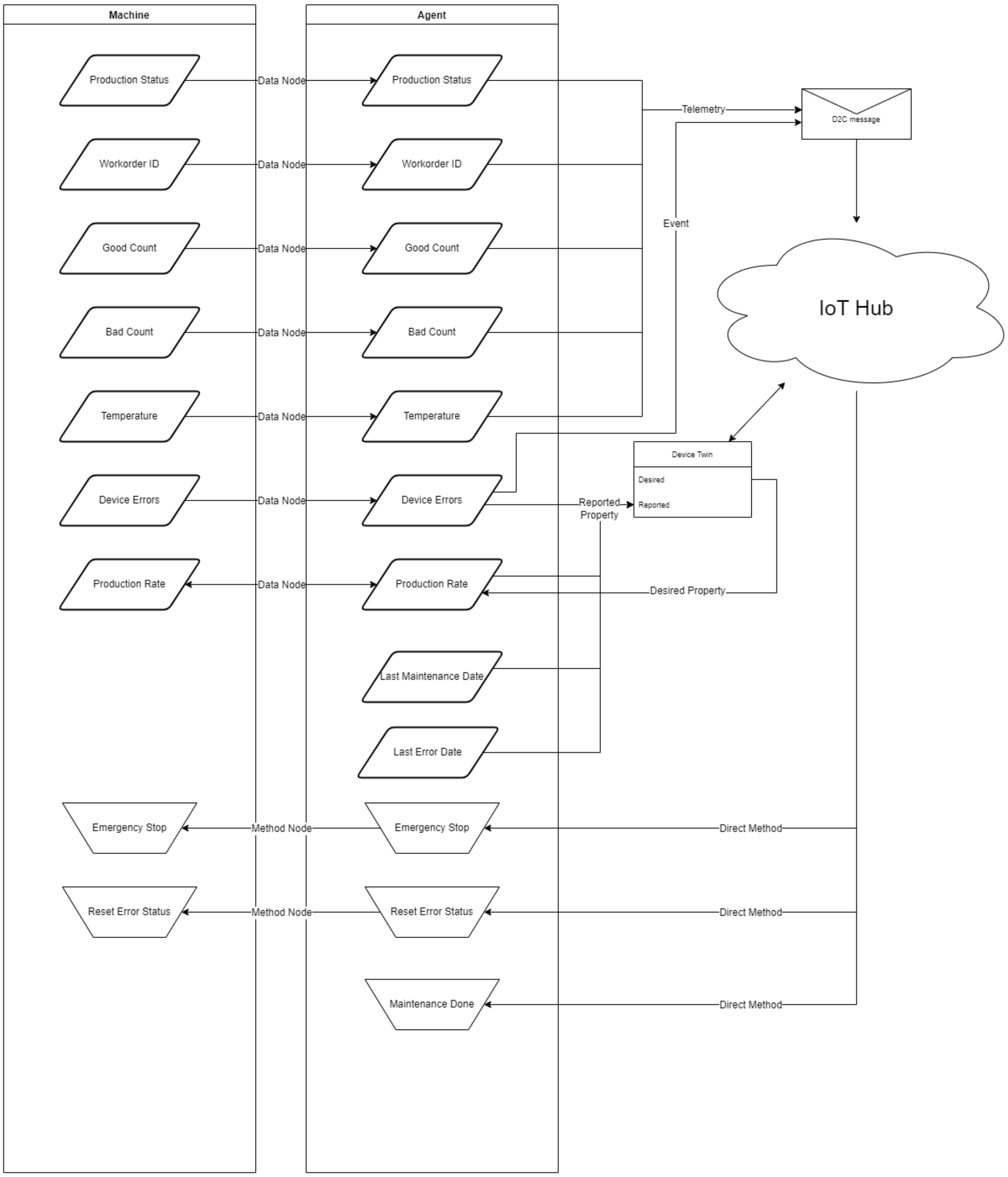
2 Technical requirements

The goal of the project is to design and develop the system in two pieces:

- Agent (may be a console app) running on-premise in the factory (can be one app instance per factory or one instance per production line – your choice)
- IoT services in Azure that handle data processing, storage, calculations, business logic, etc.

System must be able to connect multiple production lines working and transmitting data in parallel, preferably by starting just another instance of the Agent of by changing its configuration (e.g. adding another connection string).

2.1 Connectivity



A. Agent is an application running on Windows system, that:
   1. Connects to an OPC UA server to read data from connected machines.
   2. Uses Azure SDK to connect to Azure cloud IoT services and enables two-way communication

B. Every connected machine provides the following Data Nodes:
   1. Production Status - **TELEMETRY**
      - Read-only

   - Contains the current production status
     - Stopped = 0
     - Running = 1
   - Production won't run if there's a device error (see p. 7)
   - Must be transmitted to IoT platform via **D2C message**
2. Workorder ID - **TELEMETRY**
   - Read-only
   - Guid value – contains ID of currently running workorder
   - Empty when production is stopped
   - One Workorder ID is processed only by one machine (production line)
   - Must be transmitted to IoT platform via **D2C message**
3. Production Rate - **STATE**
   - Read and write
   - Contains the current/desired production rate measured in %
   - Defaults to 100%, can be set to lower value if necessary
   - Current value must be stored in the **Reported Device Twin**
   - Desired value must be read from **Desired Device Twin** and set on the machine
4. Good Count - **TELEMETRY**
   - Read-only
   - Contains the number of produced good quality items
     - Value > 0
     - Resets to zero with a new workorder
   - Increments while production is running
   - Must be transmitted to IoT platform via **D2C message**
5. Bad Count - **TELEMETRY**
   - Read-only
   - Contains the number of produced bad quality items
     - Value > 0
     - Resets to zero with a new workorder
   - Increments while production is running
   - Must be transmitted to IoT platform via **D2C message**
6. Temperature - **TELEMETRY**
   - Read-only
   - Measures the temperature in Celsius
   - Measurements are taken regardless if production is running or not
   - Must be transmitted to IoT platform via **D2C message**
7. Device Errors – **REPORTED STATE + EVENT**
   - Binary flag value:
     - None = 0 = 0000
     - Emergency Stop = 1 = 0001
     - Power Failure = 2 = 0010
     - Sensor Failure = 4 = 0100
     - Unknown = 8 = 1000
   - When value changes, a **single D2C message** must be sent to IoT platform
   - Current value must be stored in the **Reported Device Twin**

C. Every connected machine provides Method Nodes, which must be exposed as Direct Methods by the Azure IoT agent:
    1. Emergency Stop
        - Stops all production and raises the Emergency Stop error flag
    2. Reset Error Status
        - Resets all error flags to zero, allowing the machine to start production

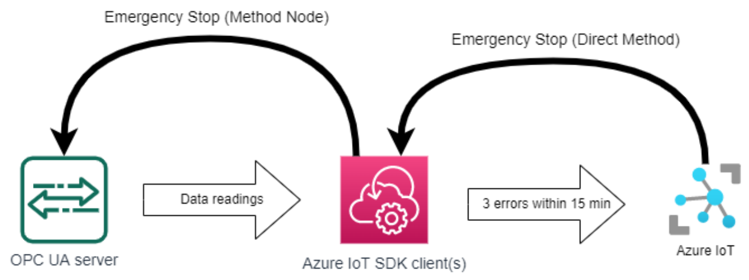
D. Additionally, the agent must store the following data in the Reported Device Twin:
    1. Last Maintenance Date – can be set to current date after executing “Maintenance Done” method on the agent
    2. Last Error Date – set by the agent when a Device Error occurs

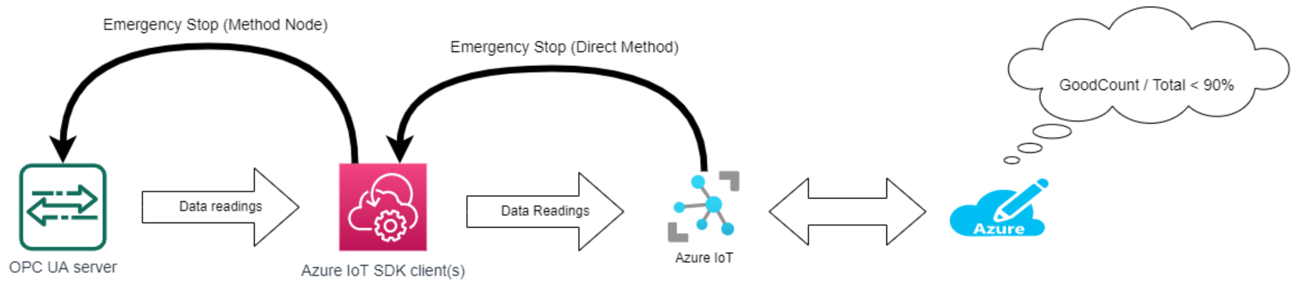
2.2 Data calculations

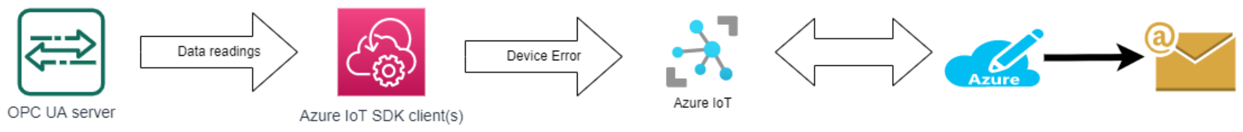
The following calculations must be performed **on the cloud** and results stored in the IoT platform:

1. Production per workorder
    - Sum of Good Count per Workorder ID
    - Sum of Bad Count per Workorder ID
2. Production KPIs
    - % of good production (vs total production) in 15-minute windows
3. Temperature per machine
    - Average temperature per machine in 5-minute windows
    - Maximum temperature per machine in 5-minute windows
    - Minimum temperature per machine in 5-minute windows
4. Errors per machine
    - Number of individual errors per machine in 30-minute windows
    - Situations, when there are more than 3 errors within 15 minutes (per machine)

2.2.1 Business logic







The following business logic must be implemented **on the cloud** (IoT platform):

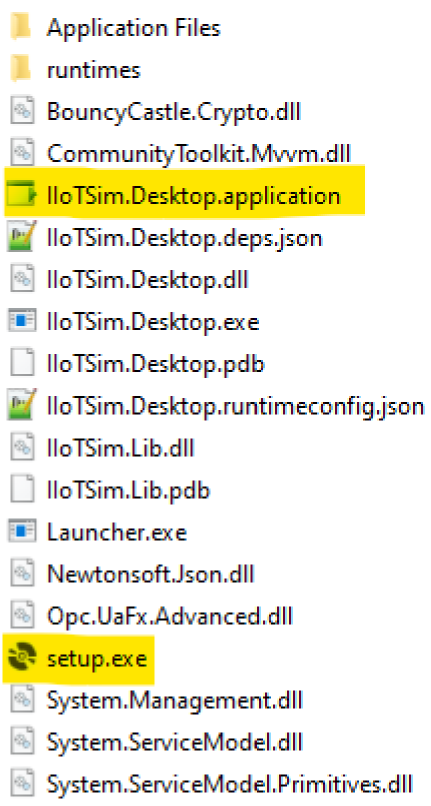
1. If there are more than 3 errors within 15 minutes (per machine):
    - Immediately trigger Emergency Stop on the machine
2. If % of good production in 15-minute window drops below 90%
    - Decrease Desired Production Rate by 10 points
3. If a Device Error occurs (of any type)
    - Send an email to predefined address

3 Device simulation

To simulate data from an actual device in Company X, run the *IIoTSim* application provided with this case study.

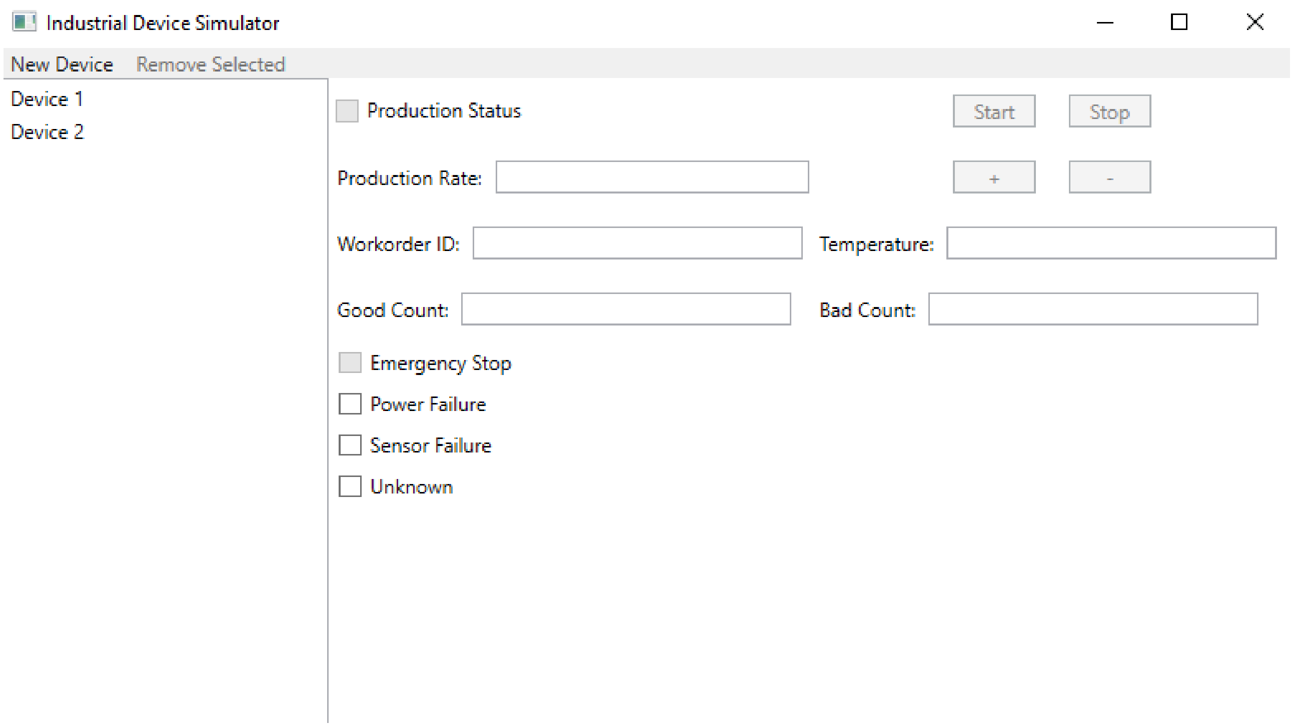
3.1 Instructions to use

1. Unpack the IIoTSim.zip package and run the *IIoTSim.Desktop.application* or *setup.exe* file.
   Follow the steps from an installation wizard (ClickOnce installer). After successful installation you'll find the IIoTSim.Desktop app in your Windows Start Menu and Control Panel.



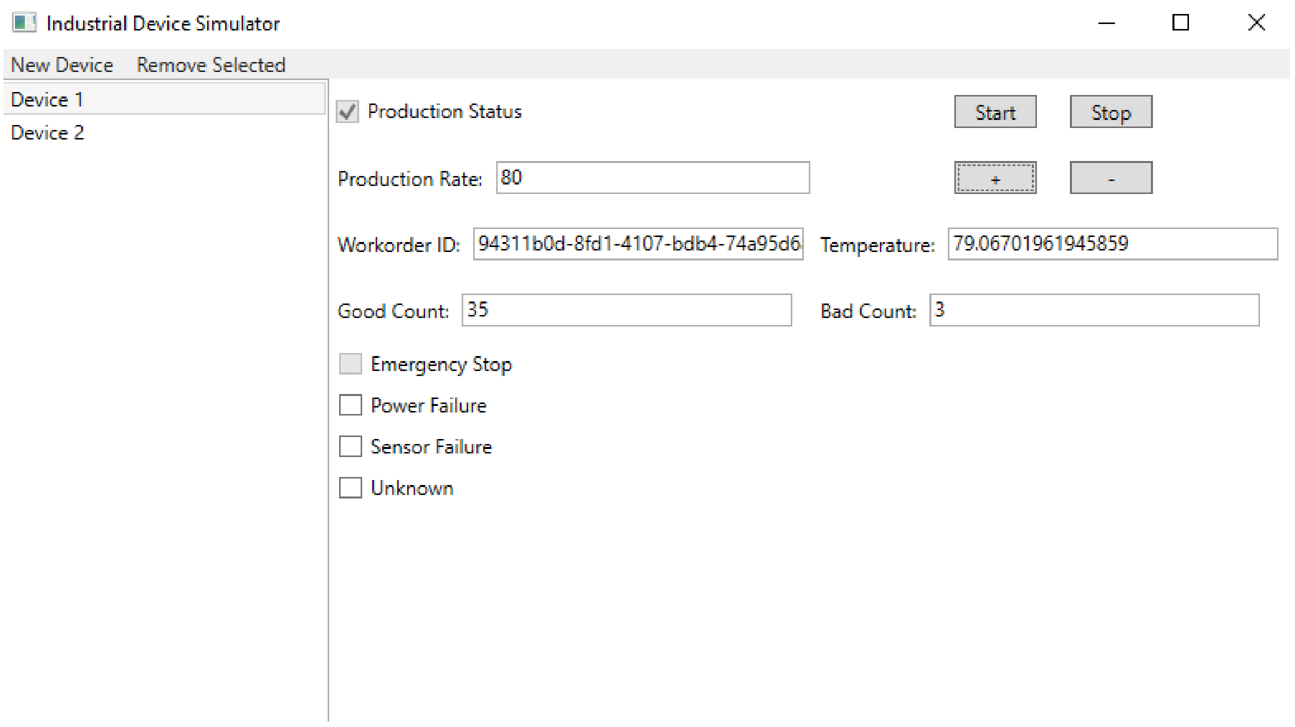
Application Files
runtimes
BouncyCastle.Crypto.dll
CommunityToolkit.Mvvm.dll
IIoTSim.Desktop.application
IIoTSim.Desktop.deps.json
IIoTSim.Desktop.dll
IIoTSim.Desktop.exe
IIoTSim.Desktop.pdb
IIoTSim.Desktop.runtimeconfig.json
IIoTSim.Lib.dll
IIoTSim.Lib.pdb
Launcher.exe
Newtonsoft.Json.dll
Opc.UaFx.Advanced.dll
setup.exe
System.Management.dll
System.ServiceModel.dll
System.ServiceModel.Primitives.dll

2. Run the IIoTSim.Desktop app.
3. Click “New Device” to create a new device. Remember that devices exist only as long as the app is running. You can also remove a device after you select it.



4. Select a device – you can now see live simulation data and change some values
   - Production Status – you can click Start or Stop to change the value;
   - Production Rate – you can increase or decrease it with +/- buttons; you can’t input a value manually
   - Workorder ID – generated automatically when production starts

- Temperature – simulated read-only value
- Good Count – simulated read-only value
- Bad Count – simulated read-only value
- Device errors:
    - Emergency Stop – sets automatically when emergency stop is triggered
    - Power Failure – can be checked in UI
    - Sensor Failure – can be checked in UI
    - Unknown – can be checked in UI



✓ Production Status

Production Rate: 80

Workorder ID: 94311b0d-8fd1-4107-bdb4-74a95d6

Temperature: 79.06701961945859

Good Count: 35

Bad Count: 3

☐ Emergency Stop

☐ Power Failure

☐ Sensor Failure

☐ Unknown

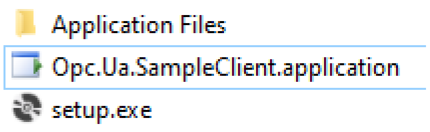
3.2 Important notes

- This app hosts an OPC UA server under *opc.tcp://localhost:4840/* URL to which you can connect with any OPC UA client.
- The OPC UA server is hosted using a trial version of *Opc.UaFx* library. This version comes with an evaluation license which has no functional limitations but the server stops after 30 minutes of use - after that the app has to be restarted.
- To develop the agent app that connects to the OPC UA server, you can use any library you want, but here are two suggestions:
    - *Opc.UaFx.Client* library – this is the Client SDK from the same maker as the library used in *IIoTSim.Desktop* app. You can find development guidelines here: https://docs.traeger.de/en/software/sdk/opc-ua/net/client.development.guide
    - *OPCFoundation.NetStandard.Opc.Ua* library – harder to use, but open source library provided by the OPC Foundation. https://github.com/OPCFoundation/UA-.NETStandard

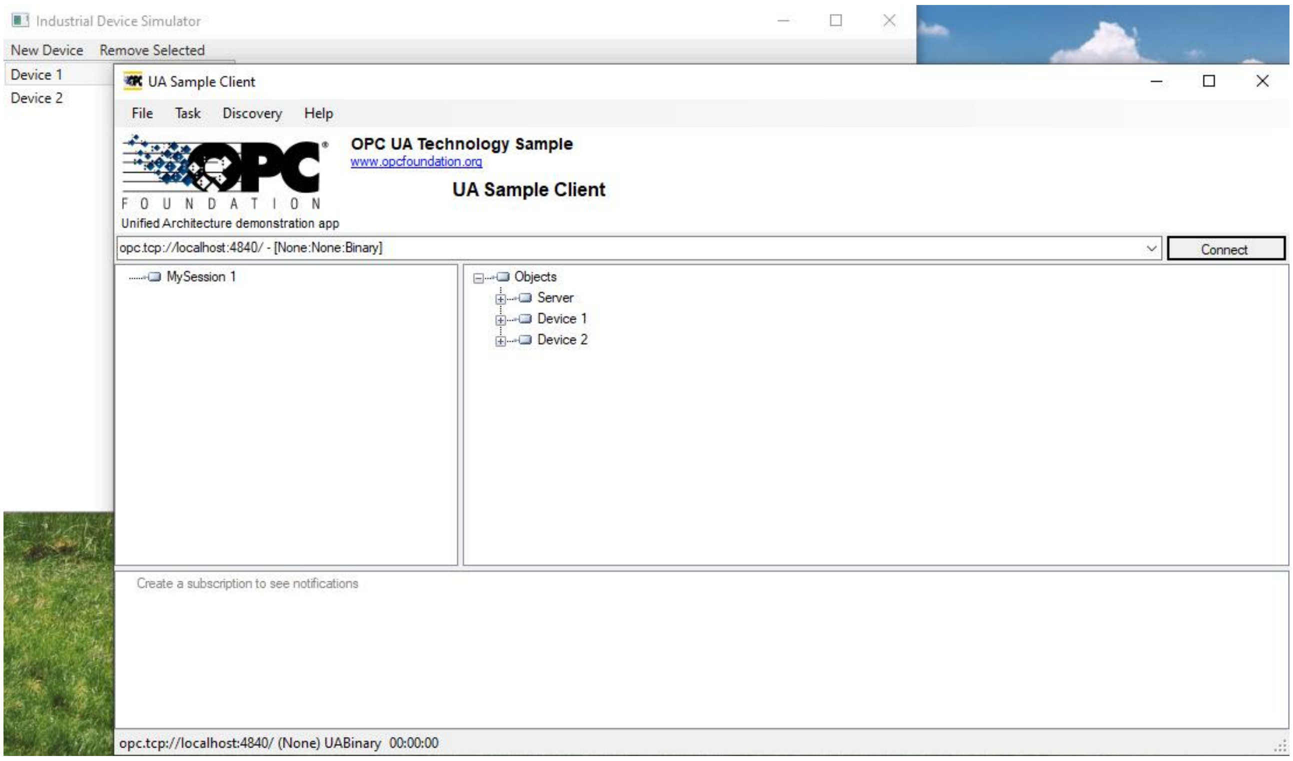
3.3 Sample OPC UA client

You can use sample OPC UA client from https://github.com/OPCFoundation/UA-.NETStandard-Samples/tree/master/Samples/Client.Net4 to connect to the IIoTSim app and preview data.

1. Install the client by running *Opc.Ua.SampleClient.application* or *setup.exe* file from the *UA Sample Client.zip* package.



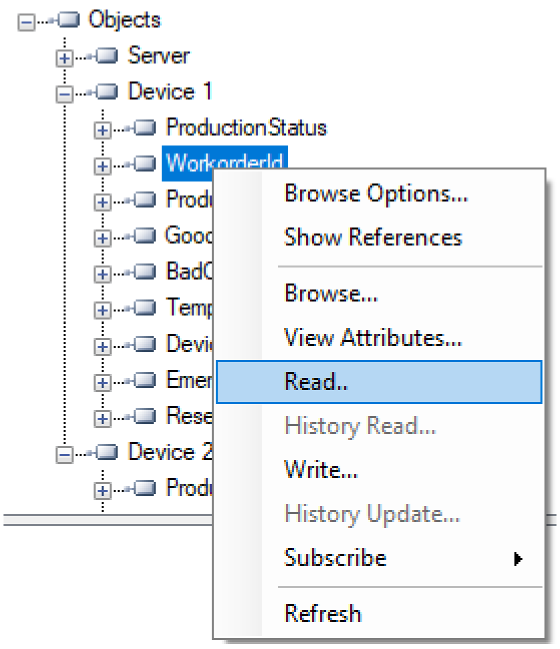
2. Run the application.
3. In the URL field put *opc.tcp://localhost:4840/* and click Connect
   - The IIoTSim app must be running at the time!
   - Confirm any pop-up dialogs

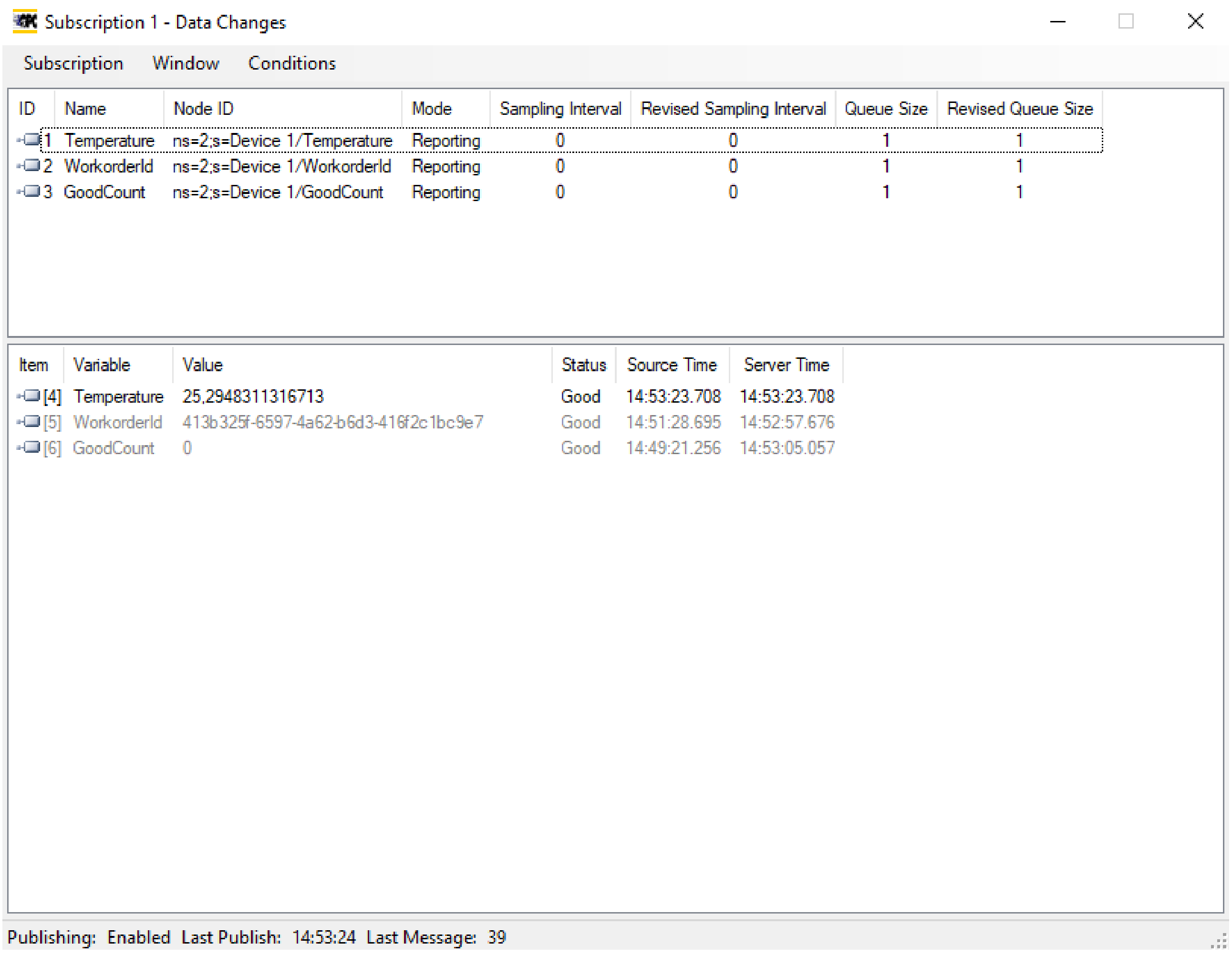


opc.tcp://localhost:4840/ - [None:None:Binary]

opc.tcp://localhost:4840/ (None) UABinary 00:00:00

4. You can read values by right-clicking a device and selecting “Read” or creating a subsctiption





Subscription 1 - Data Changes

ID	Name	Node ID	Mode	Sampling Interval	Revised Sampling Interval	Queue Size	Revised Queue Size
1	Temperature	ns=2;s=Device 1/Temperature	Reporting	0	0	1	1
2	WorkorderId	ns=2;s=Device 1/WorkorderId	Reporting	0	0	1	1
3	GoodCount	ns=2;s=Device 1/GoodCount	Reporting	0	0	1	1

Item	Variable	Value	Status	Source Time	Server Time
[4]	Temperature	25,2948311316713	Good	14:53:23.708	14:53:23.708
[5]	WorkorderId	413b325f-6597-4a62-b6d3-416f2c1bc9e7	Good	14:51:28.695	14:52:57.676
[6]	GoodCount	0	Good	14:49:21.256	14:53:05.057

Publishing: Enabled Last Publish: 14:53:24 Last Message: 39