

Educational Case Studies for Monitoring and Control of Discrete-Event Systems Using OPC UA and Cloud Applications

Erik Kučera *, Oto Haffner , Peter Drahoš  and Ján Cigánek 

Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, 812 19 Bratislava, Slovakia; oto.haffner@stuba.sk (O.H.); peter.drahos@stuba.sk (P.D.); jan.ciganek@stuba.sk (J.C.)

* Correspondence: erik.kucera@stuba.sk

Abstract: The current trend in industry is the digitalisation of production processes using modern information and communication technologies, a trend that falls under the fourth industrial revolution, Industry 4.0. Applications that link the world of information technologies (IT) and operational technologies (OT) are in particular demand. On the basis of information from practice, it can be stated that there is a shortage of specialists in the labour market for the interconnection of PLCs with information and communication technologies (cloud, web, mobile applications, etc.) in Slovakia and neighbouring countries. However, this problem is beginning to affect other countries in Europe as well. The main objective of the work was to prepare case studies suitable for educational purposes, which would address the modelling and control of a virtual discrete-event system using a PLC program and its subsequent interfacing to a cloud application. Within the scope of the work, three case studies were prepared to demonstrate the control of discrete-event system using different programming systems and their communication with the developed cloud applications. These applications are to be used for data monitoring and emergency intervention of the discrete-event system. The characteristics of the prepared case studies, which combine operational and informational technologies, predestines them for use in the sphere of education of engineers for digitalisation of production processes. They can also be helpful in research on the creation of digital twins, which represent a type of symmetry between real and virtual systems.

Keywords: discrete-event system; Industry 4.0; digital factory; system control; cloud computing; engineering education; OPC UA; Node-RED

1. Case Study No. 1: OpenPLC linked with Node-RED and Microsoft Azure

In the first case study (Figure 1), we would like to demonstrate the use of the freely available open-source application system OpenPLC [28] instead of the traditional paid PLC editors and runtimes. However, OpenPLC does not support the modern communication standard OPC UA [29], but only the conventional Modbus protocol. Although OpenPLC can communicate with Factory I/O using this protocol and implement control processes using it, we also want to send data to the cloud and possibly provide it to other clients via OPC UA. Therefore, we need a Node-RED intermediary (middleware), which is not needed for control, but we also have it there for sharing data to the cloud via MQTT and also via an OPC UA server to which we can connect a wide variety of clients. In our case, we will be testing the OPC UA client UAExpert.

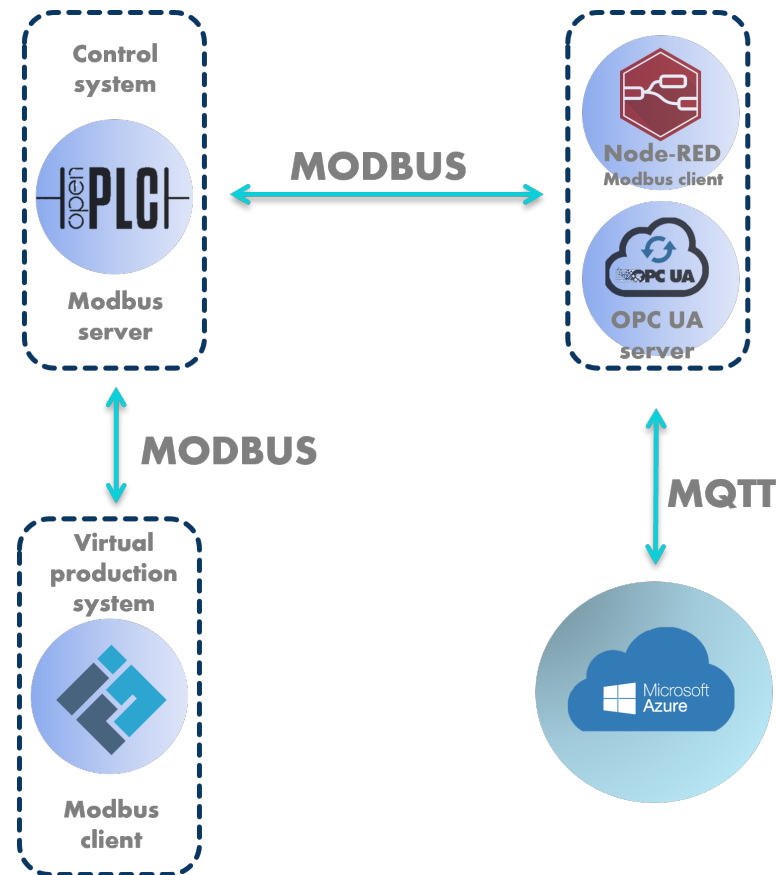


Figure 1. OpenPLC linked with Node-RED and Microsoft Azure

1.1. Discrete-Event System Specification And Behaviour

Discrete-event system could generally be defined as a system that can take on multiple states, with transitions between states being event-driven. In this case, we consider a **virtual model of a discrete-event system** in the form of a manufacturing system consisting of a production belt and a machining centre (Figure 2).

The model consists of several parts that should dynamically respond to events that occur in the system. At the **input (emitter)**, **pieces of material (semi-finished products)** are generated at certain time intervals and need to be transported to the **machining centre**. Here, they are transformed by the machining process into **products** ready for dispatch to customers. The material and products are moved by **conveyor belts**. We have six of them (C1–C6). Seven **retro-reflective sensors** (R1–R7) ensure the functionality of the conveyors. Using the emitter, we send the material onto the belt, where it is detected by the sensor R1 and then the belt C1 is started. When the product is detected by sensor R3, the corner belt C2 is triggered and then the belt C3, which takes the material to the machining centre. Here it is processed into the final product, which is passed on to belt C4, which is triggered when sensor R5 detects it. Subsequently, sensor R6 starts belts C5 and C6. The final product is recorded by sensor R7, which should also provide a count of the number of finished products. It is advisable to use an aligner to ensure that the product enters the machining centre correctly. It is also necessary to ensure that there are no collisions between pieces of material in the system and also between finished products.

Figure 3 shows an overall view of the virtual model of the production line.

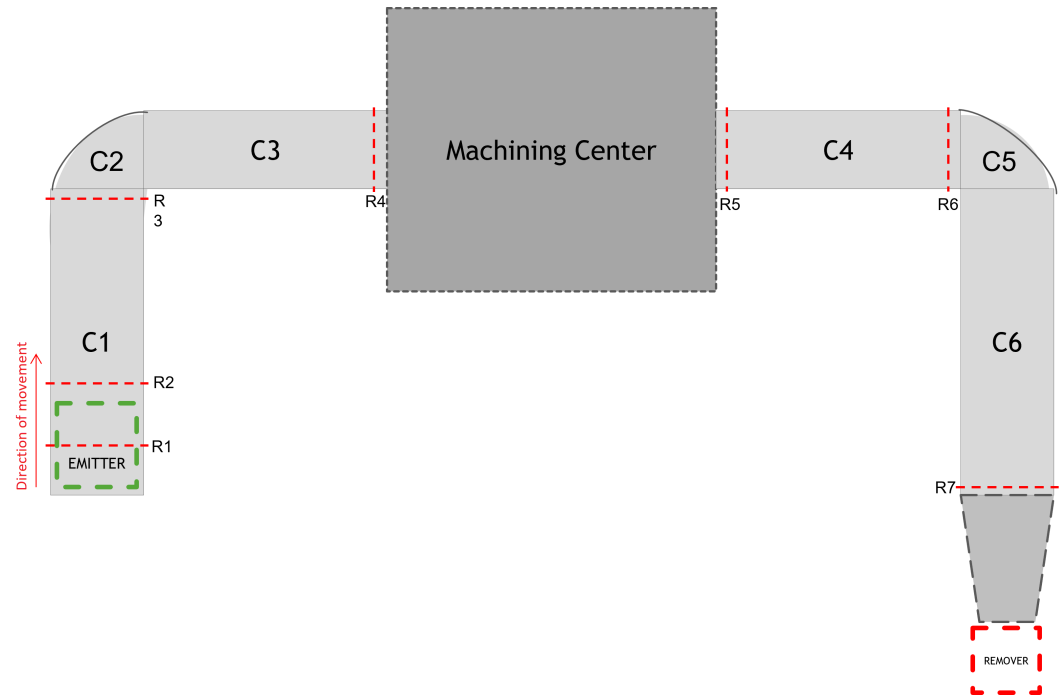


Figure 2. Scheme of production system.

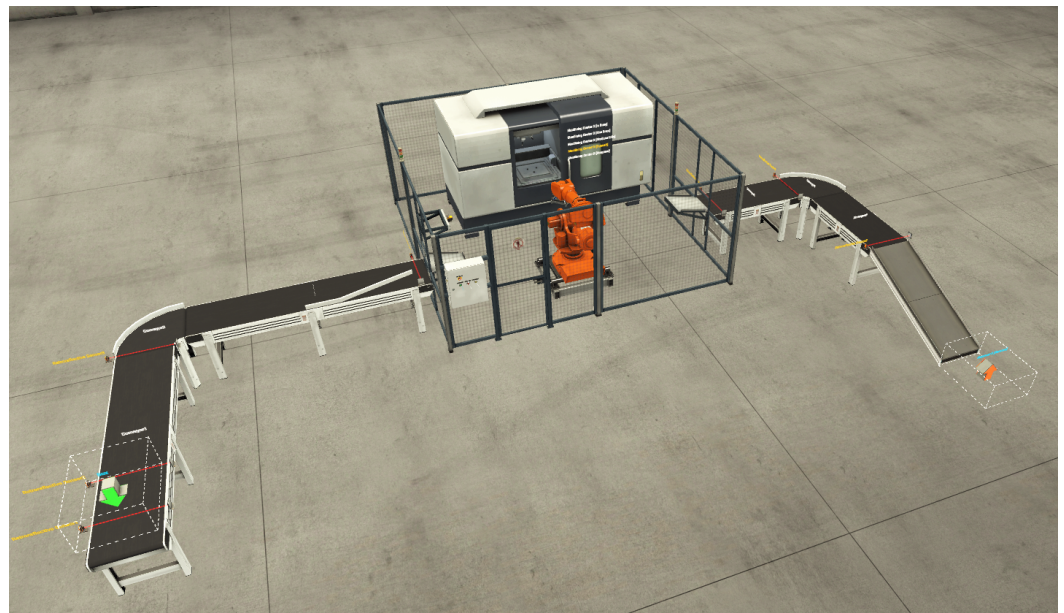


Figure 3. Overall view of the virtual production line.

To create the above production line at Factory I/O, we needed a number of parts. They will be listed in the following paragraphs.

1. **Belt conveyor** (Figure 4)—used for transporting light loads. They are available in lengths of 2, 4 and 6 metres and in analogue (we can adjust the speed of the conveyor) and digital versions;
2. **Curved belt conveyor**—used for transporting light loads and available in analogue and digital versions;
3. **Aligners**—metal structures that are attached to the conveyor to prevent the product from falling during transport. There are four types;
4. **Chute conveyor**—mostly used for dispatching items from conveyor belts;

5. **Raw Material**—metal or plastic material for the manufacturing of lids or bases. In our case we understand it as a semi-finished product that needs to be machined into a finished product;
6. **Retroreflective Sensor and Reflector** (Figure 5)—the sensor is used together with the reflector, it detects the presence of an object on the belt.
7. **Emitter** (Figure 6)—it is the entry point of the production line, which ensures the supply of production parts/raw materials to it. Raw materials are automatically generated at time intervals according to the emitter settings.
8. **Remover** (Figure 7)—removes one or more items from the scene.
9. **Machining Center**—a robot used for the production of pedestals.

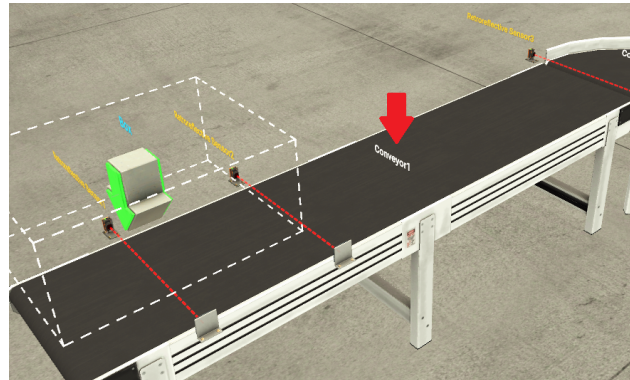


Figure 4. Belt conveyor.

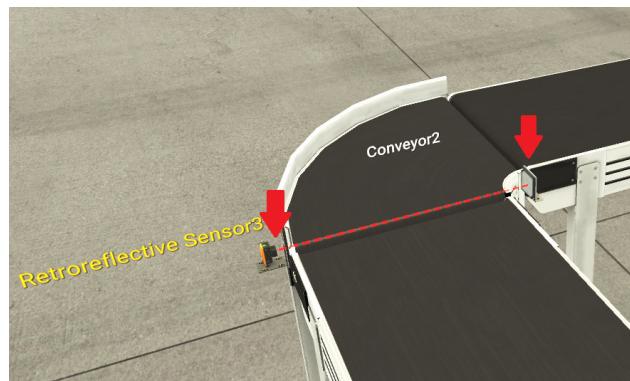


Figure 5. Retroreflective sensor and reflector.

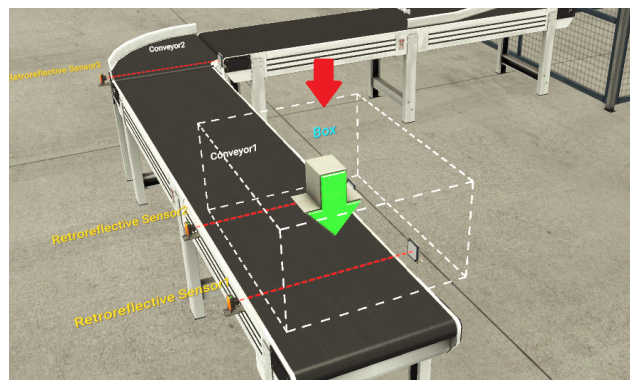


Figure 6. Emitter.

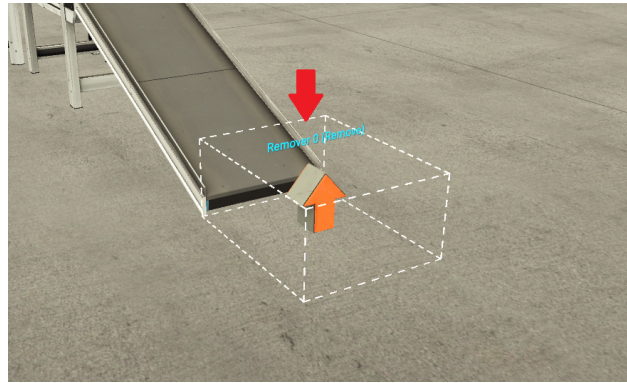


Figure 7. Remover.

1.2. Control of Discrete-Event System

In order to control the discrete-event system, we use the open-source OpenPLC editor and runtime in this case study. In OpenPLC, we use the Ladder logic. It is necessary to define the variables that are required for the creation of the control program. The variables will also be sent to the cloud later.

We use global **RetroreflectiveSensor** variables from 1 to 7 to sense the product on the belt, based on which we turn the belt on or off, or count the number of semi-products or products. The product values and subsequent calculations are handled through global variables **Result1** to 3. **Result1** counts how many semi-finished products have been dropped on the belt. **Result2** expresses how many finished end products there are and **Result3** expresses how many products are currently in production. For running the production conveyor belts, we use **Conveyor** variables 1 to 6. We also use various timers and mathematical functions. We use timers to ensure that there are no collisions. And with sensors we also count the time of the products on the belt to avoid further collisions again.

1.3. Communication between OpenPLC runtime and Node-RED Middleware

The communication takes place first between OpenPLC and Factory I/O (Figure 8). The communication is provided using the Modbus protocol because OpenPLC does not support more modern protocols (e.g., OPC UA). In this case, OpenPLC behaves as a Modbus server and Factory I/O behaves as a Modbus client. Since we want to send the data to the cloud or provide it to other clients using the OPC UA server, we need an intermediary in the form of Node-RED. Thus, Node-RED is not needed for production control itself, but we use it for the possibility of sharing data for the OPC UA server and to the cloud. We communicate from Node-RED to Microsoft Azure cloud and back using the MQTT communication protocol.

In order to properly connect the OpenPLC runtime with Node-RED, it is necessary to understand how the Modbus protocol works and what is a server and what is a client in our case. OpenPLC can even work as a server and a client at the same time, which we will not actively use. Modbus offers four types of transmitted data.

- **Discrete Input**—A single bit (BOOL) that is used for binary input (e.g., from sensors). In our case, these are addresses of type %IX. It can only be written by the Modbus server;
- **Coil**—A single bit (BOOL), which is mostly used for binary output. In our case it is addresses of type %QX. It can be written not only by the server but also by the client;
- **Input Register**—A 16-bit read-only register. It is kind of like Discrete Input, except it is not BOOL, but it is a 16-bit INT that can be unsigned or signed;
- **Holding Register**—A 16-bit register designed for both read and write. It is kind of like Coil, except it is not a BOOL, but it is a 16-bit INT that can be unsigned or signed.

As mentioned, OpenPLC can act as both a Modbus client and a Modbus server. In these modes it works simultaneously, but a different address is reserved for each. For the OpenPLC server mode we are using, it has the following addresses (see Table 1).

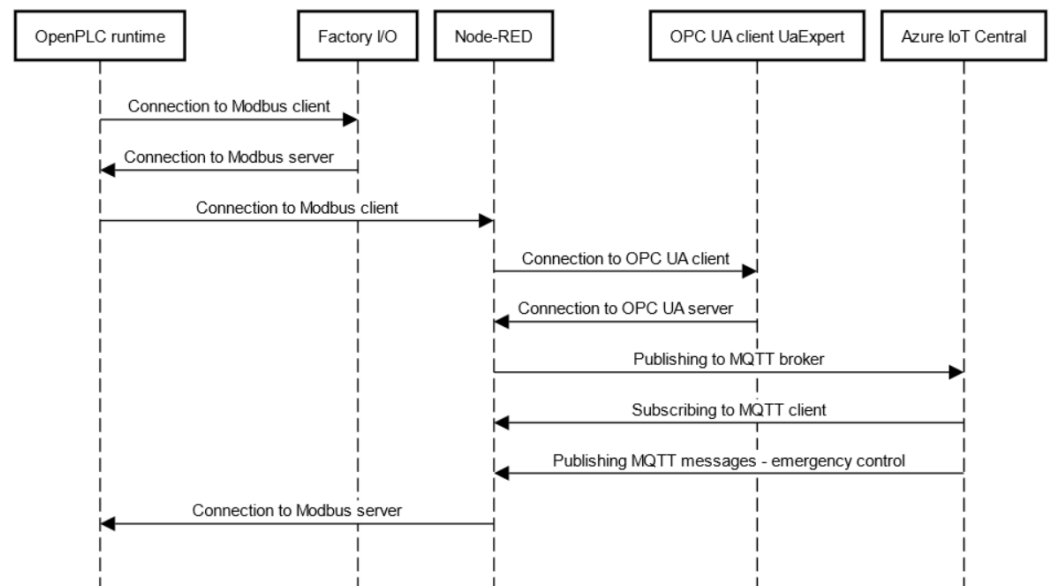


Figure 8. Sequence diagram for the first case study.

Table 1. Modbus server variables in OpenPLC engine [30].

Modbus Table	Usage	PLC Address	Modbus Data Address	Data Size	Range	Access
Discrete Output Coils	Digital Outputs	%QX0.0–%QX99.7	0–799	1 bit	0 or 1	RW
Discrete Input Coils	Digital Inputs	%IX0.0–%IX99.7	0–799	1 bit	0 or 1	R
Analog Input Registers	Analog Input	%IW0–%IW1023	0–1023	16 bits	0–65,535	R
Analog Output Holding registers	Analog Outputs	%QW0–%QW1023	0–1023	16 bits	0–65,535	RW

Next, you need to set the Factory I/O on the Modbus client. We set the localhost where we are running OpenPLC, which is 127.0.0.1. Next, we need to set the digital inputs to be used on Coils. It would be more logical to use Inputs, but since Modbus client can only write to Coils, we have to use Coils for the inputs. And last we need to set up I/O Points and there we set the inputs and outputs according to how much space we need.

In Factory I/O, the input and output variables need to be assigned correctly to the Factory I/O components (Figure 9). The addresses must be identical to those in OpenPLC.



Figure 9. Factory I/O—Modbus client.

The control program from OpenPLC editor needs to be loaded into OpenPLC runtime and run. Figure 10 shows how values are read from the OpenPLC runtime and how OpenPLC connects to Node-RED using Modbus protocol. The ability to communicate via Modbus can be obtained by installing the *node-red-contrib-modbus 5.14.1* library.

We read **inputs (from sensors)** via the node called **Modbus Read - %QX0.0-7** and read **outputs (actuators)** via **Modbus Read - %QX0.7+** node. We start with inputs from address 0, i.e., %QX0.0. The outputs start from address %QX1.0. The Modbus node always reads a whole byte, which is alright in this case since we have exactly eight input variables. The variables are of type BOOL (true/false). Modbus Read works as a client and we need to connect it to the server. We will connect it to a server that we have called OpenPLC local and set the corresponding address 127.0.0.1 and port 502. Next, it is needed to specify that we are going to read coils, which is a standard Modbus protocol command (*FC1: Read Coil Status*). In the case of our input variables, we set the address to 0, since we are reading from %QX0.0 (so in the case of %QX1.0, it would be 8). We set the quantity to 1, since we are reading 1 byte. We set the poll rate to 2 s, which means that the value is read every 2 s.

We used a similar procedure on **Modbus Read - %QX0.7+**, where we read output variables from PLC address %QX1.0 and our Modbus address is 8.

In PLC program there are also 3 values of INT type, which are stored in registers that have different addresses than the coils (these are of BOOL type). These are, for example, the number of finalised products. These values are read using the **Modbus Read Holding** node.

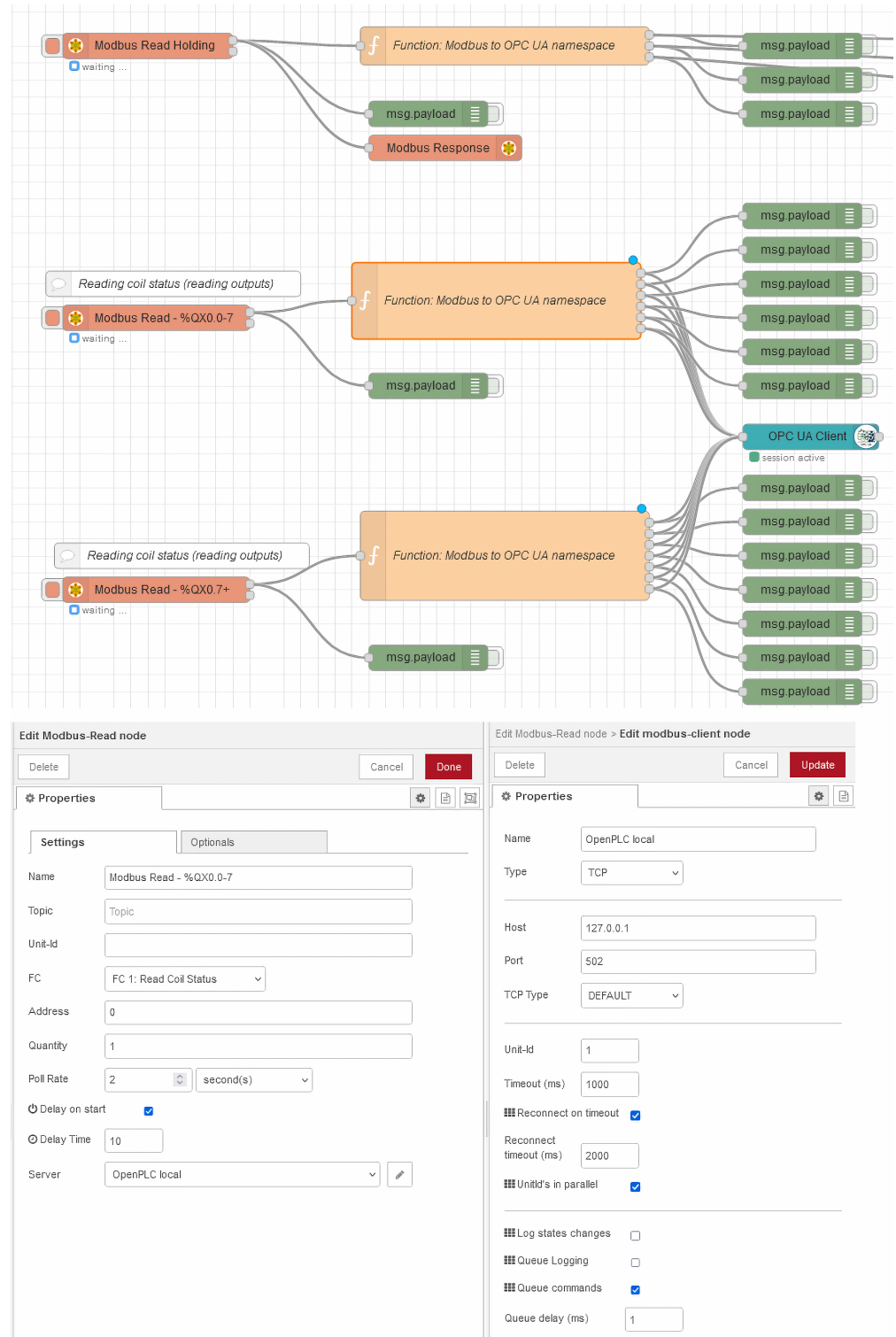


Figure 10. Node-RED as Modbus client.

1.4. OPC UA Server and Client

Since we want to offer the values from the production line to other users who may have OPC UA clients, it was necessary to implement an OPC UA server in Node-RED to provide these data.

As already mentioned, OpenPLC runtime cannot function as an OPC UA server, as it is a free open-source tool. This is the domain of more advanced PLC solutions such as

CODESYS or Siemens TIA Portal. Therefore, the OPC UA server will be created using Node-RED, to which the data arrive from OpenPLC runtime via Modbus protocol.

Library *node-red-contrib-opcua* 0.2.256 was used to create OPC UA server.

To **create a server**, it is necessary to use the **OPC UA server node**, where the port (in our case 53,880) and optionally its name are set. We use the default name. It is also possible to set authentication options, as OPC UA communication standard supports multiple security profiles. We use anonymous access for clarity (and since we are running on localhost).

The creation of the server is also related to the creation of **address space**, i.e., variables that will initially be empty or have a predefined value, and later we will fill these variables with values that OpenPLC runtime sends using Modbus protocol.

To keep our inputs and outputs clearly separated in the address space, we have created folders *FIOOutputs* and *FIOInputs* in it.

It should be noted that the individual directives that we send to the OPC UA server node are sent using the *Inject* node type. Thus, these directives need to be set to execute automatically when the Node-RED program is started, and it is logical that the timing needs to be set so that the directives that create the folders are executed first, and then the variables are created. This will successfully create an OPC UA server with the desired address space.

We will use the following directives (but there are several supported): **addFolder**, **setFolder** and **addVariable**.

The directives are bound to the *msg.payload* of messages in Node-RED and the content itself to the *msg.topic* of messages. This can be seen in Figure 11, which shows the creation of the *FIOOutputs* folder. In *msg.payload* there is a command to add the folder and in *msg.topic* there is the defined namespace and folder name.

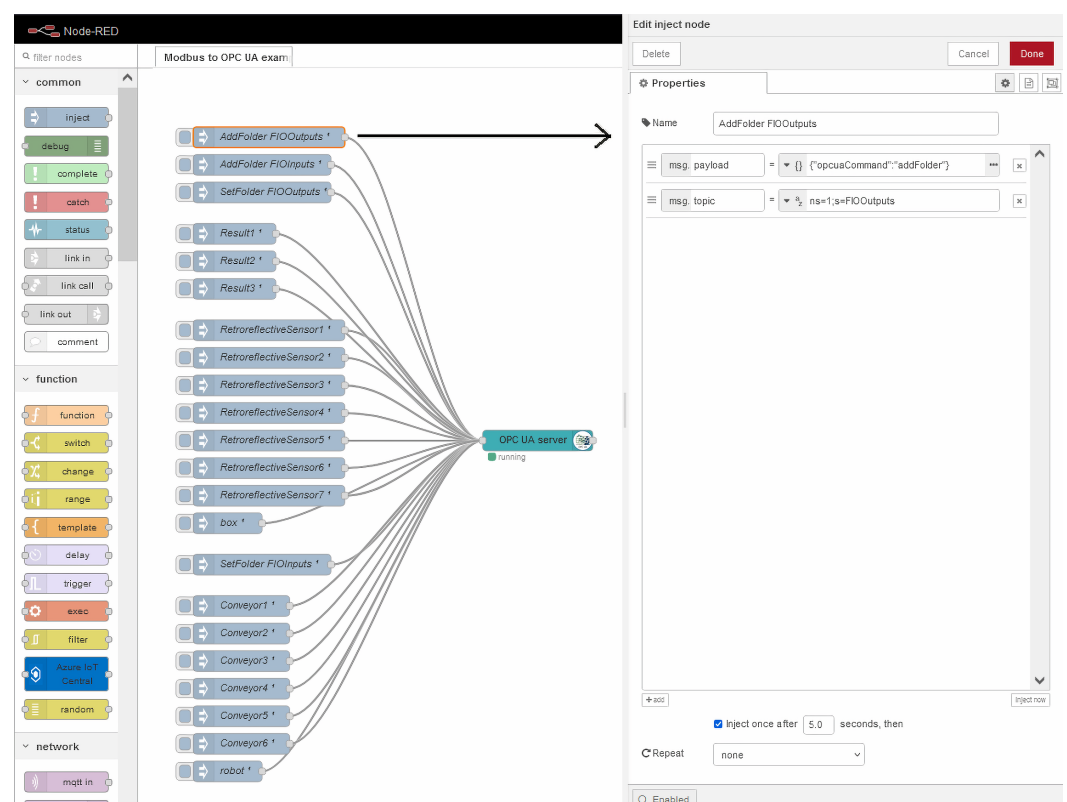


Figure 11. Creating OPC UA server and its address space in Node-RED.

Adding the *RetroreflectiveSensor1* variable would look like this, and it is obvious that the variable's data type is also set:

- `msg.payload: {'opcuaCommand':'addVariable'};`
- `msg.topic: ns=1;s=RetroreflectiveSensor1;datatype=Boolean.`

In order to be able to populate variables with data, we needed to create a custom function in JavaScript, since in Node-RED it is not necessary to use only the built-in nodes, it is also possible to create your own.

Notice in Figure 10 the node named **Function: Modbus to OPC UA namespace**, i.e., the node representing our own function. Specifically, we will describe the node that is connected to the node *Modbus Read- %QX0.0-7*. The *Modbus Read- %QX0.0-7* node sends an array of 8 values (of type BOOL) to the *Function: Modbus to OPC UA namespace* node. Figure 12 shows the contents of the node. For each value received from Modbus, we have pre-prepared an empty variable (*msg0*, *msg1*, *msg2*,...) where we store the elements of the array. In their payload we store the value itself (of type BOOL), but the more interesting part is *msg.topic*. Here, according to the documentation of the OPC UA client installed in Node-RED, we need to define which variable in the OPC UA address space the variable will go to. Thus, we have defined the namespace ID (abbreviated *ns*), the variable name and the data type that matches the input variables obtained from Modbus server.

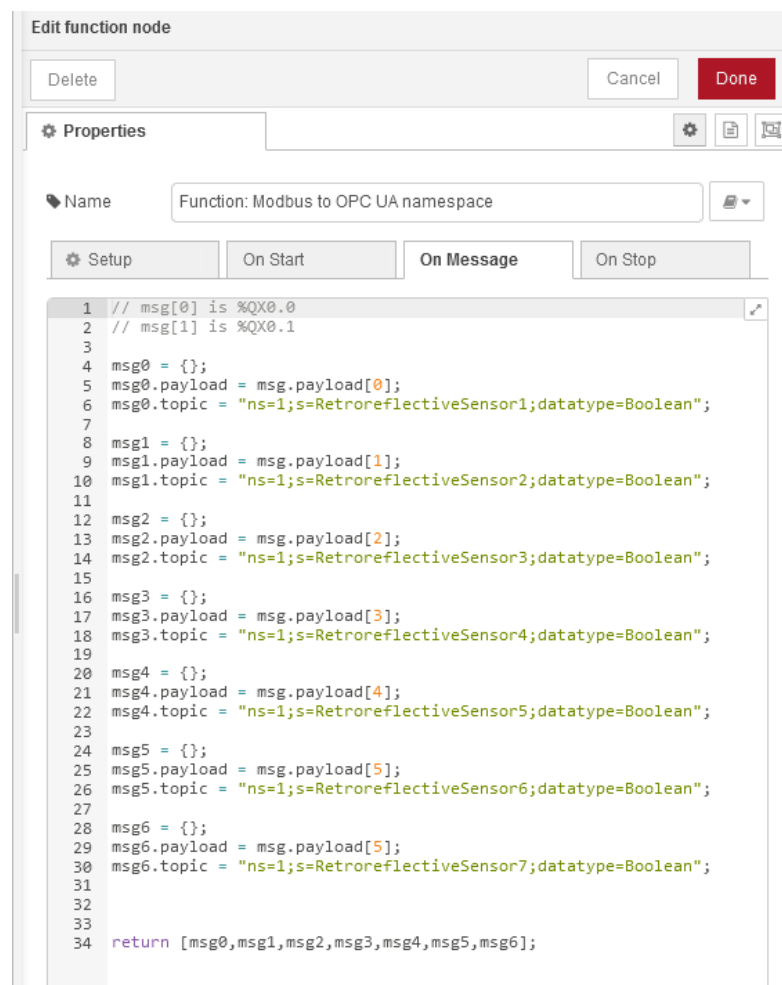


Figure 12. Function for filling OPC UA server with values obtained using Modbus protocol from OpenPLC—*Function: Modbus to OPC UA namespace*.

Then the *Function: Modbus to OPC UA namespace* node is connected to **OPC UA client** node, which stores the values in our OPC UA server.

We can also provide values from our OPC UA server to clients other than the client installed in Node-RED. Theoretically, this could be a client that does not have to run only on a computer, but also on a smartphone, tablet, in the cloud, etc.

1.5. Application in Microsoft Azure Cloud

One of the main objectives of the work was the implementation of a cloud application, which could be used to monitor discrete-event production system—to monitor the values of selected variables, to visualize the data appropriately, to process them efficiently and, if necessary, to intervene in the system.

At the beginning, it was necessary to determine what should be in the cloud application. The **functional requirements** were then determined:

- Display of read values and action buttons in a dashboard;
- Communication with Node-RED using the MQTT protocol;
- Display of the number of manufactured (final) products handed over for dispatch;
- Displaying the number of semi-finished products (pieces of raw material) that have entered production;
- Displaying the number of semi-finished products / finished products that are currently on the conveyors (or in the system as such);
- Graphical representation of the current temperature in the production hall;
- Simple processing of the current temperature values in the production hall in order to raise an alarm if the temperature rises above a certain value;
- Possibility of emergency intervention in the system - suspension and start-up of the production line.

The characteristics defined above are intended to describe what the application system (cloud application) being created will be able to accomplish. However, it is necessary to identify a second set of requirements that will not address the functionalities of the application. These will be the so-called qualitative, i.e., **non-functional requirements**:

- The cloud application should have a simple and intuitive user interface;
- Individual displayed variables should be part of a suitable object model, assuming appropriate use of Microsoft Azure cloud components;
- The application should allow for easy extensibility by displaying additional variables, or the possibility of adding additional production lines, each with its own panel;
- In terms of language internationalisation, the application should use English.

After the design, it was necessary to move on to the actual implementation. The most effective way is to use PaaS (Platform as a Service) services, i.e., to use ready developer tools to create your own cloud applications. The original plan was to bundle together multiple PaaS services, such as Azure IoT Hub for monitoring and connecting devices [31], Power BI for data visualisation [32], Azure Stream Analytics for data analytics [33], and Azure Functions for event callbacks [34], for example. During the course of the work, it was determined that the most effective solution would be to use the relatively new comprehensive aPaaS (application Platform as a Service) service **Azure IoT Central** [35], which combines the aforementioned functionalities.

Microsoft Azure IoT Central is used to connect and manage devices on a large scale and provides reliable data for business statistics, so it can be classified as an enterprise resource planning (ERP) system. It incorporates multiple PaaS services to create easily configurable, comprehensive and secure IoT solutions. A web-based user interface allows you to quickly connect devices, monitor device status, create rules, and manage millions of devices and their data throughout their lifecycle [35].

Azure IoT Central works similarly to Azure IoT Hub based on device twins, with each device based on a template. **Device template** is a so-called blueprint that defines the characteristics and behaviour of a device type. We then connect these devices to our application. For example, we can define the telemetry that a device sends so that IoT Central can create visualisations that use the right physical units and data types.

The device template we created is called **template-factoryio**. It contains the device models that we define for integration with our application. Each model has a unique ID and we also implement capabilities or semantic type for each model. The semantic type enables IoT Central, which can make some kind of assumption about how to treat the value.

The capabilities we can assign to our models are:

- *properties*—data fields that represent the state of the device;
- *telemetry*—telemetry (measurements) from sensors;
- *command*—methods that users can execute on the device (e.g., control commands).

The structure of our device named *conveyor-system1* is as follows, where the first entry is the system name and the second entry is the capability type: *turnOn (command)*, *turnOff (command)*, *produced (telemetry)*, *entered (telemetry)*, *on_line (telemetry)*, *temperature (telemetry)*, *location (property)*.

The device connection is handled using SAS (Shared access signature) authentication. We connect our device model in the cloud to Node-RED using scope ID, Device ID and Primary key data.

We can then view the data that our cloud application receives. Notice Figure 13, where we can see three values that our cloud is receiving—*Totally produced*, *Entered on line* and *Currently on line*. *Totally produced* means how many products have been produced and submitted for shipment. *Entered on line* is the number of semi-finished products (pieces of raw material) placed on the conveyor belt. *Currently on line* is the number of actual semi-finished and finished products on the belt. There is also a map that can show where a given production is taking place. Next to it is a graph that shows us the air temperature in that factory. Since our production line is not real, we just simulate the temperature by generating random values in Node-RED in the range of 18 to 25. We can also send a signal from Node-RED during the application run that the air temperature is 100 degrees Celsius. In this case, we have implemented an **event (alarm)** on the Azure IoT Central side, which ensures that an email is sent to the authorised personnel. This demonstrates the basic form of **data processing and evaluation** based upon which the event is executed. Events in Azure IoT Central can be selected from predefined types or custom functions can be programmed using Azure Functions (serverless architecture).

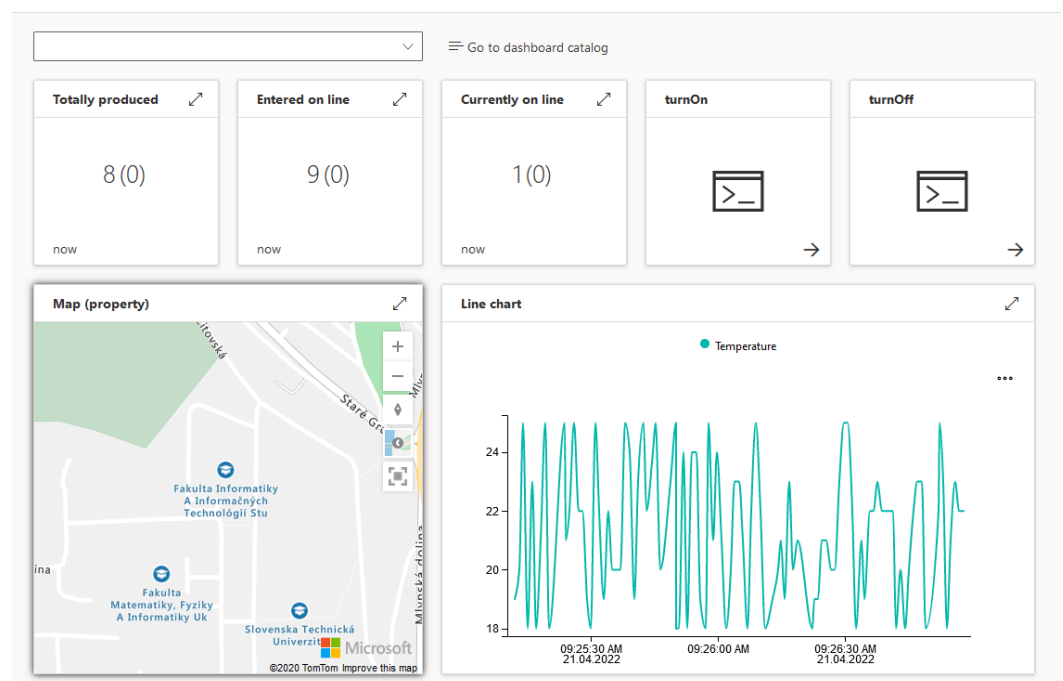


Figure 13. Dashboard in Azure IoT Central.

We will briefly describe how the data are sent in Node-RED, using the example of the generation of temperature values (Figure 14). We start by using the *timestamp* node, which provides us with the cyclic execution of a given program flow. Next, we use the *random* node (named *Temperature* in Figure 14), which generates random integers from

18 to 25. This node goes into its own *temperature* function, where we insert an identifier into the *msg.payload* of our message that is identical to the identifier of the temperature variable in the cloud on the given device model (twin). We also have an *inject 100* node ready, which we can manually push to send a temperature value of 100 degrees Celsius to simulate a fire on the production floor. Finally, the whole branch goes into the *Azure IoT Central* node where the device ID on the cloud and the access key is defined. In addition to this, the communication protocol is also defined here, in our case it is MQTT. One can also communicate over AMQP or HTTPS. We use the *azure-iot-central 1.6.0* library.

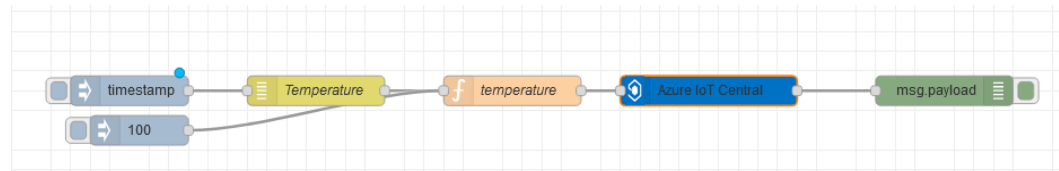


Figure 14. Node-RED data sending flow (air temperature).

We can also **intervene** in our virtual production system using the cloud application. We demonstrated this by implementing the ability to emergency **pause and start the line**. In Figure 13, we can see the buttons in the upper right corner that provide this.

First of all, it is needed to configure in Azure IoT Central node what commands this node listens for. In our case, these will be *turnOff* and *turnOn*. Another important element is the JavaScript functions that will respond to these commands from the cloud application (Figure 15). These functions must be registered in the context of our Node-RED flow (*flow context*). This is carried out (in the case of line suspend) using command `flow.set('turnOff', turnOff);`. The Azure IoT Central node takes care of the rest. After the JavaScript functions, we have other nodes connected to provide us with pause or start links. The command is sent to OpenPLC runtime using Modbus protocol, so we use nodes of type *Modbus Write*.

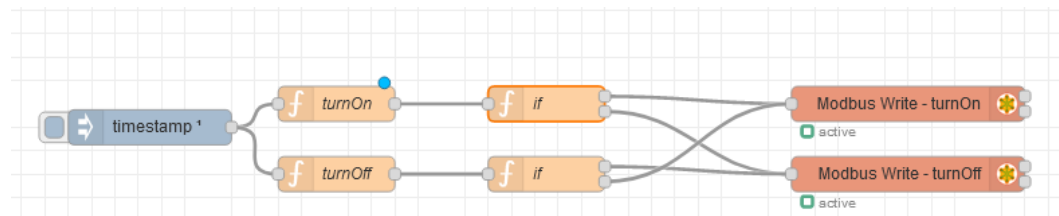


Figure 15. Flow in Node-RED ensuring the pause and start of conveyor belt.

2. Case Study No. 2: CODESYS Linked with Node-RED and Microsoft Azure

In the second case study (Figure 16), instead of using a free and open-source PLC editor, we decided to use a comprehensive automation solution called CODESYS, which provides us with a connection to Factory I/O using the modern OPC UA communication standard. OPC UA server will run through CODESYS runtime and Factory I/O will act as OPC UA client. For control purposes, we do not need an intermediary (middleware) in the form of Node-RED. However, we will use Node-RED for capturing data and sending it to the cloud and it will also be used to provide emergency stop and start of production line that can be made by the user through the cloud application. The cloud application will again be implemented using Microsoft Azure and the communication will be secured by MQTT protocol. CODESYS can be used for academic purposes, but the limitation is that the runtime can only run continuously for 1 h in free mode, which would be very limiting in production.

The specification and behaviour of discrete-event system is the same as in the first case study.

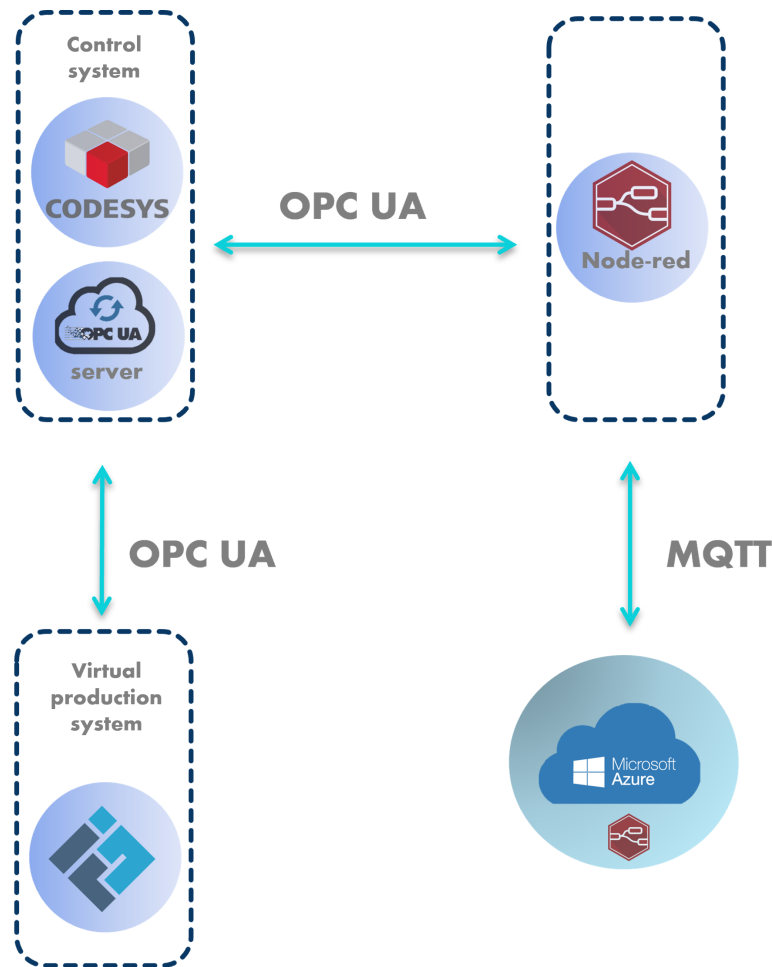


Figure 16. CODESYS linked with Node-RED and Microsoft Azure.

2.1. Control of Discrete-Event System

As in the first case study, we need to control discrete-event system. In our first case study, we used open-source and runtime OpenPLC. Now it is replaced by CODESYS. We use the Structured text language to declare variables. The control program is written by Ladder diagram.

The main variables in the control program are the same as in the first case study. However we have also added additional variables that are related to the runtime measurement of the control system. In addition, compared to the first case study, we have the CYCLE_TIME block, which is of type “Function module”. We inserted this CYCLE_TIME using the external library OSCAT_BASIC version 3.3.4.0. We use it to measure how long the control system has been running for us.

2.2. Communication

The most important aspect of communication (Figure 17) in this case is the communication between CODESYS and Factory I/O, as it provides the control process. The communication is provided by the modern OPC UA protocol. CODESYS will now behave as OPC UA server and Factory I/O will behave as OPC UA client. Thanks to the existence of OPC UA server, we can view the variables in any OPC UA client. We will use Node-RED again in this case to share data to the cloud, but it is not needed for system control. We will communicate with the cloud via MQTT protocol.

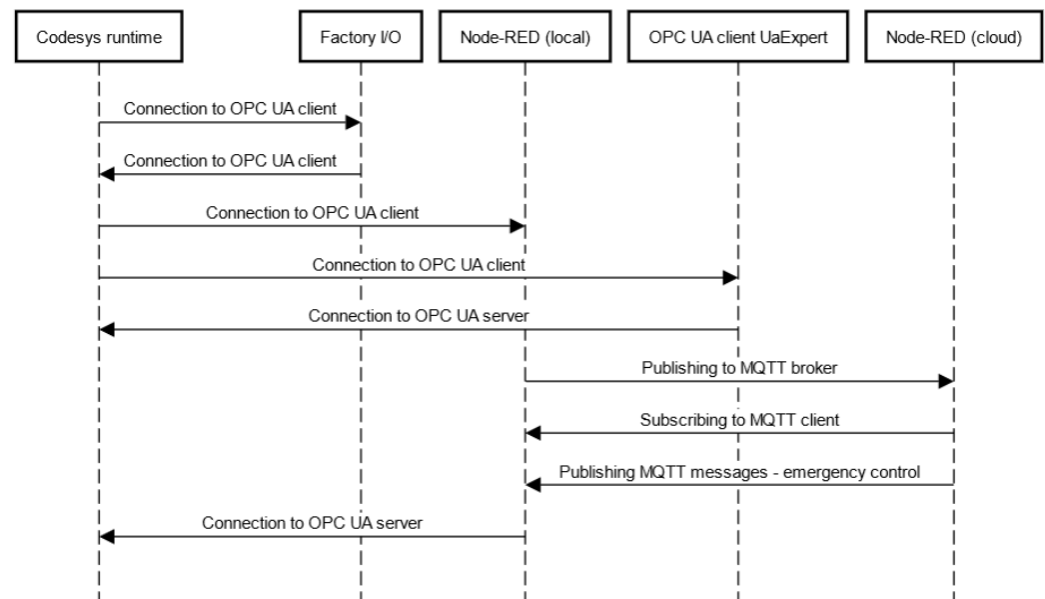


Figure 17. Sequence diagram for second case study.

2.3. OPC UA Server in CODESYS

CODESYS is relatively easy to work with. It is important to note that we need to name the variables so that Factory I/O can easily identify them and filter out the ones it needs. The entire OPC UA address space also contains a large number of different configuration and status variables that come with the CODESYS runtime. That is why all our useful variables have *FIO* prefix. We do this in order to be able to recognize our variables and make them easier to read, and most importantly, to be able to retrieve them easily in Factory I/O. In the Symbol configuration of the CODESYS project we select these variables, which actually specifies that they will be offered by the OPC UA server. Then we initialize CODESYS Control Win PLC, which is a softPLC running under Windows. Then, in the CODESYS project, we connect to this runtime and load a program into it. The connection is made by scanning the network and selecting the control unit, which can be the aforementioned softPLC or also a classic hardware PLC unit.

In Symbol configuration, we can click on our entire PLC_PRG program to mark all variables (Figure 18). In this way, all the variables we use are available in the OPC UA server. We then build and run the program.

In Factory I/O we open our scene and set the OPC Client DA/UA as driver in the configuration. We will specify *opc.tcp://localhost:4840* as the server and set “FIO” as the variable filter, this will obtain our variables from the OPC UA address space as we mentioned above. We assign all the variables and the communication between Factory I/O and CODESYS is implemented (Figure 19).

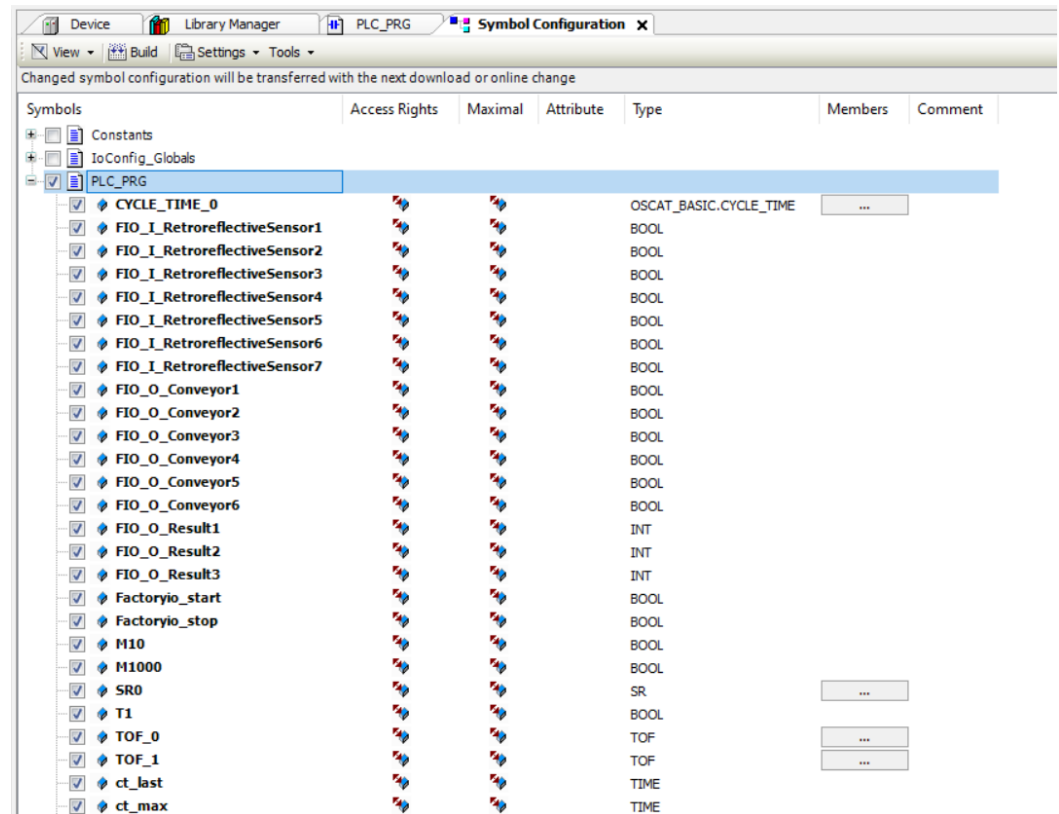


Figure 18. Symbol configuration.

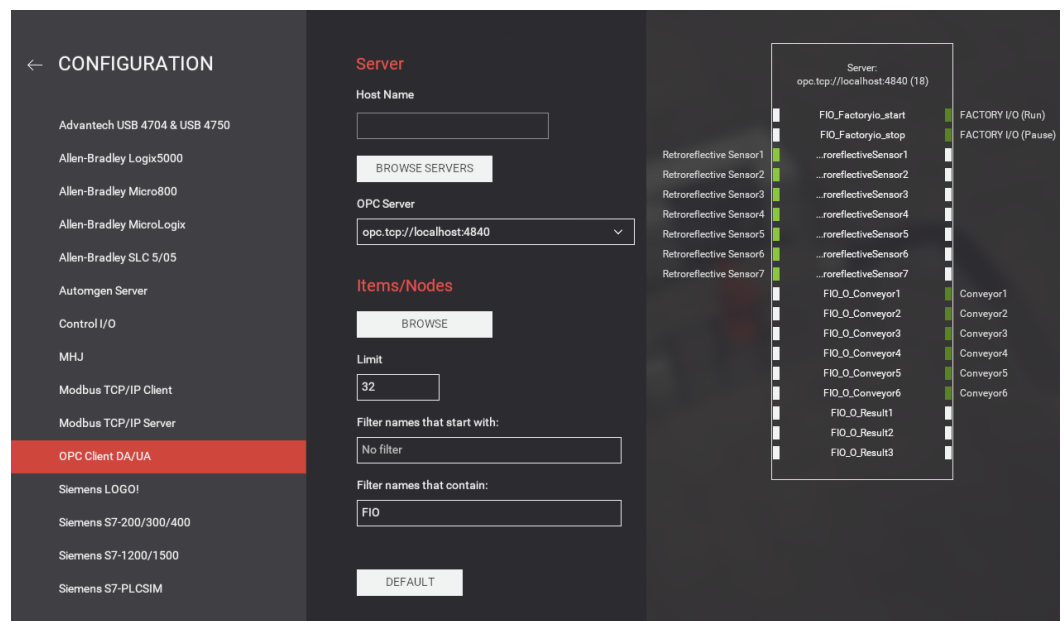


Figure 19. OPC UA Client Factory I/O.

2.4. Connection between CODESYS and Local Node-RED

After creating and running OPC UA server using CODESYS, we want to obtain the data using OPC UA to Node-RED, which runs on localhost (like OPC UA server). This will be provided by OPC UA Client node, where we set the same address as in the previous text (opc.tcp://localhost:4840). This way Node-RED acts as the OPC UA client. We accept all the variables we want to work with (Figure 20). On the left in Figure 20, we see nodes of type *Inject* querying each variable every second, defining the variable name in the *msg.topic* of the given message. The variable name is quite complex in the CODESYS OPC UA

address space. We can see the names, and also the data, clearly in any OPC UA client, for example. There we can read that the ID of one of the variables is *ns=4;s=|var|CODESYS Control Win V3 x64.Application.PLC_PRG.FIO_I_RetroreflectiveSensor1* and based on this ID we can query the variable in Node-RED.

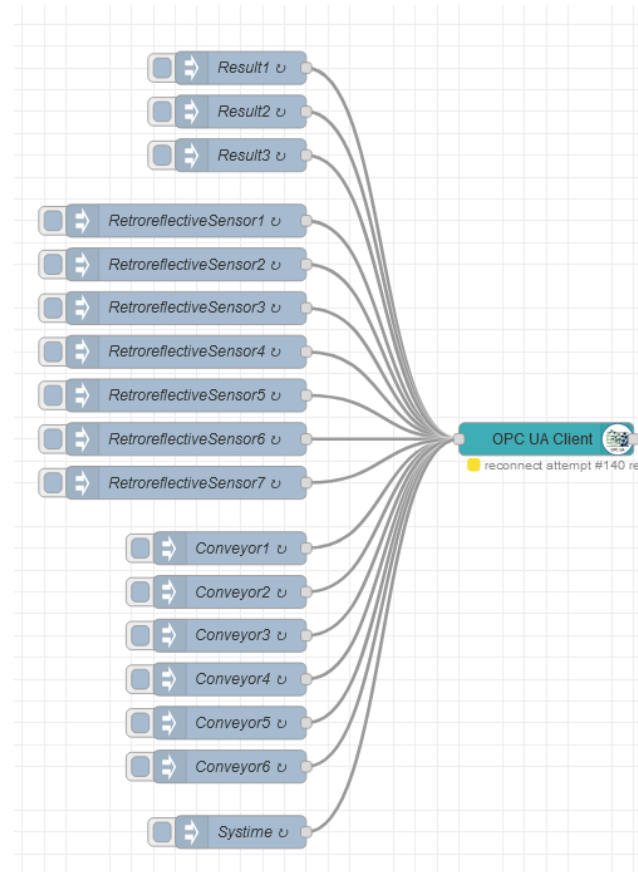


Figure 20. Local Node-RED—OPC UA client.

It should be clarified the reason for running Node-RED on localhost. Similar to the previous case study, we want to use the cloud for discrete-event system monitoring and emergency intervention. Since our OPC UA server is running locally and we do not have a public (static) IP address, we will use localhost Node-RED to send data to the cloud and receive data from the cloud.

We will send our variables to the cloud using MQTT protocol. We will use *MQTT-in* node (acting as a subscriber) and *MQTT-out* node (acting as a publisher). These nodes will connect to the broker (server) that is implemented in the cloud. Specifically, this is the Aedes MQTT broker [36]. We will read our variables in the local Node-RED using OPC UA client, from which we extract individual data (variables) using our own *Filter* functions and send them to the cloud using MQTT. When sending data via MQTT, we need to set the corresponding *MQTT topic* for it. For example, for us it looks like *inputs/FIO_I_RetroreflectiveSensor1*.

2.5. Node-RED Dashboard in Microsoft Azure Cloud

In the first case study, we implemented a dashboard for system monitoring and emergency intervention using Azure IoT Central aPaaS service. In the second case study, we opted for a different approach. Node-RED also provides the possibility of implementing a dashboard, so as suitable option turned out to be to deploy Node-RED also in the cloud and create a dashboard using it.

Deploying Node-RED to the cloud is relatively easy, as it is actually an application running on the Node.JS runtime. So we used a virtual machine, specifically the Azure Virtual Machine IaaS service [37]. Azure Virtual Machine is one of several types of scalable on-demand compute resources that Azure offers. Before we create one, we need to think about a few things such as the application name, where the resources are stored, virtual machine size, operating memory, maximum number of virtual machines, operating system, configuration, and related resources. Further, it gives us the flexibility of virtualisation without having to buy and maintain the physical hardware on which it runs. We used a Linux-based operating system, specifically Ubuntu Server 20.04 (Focal).

Thus, we send data from the local Node-RED to the Node-RED in the cloud using the MQTT protocol. Here, we read the data, i.e., receive it using the MQTT protocol, and then display it in the dashboard. We use the *node-reddashboard* 3.1.6 library and *node-red-contrib-ui-led* 0.4.11 to display the data in the dashboard, since the nodes that allow us to do this are not part of the base installation and need to be installed separately. Table 2 gives us a more detailed description of Figure 21, where we can display data in the dashboard using these nodes.

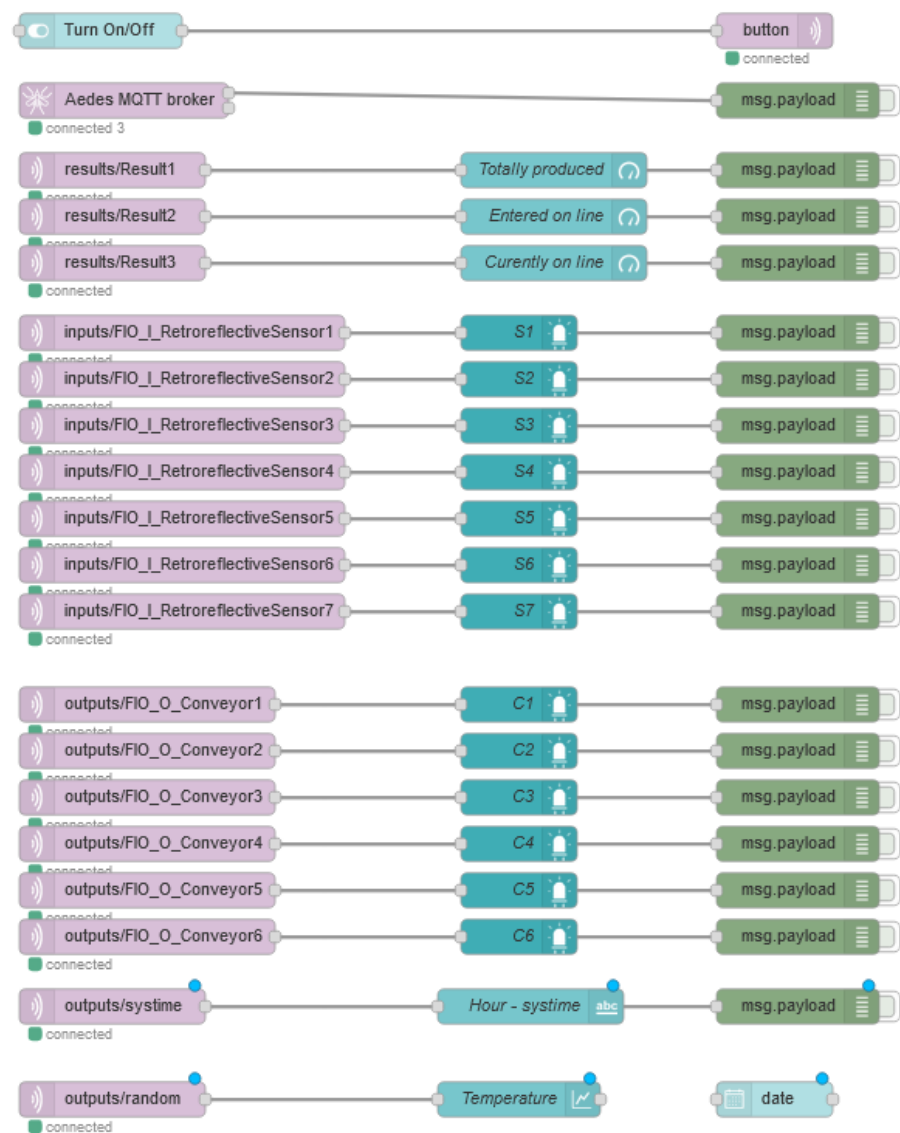
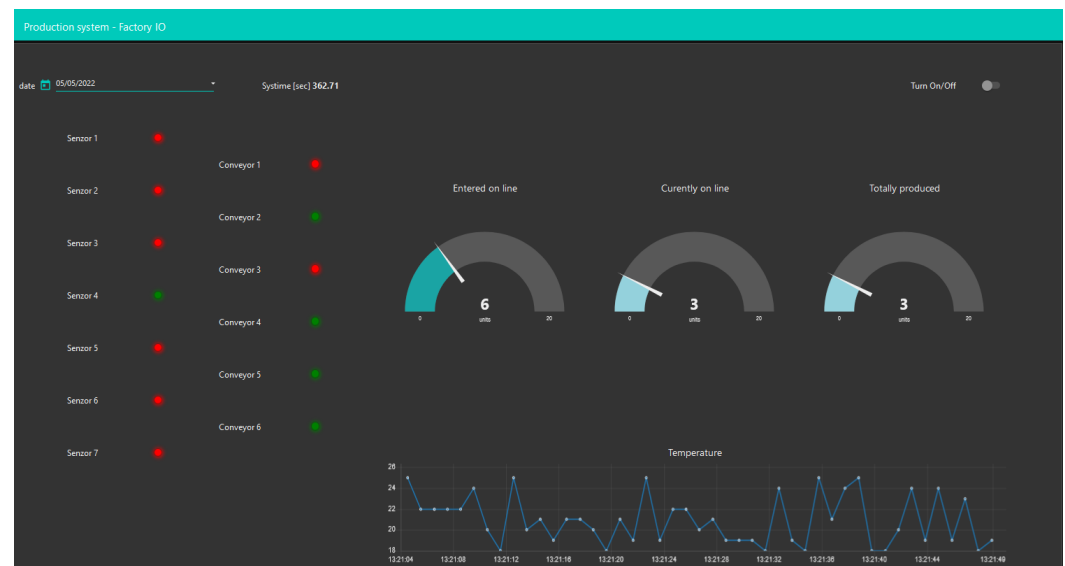


Figure 21. Node-RED in cloud.

Table 2. Variables in local Node-RED.

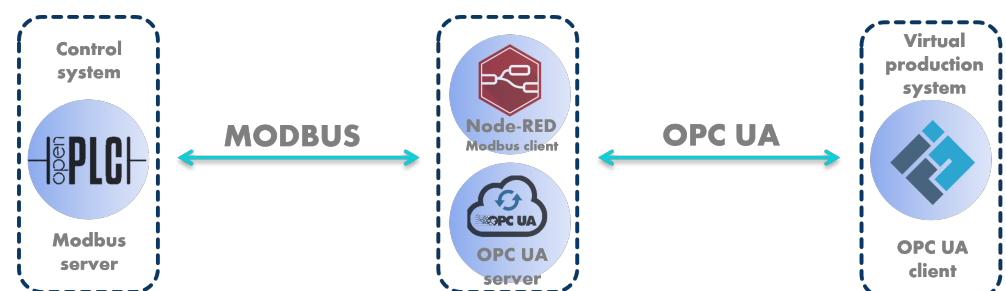
Node Type	Node Name
led	S1–S7 , C1–C6
gauge	Totally produced, Entered on line, Curently on line
text	Systime
chart	Temperature
button	Turn On/Off
date picker	date

Our final dashboard is shown in Figure 22. It is important to note that we can not only monitor the variables, but again we can also intervene in the production process in an emergency.

**Figure 22.** Dashboard in Node-RED (cloud).

3. Case Study No. 3: OpenPLC Linked with Node-RED Acting as a Software IIoT Gateway

In the third case study (Figure 23), we will again use the free OpenPLC tool. However, let us imagine a situation where we want to use OpenPLC to control a system that can be remote (accessible via a local network or Internet). OpenPLC only supports the old protocol Modbus, but we want to access this remote system via the modern OPC UA standard for security reasons. Network elements are used for this - for example IIoT gateways, which, in simple terms, provide translation of Modbus messages into the OPC UA address space (and vice versa). In our case study, we will show a software implementation of an IIoT gateway using Node-RED, which can be implemented, for example, using a Raspberry Pi microcomputer. Thus, Node-RED will act as an OPC UA server and at the same time as a Modbus client. Factory I/O will be the OPC UA client and OpenPLC runtime will be the Modbus server.

**Figure 23.** OpenPLC linked with Node-RED acting as a software IIoT gateway.

Of course, such communication cannot be considered as real-time, so such a solution is only suitable for non-time-critical systems (e.g., in a smart home). The control of systems over a network, where substantial delays can occur, is the subject of a special area of control theory focusing on networked control.

For simplicity and greater clarity, we have chosen only a fragment of the production event system for demonstration (Figure 24). It is a simple system that contains parts from both case studies. In the case study, we will use OpenPLC, which will communicate using Modbus. Then the Modbus data will be converted using Node-RED to communicate via OPC UA to Factory I/O.

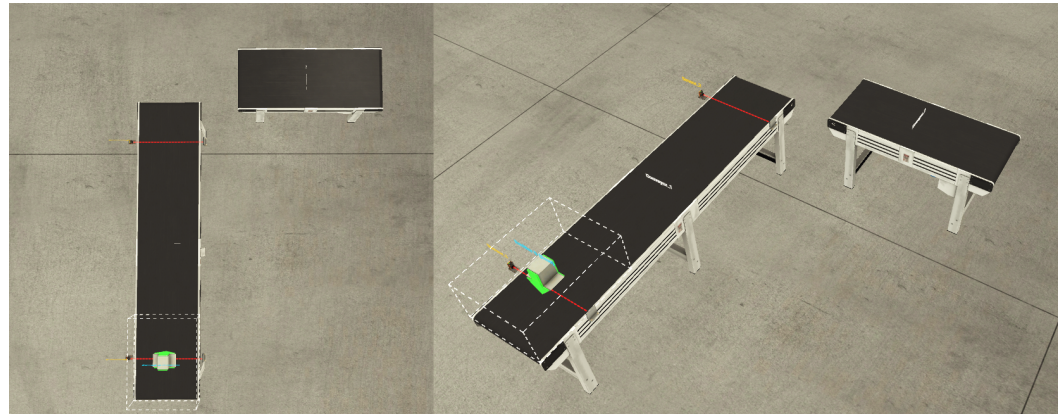


Figure 24. Fragment of discrete-event system.

3.1. Control of Discrete-Event System

As in the previous case studies, we will need to control discrete-event system. In this third case study, we use open-source editor and runtime OpenPLC. In OpenPLC, we again use Ladder logic. However, in this case study we use only four global variables named **Sensor_1** and **Sensor_2**, **Conveyor_1** and **Conveyor_2**.

Sensor_1 checks if the product is on the conveyor belt, based on which we switch on the **Conveyor_1** belt. **Sensor_2** stops **Conveyor_1** by sensing the product on the belt. We have the **Conveyor_2** variable for the purpose of checking if we are communicating, so it is set to TRUE by default. So we can think of the whole process as moving the product from point A to point B, where it stops.

3.2. Communication

The communication will be between OpenPLC and Factory I/O. However, we want to communicate with Factory I/O using the modern OPC UA protocol. As we already know from previous case studies that OpenPLC cannot communicate with modern protocols, so we will need a so called intermediary (middleware). The intermediary in this case will be Node-RED. OpenPLC will communicate with Node-RED using the Modbus protocol and Node-RED will further communicate with Factory I/O using the OPC UA protocol. Node-RED will provide us with the encapsulation of Modbus data to OPC UA address space as we mentioned above.

3.3. Creation of OPC UA Server in Node-RED

In the local Node-RED, we first need to create an OPC UA server using the OPC UA server node (Figure 25) and the procedure already mentioned in the previous section. At the beginning, we create folders in the address space in order to have separate inputs and outputs in the address space. Next, we create 4 variables. These variables are the same as in OpenPLC and we named them **fio_Sensor_1_Q1-0**, **fio_Sensor_2_Q1-1**, **fio_Conveyor_1_Q0-0** and **fio_Conveyor_2_Q0-1**.

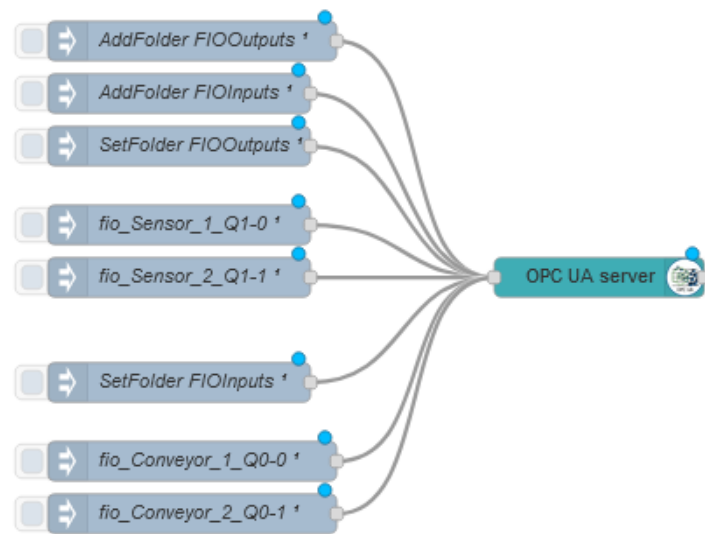


Figure 25. OPC UA server.

3.4. Communication from OpenPLC to Factory I/O

First we need to describe how the communication from OpenPLC runtime towards Factory I/O works. From OpenPLC, Factory I/O needs to read the values of the outputs. Therefore, we need to read the outputs using the **Modbus Read - %QX0.0-7** node, since the PLC program in OpenPLC evaluates whether a given Conveyor should be started or not. We connect this *Modbus Read* to our server, which we have called OpenPLC local, where we have again (as in the first case study) set the corresponding address 127.0.0.1 with port 502. Next, we again set the quantity to 1 (because we are reading 1 byte), the address to 0 (the outputs are available from address %QX0.0). This is a reading, so we set the function to *FC: Read Coil Status*.

Once we have read the values, we need to obtain them to OPC UA server so that Factory I/O can read them from it. Using the **Conveyors: Modbus to OPC UA namespace** custom function (Figure 26), we assign a specific *msg.topic* to the Modbus messages to ensure that the Modbus data are placed in the correct variable in the OPC UA address space. Sending the data to the OPC UA server is handled by the *OPC UA client* node.

The nodes at the bottom of Figure 26 are for program testing purposes only and specifically return the current value of the inputs from the Modbus server (addresses %QX1.0 and %QX1.1).

In Factory I/O, the input and output variables of the address space of the OPC UA server must be correctly assigned to the Factory I/O components (Figure 27).

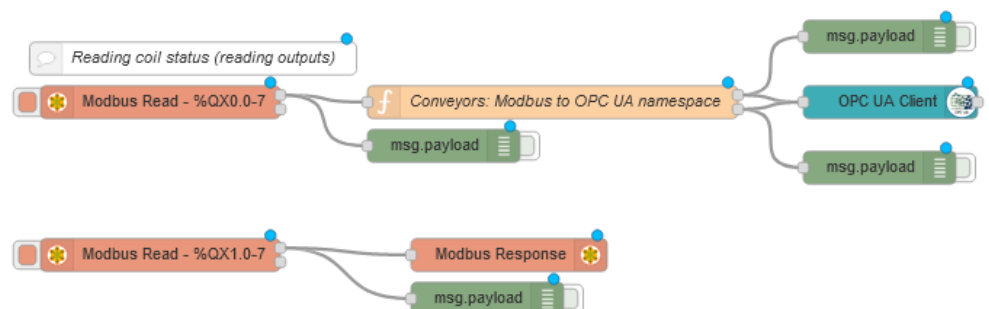


Figure 26. Communication from OpenPLC to Factory I/O.

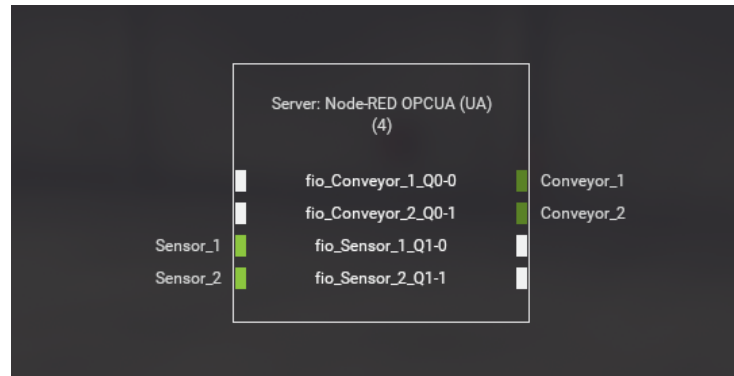


Figure 27. Factory I/O—connection.

3.5. Communication from Factory I/O to OpenPLC

In this subsection, we will show how the communication from Factory I/O towards the OpenPLC runtime takes place. Factory I/O needs to send values to the OpenPLC runtime from inputs, i.e., from sensors that detect the presence of a product. For this we need the variables **fio_Sensor_1_Q1-0** and **fio_Sensor_2_Q1-1**, which we will send to OpenPLC. In Figure 28 we see on the left two nodes of type *Inject* named after specific variables. These nodes are queried every 500 ms for the values of these variables by the *OPC UA client* node. This node then returns the value of the first or second sensor, and this is detected using the custom functions that follow in the flow (functions starting with the word *if*). Depending on the type of value, the value is written using *Modbus Write* to address %QX1.0 (address 8 in Modbus) or %QX1.1 (address 9 in Modbus) in OpenPLC runtime.

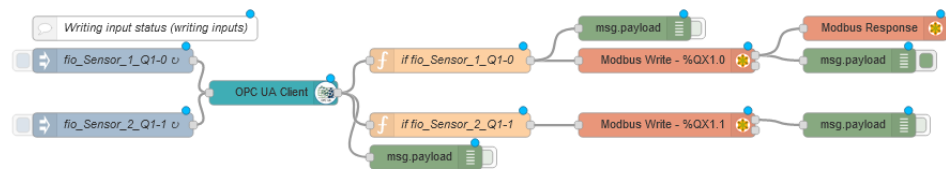


Figure 28. Communication from Factory I/O to OpenPLC.

In a similar way, a program in Node-RED could be extended to control an entire production line.

Funding: This research was funded by the Educational Grant Agency of the Ministry of Education, Science, Research and Sport of the Slovak Republic 039STU-4/2021.

Acknowledgments: We would like to thank to Simona Lopatniková for helping with programming the implementation.

References

1. Kamal, S.; Al Mubarak, S.; Scodova, B.; Naik, P.; Flichy, P.; Coffin, G. IT and OT convergence-Opportunities and challenges. In Proceedings of the SPE Intelligent Energy International Conference and Exhibition, Aberdeen, Scotland, UK, 6–8 September 2016.
2. Shahroom, A.; Hussin, N. Industrial Revolution 4.0 and Education. *Int. J. Acad. Res. Bus. Soc. Sci.* **2018**, *8*, 314–319. <https://doi.org/10.6007/IJARBS/v8-i9/4593>.
3. González-Pérez, L.I.; Ramírez-Montoya, M.S. Components of Education 4.0 in 21st Century Skills Frameworks: Systematic Review. *Sustainability* **2022**, *14*, 1493. <https://doi.org/10.3390/su14031493>.
4. Information Technology (in German). Available online: <https://theastrologypage.com/information-technology> (accessed on 26 April 2022).
5. Operational Technology. Available online: <https://www.gartner.com/en/information-technology/glossary/operational-technology-ot> (accessed on 26 April 2022).
6. What's the Difference between IT and OT? Available online: <https://www.3pillarglobal.com/insights/how-does-iiot-bring-it-and-ot-together/> (accessed on 26 April 2022).
7. Huba, M.; Kozák, Š. From E-learning to Industry 4.0. In Proceedings of the 2016 International Conference on Emerging e-Learning Technologies and Applications (ICETA), Sary Smokovec, Slovakia, 24–25 November 2016; pp. 103–108.
8. Leiden, A.; Posselt, G.; Bhakar, V.; Singh, R.; Sangwan, K.; Herrmann, C. Transferring experience labs for production engineering students to universities in newly industrialized countries. In Proceedings of the IOP Conference Series: Materials Science and Engineering (TSME-ICoME 2017), Bangkok, Thailand, 12–15 December 2017; Volume 297, p. 012053.
9. Souza, R.G.d.; Quelhas, O.L.G. Model proposal for diagnosis and integration of industry 4.0 concepts in production engineering courses. *Sustainability* **2020**, *12*, 3471.
10. Assante, D.; Caforio, A.; Flamini, M.; Romano, E. Smart Education in the context of Industry 4.0. In Proceedings of the 2019 IEEE Global Engineering Education Conference (EDUCON), Dubai, United Arab Emirates, 8–11 April 2019; pp. 1140–1145.
11. Sackey, S.M.; Bester, A. Industrial engineering curriculum in Industry 4.0 in a South African context. *S. Afr. J. Ind. Eng.* **2016**, *27*, 101–114.
12. Ciolacu, M.; Svasta, P.M.; Berg, W.; Popp, H. Education 4.0 for tall thin engineer in a data driven society. In Proceedings of the 2017 IEEE 23rd International Symposium for Design and Technology in Electronic Packaging (SIITME), Constanta, Romania, 26–29 October 2017; pp. 432–437.
13. Pierleoni, P.; Belli, A.; Palma, L.; Sabbatini, L. A versatile machine vision algorithm for real-time counting manually assembled pieces. *J. Imaging* **2020**, *6*, 48.
14. Merkulova, I.Y.; Shavetov, S.V.; Borisov, O.I.; Gromov, V.S. Object detection and tracking basics: Student education. *IFAC-PapersOnLine* **2019**, *52*, 79–84.
15. Dr, P.; Kumar, P.; Johri, P.; Srivastava, S.; Suhag, S. A Comparative Study of Industry 4.0 with Education 4.0. In Proceedings of the 4th International Conference: Innovative Advancement in Engineering Technology (IAET), Jaipur, India, 21–22 February 2020. <https://doi.org/10.2139/ssrn.3553215>.
16. Hussin, A.A. Education 4.0 Made Simple: Ideas For Teaching. *Int. J. Educ. Lit. Stud.* **2018**, *6*, 92–98. <https://doi.org/10.7575/aiac.ijels.v6n.3p.92>.
17. Coşkun, S.; Kayıkcı, Y.; Gençay, E. Adapting Engineering Education to Industry 4.0 Vision. *Technologies* **2019**, *7*, 10. <https://doi.org/10.3390/technologies7010010>.
18. Mian, S.H.; Salah, B.; Ameen, W.; Moiduddin, K.; Alkhalefah, H. Adapting Universities for Sustainability Education in Industry 4.0: Channel of Challenges and Opportunities. *Sustainability* **2020**, *12*, 6100. <https://doi.org/10.3390/su12156100>.
19. Grenčíková, A.; Kordoš, M.; Navickas, V. The impact of Industry 4.0 on education contents. *Business: Theory Pract.* **2021**, *22*, 29–38. <https://doi.org/10.3846/btp.2021.13166>.
20. Produktion2030 Ingenjör4.0. Available online: <https://produktion2030.se/en/ingenjor-4-0/> (accessed on 27 December 2021).
21. Chanthakit, S.; Rattanapoka, C. Mqtt based air quality monitoring system using node MCU and node-red. In Proceedings of the 2018 Seventh ICT International Student Project Conference (ICT-ISPC), Nakhonpathom, Thailand, 11–13 July 2018; pp. 1–5.
22. Langmann, R.; Stiller, M. The PLC as a Smart Service in Industry 4.0 Production Systems. *Appl. Sci.* **2019**, *9*, 3815. <https://doi.org/10.3390/app9183815>.
23. Ferrari, P.; Flammini, A.; Rinaldi, S.; Sisinni, E.; Maffei, D.; Malara, M. Impact of Quality of Service on Cloud Based Industrial IoT Applications with OPC UA. *Electronics* **2018**, *7*, 109. <https://doi.org/10.3390/electronics7070109>.
24. Lakhan, A.; Mohammed, M.A.; Abdulkareem, K.H.; Jaber, M.M.; Nedoma, J.; Martinek, R.; Zmij, P. Delay Optimal Schemes for Internet of Things Applications in Heterogeneous Edge Cloud Computing Networks. *Sensors* **2022**, *22*, 5937. <https://doi.org/10.3390/s22165937>.
25. Indexed Line with Two Machining Stations 24V—Simulation. Available online: <https://www.fischertechnik.de/en/products/simulating/training-models/96790-sim-indexed-line-with-two-machining-stations-24v-simulation> (accessed on 27 April 2022).
26. Pajpach, M.; Haffner, O.; Kučera, E.; Drahoš, P. Low-Cost Education Kit for Teaching Basic Skills for Industry 4.0 Using Deep-Learning in Quality Control Tasks. *Electronics* **2022**, *11*, 230. <https://doi.org/10.3390/electronics11020230>.
27. Factory IO. Available online: <https://docs.factoryio.com/> (accessed on 27 April 2022).
28. OpenPLC Editor. Available online: (accessed on 26 April 2022).

29. Philip Samuel, A.K.; Shyamkumar, A.; Ramesh, H. Industry 4.0-Connected Drives Using OPC UA. In *Industry 4.0 and Advanced Manufacturing*; Chakrabarti, A., Arora, M., Eds.; Springer: Singapore, 2021; pp. 3–12.
30. Modbus Address Mapping. Available online: (accessed on 26 April 2022).
31. Azure IoT Hub. Available online: <https://azure.microsoft.com/en-us/services/iot-hub/> (accessed on 27 July 2022).
32. Power BI. Available online: <https://powerbi.microsoft.com/en-us/> (accessed on 27 July 2022).
33. Azure Stream Analytics. Available online: <https://azure.microsoft.com/en-us/services/stream-analytics/> (accessed on 27 July 2022).
34. Azure Functions. Available online: <https://docs.microsoft.com/en-us/azure/azure-functions/> (accessed on 27 July 2022).
35. Azure IoT Central. Available online: <https://azure.microsoft.com/en-us/services/iot-central/> (accessed on 27 July 2022).

-
36. Aedes MQTT Broker. Available online: <https://github.com/moscajs/aedes> (accessed on 27 July 2022).
 37. Azure Virtual Machines. Available online: <https://azure.microsoft.com/en-us/services/virtual-machines/> (accessed on 27 July 2022).