

Message Correlation and Inter-Instance/Process Communication in Process Aware Information Systems Final Report

Umay Bengisu Bozkurt
ge85vat@mytum.de

Detailed hardware description.....	2
Code.....	3
Clean Glass Use Case.....	7
Clean Glass Use Case.....	8
Extra use case.....	8

Detailed hardware description

The newly added components are 5 distance sensors (E18-D80NK 5VDC), 5 USB to TTL adapters (DSD TECH SH-U09C2 USB), a USB hub, and a treadmill. 5 distance sensors are used to catch the glasses' locations and sensors should be wired to the treadmill, facing the glasses. 5 USB to TTL adapters are used for programming and the USB hub is used for wiring and managing these 5 USB's in a parallel fashion. The treadmill will be used to store the glasses in an ordered manner so that they can be cleaned and get ready for the robotic arm to pick them up for new cocktails.

1. Sensor wiring:

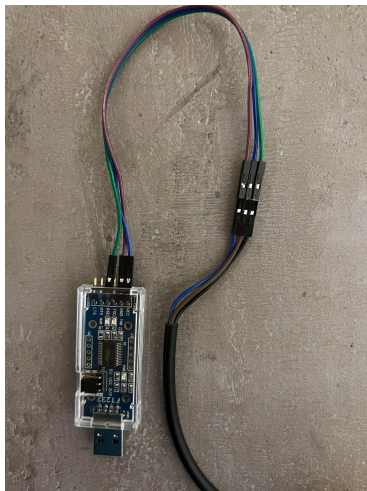


Figure 1. Wiring of the USB adapter and the distance sensor

I wired the sensors to the USB adapters in the following way:

- Sensor's GND, the blue cable, to the GND of the USB converter
- Sensor's VCC, the brown cable, to the VCC of the USB converter
- Sensor's OUT, the black cable, to the RXD of the USB converter

2. The circuit

Five sensors are wired to USBs as in the above section. Then, all the USBs are plugged into the USB hub. The USB hub is plugged into one USB port. In the future, all sensors will be placed on the treadmill.



Figure 2. Wiring of all sensors

Code

I have chosen Python as my programming language as I thought it would be suitable for this task and also I had prior experience with Python.

sensor_reader.py

After wiring the sensors, I defined the sensors in the code. An example of defining a sensor can be seen in Figure 3. The baud rate depends on the sensor. Now that I am connected to the lab's server and the sensors are not connected to my local computer anymore, I have changed the port's name, which had COM port values, to the USB's device names. I found the USB device names are found by `udevadm monitor` command.

I also initialize a thread for each sensor. Before I added threads, the sensors were only able to read in order. With the implementation of threads, the sensors should also be read out of order and therefore it will also support error checking. For instance, if sensor 3 catches the glass before sensor 1, which is an error for our use case, it can be further investigated for error handling.

```
def __init__(self, usb_name, sensor_name):
    self.ser = serial.Serial('/dev/tty'+ usb_name, baudrate=9600,
bytesize=8, parity='N', stopbits=1, timeout=None)
    self.thread = threading.Thread(target=self.read_sensor)
    self.sensor_value = None
```

Figure 3. Initializing sensors in `sensor_reader.py`

To read the sensors defined in Figure 3, I added the `read_sensor` method which prints the sensor's name when the sensor catches the glass. The `read_sensor` method is defined in Figure 4.

```
def read_sensor(self):
    while True:
        recv = self.ser.read(1)
        if recv:
            self.sensor_value = recv.decode('utf-8')
            return self.sensor_value
```

Figure 4. Defining the method for reading sensors.

There is also code for keeping these threads alive and stopping them, but they were omitted for simplicity and can be found in the code. The code snippets above are all from `sensor_reader.py`.

sensor_api.py

The API implementation is in the `sensor_api` file. The sensor API is designed to facilitate the interaction with multiple sensors connected to a system. It provides endpoints to retrieve data from individual sensors, check sensor status, and obtain all sensor values collectively.

The RESTful API endpoints are created using the FastAPI Framework, with uvicorn. To use the sensor readings, I imported the `sensor_reader.py` code. Reading data from specific sensors is the responsibility of the `SensorReader` class. There is an instance of this class for each sensor.

Initializing an instance of the Sensor Reader class with the USB device name implemented as follows:

```
app = FastAPI()
# Create instances of SensorReader for each sensor
sensors = {}
usb_names = {
    1: "USB1",
    2: "USB2",
    3: "USB3",
    4: "USB4",
    5: "USB5"
}

for sensor_id, usb_name in usb_names.items():
    sensors[sensor_id] = SensorReader(usb_name, f"Sensor {sensor_id}")
    sensors[sensor_id].start_reading()
```

Figure 5. Reading sensor in the sensor_api.py

I have also utilized multiprocessing for a simultaneous execution and manage several sensor instances concurrently. I defined asynchronous methods in order to read sensor data asynchronously, and to have non-blocking I/O operations. I used a JSON file to write sensor values to debugging and logging.

Inside the sensor_api.py class I have multiple get endpoints and methods to call, here is an overview of the methods in the API:

- read_root(): This function returns data about all connected sensors while serving the root endpoint /.
- read_sensor_value(sensor id): This function reads the current value of the designated sensor for the endpoint /sensor_api/{sensor_id}.
- write_to_json(sensor_data): This method publishes sensor data to a JSON file in an asynchronous way.
- read_sensors_data(): This method writes to the JSON file the first detection of the sensors by using write_to_json method defined before. This method also uses Python's time module used to sleep so that it does not write for the same detection again and again.

```
• async def read_sensors_data():
•     #sensors_data = {}
•     while True:
•         start_time = time.time()
•         sensor_values = {sensor_id: sensor.sensor_value for sensor_id,
sensor in sensors.items()}
•         #sensor_values = {1: sensor1.sensor_value, 2:
sensor2.sensor_value, 3: sensor3.sensor_value, 4: sensor4.sensor_value,
5: sensor5.sensor_value}
•         # Update the sensors_data dictionary
•         non_null_sensor_values = {key: value for key, value in
sensor_values.items() if value is not None}
•         sensors_data = non_null_sensor_values
•
•         for sensor_id, value in sensor_values.items():
•             if value is not None:
•                 if not sensor_first_detection[sensor_id]:
•                     print(f"Object detected by Sensor {sensor_id} for the
first time: {value}")
•                     await write_to_json({sensor_id: value})
•                     sensor_first_detection[sensor_id] = True
```

```

•
•     # Write sensor values to JSON file asynchronously
•
•     elapsed_time = time.time() - start_time
•     # Sleep for a while before reading again
•     await asyncio.sleep((max(0.0, 1 - elapsed_time)))

```

Figure 6. Read_sensor_data method.

- get_all_sensor_values(): This asynchronous function provides the current values of every sensor to when the get endpoint with /sensors is called.
- check_sensors(): This function checks all the sensors' status and returns their current value when the /check_sensors endpoint is called.
- check_sensor(sensor_id): Serving the endpoint /check_sensor/{sensor_id}, this method retrieves the current value of the provided sensor after verifying its status. This is used as the endpoint to check the sensor in CPEE. This function is defined as follows:

```

• @app.get("/check_sensor/{sensor_id}")
• def check_sensor(sensor_id: int):
•     if sensor_id not in sensors:
•         raise HTTPException(status_code=404, detail="Sensor not found")
•
•     sensor = sensors[sensor_id]
•     sensor_value = sensor.sensor_value
•     if sensor_value is not None:
•         return sensor_value
•     else:
•         return 0

```

Figure 7. Check sensor endpoint.

- startup_event(): This asynchronous function starts the asynchronous task of reading sensor data when the server boots up.

To run the API in the server run_server method is implemented. Inside this function uvicorn.run method is called so that I can use uvicorn to run my project. For running only one process in the server, it is checked if there is already an existing process inside the main method. If it does, it kills and removes this process before starting the process again.

```

def run_server():
    pid = os.fork()
    if pid != 0:
        return

```

```

print('Starting ' + str(os.getpid()))
print(os.getpid(), file=open('sensor.pid', 'w'))
uvicorn.run("sensor_api:app", host="::", port=9119, log_level="info")

if __name__ == "__main__":
    if os.path.exists('sensor.pid'):
        with open("sensor.pid", "r") as f: pid = f.read()
        print('Killing ' + str(int(pid)))
        try:
            os.kill(int(pid), signal.SIGINT)
        except ProcessLookupError:
            print("Previous process already terminated.")
            os.remove('sensor.pid')
    proc = Process(target=run_server, args=(), daemon=True)
    proc.start()
    proc.join()

```

Figure 8. Run server and main method implementations

You can see the effect of the main method below from the terminal:

```

mangler@hairychest:~/praktikum$ ./sensor_api.py
Killing 3217110
Starting 3218067
INFO: Started server process [3218067]
INFO: Waiting for application startup.
INFO: Application startup complete.
ERROR: [Errno 98] error while attempting to bind on address ('::', 9119, 0, 0): address already in use
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: Shutting down
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: Finished server process [3217110]

```

requirements.txt

The requirements.txt file contains the dependencies and their versions used in my project such as fastapi, pyserial and uvicorn. The packages listed inside requirements.txt are necessary for the proper functioning of the project and should be installed with the given versions to ensure compatibility.

Check Sensor Use Case

The setting explained above will be used as a way to check if all the glasses are placed before cleaning the glasses. For this use case, all the sensors will be checked to see how many glasses are on the treadmill and if all the sensors are not null, meaning that five glasses are waiting to be cleaned and the treadmill is full, the glasses are cleaned. If all the checks return not null then that means the glasses are ready to be cleaned.

For this use case we assume we have five glasses, one sensor for each and five sensors in total. We proceed if only the sensor values are different from zero. As defined in the API, if the sensor value is zero it means the sensor did not detect anything. Therefore, we only need to proceed to the next sensor if the sensor has detected a glass.

The screenshot displays a BPMN editor interface with three tabs: **Data Elements**, **Endpoints**, and **Attributes**. The **Data Elements** tab is active, showing a list of five sensors: `check_sensor_1`, `check_sensor_2`, `check_sensor_3`, `check_sensor_4`, and `check_sensor_5`. Each sensor is associated with a specific endpoint URL: `https://lab.bpm.in.tum.de/ports/9119/check_sensor/2` through `https://lab.bpm.in.tum.de/ports/9119/check_sensor/5`.

Below the list, the **Graph** tab is active, showing a BPMN diagram. The diagram starts with a start event (circle with a plus sign), followed by a task (rounded rectangle) labeled "Check sensor 1". This task is connected to an exclusive gateway (diamond with a cross). The gateway has two outgoing flows: one leading to a data store (cylinder) labeled "data.sensor1 > 0" and another leading to a task labeled "Check sensor 2". The "Check sensor 2" task is also connected to an exclusive gateway. This gateway has two outgoing flows: one leading to a data store labeled "data.sensor2 > 0" and another leading to a final event (circle with a thick border).

On the right side, the **Properties** panel is visible. It shows the following configuration for the selected task:

- ID**: `a1`
- Endpoint**: `check_sensor_1`
- Label**: `Check sensor 1`
- Method**: `:post`
- Arguments**: A button labeled "Create Argument Pair" is present.
- Output Handling**: A section with four expandable options: **Prepare**, **Finalize**, **Update**, and **Rescue**.

Check sensor is under ge85vat in the process hub.

Future work

The next step of the project was to place the sensor on a treadmill. Since there was a delay with its implementation throughout this course, when it arrives the sensor can be taped on the treadmill and the whole configuration and the implementation can be used with the treadmill.

Also, when another project implements a cleaning glass method, it can be added in the clean glass subprocess field inside the last condition in Check Sensor.

Extra use case

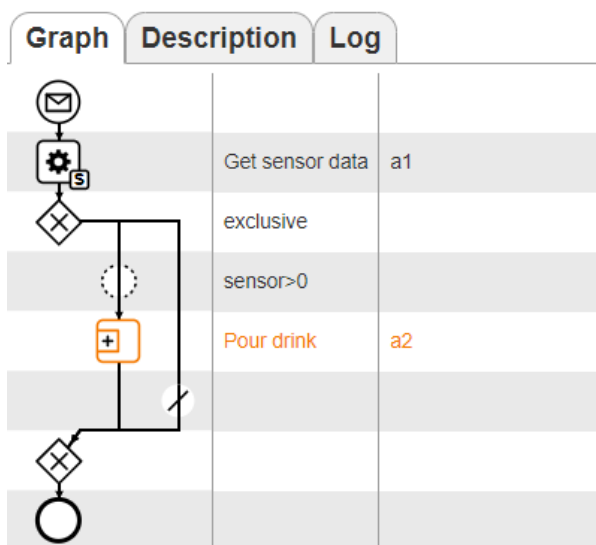
In the future, this sensor setup can also be used to take away some of the responsibilities from the robotic arm and delegate these responsibilities to newly added components such as the sensors and a treadmill. For this use case, an electronic treadmill that flows horizontally is needed. The responsibilities taken away from the robotic arm and by which component they are done in my project are as follows:









1. Picking up the glasses under the bottles and placing them in their next position under one of the glasses is not necessary as the treadmill will be used to move the glasses forward.
2. Sensors will also help to replace the robotic arm's responsibility of carrying the glasses around as now the glass' locations are also known by the sensor; therefore, the robotic arm still can tell which bottle's lever to push, even though it did not place the glass itself.

Therefore, only pushing the lever and picking up the glass to place it under the mixer is left for the robotic arm. If this use case is implemented the cocktail-making process will be faster.

If this use case is implemented the flow would be as follows:

1. The robotic arm picks the glass from its initial place, which is the back of the station (where the beverage bottles are located and drinks are poured).
2. The robotic arm carries the glass to the beginning of the treadmill.
3. The treadmill moves until the sensor located under the first ingredient catches the glass.
4. The treadmill stops.
5. The robotic arm pushes the lever of the ingredient and the drink is poured. When the pouring is over, the treadmill starts running again until the second ingredient's sensor catches the glass.
6. Steps 2-4 are repeated until they are done for all the ingredients of the cocktail.
7. After all the ingredients are poured the robotic arm picks up the glass and puts it under the mixer.



Graph	Description	Log
		
	Set position	a1
	Set amount	a2
	Pull lever	a3
	exclusive	
	$\text{data.bottle_number} < 4$	
	Pick glass up from under one of the first 4 bottles	a4
	Pick glass up from under the last bottle	a5
