Inform : Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi
Vol.6 No.1 January 2021, P-ISSN : 2502-3470, E-ISSN : 2581-0367

Implementation of Multiprocessing and Multithreading for End Node Middleware Control on Internet of Things Devices

Iwan Kurnianto Wibowo[1], Adnan Rachmat Anom Besari[2], Muh. Rifqi Rizqullah[3]

[1,2,3]*Program Studi Teknik Komputer, Politeknik Elektronika Negeri Surabaya, Indonesia*
[1]eone@pens.ac.id*
[2]anom@pens.ac.id, [3]rifqimr@ce.student.pens.ac.id

Abstract— Previously, an educational robot system was built by incorporating Internet of Things (IoT) elements. Over time, this educational robot has been implanted with a middleware. Middleware has a role in receiving command data from the real-time database, access sensors, actuators, and sending feedback. Middleware contains protocols that translate commands between high-level programming and Raspberry Pi hardware. The focus of this research is to improve the performance of the middleware to pursue processing time efficiency. For this reason, it is necessary to implement multiprocessing and multithreading in handling several tasks. The CPU division has been adjusted automatically to not work on just one core or block of memory. Several program functions can run in parallel and reduce program execution time efficiently. The tasks handled are sensor reading and actuator control in the form of a motor. Testing has been carried out to perform multiprocessing and multithreading tasks to process six sensors and five actuators. Multiprocessing requires an average of 1.00% to 15.00% CPU usage and 2.70% memory usage. Meanwhile, multithreading involves an average of 1.00% to 71.00% CPU usage and 3.30% memory usage.

Keywords— Raspberry Pi, Internet of Things, Multiprocessing, Multithreading, Middleware**,** Educational Robot.

I. Introduction

Various aspects of human life have been affected by technological advances that continue to develop so far. The Internet of Things (IoT)[1] is one of the big technological advances[1]. The advancement of the IoT field itself enters the technical or industrial sector, but with the emergence of a development board with such requirements to become a learning module, development in the world of education has also increased. Despite the many historical developments, IoT developers have only one goal, namely to support human life.

Unfortunately, the rapid development of IoT has not been matched by adequate learning methods. At present, the interest in studying the IoT field is very strong, but there is not much software that provides IoT building learning media. It is also hard for beginners who want to learn about IoT to learn the fundamentals of programming comprehension since one of them is a programming grammar that is very hard to understand. In developing your own IoT device, you also need hardware or modules that are not cheap, especially when doing trial experiments and error is possible.

Generally, the IoT infrastructure is designed with the Raspberry Pi [2]. Raspberry Pi is an SBC (Single Board Computer), which in terms of size is practically the size of an ATM card, has a 40-pin GPIO like a microcontroller, has computer capabilities, and is relatively affordable. Middleware can also be built using Raspberry Pi to solve one of the challenges of the industrial revolution 4.0 for controlling electronic devices [3]. This study uses Raspberry as SBC. This is because there is a lot of support from forums for continuous research, the availability of sensor and actuator modules in the open market, and many people have used it for the development of IoT infrastructure.

Applications must pay attention to the needs of the crowd, especially for a beginner who wants to learn about IoT. They need help and explanation regarding the learning process of understanding programming logic in the IoT area. From here on, the author wants to build a Raspberry Pi that can be assembled modularly and operated by ordinary people via the Raspbian OS Interface framework.

This research focuses on designing middleware on educational robots so that it is hoped later to create a protocol for accessing sensors and controlling actuators with commands that have been developed. The software is often used to coordinate embedded system implementations and so that the system works well and can also be timely and effective. In related research [4], middleware is a programming layer that links high-level programming on the Raspberry Pi with block programming.

This programming layer is used based on the user method of interpreting commands with high-level programming, which are then passed on to the middleware sensors or actuators. Based on [5], It is also possible to view middleware as a protocol that translates commands between the Raspberry Pi and high-level programming. The goal of this development is to encourage optimum hardware performance. This research aims to embed the middleware built on the Raspberry Pi in the form of wiring on the GPIO, which is modular.

II. Research Method

A. *Middleware for IoT Devices*

Research of [6] provides an Information Flow of Things (IFoT) middleware architecture, a system for the collection, interpretation, and mixture of real-time and scalable data based on the sharing of data processing between IoT devices. There are several parameters in the basic definition of IFoT. Each layer of the IFoT middleware has a function:

DOI : 10.25139/inform.v6i1.3346

1. *Task Allocation Mechanism*: consists of split class and task assignments class. Recipe split class reads recipes between applications and divides them into tasks that can be executed in parallel. Assignments class distributes assignments divided among IFoT modules.
2. *Flow Analysis Function*: consists of learning class, judging classes, managing classes. The learning class analyzes the time series of sensor data, sequentially ordering, and building / updating the model. The judging class analyzes the flow of data using the built model. The managing class manages cooperative operations for distributed processing.
3. *Flow Distribution Function*: consists of a publishing class, broker class, subscribe class. In the IFoT middleware, a publish/subscribe system is adopted to distribute flows between IFoT nodes, which aims to realize loosely coupled flows and scalable messages. The publish class is placed on the sending side, the subscribe class is placed on the receiving side in the communication between the IFoT nodes. The broker class manages the distribution of data according to the topics defined by the subscription class.
4. *Sensor/actuator Integration Function*: consists of sensor class and actuator class. Each class of hardware and sensor/actuator communication interfaces and provides an interface for streaming distribution functions.

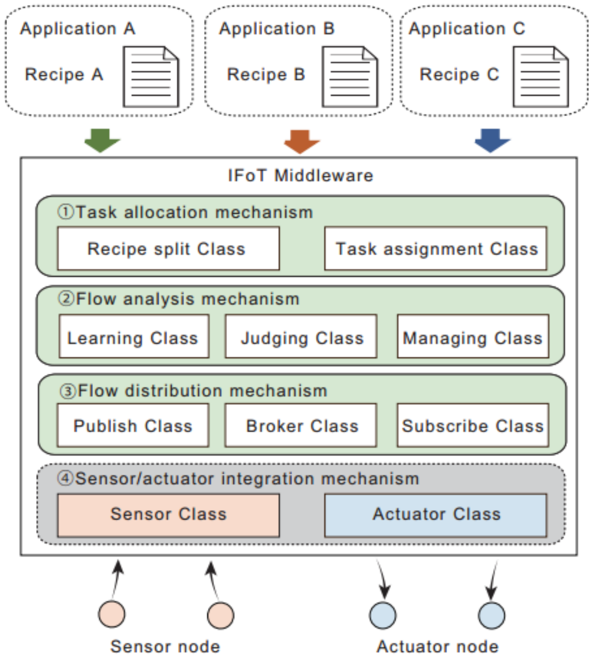

Figure 1. The logic architecture of the IFoT middleware

Figure 1 presents IFoT's logical architecture. Restricted functions, namely the flow distribution function and the flow analysis function, are necessary to implement the IFoT prototype. The first feature is a Mosquitto-built flow propagation function, an application that uses the communication protocol of MQTT. In terms of online machine learning, the flow analysis feature was developed using Jubatus, which has more capabilities. A framework handles any process between many modules. Using OpenRTM-aist, the program was constructed.

B. *Multithreading and Multiprocessing on Python*

Based on [7], a thread is the sequential execution of a program from a machine instruction. A thread can run in parallel with other threads in a process. A process can contain multiple threads. Each thread can execute a set of instructions (a function) independently and run parallel to other processes or threads. To be an active thread that is different in a process, the thread will divide the empty memory address, then share its data structure.

Thread-based parallelism is a standard for making parallel programming. Note that the python interpreter is not completely thread-safe [8]. In python, fully able to use multithreading, it requires a global lock called GIL (Global Interpreter Lock). The essence of GIL is that it can only execute a thread from a python program. A GIL is never enough to avoid the trouble of a program [9]. If multiple threads try to access the same data in an object, the program will terminate in an inconsistent state.

Process-based parallelism is parallel programming that implements the shared memory paradigm. A python program that implements multiprocessing will use one or more processors to access the main memory. In python, multithreading does not require GIL because each process will run on a different CPU to access the main memory. In contrast to multithreading, which uses multiple threads.

C. Firebase Real-time Database

Firebase real-time database is a product in the form of a cloud-hosted database from Google [10]. Firebase uses JSON to sync data in real-time whenever a client connects. This study uses the REST API to access the Firebase real-time database URL as a REST endpoint. Firebase allows users to access the real-time database directly from the program on the client and access it securely. The database of the Firebase real-time database is NoSQL[11]. In Firebase, the real-time database API is designed only for operations for fast execution.

D. System Design

The architecture starts from giving the system driver instructions that include the class of sensor and actuator. The only way to control sensors and actuators is to give a command line with any declared parameters so that the command can read/write the sensor or actuator. Figure 2 shows the types of sensors and actuators that have been specifically defined in our research middleware system.

There are some key points in applying middleware on the Raspberry Pi, including the reading of the command notation sent and the control over the devices to be used. In the middleware, the processing of incoming data, the translation, the operation of functions to access sensors and actuators as a process or thread is performed. Starting from reading the header and then matching it to the given function code map, each data obtained will be parsed

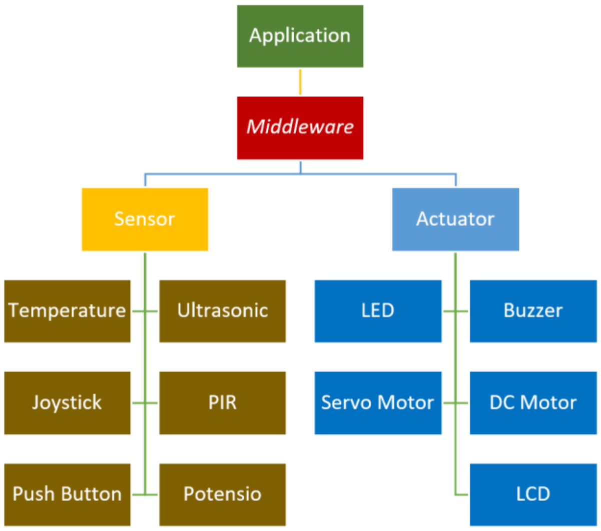

Figure 2. Sensors And Actuators That Can Be Accessed On The Raspberry Pi With Our Middleware Framework

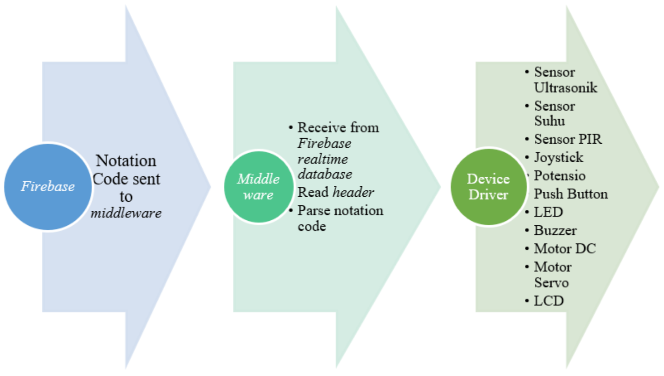

Figure 3. Workflow system

The initial stage in Figure 3 is that the Firebase real-time database's incoming data will be accommodated in a variable first. Next, the Firebase real-time database's data will be parsed with the specified parameters or function code. Each data that is sent is a set of instructions for access sensors or actuators with feature and parameter codes. Each data information structure has the same information structure, but each sensor and actuator incorporate a distinctive number of parameters. Figure 4 is the program code notation for each block.

```
<header>,<parameter - n>,<parameter - n>,…....<parameter - n>
```

Figure 4. Program code notation for each block

There are two headers, namely a header for sensors and a header for an actuator. Then followed by several parameters which contain the type of sensor, arguments, and feedback. The ON or OFF state is the next parameter. Where ON is used to activate a sensor or actuator as a process. Meanwhile, OFF is used to deactivate sensors or actuators by stopping the process/thread. There is an additional parameter which, according to its characteristics, functions to access sensors and actuators. Headers and parameters are separated using a comma (","). The temperature sensor access notation parameters are shown in Figure 5.

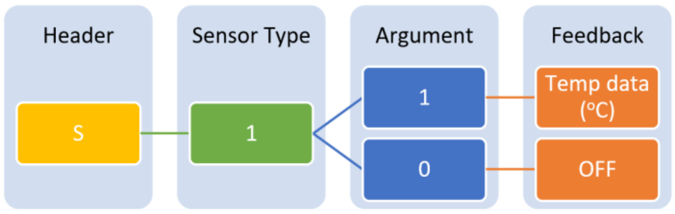

Figure 5. Temperature sensor access notation

The distance sensor's notation is to retrieve and display data in the form of the distance in front of it in centimeters. The following is a notation for retrieving data from a distance sensor which is presented in diagrammatic form shown in Figure 6. The feedback from this notation is the output of the activated distance sensor reading process/thread.

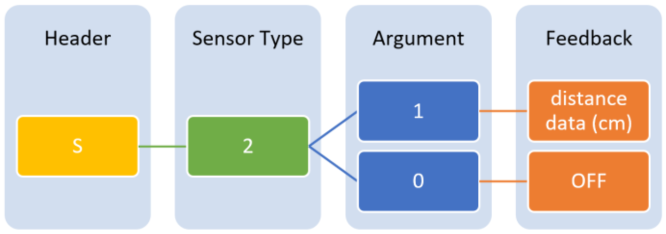

Figure 6. Distance sensor access notation

The notation on the PIR sensor is to retrieve and display data in the form of 1/0 logic when there is movement or not around the sensor. The notation of data collection from the PIR sensor can be seen in Figure 7.

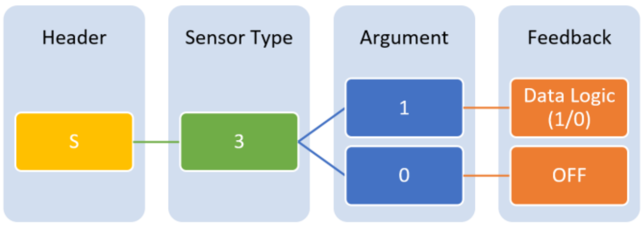

Figure 7. PIR sensor access notation

The joystick's notation is for retrieving and displaying data in the form of X and Y coordinate values when the lever is moved. Figure 8 is a diagram for retrieving data from the joystick. The feedback from this notation is the X and Y coordinate of the joystick.

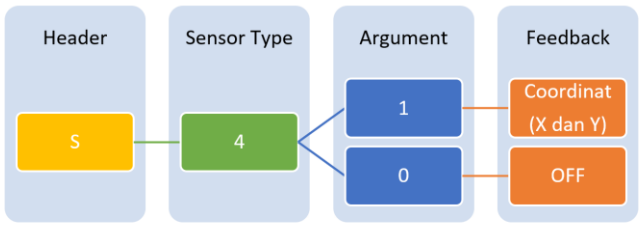

Figure 8. Joystick access notation

DOI : 10.25139/inform.v6i1.3346

Figure 9 is a notation for retrieving data from a potentiometer, which is presented in diagrammatic form. The potentiometer's notation is to retrieve and display data in the form of ADC values 0 to 1024 when the lever is rotated right or left.

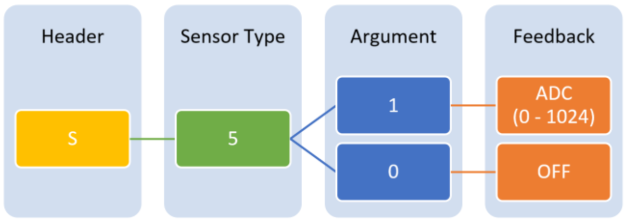

Figure 9. Potentiometer access notation

The notation for turning the LED on or off with time parameters is presented in Figure 10. The parameters provided are the LED label number and time. Meanwhile, the feedback that can be read is the output of the activated LED thread or process.

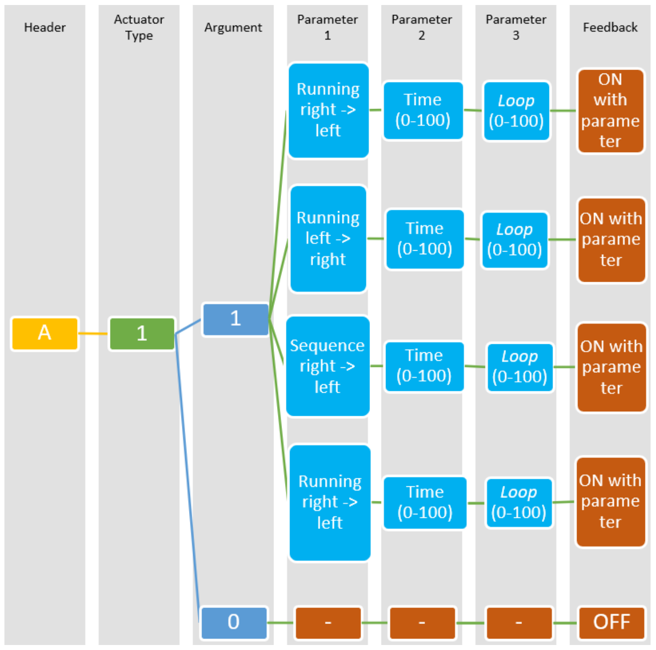

Figure 10. LED access notation

E. Multiprocessing and multithreading

In an operating system, the multiprocessing approach is a parallel computation model, where each task performed is considered a process. [12]. In general, small tasks are always in a process (big task). Please note that the threads are always in the same block of memory addresses. Therefore, data processing uses shared memory in one logical processor. This is different from middleware that uses multiprocessing. Instead of being threads, the middleware process that uses multiprocessing is divided into several sub-processes. This makes it possible to use another logical processor contained in the CPU in some of the jobs that are subprocesses. There is a 4-core or logical processor on the Raspberry Pi.

It is expected that from the use of this multiprocessing, the task contained in the middleware can be immediately divided into 4 cores. But back to the nature of an operating system, which has the right to fully control each process or thread which will run on which core in a CPU. In this study, a comparison was made between the use of multiprocessing and multithreading in the middleware that was built.

A declaration is required in the main program to make every function that runs each sensor and actuator a process. The declaration on the main program is shown in Figure 11. If each function that runs each sensor and actuator wants to be a thread, it must declare the main program to show a function to be a thread when executed. The declaration on the main program is shown in Figure 12.

```
if __name__ == '__main__':

        battery=multiprocessing.Process(target=statb
attery)
        temperature=multiprocessing.Process(target=s
uhu)
        us=multiprocessing.Process(target=jarak)
        pir=multiprocessing.Process(target=ir)
        joystick=multiprocessing.Process(target=joy)
        potensio=multiprocessing.Process(target=trim
pot)
        button=multiprocessing.Process(target=push)
        barled=multiprocessing.Process(target=led,
args=(lednum,waktu,))
        buzzer=multiprocessing.Process(target=buzz,
args=(pitch,duration,))
        rotomfw=multiprocessing.Process(target=motor
fw, args=(waktu,kn,kr,))
        rotombw=multiprocessing.Process(target=motor
bw, args=(waktu,kn,kr,))
        servo=multiprocessing.Process(target=srv,
args=(sudut,))
        lcd=multiprocessing.Process(target=tampil,
args=text,))
```

Figure 11. Declaration of the main program to be a process

```
if __name__ == '__main__':

        battery=threading.Thread(target=statbattery)
        temperature=threading.Thread(target=suhu)
        us=threading.Thread(target=jarak)
        pir=threading.Thread(target=ir)
        joystick=threading.Thread(target=joy)
        potensio=threading.Thread (target=trimpot)
        button=threading.Thread(target=push)
        barled=threading.Thread(target=led,
args=(lednum,waktu,))
        buzzer=threading.Thread(target=buzz,
args=(pitch,duration,))
        rotomfw=threading.Thread(target=motorfw,
args=(waktu,kn,kr,))
        rotombw=threading.Thread(target=motorbw,
args=(waktu,kn,kr,))
        servo=threading.Thread(target=srv,
args=(sudut,))
        lcd=threading.Thread(target=tampil,
args=text,))
```

Figure 12. Declaration of the main program to be a thread

DOI : 10.25139/inform.v6i1.3346

III. Results and Discussion

Testing is done via a remote connection from the laptop to the raspberry device. The laptop has core specifications Intel i3-7100U @ 2.4 GHz x 4, 4GB RAM with OS Win 10 64 bit. While the raspberry has specifications a Broadcom BCM2837 @ 1.2 GHz x 4 processor, 1 GB RAM, and Raspbian Stretch 2017 OS. The tests that have been carried out include every command notation sent, the response time between the start of the command being sent to the middleware being able to execute a function. The middleware's speed in making a process, the speed of middleware in creating a thread, and the comparison of multiprocessing usage with multithreading on CPU usage and memory usage.

Table I shows the response time between the command notation sent from Apps until the middleware can respond to command data. Middleware is capable of executing a process in seconds. Testing is carried out for each process, and data is taken from 5x sending orders individually.

TABLE I
TEMPERATURE SENSOR ACCESS TESTING WITH MULTIPROCESSING

Notation	Response time (s)	Multiprocessing execution time (s)	Trial
S,1,1	01s:389ms	1,1056	1
	01s:102ms	1,1065	2
	01s:789ms	1,1049	3
	02s:129ms	1,1079	4
	01s:827ms	1,1089	5

Table I shows that the response time between commands sent from Apps to the middleware to respond is between 1 second to 2 seconds. The test in Table I is carried out at night to get a good and stable internet speed. While the execution time of a process starts from initialization to become a process, and sending 1x data takes > 1 second. The next test is to take the response time between the command notation sent from Apps until the middleware can respond to command data and the middleware's speed in executing a thread in seconds. Tests are carried out on each thread, and 5x data for sending orders are taken individually, shown in Table II.

TABLE II
TEMPERATURE SENSOR ACCESS TESTING WITH MULTITHREADING

Notation	Response time (s)	Multiprocessing execution time (s)	Trial
S,1,1	01s:829ms	1,1329	1
	01s:126ms	1,1442	2
	01s:912ms	1,1484	3
	01s:127ms	1,1582	4
	02s:892ms	1,1393	5

Table II shows that the response time between the commands sent and the middleware able to respond is between 1 second to 2 seconds. Meanwhile, the thread execution time starts from initialization to become a thread, and sending 1x data takes> 1 second. The test was conducted at night with good and stable internet conditions. The next test is carried out sequentially, and data is taken from 5x individual orders which are shown in Table III.

TABLE III
TEMPERATURE SENSOR ACCESS TESTING SEQUENTIALLY

Notation	Response time (s)	Multiprocessing execution time (s)	Trial
S,1,1	01s:102ms	1,1279	1
	01s:923ms	1,1397	2
	01s:124ms	1,1383	3
	01s:721ms	1,1472	4
	01s:436ms	1,1394	5

Table IV is the test results of CPU usage and memory usage data between multiprocessing, multithreading, and sequential access to temperature sensor functions. The function is carried out individually, and 5 times the sampling is taken.

TABLE IV
CPU AND MEMORY USAGE TESTING FOR MULTIPROCESSING, MULTITHREADING, AND SEQUENTIAL TEMPERATURE SENSOR ACCESS

Multiprocessing (%)		Multithreading (%)		Sequential (%)	
CPU	Memory	CPU	Memory	CPU	Memory
8,7	2,4	8,7	2,8	10,5	2,7
7,7	2,4	7,9	2,8	10,3	2,7
8,3	2,4	8,9	2,8	9,9	2,7
9,1	2,4	9,7	2,8	10,7	2,7
8,9	2,4	9,3	2,8	9,7	2,7

From the 4 test points whose results are shown in Table I-IV, it can be concluded that the command notation test was successful. In testing the second point, namely retrieving response time data between commands sent from Apps until the middleware can respond to commands and retrieving data when creating a process or thread from initializing to becoming a process/thread. Multiprocessing and multithread tests are carried out with 5 individual running processes/threads. The sequential method is carried out 5 times the program runs because sequential programming can only run 1 loop at a time.

CPU usage and memory usage are also compared when a function is accessed using multiprocessing or multithread or sequential programs. Figure 13 is a graph showing the comparison of CPU and memory usage for temperature sensor access.

Analysis on temperature sensor testing using multiprocessing, multithreading, and sequential programs is that there are differences in the completion of the 1x program loop and CPU and memory use. After a 1x program loop, multiprocessing records a slightly faster time than multithreading or sequential. This happens because multiprocessing uses different addressing space and CPU-core in carrying out each process. While multithreading uses the same CPU-core addressing space, the process execution is slightly faster than the execution of a thread. A thread's execution uses more CPU usage and memory usage because it is in the same addressing space. A sequential program has the same character as the execution of a thread. This is because, in general, a program that is executed will become a thread. So there is not much difference between CPU usage and memory usage between multithreading and sequential.

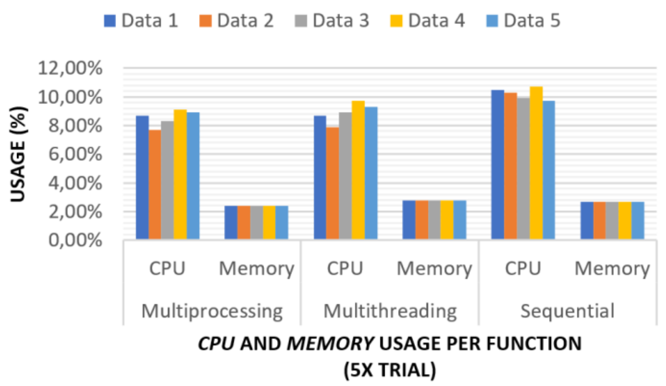

Figure 13. Comparison graph between CPU and memory usage on temperature sensor access

The next test is to test the system when it is used to access a servo motor. Table V shows the response and execution times using multiprocessing on servo motor access.

TABLE V
SERVO MOTOR ACCESS TESTING WITH MULTIPROCESSING

Notation	Response time (s)	Multiprocessing execution time (s)	Trial
A,4,1,10	02s:696ms	2,9388	1
	02s:205ms	2,1954	2
	02s:205ms	2,0789	3
	01s:806ms	2,1595	4
	02s:154ms	2,1391	5

The test in Table V shows that the response time between commands sent from Apps to the middleware to respond is between 1 second to almost 3 seconds. Meanwhile, the process execution time starts from initialization to a process, and sending 1x data takes> 2 seconds.

TABLE VI
SERVO MOTOR ACCESS TESTING WITH MULTITHREADING

Notation	Response time (s)	Multiprocessing execution time (s)	Trial
A,4,1,10	01s:478ms	2,1688	1
	01s:579ms	2,2465	2
	01s:442ms	2,3921	3
	02s:581ms	2,6232	4
	02s:251ms	2,2477	5

The results of the multithreading servo motor access test can be seen in Table VI. The response time between Apps' commands until the middleware can respond is between 1 second to almost 2. While the thread execution time starts from initialization to becoming a thread, and sending 1x data takes> 1 second.

TABLE VII
CPU AND MEMORY USAGE TESTING FOR MULTIPROCESSING, MULTITHREADING, AND SEQUENTIAL SERVO MOTOR ACCESS

Multiprocessing(%)		Multithreading(%)		Sequential (%)	
CPU	Memory	CPU	Memory	CPU	Memory
1,70	2,70	0,70	2,70	0,70	2,70
1,10	2,40	0,70	2,70	0,70	2,70
1,20	2,70	0,70	2,70	0,70	2,70
1,70	2,70	1,30	2,70	1,30	2,70
1,80	2,40	0,87	2,70	0,70	2,70

Table VII is the test results of CPU usage and memory usage data between multiprocessing, multithreading, and sequential when accessing servo motors. This time, the function is also carried out individually, and 5 times the sampling is taken.

Comparison of CPU usage and memory usage when a servo motor function is accessed using multiprocessing or multithread or sequential programs is shown in Figure 14.

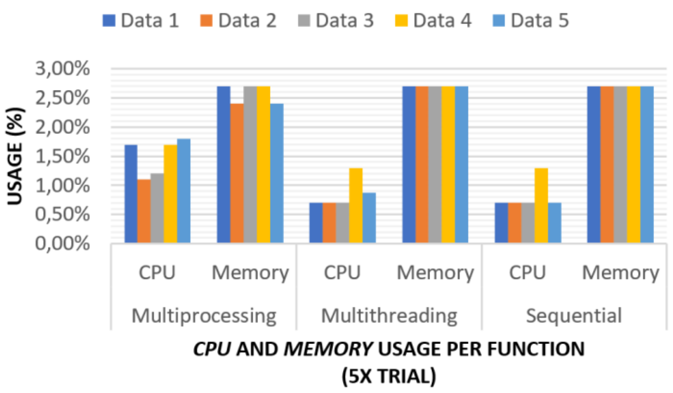

Figure 14. Comparison graph between CPU and memory usage on servo motor access

Analysis on servo motor testing using multiprocessing, multithreading, and sequential programs is that there are differences in the completion of the 1x program loop and CPU and memory use. After a 1x program loop, multiprocessing records a slightly faster time than multithreading or sequential. However, in experiments 1 and 2, the completion was slower. This happens because multiprocessing uses different addressing space and CPU-core in carrying out each process. While multithreading uses the same CPU-core addressing space, the process execution is slightly faster than the execution of a thread. A thread's execution also uses CPU usage and memory usage more because it is in the same addressing space. Besides, the difference in experiments 1 and 2 occurs because it coincides with extensive processes carried out by the OS so that the process carried out by the user is slightly slowed down. There isn't much difference between CPU usage and memory usage between multithreading and sequential. Several conditions indicate that a small task such as servo motor access is sometimes better done as a threat than a process because the job is only one time, not continuous like the sensor reading function.

IV. CONCLUSION

In multiprocessing, it will be more optimal to do complex or continuous work such as reading sensors. When used to access a sensor, a process is not always faster than a thread. Such as temperature sensor access, whose execution time is in the range <1,0001s, when it becomes a process and execution time, is in the range> 1,1001s when it becomes a thread. Meanwhile, multithreading will be more optimal for doing small or non-continuous work such as actuator access. When used to run sensors, not always a thread will execute faster than a process. For example, the servo motor access has an execution time of up to 2.9388s on the first try when it becomes a process and up to 2.6232s execution time on the fourth try when it becomes a thread.

ACKNOWLEDGMENT

The Highest appreciation for fellow researchers of the Embedded Systems Laboratory Computer Engineering Study Program, Surabaya State Electronics Polytechnic, and a big thank you to the CE IoT community members.

REFERENCES

[1] S. Li, L. Da Xu, and S. Zhao, "The internet of things: a survey," *Inf. Syst. Front.*, 2015, DOI: 10.1007/s10796-014-9492-7.

[2] J. F. Nusairat, "Raspberry Pi," in *Rust for the IoT*, Berkeley, CA: Apress, 2020, pp. 391–427.

[3] A. Jalil, "Pemanfaatan Middleware Robot Operating System (ROS) Dalam Menjawab Tantangan Revolusi Industri 4.0," *Ilk. J. Ilm.*, 2019, doi: 10.33096/ilkom.v11i1.412.45-52.

[4] M. R. Rizqullah, A. R. Anom Besari, I. Kurnianto Wibowo, R. Setiawan, and D. Agata, "Design and Implementation of Middleware System for IoT Devices based on Raspberry Pi," in *2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC)*, Oct. 2018, pp. 229–234, DOI: 10.1109/KCIC.2018.8628528.

[5] A. H. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and M. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling technologies," *IEEE Internet Things J.*, pp. 1–1, 2016, DOI: 10.1109/JIOT.2016.2615180.

[6] Y. Nakamura, H. Suwa, Y. Arakawa, H. Yamaguchi, and K. Yasumoto, "Design and Implementation of Middleware for IoT Devices toward Real-Time Flow Processing," 2016, DOI: 10.1109/ICDCSW.2016.37.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multi programmed Multiprocessors," *Theory Comput. Syst.*, vol. 34, no. 2, pp. 115–144, Jan. 2001, DOI: 10.1007/s002240011004.

[8] F. Menczer, S. Fortunato, and C. A. Davis, "Python Tutorial," in *A First Course in Network Science*, Cambridge University Press, 2020, pp. 221–237.

[9] R. Odaira, J. G. Castanos, and H. Tomari, "Eliminating global interpreter locks in ruby through hardware transactional memory," *ACM SIGPLAN Not.*, 2014, DOI: 10.1145/2692916.2555247.

[10] L. Moroney and L. Moroney, "The Firebase Realtime Database," in *The Definitive Guide to Firebase*, 2017.

[11] W. J. Li, C. Yen, Y. S. Lin, S. C. Tung, and S. M. Huang, "JustIoT Internet of Things based on the Firebase real-time database," 2018, DOI: 10.1109/SMILE.2018.8353979.

[12] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared-Memory Multiprocessing," *IEEE Micro*, vol. 30, no. 1, pp. 40–49, Jan. 2010, DOI: 10.1109/MM.2010.14.