International Journal of Robotics and Automation (IJRA)
Vol. 8, No. 4, December 2019, pp. 217∼244
ISSN: 2089-4856, DOI: 10.11591/ijra.v8i4.pp217-244

Concurrent asynchronous robotic subsumption using actors & promises

Ratnesh Srivastav[1], **G. C. Nandi**[2], **Rohit Shukla**[3], **Harsh Verma**[4]
[1,3,4]Department of Information Technology, College of Technology, Pantnagar
[4]Indian Institute of Information Technology, Allahabad

Article Info

Article history:

Received Jun 9, 2019
Revised Aug 20, 2019
Accepted Sep 11, 2019

Keywords:

Actor
Concurrency
Promise
Petrinet
Player
Stage
Subsumption

ABSTRACT

Fascinated with robotics open source integrated components MARIE [1], ROS [2] PLAYER/STAGE [3] and CARMEN [4] are available in the field of robotics application. We aim at developing an asynchronous concurrent robotics application based on subsumption [5] architecture using PLAYER/STAGE and already freely available services on the internet. We used wanderer and wall follower services embedded inside an actor [6] [7] and integrated these services to two different robots of same configurations in our promise based framework that provides actor as a service. The two robots switch their services based on color detected (red,green,blue) during their movement in an environment asynchronously. Using actor as a service fills the gap of SOA [8] & EDA [9] by providing synchronous and asynchronous support for communication. We measured the performance of time taken in completion of services in promise [10] based implementation, synchronous and asynchronous callback [11] based implementation. We developed a model to prove deadlock freeness in our integrated architecture using petrinet [12] interface composition. We have also been able to justify that component integrated on promise based framework takes less time in service completion than synchronous and asynchronous callback based services.

Copyright © 2019 Insitute of Advanced Engineeering and Science.
All rights reserved.

Corresponding Author:

Name: Ratnesh Srivastav
Affiliation: Assistant Professor
Address: Department of Information Technology, College of Technology, Pantnagar
Phone: +91-9458359031
Email: write2ratnesh@gmail.com

1. INTRODUCTION

For the evaluation of the field, it is required to integrate available services so that complex systems can be built upon these available services without wasting time in code replications. Components are built independently following an own set of protocols and communications strategies, integrating these together is not a trivial task. This is also favored because services that are well tested and debugged can be reused. Proposed Architecture is implemented using principles of Service Oriented Architecture(SOA) & Event Driven Architecture(EDA). We have used Service Oriented Actors which fulfill the gap of SOA and EDA by exposing services as actors and providing asynchronous communication. Here an actor containing a list of robotic services are exposed and used for communication by another client actors. Actor & Promise is used to facilitate asynchronous concurrent communication using promise and maintains the state of messages using actors. We have developed an extension of actor model discussed in [13] and provided an abstraction of promises to accept and deliver concurrent request and response. The service oriented actor provides services which have return type either void or promise. Two actors communicate with each other by implicit message passing. None of

the other actor-based framework for Java developed earlier supports promises and implicit message passing mechanism as they are developed on an older version of Java 8.0.

While developing the application we have kept in mind that integrated services should be cost efficient and easy. Moreover, architecture should be flexible enough to integrate modified and new service without obsoleting or rebuilding the architecture [14]. Such integration should not be too complex. Integrating new service to architecture should not involve rebuilding whole architecture again thus service can be connected on the fly to the architecture.

If service is on the host computer, it is being invoked from the same host and if not, service is requested from the host having that service. In case, if a service is not present then system should terminate gracefully with proper error messages so that user can diagnose the problem with the application created. SOA based architectural concept uses description language to define the function and services of the component interfaces to be discovered over the network. SOA facilitates isolation practice of software development. Earlier SILO [15] based approach had been inefficient where same functionalities were being developed, deployed and maintained multiple times by other organizations refer to Figure 1 and 2.

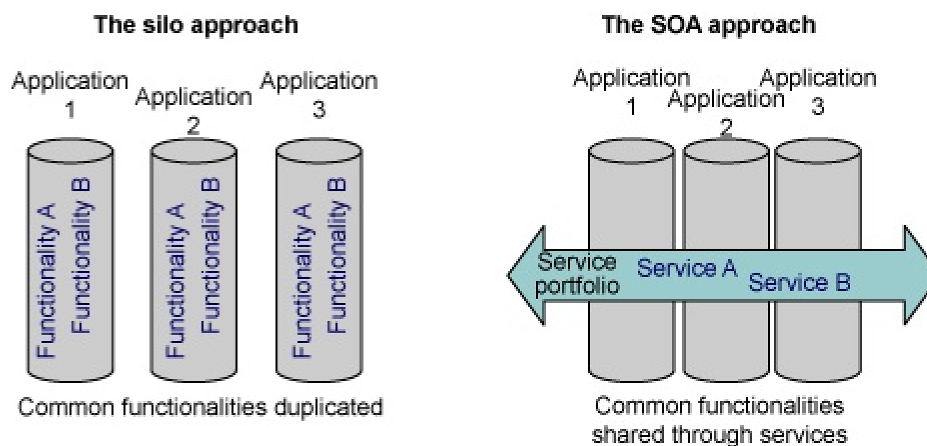

Figure 1. SILO Vs SOA

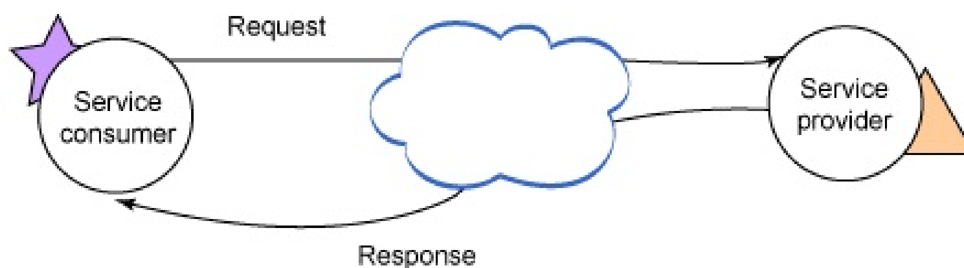

Figure 2. Request/Reply Mechanism in SOA

Gartner, in 2003 proposed a new design paradigm based on events using publisher/subscriber pattern as event-driven architecture(EDA). EDA is not a replacement of SOA but complements SOA by asynchronous message communication than request/reply used in SOA.

Every actor writes their messages to other actor's mailbox for sending their request to be processed. The response sent by another actor is written back to callback queue of the actors. Every actor reads its mailbox queue after reading the messages of its callback queue. Finally, each actor is copied to global actor space of a single thread. Thus works as a single threaded model and each actor may work as publisher and subscriber. We are using our actor-based framework as shown in Figure 3, it uses different actors with their mailbox and callback queue.

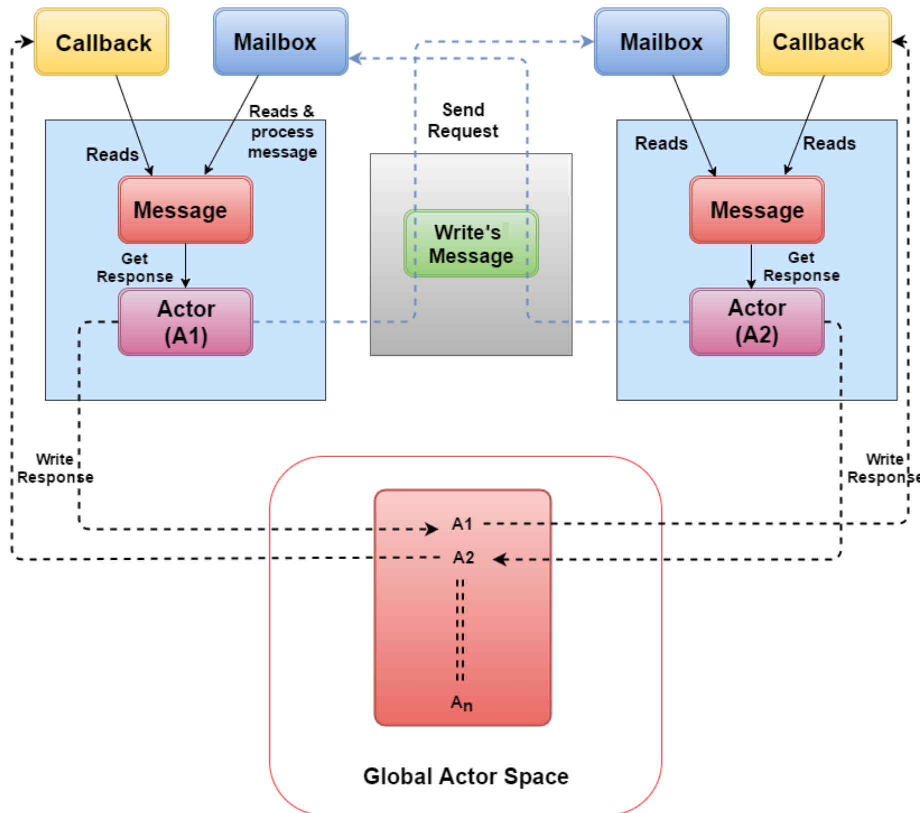

Figure 3. Request/Reply in actor-based framework

This paper is organized as followed: section II introduces the implementation architecture, section III models services and discuss the challenges and implementation details of subsumption architecture. Section IV discuss the performance evaluation of services on synchronous vs. asynchronous using callback vs. asynchronous using promises, section V deals with verification of deadlock in the composed system using Petri net model and finally section VI introduces conclusion and future work.

2. IMPLEMENTATION ARCHITECTURE

Our architecture integrates loosely coupled components based on Service Oriented Actors. An application needs to be developed using the functionalities provided by different components. A single component may provide more than one functionality which may or may not interrelated, such as audio and video processing or collision avoidance and path planning. Further, it's possible that these packages are not present on the same machine hence creating a distributed environment. At the highest level of abstraction, the proposed architecture will look as illustrated in Figure 4.

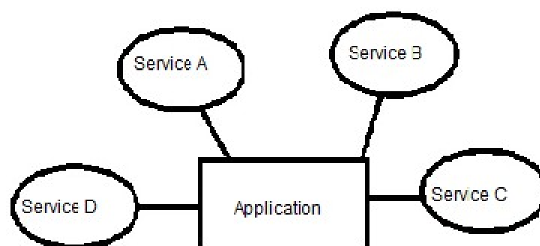

Figure 4. High level abstraction of Application

Proposed architecture can be understood from the following two perspectives:
A. Abstraction Layers
B. Component Framework

2.1. Abstraction Layers

To provide multiple levels of abstraction three layers are defined in proposed architecture [16]. Each layer provides the certain set of functions to the layer above it. The three layers are illustrated in Figure 5.

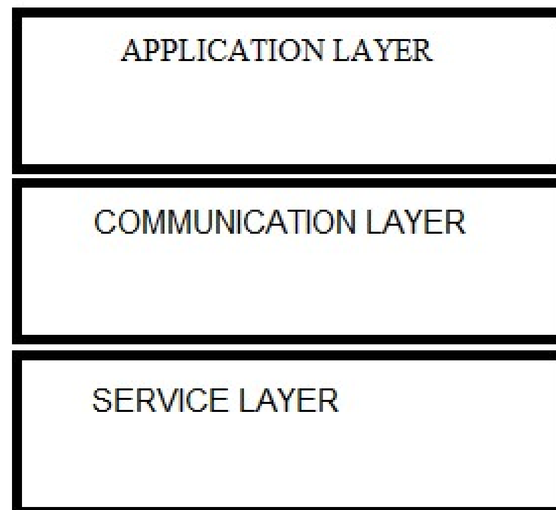

Figure 5. Different Layers of Architecture

2.1.1. Service Layer

It is the lowest layer where the services reside. This layer is also the core layer having all the functionalities required by the application need to be developed. Each service is nothing but functionality that a component provides [17]. All the components integrated are an integral part of the Service Layer. These components may provide much functionality. These packages may be heterogeneous, built in different programming languages, using different communication protocols and may present on different machines. Hence in SOA's terminology Service layer is a collection of Service Providers.

2.1.2. Communication Layer

This layer is present above Service layer and responsible for all types of communication between the services and application. This layer defines the communication mechanism, strategies, and protocol. The communication mechanisms used are publish/subscribe and request/reply [18].

2.1.3. Application Layer

This layer sits right above Communication Layer. The user interacts with this layer. It provides functions like service discovery, service invocation and synchronization between services.

2.2. Component Framework

Proposed architecture composed of different building blocks. These blocks are services, interfaces, bus, broker and application. These building blocks are defined below in Figure 6, which shows component together with the interface internal working.

2.2.1. Service

Each of the robotics components can be treated as the service. The application requires these services to perform the designated task. A service may provide many functionalities like navigation and path planning.

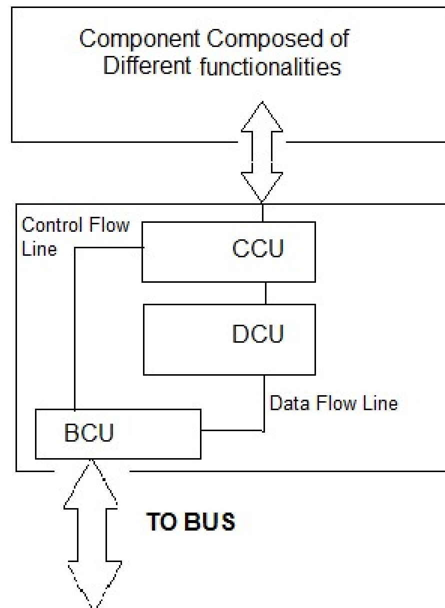

Figure 6. Interface

2.2.2. Interfaces

The interface is component specific that makes component compatible with architecture. Such an interface can be considered as middleware between architecture and component. All the communications between component and architecture occur through the interface only. All the components need to be connected, they must have an interface. If a new component needs to be connected then an interface is required according to requirement of that component. Developing an interface requires the deep study of a component to be connected for the insight knowledge of communication protocols, data formats, and robotics standards. Interface has different units as listed below:

2.2.2.1. Component Communication Unit (CCU)

This unit communicates with the component, more precisely with the service in which an application is interested. CCU supports both control signal and the data signal. Control signals are responsible for invoking and termination of required service. Data signals are for the transfer of service specific data.

2.2.2.2. Data Conversion Unit (DCU)

Robotics packages are data specific. They require the data in particular format and type. This unit is responsible for converting data from application-specific data to service specific data and vice versa. This data is read and written to the buffer built inside DCU for providing latency.

2.2.2.3. Bus Communication Unit (BCU)

It is responsible for communication between interface and bus and vice versa.

2.2.3. Bus

The bus is a messaging infrastructure to allow different systems to communicate through shared set of interfaces. This is analogous to bus in a computer system which serves as the focal point for communication between CPU, Main memory, and peripherals. Messages may be data messages or control messages. Control messages enable managing services while data messages are service specific data that is input and output data of services.

2.2.4. Broker

It synchronizes different services. The duty of the broker/arbitrator is to discover services required for the application to work. For this record of services are maintained for rapid service discovery. This also involves invoking the service in case service is not running.

3. IMPLEMENTATION DETAILS

Problem Statement:

There are two robots which we need to run asynchronously from a single JVM. We need to implement subsumption of the services by using different colors. There are two main services (wall follower and space wanderer) which can be run on any of the robots. These robots are equipped with sonar, position2d, and blob finder sensors. We want to implement this subsumption architecture on promise based asynchronous framework by measuring the processing time of services running asynchronously. We have an actor as a service provider for providing asynchronous services wall follower and space wanderer. Two robots (robot1 and robot2) as an actor using or switching these services based on the color detection. An actor as an arbitrator is used to switch service from one robot to another, based on the color detection.

The challenges in the implementation are:

[i] How to ensure required sensor data is ready before invoking the service?

[ii] To define a rule to switch services from one robot to other based on the color detection.

[iii] Starting and stopping services running on different robots asynchronously based on color detected.

[iv] How to update the color detected by a robot at the run time to the main code?

[v] How to model a system which has N number of services running on N different robots detecting N number of different colors?

[vi] How to verify that integrated components do not produce any deadlock?

We have two robots:

1. Robot1 (id = 0, port 6665, color orange)

Sensors

i. Sonar sensor (distance with the obstacles)

ii. Position2d (used to control the robot movements)

iii. Blob finder (used to detect the color obstacles in range) and

2. Robot2 (id = 1, port 6665, color black)

Sensors

i. Sonar sensor

ii. Position2d

iii. Blob finder

They are connected with PLAYER server for transmitting and receiving data on the network and the simulation environment is provided by STAGE. The details of the two services are:

3.1. Wall follower service

In this service, first, the connection is established with the robot. After connection, the sonar values are checked if they are ready or not. When sonar values are ready, these values are used to find the distance of the obstacles in front and left side. First, the robot is aligned to the left wall then the robot is made to follow the wall. The logic behind alignment with the wall and follow the wall is written separately. Both the codes are differentiated by a temporary (temp) variable. First, the robot left side is aligned with the wall (Table 1). When the robot left side is aligned with the wall, the value of the temporary (temp) variable is changed to 0 which causes second part of the code to run i.e. follow the wall. The logic behind getting the wall aligned to its left side is that the robot will move forward until its left side sonar or front side sonar detects any obstacle.

Max wall threshold = 0.4 cm

Min wall threshold = 0.3 cm

x: translational speed &

y: angular speed (radian/sec)

If none of these conditions is satisfied then our robot detected the wall to be followed and conditions in Table 2 are checked. This phase is known as following the wall present at left side. The temporary (temp) variable value is made 0 because there is no need to align the wall again to the left.

These conditions in Table 2 help the robot to follow the wall. If the wall in the front is less than max wall threshold then the robot will turn right and if the left side wall is too near then also the robot will turn right. If the left side sonar value is more than max wall threshold then it will turn left. By default, the robot will move forward when no above condition is satisfied. There is a priority order in the above conditions. Upper conditions have more priority then below conditions.

Table 1. Logic for wall alignment to the left side of the robot

Left sonar value	Front sonar value	Result
<max wall threshold	>max wall threshold	Move forward (x=0.2, y=0)
<previous left side	<Left side	Turn right (x=0, y= -0.45)
<previous left side	>left side	Turn right (x=0, y= -0.15)

Table 2. Logic for following the wall

Left sonar value	Front sonar value	Result
No condition	<max wall threshold	Turn right (x=0.1, y= -0.6)
<min wall threshold	No condition	Turn right (x=0.1 , y= -0.15)
>max wall threshold	No condition	Turn left (x= 0.1, y= 0.15)
By default(no condition)	By default(no condition)	Move forward (x=0.2, y=0)

3.2. Space Wanderer and Color Finder

In this service, first, the connection is established with the robot. After connection, the sonar values are checked if they are ready or not. When sonar values are ready, these values are used to find the distance of the obstacles in right and left side of the robot. In this service, a robot is made to travel in between the obstacles or wall and if the left sonar distance is less than right sonar distance then turn right with y = 0.5 radian/sec and x =0. To move in between the walls, x is set as ((left/10)+(right/10))/2 and y = ((left/10)-(right/10))*(180/pie)/wheel diameter.

x is translational speed and y is rotational speed in radian. While the robot wanders, the color finding service is also running. This service uses blob finder sensors to detect the color. The logic for sending the color to the main code is different in callback and promise.

3.3. Synchronous, Callback and Promise based implementation

When subsumption occurs, or change in the running state, services switches to other services. The switching is fast so that robot does not colloid with the wall when no one is controlling the actions of the robot.

The code shown in Figure 7 has used runThreaded a function of the player client. This function calls the start() function of thread class. Run function is called asynchronously. Run function in player client interact with the player server and gets the sensors values. This function updates the isDataReady function of player client about the sensors value. isDataReady function is used in the services for checking the sensors values are ready or not. When isDataRedy function replies truly then only we start the service. This is the solution to the first challenge discussed above.

```
PlayerClient        robot = null;
Position2DInterface posi1  = null;
SonarInterface      rngi1  = null;
BlobfinderInterface blfi1  = null;

try {
    // Connect to the Player server and request access to Position and Sonar
    robot  = new PlayerClient ("localhost", 6665);
    posi1 = robot.requestInterfacePosition2D (0, PlayerConstants.PLAYER_OPEN_MODE);
    rngi1 = robot.requestInterfaceSonar     (0, PlayerConstants.PLAYER_OPEN_MODE);
    blfi1 = robot.requestInterfaceBlobfinder (0, PlayerConstants.PLAYER_OPEN_MODE);
} catch (PlayerException e) {
    System.err.println ("robot1: > Error connecting to Player: ");
    System.err.println ("    [ " + e.toString() + " ]");
    System.exit (1);
}

robot.runThreaded (-1, -1);
```

Figure 7. Connection with robot sensors

Here we have used color as subsumption event. If a color detected other than orange (another robot color), black (walls) and the color of the state they are in, then the state of the model will change. On the basis of color detected by the robots, service is decided. We have two main services, the possible arrangement of services among two robots is 4 as shown in Table 3.

Table 3. Possible arrangements of services

Robot1	Robot2
Wall Follower	Space Wanderer
Space Wanderer	Wall Follower
Space Wanderer	Space Wanderer
Wall Follower	Wall Follower

The state transition diagram of the services based on color detection is shown in Figure 8, which works on 3 states of the model.

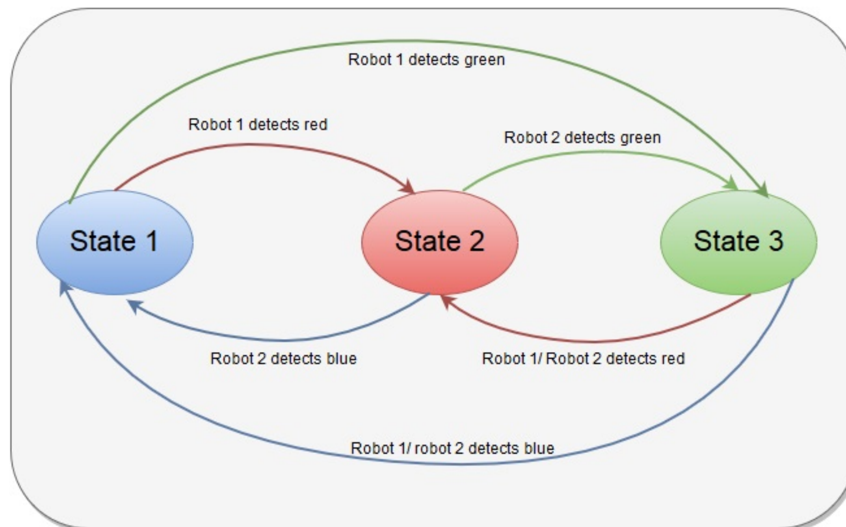

Figure 8. State transition diagram of service switching

In state 1, as shown in running demo of Figure 9, wall follower service in robot1 and space wanderer with color finder service in robot 2, it is default services. This state occurs when robot senses blue color. Note, color finder service is only with space wanderer service. Only the robot running the space wanderer can detect the color and perform the subsumption of its own service with the other robot service.

In state 2, as shown in running demo of Figure 10, space wanderer and color finder service runs in robot 1 and wall follower service runs in robot 2. This state occurs when robot senses red color.

In state 3, as shown in running demo of Figure 11, both the robots runs with space wanderer and color finder service. This state occurs when robot senses green color. In this state, both the robots can detect the color and change the state of the model.

The switching of the services can be depicted as in Figure 12, which works on the following rules which solves our second challenge discussed above are:

[1] The default condition for state1 is, lets assume this condition as A= robot1.color != black & robot1.color != orange & robot1.color != state.color.

[2] The robot changes its priority and goes to state2 based on condition, lets assume this condition as B= robot1.color = blue or robot2.color = blue.

So priority of services are (robot1.service = SW >robot1.service = WF & robot2.service = SW <robot2.service = WF) if condition A and B are true.

[3] Again the robot changes it's priority and goes to state2 based on condition, lets assume this condition as C = robot1.color = red or robot2.color = red.

So priority of services are: (robot1.service = SW <robot1.service = WF & robot2.service = SW >robot2.service = WF) if condition A and C are true.

[4] The robot changes it's priority and goes to state3 based on condition, lets assume this condition as D = robot1.color = green or robot2.color = green.

So priority of services are (robot1.service = SW >robot1.service = WF & robot2.service = SW >robot2.service = WF) if condition A and D are true.

where State.color: color of the present state, WF: Wall follower service, SW: Space wanderer service, Robot1.color: color detected by robot1 and Robot2.color: color detected by robot2. A Table 4 based on the above rule is presented below.

Table 4. Conditions and outcome of different states

Blue Color	Red Color	Green Color	Robot1	Robot2	State
Yes	No	No	Wall Follower	Space Wanderer	1
No	Yes	No	Space Wanderer	Wall Follower	2
No	No	Yes	Space Wanderer	Space Wanderer	3

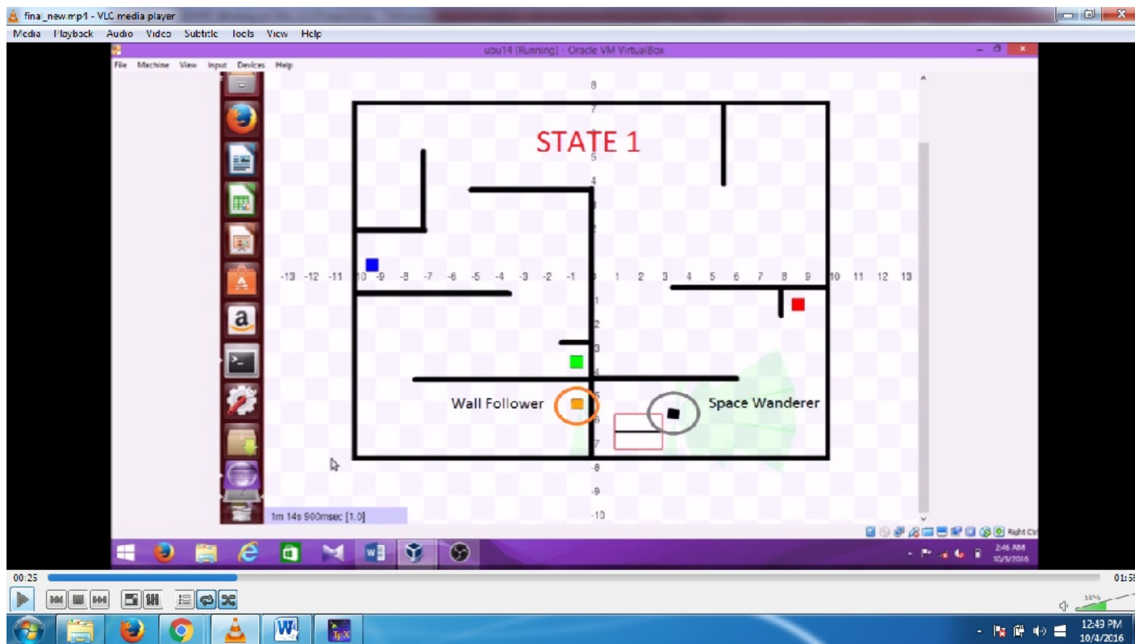

Figure 9. Running demo of two robots exchanging their services on color detection in state1

The BIP [19] model of the implementation of the problem is presented below in Figure 13 for the two robots. We have used 3 different implementations of the same problem using:

i) Subsumption using Synchronous mode.

ii) Subsumption using Asynchronous model of Actor & Actor with callback.

iii) Subsumption using Asynchronous model of Actor & Actor with Promise.

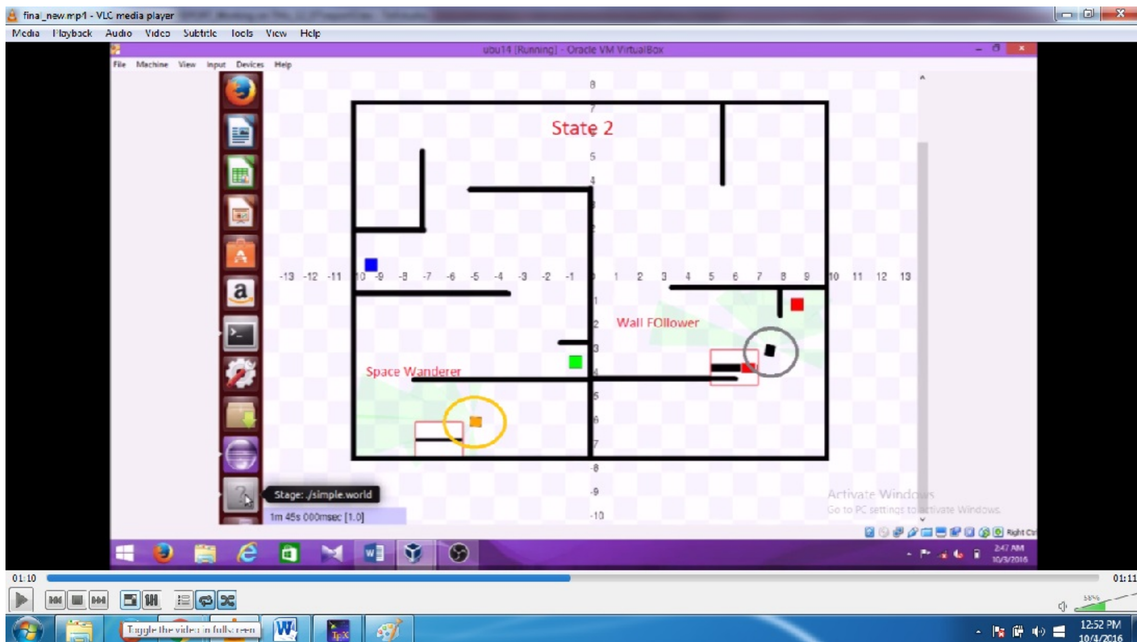

Figure 10. Running demo of two robots exchanging their services on color detection in state2

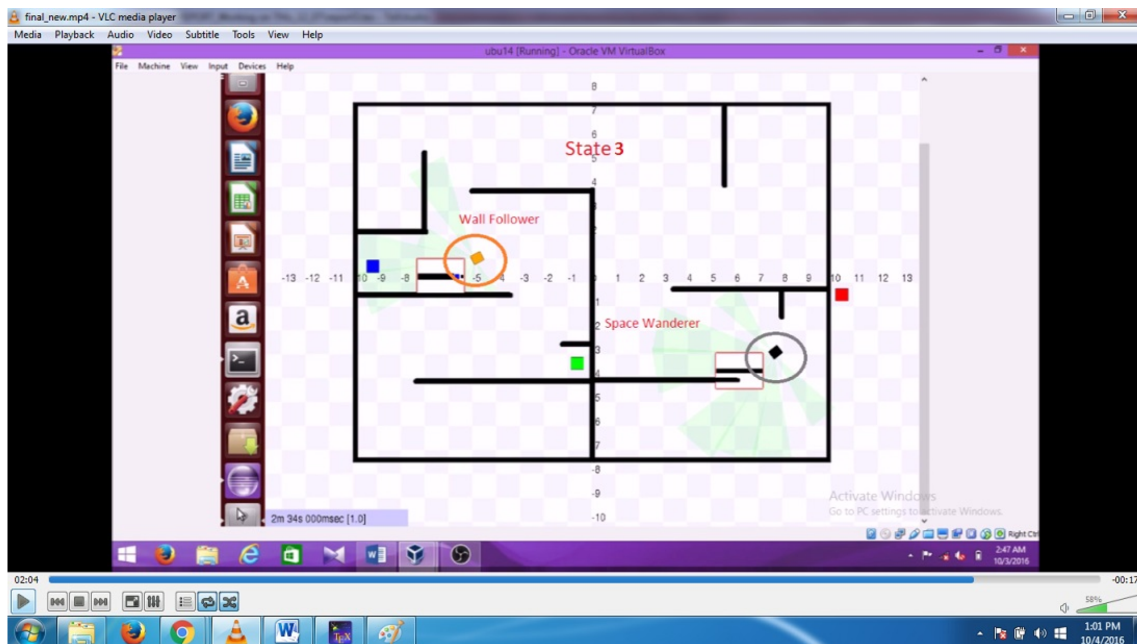

Figure 11. Running demo of two robots exchanging their services on color detection in state3

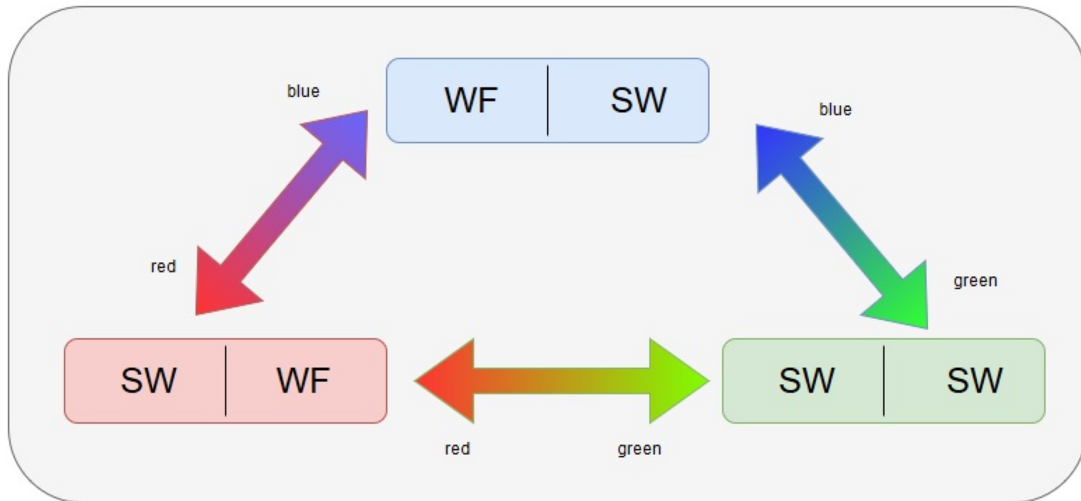

Figure 12. State transition diagram of service switching

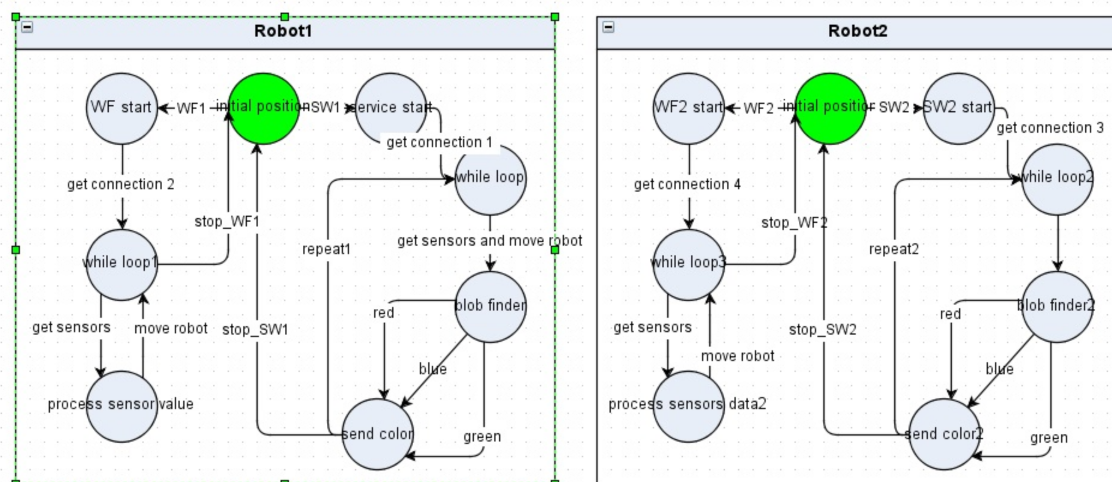

Figure 13. BIP model of two running robots switching their services

3.4. Subsumption using Synchronous model

Wall Follower service and Space Wanderer service functions are present in a single file. The connection with the robots is done only single time in the starting of the main code. Then in while loop, separation of colors are done for the states or subsumption. When the service starts in a state, the functions of service are repeatedly called till any different color is detected. This color should be different than the present color of the state, not equal to 0 (black) and orange (another robot color). When a color is detected, then subsumption occurs and state changes accordingly. Services are running one by one. For example, in state 1, robot1 is running wall follower and robot2 is running space wanderer.

In the Figure 14, the synchronous implementation is shown, when wall follower service completes its cycle then space wanderer cycle will start. At a single time, only single service is running and commanding the respective robot. The switching is fast (200-300 milliseconds) therefore we cannot differentiate with our eyes.

We see both the robots are running their services. The total time taken is equal to a sum of the time taken by both the services. In the code, color1 is for saving the color received by the space wanderer service and later used for checking the conditions for switching the states. Color4 variable temporary saves the value received by the service, before copying the color4 value in color1, color4 is checked for black color and orange color because these colors are not used in subsumption.

```
if(color1 == 255){
    System.out.println("State 1");
    do{
wallF(posi1, rngi1, blfi1);        ——> WallFollower
color4 = SW( posi, rngi, blfi);   ——> Space Wanderer
System.out.println(color4);
if(color4 !=0 && color4 != 16753920){
    color1 = color4;
}
```

Figure 14. Code for services running synchronously

We compared the time taken to complete 1000 loops of both the services in core2 and core3 machine, shown by graph presented in Figure 15.

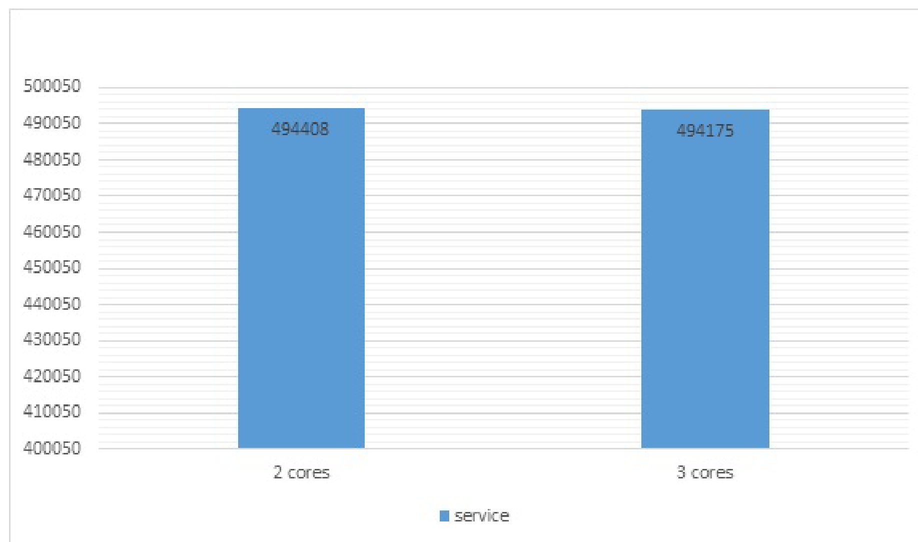

Figure 15. Time taken in running services synchronously

3.5. Subsumption using an Asynchronous model of Actor & Actor with callback

Wall follower and Space Wanderer services are inside an actor and main code for calling the services are in another actor. Two reference variables of services class are made as actor. From the main code functions of services are called to run the services in the robot. Connection and control of the robot is defined in the services. There is a reconnection every time when the services are switched or subsumption occurs.

In the Figure 16, the services are called asynchronously, it is difficult to stop the service from other thread. For stopping and starting a service, AtomicReference is used. The AtomicReference class provides an object reference variable which can be read and written atomically. By atomic meant that multiple threads attempting to change the some AtomicReference will not make the AtomicReference end up in an inconsistent state. Before starting the service in a robot, atomic reference of that service is set to 1. There is a loop in the service which checks the value of the atomic reference of the service. When we need to stop the service, we can set the value of the atomic reference 0 from the main code. For this purpose, we pass the atomic reference and id of the robot in the Wall follower service. This solves our third challenge of starting and stopping robot services. The id of the robot is used for connection with the robot. Space Wanderer has two services, therefore, two reference ids are passed for a robot id. To deal with the fourth challenge as above we have used a callback. For callback, callback as a function passed to the function. The color detected other than black is directly send to the main code where the color is checked. If the color defined for the running state is same as the color detected then no change in the state or subsumption. If the color detected is the orange i.e. color of another robot then it is also avoided. If a color other than these defined for different states detected than the change in the state or subsumption is done. It is faster than the synchronous model.

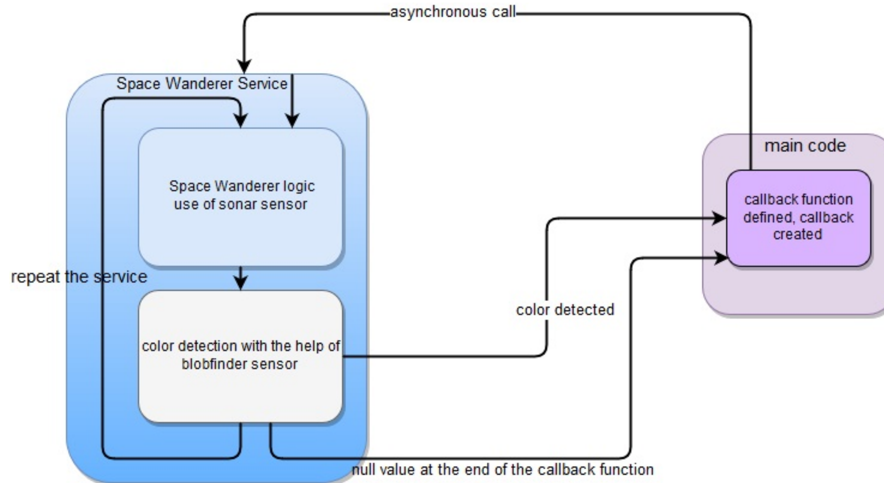

Figure 16. Messaging between space wanderer services with main code using callback

The code presented for this model is given in Figure 17 below:

```
    if (color2==255){
        System.out.println("State 1");
        while(!(atomicRefSW2.get()==2));
        atomicRefWF1.set(1);

        atomicRefSW2.set(1);
        atomicRefBF2.set(1);
        robot2.spaceWanderer((ch,err) -> {          Space Wanderer
            color2 = (int) ch;
            System.out.println(color2);
            System.out.println("received color "+System.currentTimeMillis());
            if(color2!=255 && color2!=16753920){
            atomicRefWF1.set(0);
            atomicRefSW2.set(0);
            atomicRefBF2.set(0);
            ((Demo2) self()).live();
            }

},1, atomicRefSW2,atomicRefBF2);                     Wall Follower
        robot1.wallFollower(0, atomicRefWF1);
```

Figure 17. Code for Actor model using callback

In this code, the color2 is used for saving the color received from the space wanderer service. Color2 variable is used for switching the states by checking its value. When color is received and satisfy all the conditions, all the services are stopped by changing the values of atomic reference of respective services and then thread recalls the function live().

In the above case, we also compared the time taken to complete 1000 loops of both the services in core2 and core3 machine, shown by graph presented in Figure 18 below:

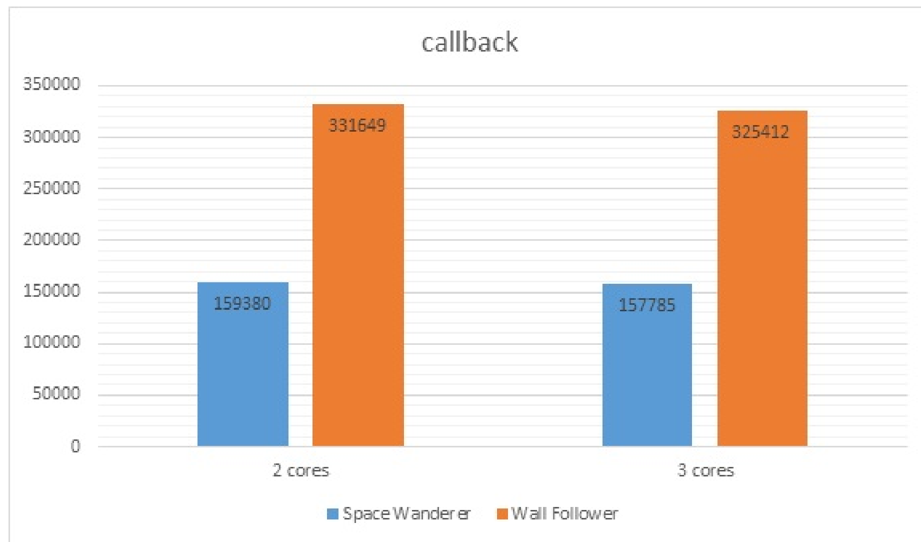

Figure 18. Time taken in running services asynchronously using callback

3.6. Subsumption using Asynchronous model of Actor & Actor with Promise

In the Figure 19, all the services are same as Actor with callback and use of atomic reference is also the same. The change is that the space wanderer is now had promise return type, which contains the color detected. The function called from main code provide atomic references, robot id, and the color of the present state as an argument. We use then function to use the color received after completion of the function. This color is used as a control point for subsumption by selecting suitable services. For subsumption, all the 3 states as discussed above, logic is written in a single function of the main code. The code presented for this model is given in Figure 20.

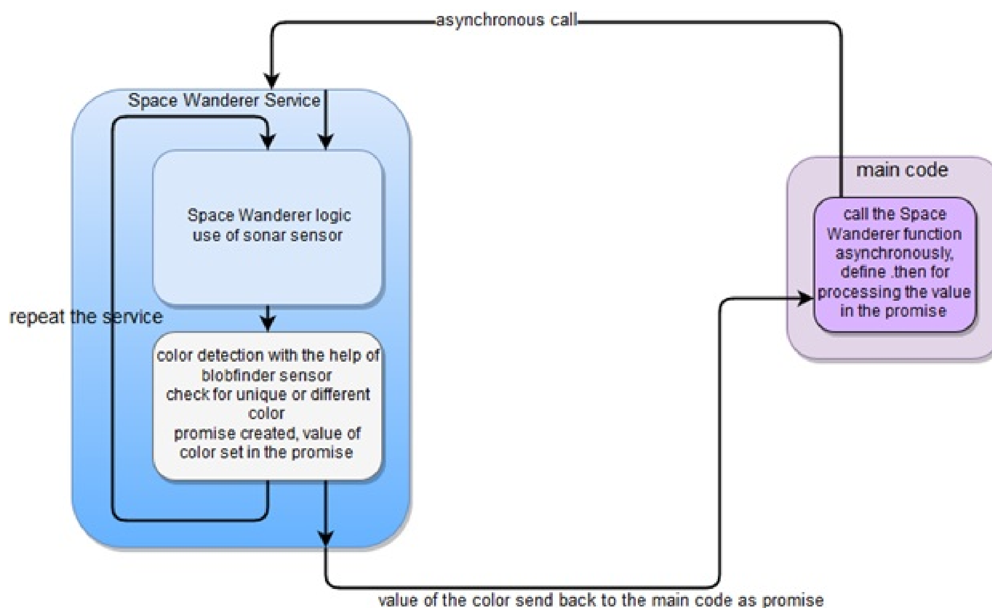

Figure 19. Messaging between Space wanderer service with main code using promisek

```
if (color2==255){
    System.out.println("State 1");
    atomicRefWF1.set(1);
    robot1.wallFollower(0, atomicRefWF1);    Wall Follower
    atomicRefSW2.set(1);
    atomicRefBF2.set(1);
    robot2.spaceWanderer(1,atomicRefSW2,atomicRefBF2, 255).then(result-> {
        color2 = result;                      Space Wanderer
        System.out.println("color"+color2);


        atomicRefSW2.set(0);
        atomicRefBF2.set(0);
        ((Demo3) self()).live();

    });
```

Figure 20. Code for Actor model using promise

By default value of color is blue. This causes state 1 run first. Any service, which needs to run on a robot, is set to 1 for its atomic reference and then call the respective function from the actor. In the space wanderer call, after receiving the color, wall follower service and space wanderer service is switched off by making their atomic references to 0. After this, the function calls itself. The function checks the color value and selects the state. This process repeats again and again.

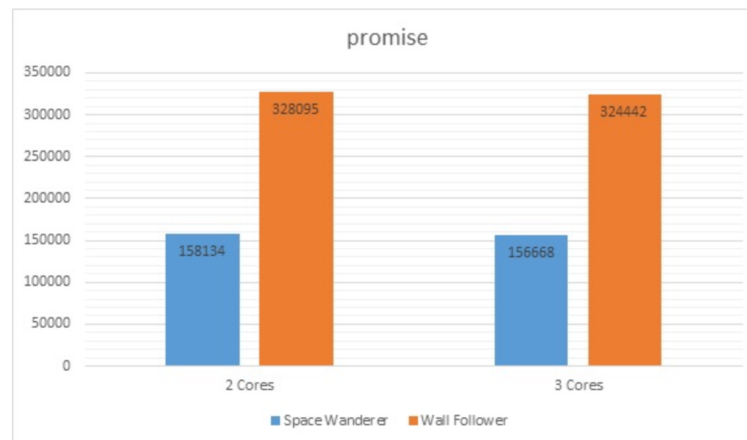

Figure 21. Time taken in running services asynchronously using promise

It is faster than synchronous and actor with callback. It sends a color only single time whereas a callback sends the color repeatedly till the service is switched off.

In this code, color2 is set as 255 (color code for blue) in the parameter of the space wanderer service. Color2 variable is used for switching the states by checking its value asynchronously using then method. When the color2 is received other than blue, all the services are stopped by changing the values of atomic reference of respective services and then thread recalls the function live(). In the above case, we also compared the time taken to complete 1000 loops of both the services in core2 and core3 machine, shown by graph presented in the Figure 21 above which shows that services running on promise takes less time than callback.

4. PERFORMANCE EVALUATION

Space Wanderer service takes very less time than Wall Follower in covering 1000 loops of the service. Y axis: time in milliseconds & number of loops of service: 1000 & Cores = 3. Time difference between synchronous model, an actor with callback model and the actor with promise model. Y axis: time in milliseconds & number of loops of service: 1000 & Cores = 3.

The graph shown in Figure 22 shows that there is not much time difference between actor with callback model and actor with promise. The small time difference is because callback code needs to send the color back to main code repeatedly whereas in promise only one color is sent back to the main code in the end. To receive the color sent by the callback, the main function needs to synchronize with the callback function and process the data for subsumption. In the callback case, the main function stops the space wanderer service by atomic reference after processing the data received whereas in promise case, the space wanderer service close itself and send the color value as a promise to the main code. For time calculation, only the main logic part is used. Connection with the robot is not taken into consideration.

In callback and promise, the while loop in the services are used in time calculation whereas in a synchronous model, both the function of the services are used for time calculation. The time taken by the callback function to send messages to the main function is almost equal to 1-2 milliseconds. In promise, the message is sent only once after closing the space wanderer service.

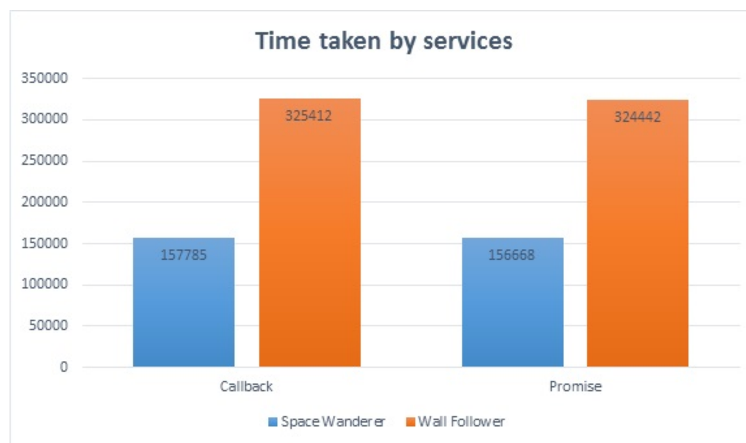

Figure 22. Time taken in running services asynchronously using promise and callback

The graph shown in Figure 23, depicts overall time taken by running services synchronously, asynchronous with callback and promise. The time taken in running services synchronously takes much time than the other because both the services run one after the other.

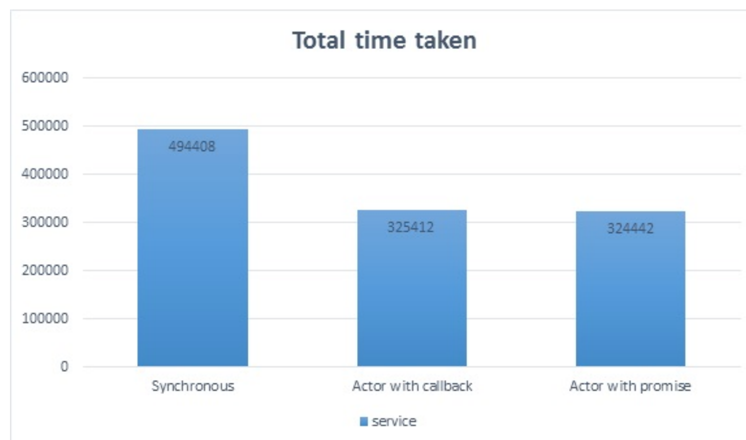

Figure 23. Time comparison graph in running services asynchronously

5. VERIFICATION OF MODEL WITH THE HELP OF PETRI NET

A Petri net (also known as a place/transition net or P/T net) is one of the several mathematical modeling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, represented by bars) and places (i.e. conditions, represented by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows).

There are four primary components involved in the system: the robot, player server, middleware as promise based framework and services as shown in Figure 24. Service interact with middleware only. Other three interfaces of robot, player, and middleware interact with one another at some point during their respective lifetimes.

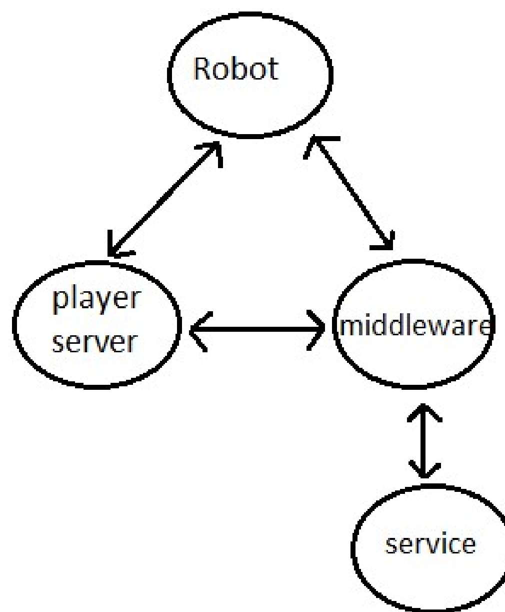

Figure 24. Component model of a Robot system

The basic sequence of events or use case can be outlined as follows:

[1] The robot is equipped with sensors. The robot takes the data of Laser Sensor/IR Sensor and put them in the proxies of the Player Server. Since the robot initiates the interaction, it is the requester and player component is the provider.

[2] Player Server component sends the data to the middleware and then player server initiates, it becomes requestor and middleware become the provider.

[3] The service component interacts with Middleware in order to register its services and to obtain the data. During the first phase, the role of the service component is requester and the middleware is provider. In the second phase, the role changes as the middleware initiate and sends the data to the services, it becomes requestor and service component becomes the provider. The reference of player sensors proxy is passed to the services.

[4] Once the service component has all the permitted information than a decision is made to start the service. Here it chooses between wall-follower and wanderer services.

[5] Finally Service is started and the Robot continuously sends the sensors data to the services component indirectly through player server and the middleware. This is continued till the robot wants to stop.

5.1. Robot Interface:

When the robot is started, it interacts with the player server and creates player proxies. Here Robot is requester and player is the provider. The values from the sensors present in the robot are now set in the proxies of player. Robot Interface is shown in Figure 25.

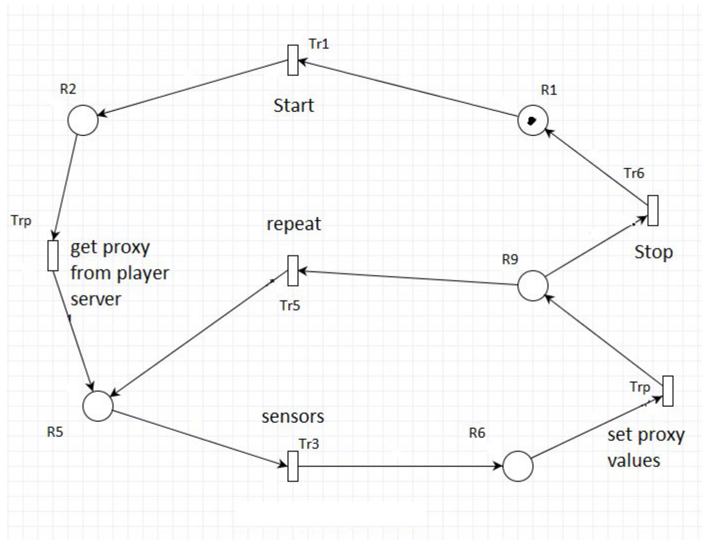

Figure 25. Robot interface

5.2. Player Interface:

Player provide proxies to the robot. The values are set on these proxies by the robot. These proxies are now transmitted to the middleware. Setting of the sensor data in the proxies and transmitting it to the middleware is continuously done till the robot is stopped. Player Interface is shown in Figure 26.

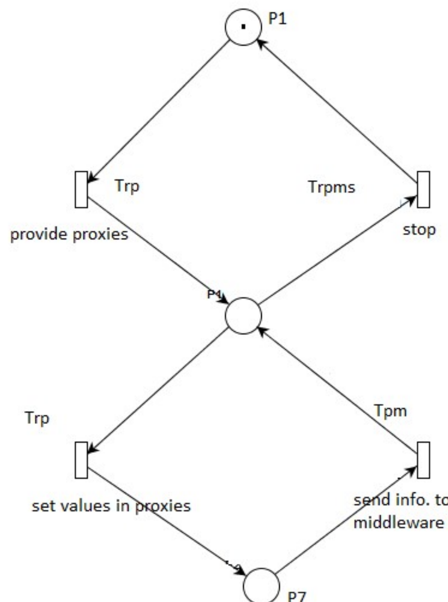

Figure 26. Player Server Interface

5.3. Middleware Interface:

Services are registered by the service in the middleware. After that, it receives the data from the player server. Middleware provides this data to the service and request to run the service. Receiving of data from the player server and providing it to the service interface continues till the robot is stopped. Middleware Interface is shown in Figure 27.

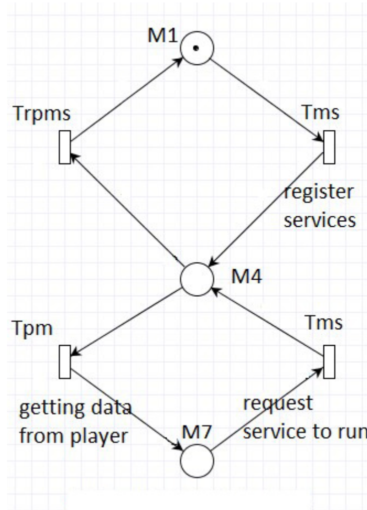

Figure 27. Middleware Interface

5.4. Service Interface:

Service register its services to the middleware. In this case, it registers wall follower service and wanderer service. Service receive the data from middleware and decide which service needed to run. This process continues till the robot is stopped. Service Interface is shown in Figure 28.

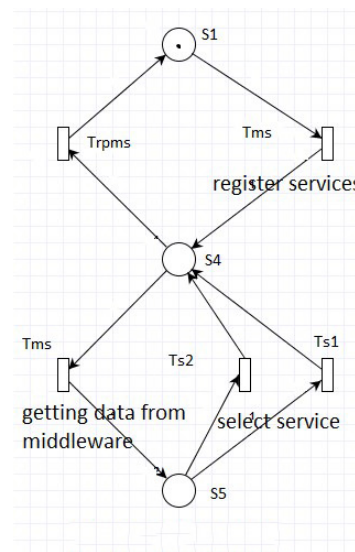

Figure 28. Service Interface

5.5. Composition

The composition of requestor and provider net is done by the melding of requestor interface with provider interface. The resultant composition has 4 new place and 3 new transitions for each common single transition. The transition is removed and 2 new places added in the requestor. The other 2 transitions added are synchronized with the provider. The place and transition added before the synchronous transition in the requestor are for removing the problem caused by the provider which changes the sequence of the requested operations. They allow the interaction with the provider and maintain the requestor sequence. In the composite net, the sequence of operations of the provider should not affect the sequence of requested operations. The other place of the requester is in between the synchronous transitions.

5.5.1. Robot-Player Composition

The composition of the robot request for the proxies from the player server composition is shown in Figure 29. R3", R4", P2", P3" and Tr2", Trp1", Trp2" are new places and transitions respectively, added in the net for composition. Tr2" and R3" added in the robot (i.e. requestor) before synchronous transition (Trp1"). They allow the requestor to interact with the provider. The initial transition of the requestor is removed and two synchronous transitions (Trp1" and Trp2") and 1 place (R4") added. In the provider side, the initial transition (Tp1) remain unchanged between two new places P2" and P3". They all together are in between the synchronous transitions Trp1" and Trp2". These synchronous transitions are connected with the both player server (provider) and robot (requestor), places either sides (Trp1" input side and Trp2" output side). In this composition Figure 30, the robot set values in the proxies of the player server. Here robot is requestor because it initializes the interaction with the player server. Tr4", R7", R8", Trp3", Trp4", P5" and P6" are new transitions and places added in the robot and player server interface for composition.

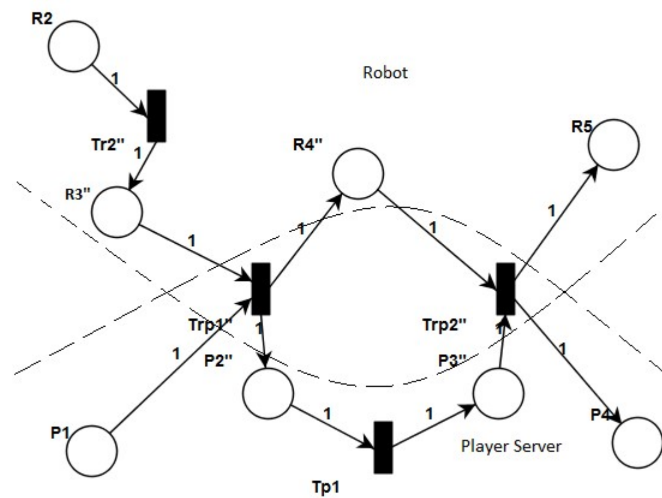

Figure 29. Robot-Player Composition-I

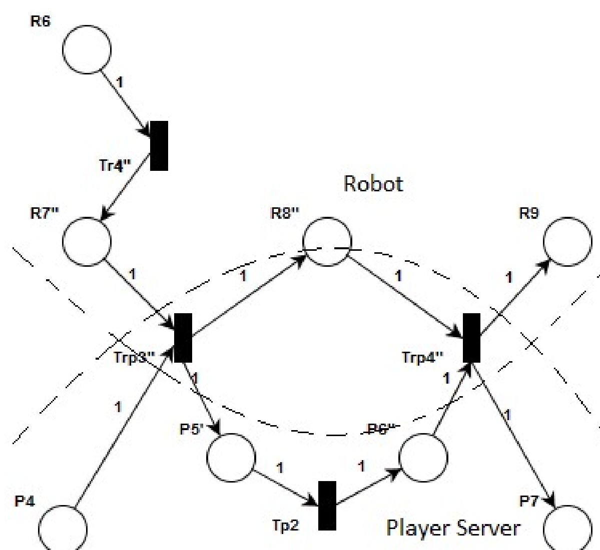

Figure 30. Robot-Player Composition-II

5.5.2. Player Server-Middleware Composition

In this composition shown in Figure 31, the player server sends the sensors data to the middleware. Here player server is requestor because it initializes the interaction with middleware. Tp3", P8", Tpm1", Tpm2", P9", M5" and M6" are new transitions and places added in the player server and middleware interface for composition.

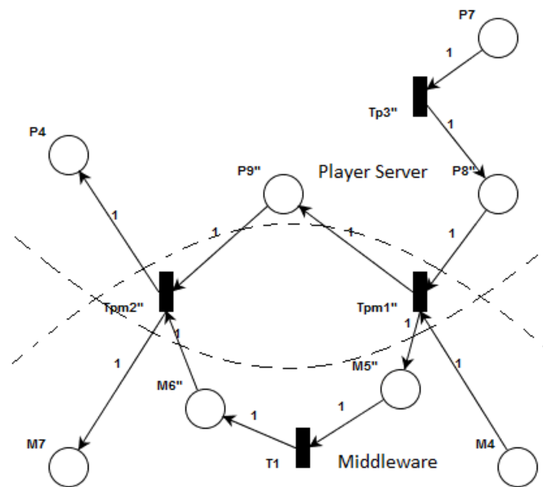

Figure 31. Player Server-Middleware Composition

5.5.3. Middleware-Service Composition

In this composition shown in Figure 32, 33, middleware sends the data to the service. Service after receiving data, it selects the service needed to run. Here middleware is requestor because it initializes the interaction with service. Tm2", M8", Tms3", Tms4", M9", S5" and S6" are new transitions and places added in the middleware a service interface for composition.

In this composition, Service registers its services in the middleware. Here service is requestor because it initializes the interaction with middleware. Ts1", S2", Tms1", Tms2", M2", and M3" are new transitions and places added in the middleware a service interface for composition.

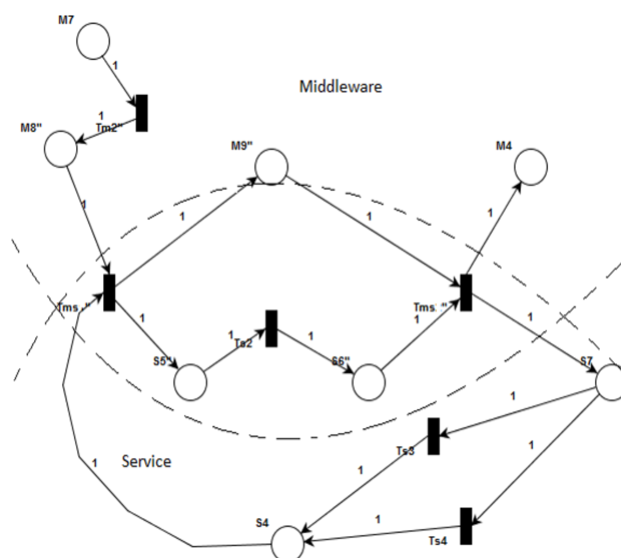

Figure 32. Middleware-Service Composition-I

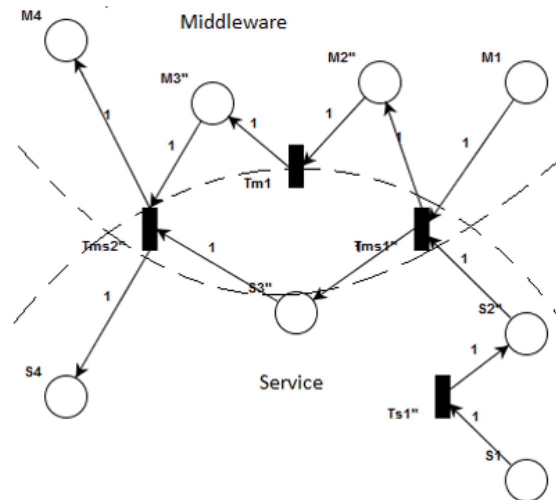

Figure 33. Middleware-Service Composition-II

5.5.4. Robot-Middleware-Service-Player Server Composition

An overview of the net composition is outlined in Figure 34. After analyzing this Petri Net model, we got 44 minimal Siphons [20]. To detect the deadlock we need to minimize the no. of tokens in minimal siphons by using linear programming [21] with the help of constraints.

These constraints correspond to places of the net and specify that the number of tokens in each place cannot be negative. After analyzing the model we found deadlock. The deadlock was arising because of stop transitions of these 4 interfaces were not synchronized. This caused irregular stopping and starting of the interfaces which resulted in a circular wait. When the token arises at P4 after player server providing the proxies to the robot, it has a choice to select between two paths that lead to the player server stop transaction and setting values in the proxies by the robot. All the transaction have the same priority, the player server can choose any path. Instead of waiting for the robot to get ready for interaction between them for setting the values in the proxies, player server triggers the stop transaction. This causes deadlock at the synchronized transactions Trp1" and Trp3". Here circular wait problem arises. At Trp1", the player server is waiting for the robot to accept the proxies and at Trp3" robot is waiting for the player server. Same kind of thing happens with middleware and service composition. This situation comes when Tms2" fired after the service registers its service in middleware. All the transactions have the same priority, service and middleware can take any path at this point. If service select the stop path and middleware select the Tpm1" path then there will be a deadlock at Tms1" and Tms3" in future. A circular wait problem arises here. Middleware is waiting for services to accept the sensors data and service are waiting for middleware to register the services. This caused deadlock in the system. This is also verified by using State Space analysis module of PIPE(Platform Independent Petri net Editor) tool shown in the Figure-35.

Changing the priority of the transaction, stop transaction with lower priorities than the normal transaction so that stop transaction will fire at last, does not help. It removes the deadlock but the robot never stops. The stop transactions have the least priority, they never fired and the process of sending sensor data to the services never end. To overcome this problem, we synchronized the stop transitions of other interfaces with the robot stop transition. When the robot wants to stop, it fires its stop transition (Tr6) and token reaches at R10 place, see in Figure 36, 37.

The next transition is synchronized stop of other three interfaces. When the token reaches R10, now it becomes necessary for other interfaces to stop because there no other path to select. After firing the synchronized stop, the robot model comes to its initial state or starting state. Synchronize stop remove the circular wait and irregular stopping and starting of the interfaces.

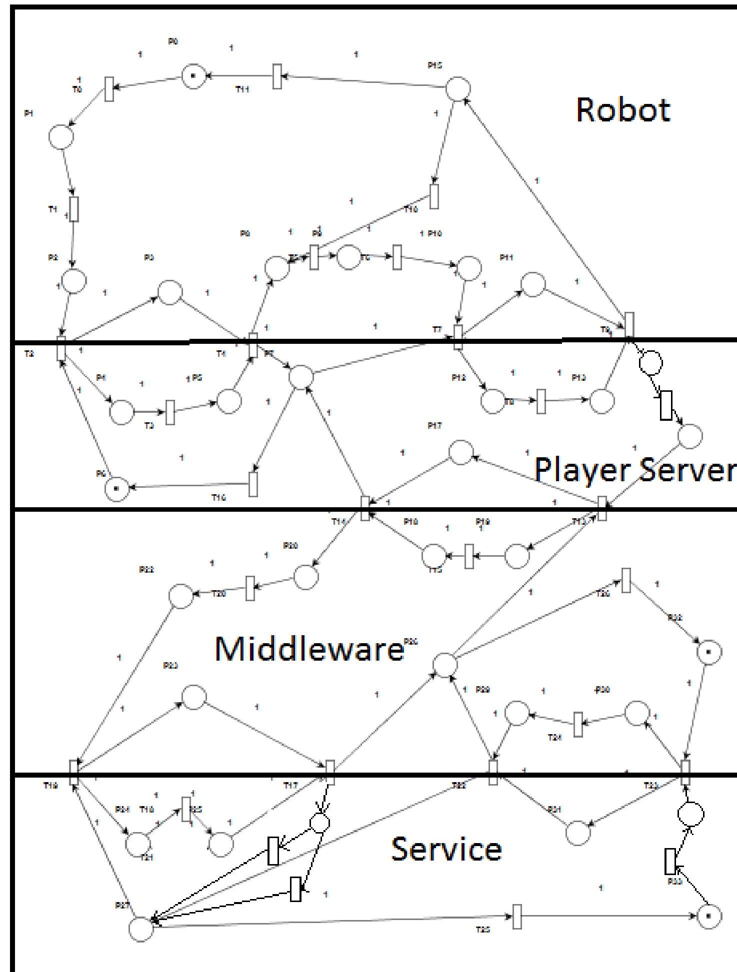

Figure 34. Robot-Middleware-Service-Player Server Composition

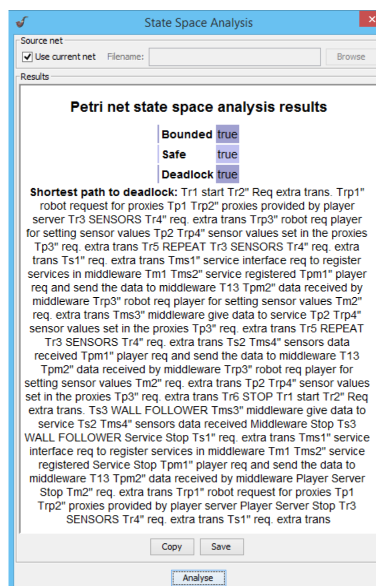

Figure 35. State Space Analysis

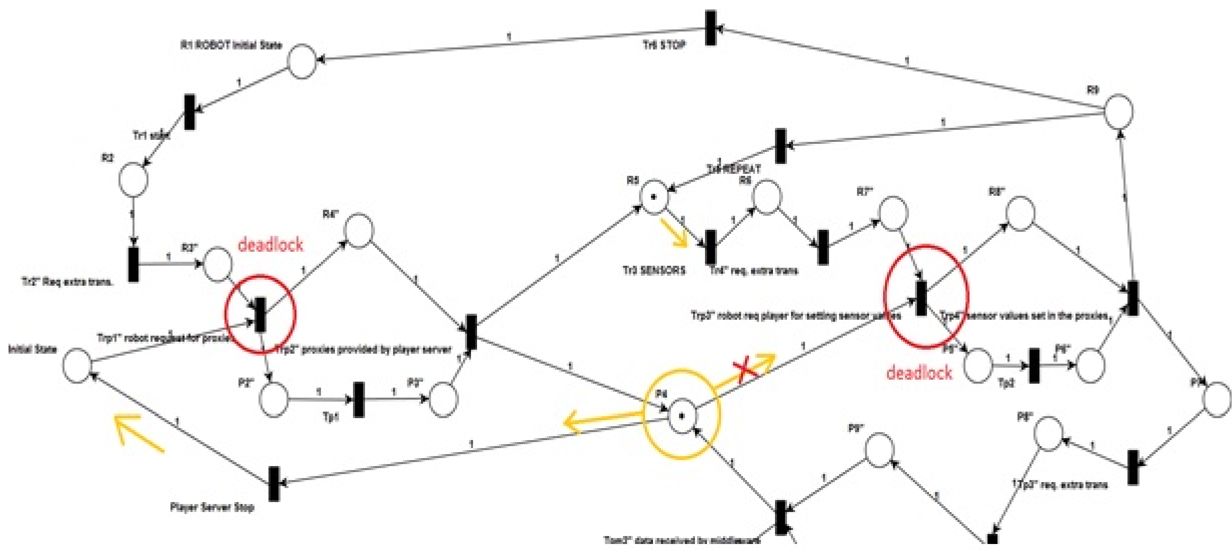

Figure 36. Robot-Player Composition part of Figure 38

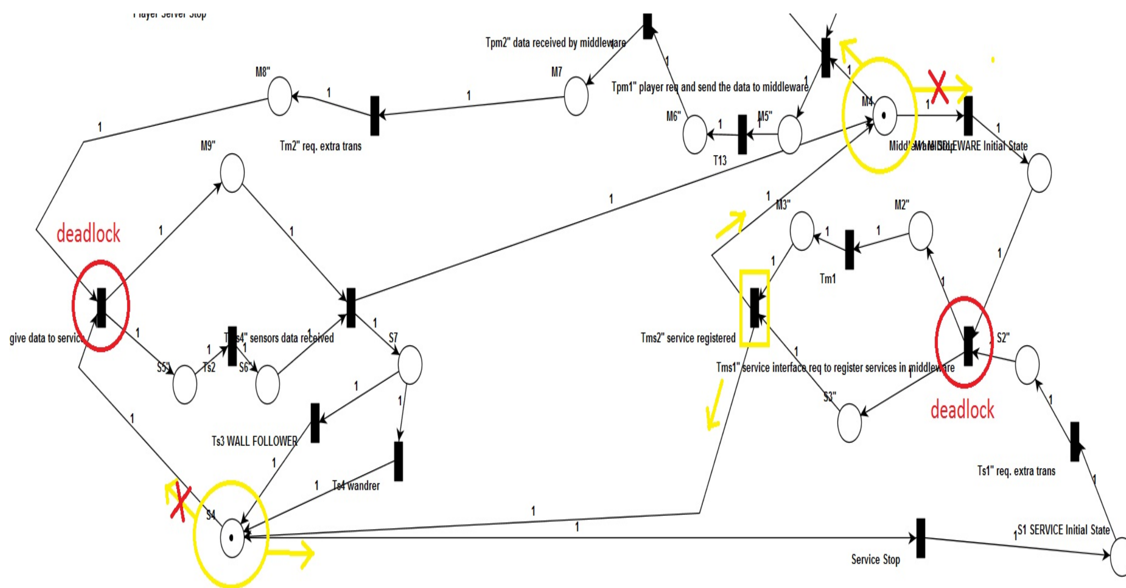

Figure 37. Middleware-Service Composition part of Figure 38

Now the player server provides proxies to robot and service register its services in middleware one time only when the model runs from starting. This removes the unnecessary repetition of transactions which was happening in the previous model. Now the player server, middleware, and services will stop only when the robot wants to stop. The new Petri net model is shown in the Figure 38. Where yellow path shows the continuous working of the robot, red path shows Stop and blue path shows back to starting position.

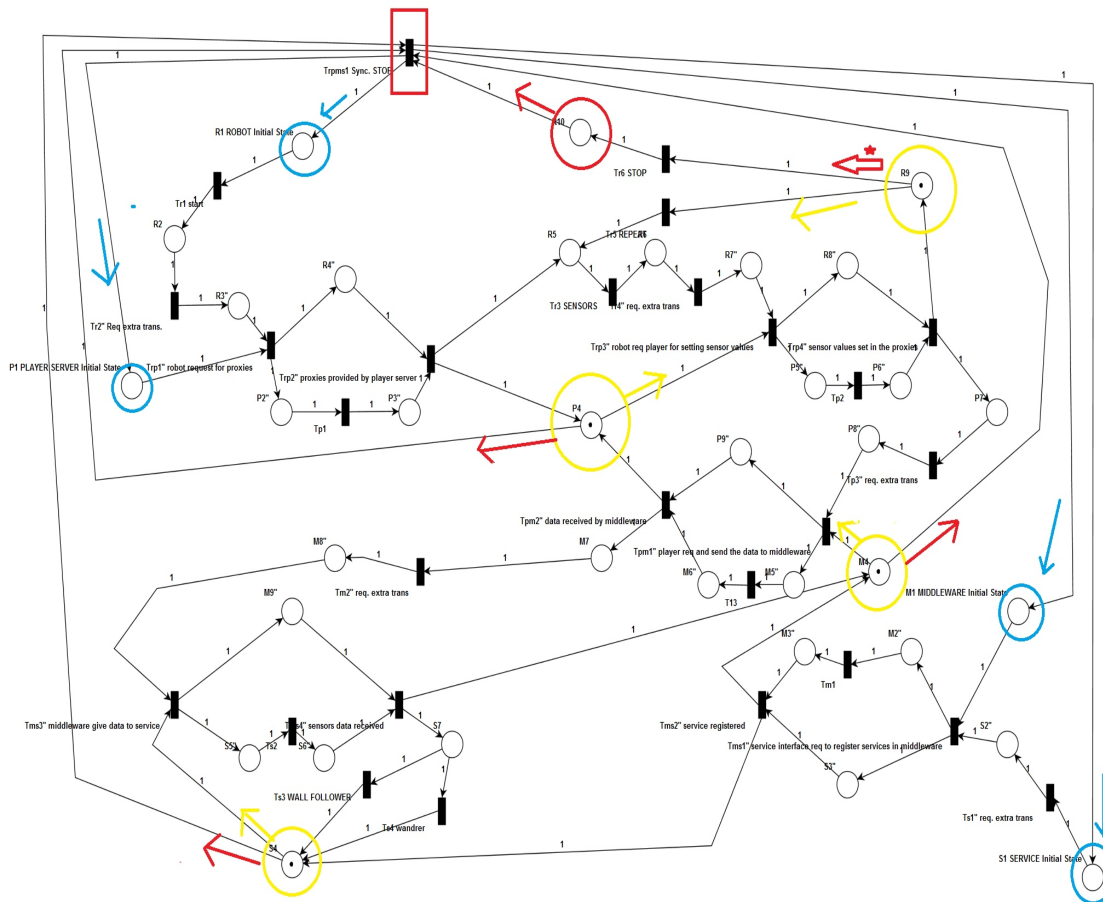

Figure 38. Corrected Robot-Middleware-Service-Player Server Composition

In the new net, we tried to minimize the tokens in siphons with the help of constraints. At last, we didn't find any siphon with 0 tokens. We also did the reachability analysis on the net where we did not find any dead place. So, we can say that the model is deadlock free. This is also verified by using State Space analysis module of PIPE(Platform Independent Petri net Editor) tool shown in the Figure 39.

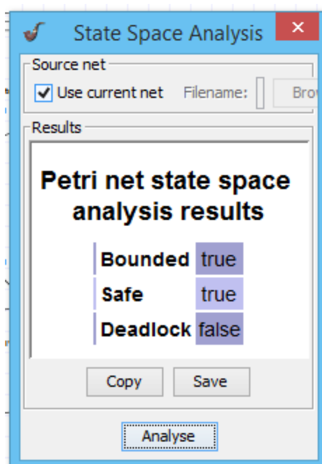

Figure 39. State Space Analysis of new net

6. CONCLUSION

We conclude that in synchronous and asynchronous communication promise are better than using a callback. We have compared the time taken in completion of services and found that services running using promise take less time as compared to callback and synchronous. Petri nets used for verification of concurrent robotic system integration helps us to find out the issues of deadlock using siphons prior to building the system for integration. Systems composed using Petri nets adds extra transition and places to compose common transition between requestor and provider. Our promise based framework provides the facility of using already available robotic components as an actor which can communicate with another the actor asynchronously using a single threaded model as supported by Node.js [22]. In future, we aim to verify the architecture of our integration system by checking the coordination control constraints in different Petri net composition model as proposed in paper [23].

REFERENCES

[1] Carle Cote, Yannick Brosseau, Dominic Letourneau, Clement Raievsky and Francois Michaud, "Robotic Software Integration Using MARIE", *International Journal of Advanced Robotic Systems, ISSN: 1729-8806, INTECH, 2008.*

[2] S. Cousins, "Is ROS Good for Robotics? [ROS Topics]", *in IEEE Robotics & Automation Magazine, vol. 19, no. 2, pp. 13-14, June 2012. doi: 10.1109/MRA.2012.2193935.*

[3] Xin-qing Yan, Wen-feng Li and Ding-fang Chen, "A New Mechanism for Robots Control Based on Player/Stage", *Robotics and Biomimetics, 2006. ROBIO '06. IEEE International Conference on, vol., no., pp.750-754, 17-20 Dec. 2006.*

[4] M. Montemerlo, N. Roy and S. Thrun, "Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit", *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003). Las Vegas, October, 2003. Vol. 3, pp 2436-2441.*

[5] Eyal Amir and Pedrito Maynard-Reid, "Logic-based subsumption architecture", *In Proceedings of the 16th international joint conference on Artifical intelligence - Volume 1 (IJCAI'99), Vol. 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 147-152, 1999.*

[6] Nobakht, Behrooz, Frank S. and de Boer., "Programming with actors in Java 8", *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications. Springer Berlin Heidelberg, 2014. 37-53.*

[7] Haller, Philipp and Martin Odersky, "Actors that unify threads and events", *Coordination Models and Languages. Springer Berlin Heidelberg, 2007.*

[8] Soumen Chatterjee, Cap Gemini Ernst and Young, "Messaging Pattern in Service oriented Architecture", *http://msdn.microsoft.com/en-us/library/aa480027.aspx, retrieved September 2010.*

[9] H. Tran and U. Zdun, "Event Actors Based Approach for Supporting Analysis and Verification of Event-Driven Architectures", *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International, Vancouver, BC, 2013, pp. 217-226. doi: 10.1109/EDOC.2013.32.*

[10] Brodu, Etienne, Stéphane Frénot and Frédéric Oblé, "Toward automatic update from callbacks to Promises", *AWeS. 2015.*

[11] Kai Qian, Jigang Liu and LiXin Tao, "Asynchronous Callback in Web Services", *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06), Las Vegas, NV, 2006, pp. 375-380. doi: 10.1109/SNPD-SAWN.2006.23.*

[12] Barkaoui K. and Lemaire, B., "An effective characterization of minimal deadlocks and trapsin Petri nets based on graph theory", *In Proc. 10th Int. Conf. on Applications and Theory of Petri Nets, pp.121., 1989.*

[13] Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam and Vivek Sarkar, "A Distributed Selectors Runtime System for Java Applications", *In Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16). ACM, New York, NY, USA, Article 3 , 11 pages. DOI: http://dx.doi.org/10.1145/2972206.2972215.*

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design patterns :Elements of reusable object-oriented software", *Addison-Wesley, 1994.*

[15] Margaret M., Burnett and Brad A. Myers, "Future of end-user software engineering: beyond the silos", *In Proceedings of the on Future of Software Engineering (FOSE 2014). ACM, New York, NY, USA, 201-211. DOI: http://dx.doi.org/10.1145/2593882.2593896, 2014.*

[16] Diomidis Spinellis, "Another level of indirection. In Andy Oram and Greg Wilson, editors, Beautiful Code: Leading Programmers Explain How They Think", *chapter 17, pages 279291. O. Reilly and Associates, Sebastopol, CA, 2007.*

[17] "Software as a Service: Strategic Backgrounder", *Software and Information Industry Association, Feb 2001, http://www.siia.net/estore/pubs/SSB-01.pdf, retrieved September 2010.*

[18] Diomidis Spinellis, "Messaging Pattern in Service oriented Architecture", *http://msdn.microsoft.com/en-us/library/aa480027.aspx, retrieved September 2010.*

[19] A. Basu et al., "Rigorous Component-Based System Design Using the BIP Framework", *in IEEE Software, vol. 28, no. 3, pp. 41-48, May-June 2011. doi: 10.1109/MS.2011.27.*

[20] Abdallah, I. B., ElMaraghy, and H.A.ElMekkawy, "A logic programming approach for nding minimal siphons in S3PR nets applied to manufacturing systems", *In Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics, pp.17101715, 1997.*

[21] Barkaoui and K. Lemaire, "An effective characterization of minimal deadlocks and trapsin Petri nets based on graph theory", *Proc. 10th Int. Conf. on Applications and Theory of Petri Nets, pp.121., 1989.*

[22] McCune and Robert Ryan, "Node. js paradigms and benchmarks", *STRIEGEL, GRAD OS F 11 (2011).*

[23] Attie, Paul, Baranov, Eduard, Bliudze, Simon, Jaber, Mohamad, Sifakis and Joseph, "A general framework for architecture composability", *Formal Aspects of Computing,vol.28,no. 2,pp. 207-231, 2016.*

BIOGRAPHY OF AUTHORS

Ratnesh Srivastava is currently a research scholar at Indian Institute of Information Technology, Allahabad and full-time Assistant Professor in the department of Information Technology, College of Technology, GBPUAT, Pantnagar since 2011. Prior his current job, he was working with software industry for 08 years. He has experience of using Java based technologies on various domains including health and banking.

Gora Chand Nandi received the degree from the Indian Institute of Engineering, Science and Technology, in 1984, the master's degree from Jadavpur University, Calcutta, in 1986, and the Ph.D. degree from the Russian Academy of Sciences, Moscow, in 1992. He received the National Scholarship by Ministry of Human Resource Development (MHRD), India, in 1977, and a Doctoral Fellowship by the External Scholarship Division, MHRD, India, in 1988. In 1997, he was a Visiting Research Scientist with the Chinese University of Hong Kong, and he was visiting Faculty at the Institute for Software Research, School of Computer Sciences, Carnegie Mellon University, USA, (2010–2011). He served as Senior Professor and Dean of Academic Affairs with the Indian Institute of Information Technology, Allahabad. Currently and in 2014, he served as the Director-in-Charge of the Indian Institute of Information Technology, Allahabad. He has authored over 100 papers in the various refereed journals and international conferences. His research interest includes robotics specially biped locomotion control and humanoid push recovery, artificial intelligence, soft computing, and computer controlled systems. He is a Senior Member of ACM, Chairman, ACM-IIIT-Allahabad Professional Chapter, (2009–2010), Chartered Member of the Institute of Engineers (India), member of the Department of Science and Technology, India, Program Advisory Committee Member of Robotics, and Mechanical and Manufacturing Engineering.

Rohit Shukla is an undergraduate student pursuing his Bachelor Degree in Information Technology from College of Technology which is under Govind Ballabh Pant University, Pantnagar(India).

Harsh Verma is an undergraduate student pursuing his Bachelor Degree in Information Technology from College of Technology which is under Govind Ballabh Pant University, Pantnagar(India).