Publications

Available Online

J. Sci. Res. **15** (1), 11-29 (2023)

JOURNAL OF SCIENTIFIC RESEARCH

www.banglajol.info/index.php/JSR

A SOA and DBUS-based Robotic Component Interaction

R. P. Srivastava[1], L. S. Umrao[2*], R. S. Yadav[3]

[1]Department of CSIT, Guru Ghasidas Vishwavidyala, Bilaspur (C.G.), India

[2]Department of Computer Science & Engineering, Faculty of Engineering and Technology, Dr. Rammanohar Lohia Avadh University, Ayodhya (U.P.), India

[3]Department of Business Management & Entrepreneurship, Dr. Rammanohar Lohia Avadh University, Ayodhya, UP, India

Received 13 January 2022, accepted in final revised form 8 September 2022

Abstract

Rewriting and executing codes violate the SOLID principle of object-oriented programming. Robotics being an emerging platform, it becomes hard to write the code from scratch. We focus on using what is already built in as a Robotics Component instead of writing something from scratch. The intention is to minimize development effort and produce the desired result on time without rewriting the software components. By allowing functionalities of different components to be brought into a single component, we are saving the time required for code conversion or replication of functionality written in one language to another. So an architecture with these properties seems ideal for integrating different components. Integration is one of the most fundamental problems in designing autonomous mobile robots, especially those interacting with people in real-life settings. This paper presents an approach to building complex systems from different robotics packages available as open source components (PLAYER, STAGE, GAZEBO, CARMEN). DBUS is a message-oriented protocol for communicating among processes running on the same desktop. SOA provides access to the services over the network and adds novelty to our application.

Keywords: CBARPI; MARIE; ROBOTICS; PLAYER; CARMEN; GAZEBO; Service-oriented architecture; CCU; DCU; BCU; Interface.

doi: http://dx.doi.org/10.3329/jsr.v15i1.57545 J. Sci. Res. **15** (1), 11-29 (2023)

1. Introduction

Robotics [1] is still a new field, and till now, there is no standard proposed. Lacking a defined standard has created a scenario where different communities use different robot programming software. This has restricted knowledge propagation. Either you need to know their standard, or you have to convert the same into yours. Moreover, this has also created a limited functionality situation, i.e., we are limited to the functionality provided by the component that a particular community uses. The main idea of this project is to

[*] *Corresponding author*: lokendra.manit@gmail.com

develop an architecture that provides different functionalities provided by different components [2].

By presenting such architecture to the robotics community, we can equip developers with functionalities of different components. Robotics which is still a new field, lacks standards. Thus no standard component is available for robot programming. Since Robotics is in the early exploration phase, identifying common requirements is still challenging. So, many communities worldwide are developing different programming environments for controlling and programming robots. Most of these components are incompatible, so the developer has to choose between available components. This incompatibility may be because of different communication protocols, robotics platforms, architectural design, programming language, proprietary source code, etc. But such non-standard developments have made developers restricted to one component. This makes them code functionalities already present in other components. This has restricted knowledge propagation and consumes more time in developing controlling code for robots. Moreover, to use the function of another component first, the developer needs to understand and study that component and then deploy the already programmed function in that component to the component which the developer is using for the programming of the robot. Table 1 summarizes the issues addressed to find the research gaps, which are then solved using DBUS and SOA.

Table 1. Summary shows need of DBUS & SOA.

Author	Approach and Process	Existing Problem
Zwicky [3]	Focuses on designing Java based actor framework	These framework designated on Java do not support promise based construction and also do not provide separation of concerns for coordination. These frameworks were designed before Java 8 was released, hence they do not support functional programming construct necessary for immutability. Hence these frameworks are error prone to race conditions.
Karmani *et al*. [4]	In order to deal with the concurrency issue, emphasis has been given to design isolated computations.	All the sequential programming languages (Java, C++ etc.) are not considered fit to implement Shared Memory concurrent communications, because they share values which results in race condition.
Karmani *et al*. [4]	Actor based functional languages (Scala, Salsa etc.) addresses the concurrency constructs such as immutability and pure functions.	These actors based functional frameworks use continuation mechanism to implement synchronization which suffers with inversion of control issues.
Elkady *et al.* [5]	Represents the survey of different existing components models such as MARIE, OROCOS, ROS etc.	They have been designed using high level language and they do not support concurrency by default.

Thus by combining different components we can equip developers with different components functionalities. Such environment helps quick deployment of robot controlling code. The thread based integration approach is also tested in [6] which produce certain bottlenecks due to serialization. The integration approach of various components is also tested for deadlock analysis using [7] which is caused due to multi thread based implementation. Therefore it has been decided to use DBUS as suggested in and SOA which is widely accepted framework of communication and service exchange [8,9]. The robotic component models ROS, CORBA, Player/Stage, MARIE, OpenCom, and OROCOS are domain-specific reusable collections and must execute on the underlying ROS platform. The use of ROS discussed in [21,22] provides an implementation of platform-specific implementation and does not provide services on demand. CORBA component model uses its built-in CORBA processes for component inter-process communication. OpenCom component model is designed for embedded system environments that use resources through a kernel API, and the component model mediates a call to this API. MARIE uses the ACE framework for component communication and Player/Stage as a network server to access sensor data. MARIE takes advantage of distributed support of execution by accepting and offering sensor data from Player/Stage.

This paper is organized as follows: section 2 introduces the details of CBARPI [Component Based Architecture for Robotics package integration], section 3 introduces the detailed study of MARIE, section 4 introduces the proposed architecture for PLAYER and CARMEN interaction, and section 5 discusses the implementation methodologies section 6 introduces implementation details, and section VII introduces the developed model of robots interaction and finally conclusion and future work.

2. CBARPI

CBARPI [Component Based Architecture for Robotics package interaction] is an approach toward developing such an architecture that can use for integrating different robotics components together. This approach demands architecture to be flexible enough so that it can handle:

Rapid interaction of Robotics Components: Components can easily be connected to each other through architecture. If connecting components requires too much expertise, that would be useless again as it would make the architecture obsolete after some time as it would not be able to integrate future components.

Support Different Communication Protocols: This allows architecture to connect components from different domains. Components using different communication protocols can be integrated using architecture.

Support Different Programming languages: Components written in different programming languages can be connected, so the language barrier is removed. As we know, every programming language has its advantages and disadvantages. Thus such architecture enables developers to exploit good aspects of different programming languages.

Simple and well-designed: Simple and well-defined architecture helps in a better understanding of the architecture. Such architecture can be easily modified and worked on by different developers. To view more conveniently, architecture can be broken into different layers. Well-defined layers help developers to contribute to a particular layer without caring much about other layers. So, architecture should be such that there is minimum coupling between layers defined in architecture. Thus, understanding architecture is limited to layers, and moreover, developers can contribute to a particular layer without caring or understanding the whole architecture. Such an approach develops an expertise approach by equipping developers to contribute to their well-defined area. So, one of the most important steps toward building the architecture is identifying different functional layers that allow a high-level view of the architecture.

CBARPI utilizes all the features mentioned above by presenting simple and well-defined architecture. These features make architecture more usable by presenting it in an easily modifiable way. CBARPI is motivated by MARIE [5].

3. MARIE

MARIE [Mobile and autonomous robotics integration environment] is an architecture already available for integrating different robotics packages but has become obsolete now. But the study of MARIE architecture helps a lot in finding out various aspects of CBARPI's architecture. It has a mediation layer architecture in which a central manager manages all integration and communication work. This centralized control unit also termed a mediator, interacts with each class independently. Further helps coordinate and synchronize global interaction between classes to realize the desired system. BY allowing such architecture, integration of components having different communication protocols and mechanisms are possible as long as the mediator supports it.

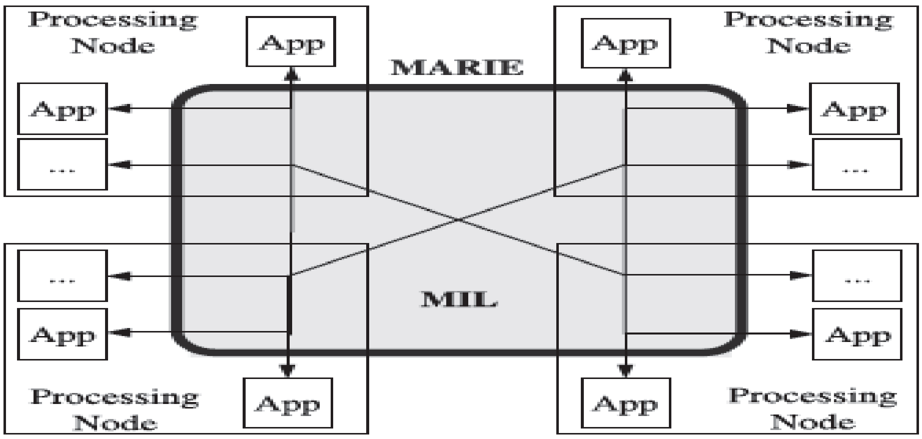


Fig. 1. Mediation pattern MARIE.

Four types of components are used in the MIL they are Application Adapter, Communication Adapter, Application Manager, and Communication Manager:

Application Adapter (AA): component interfacing useful applications within the MIL and enabling them to interact with each other through their standardized interface (i.e., Ports) and using MARIE's shared data types. Interconnections using Port communication abstraction are illustrated in Fig. 3 with a small dot between communication links represented by arrows.

Communication Adapter (CA): a component that ensures communication protocols or by implementing traditional routing communication functions. Available Communication Adapters in MARIE are Splitters, and Switches, between other components by adapting incompatible communication mechanisms and Mailboxes and Shared Maps. A Splitter sends data from one source to multiple destinations without the sender being aware of the receivers. A Switch acts like a multiplexer sending data to the selected output. A Mailbox creates a buffering interface between asynchronous components. A Shared Map is used to share data between multiple components in the key-value form.

Application Manager (AM): System-level component managing on local or remote processing nodes, Application Adapters, and Communication Adapters. Application and Communication Adapters initialization, configuration, start, stop, suspend and resume are handled by the Application Manager. When starting the system, the Application Manager initializes the components following the adequate sequence.

Communication Manager (CM): System-level component dynamically managing, on local or remote processing nodes, the communications mechanisms (socket, port, shared memory, etc.).

3. Communication Protocol Abstraction

The development of robotic applications using MARIE is based on reusable software blocks, referred to as components, which implement functionalities by encapsulating existing applications, programming environments, or dedicated algorithms. Components are configured and interconnected to implement the desired system, using the software applications and tools available through MARIE.

But this architecture has become obsolete now. It supports only old versions of packages and is not compatible with today's versions of compilers and hardware. Some of the drawbacks of MARIE Architecture:

- Doesn't compile with today's version of compilers.
- Marie supports an old version of Robotics packages
- It is a must to have a closer understanding of Marie's architecture for designing adapters.

4. Proposed Architecture

Contexts: While designing new software components, reusability, interoperability, extensibility, adaptability & modularity are important software attributes that should be considered as a requirement. To produce useful components in Robotics & to progress in the field of Robotics activities, It is a must to take benefit of the components developed earlier by different organizations for different purposes.

Problem and contribution: Such attributes can be obtained while designing components by using the standard framework for data representation & interfaces. Since the Robotics activities and standard is developed and adopted by different communities, they provide a different context, i.e., using different operating systems, programming languages, inter-process communication, data representation, and communication protocols.

Therefore integrating such heterogeneous components is often a difficult job that needs to be simplified.

Contributions:

- Exploiting already available programming environments and libraries
- Accelerating developments by using previous development
- Increasing quality attributes of heterogeneous components, e.g., modularity, adaptability, reusability, extensibility, and interoperability
- Minimize or overcome in lack of a standard for developing heterogeneous components based on application
- Exploit different communication protocol and their advantages and disadvantages.
- Avoid reimplementation from scratch.
- Development and Implementation of DBUS and SOA based Architecture for addressing heterogeneity among robotic component.
- Performance Evaluation based on response time and concurrent service running time.

Authors though pointed out [14] Promise and callback based mechanism for asynchronous implementation of thread based services but if suffers with deadlock due to their concurrent services. Proposed Architecture is implemented using principles of Service Based Architecture (SOA) [15,16]. Here robotics components are treated as services in SOA. Here robotics components are treated as services in SOA. SOA-based architecture is chosen because of the following features:

Reusability: 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 on code replications [6]. As components are built independently following their 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 re-used.

Easy integration: Integrating services should be cost-efficient and easy. Moreover, architecture should be flexible enough to integrate modified and new services without obsoleting or rebuilding the architecture [7]. Such integration should not be too complex. Integrating new services into architecture should not involve rebuilding the whole architecture again. Thus, service can be connected on the fly to the architecture.

Support for different communication protocols and robotics standards: Since there is no defined standard yet established in robotics, various robotics communities use different robotics standards and different communications protocols. Support of various communication protocols and robotics standards allows components from different domains to be integrated.

SOA seems to provide all the above-desired features. SOA allows using a well-defined set of services according to the user requirement. Users can visualize a pool of services available. They select the services they require for the building of the application. All these services are loosely coupled and work independently of each other. Synchronization and consistency are being maintained and monitored so that all the services can work concurrently. Some services may not be compatible with each other supporting different incompatible communication protocols or robotic standards. In that case, it's one of the duties of the architecture to make services talk to each other without being worried about the protocols and other standards requirements. Such a feature requires protocol conversion to make components compatible with each other. Moreover, some of the services may not be on the host computer but rather on the network. A record of all these services needs to be kept such an efficient service discovery can be made. The record is checked for the service being called by the user. If service is on the host computer, it is invoked from the same host; if not, service is requested from the host having that service. If a service is not present, the system should terminate gracefully with proper error messages so that the user can diagnose the problem with the application created.

The architecture consists of three layers based on the developer's viewpoints. Viewpoints categorize the developer's expertise. In a team of developers, few may be good in knowledge of the Robotics domain; few may be good in data handling, communication and serialization, and de-serialization. So based on the viewpoints, the three layers are:

- Builder Abstraction Layer
- Core Abstraction Layer
- Components Management and Creation Layer

Fig. 2 depicts how the two components interact with each other with various subcomponents.

The application builder layer of abstraction consists of tools to develop an application using a set of available components. In this layer, the developer does not need to know the inner details of components, e.g., programming, business domains, etc. The core layer comprises tools for communication, data handling, and low-level issues such as serialization and de-serializations. The component management and creation layer supports domain-specific behavior by adding new components which specify and implement a particular framework. Each component in the framework has its lifecycle, which the Application Manager initializes. The component Initialization life cycle has four software modules that are dedicated to communicating with the Application Manager. The modules that included managing component life cycles are Director, Spawner, Configurator, and Component Initializer. Director is a port listener application that accepts requests sent by the Application Manager and passes the request to Spawner, which checks whether the component is already running and passes the parameters required to Configurator to configure the final component is initialized, and the status is updated to the Application Manager. Initially, CBARPI's architecture can be seen as different components working together to get the desired work done. This architecture is still in its initial stage. It needs a more detailed explanation of its components. Currently, we have divided it into three components which have sub-components.

- Robot-Interface
- Buffer

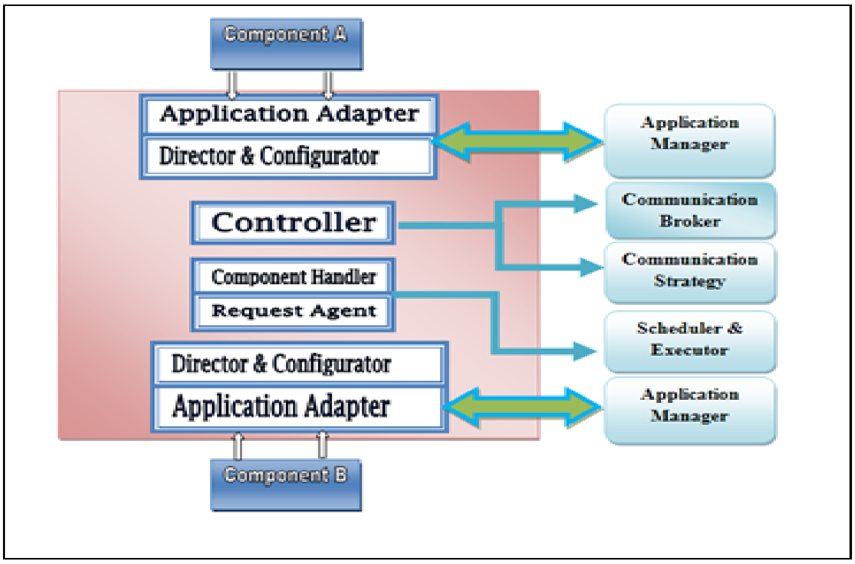


Fig. 2. Component integration middleware design.

Robot-Interface: This is the interface between the robot and another robotics package. All the commands that need to be executed are sent to the robot interface that sends the same to the robot. All data that needs to be sent to packages like laser data, sonar data, and other data is collected by the interface from the robot. This interface can be assumed as just the data communication factory through which data communication is taking place. For every application, we have an interface. Through the interface, the application can read data from the buffer and write data to a buffer. Fig. 4, shows the interface, which also acts as a data conversion factory that converts data in global format when writing to a buffer and converts data to a format understandable by the application. For global data format, we can use XML format. By XML global data format, we can define our own tags, and using DTD and schemes; we can also restrict the use of invalid tags. Just by creating a simple parser, we can extract data from the xmL file and convert same to the format recognized by the application.

ADAPTERS: Adapters for data conversion are shown in Fig. 3. This component of the architecture interfaces with the package that is required to be connected. It is the adapter that exposes the functionality of that package to other robotic packages. So a package that needs to be connected must first define its adapter. Just by creating an adapter, we can add the functionality of that package to the robot. Building an adapter may require in-depth knowledge of the package, but it can be easily deployed to the architecture once it's done. This adapter is also responsible for data conversion. It converts data from the package to global data and global data to package-specific data. Global Format is already defined in the architecture so that it is understandable by another component in case they also require that data. This global data format is XML because it's an easy way to communicate data between components. Such a format can be understood easily just by creating parsers. Now data that is required by the package should be in the format that is understood by the package. So we need global data conversion to standard package type. This work is also done by the adapters. So building an adapter requires a deep understanding of the package. Peeking inside proprietary packages is difficult, so creating adapters for such packages may be difficult. But giving adapters a well-defined format, we can get the adapter from the developers of the package only. Open-source packages can be integrated easily as the source is visible.

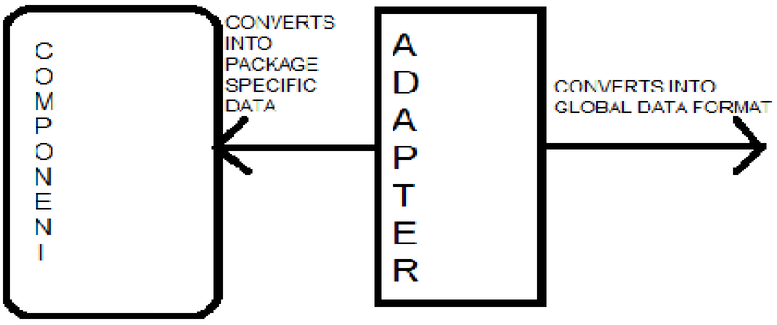

COMPONENT
CONVERTS INTO PACKAGE SPECIFIC DATA
ADAPTER
CONVERTS INTO GLOBAL DATA FORMAT

Fig. 3. Data conversion from adapters.

BUFFER: Fig. 4 shows the working of Buffers. This is the temporary storage where data is written and read from. Robot data obtained from the robot interface is written to this storage in global data format through the robot interface. Now any application requiring any robot data can read the data from this interface through their adapters. Now, this is the duty of the adapter to convert the data required by the application from buffer space into the format in which the application requires the data. So buffer can be visualized as the data storage factory where just data is stored, which different applications can read through their adapters. Since it's a data storage factory, we need a buffer manager which manages this data. The work of the buffer manager includes cleaning obsolete and old data that is not required. Change the buffer size according to the requirement.

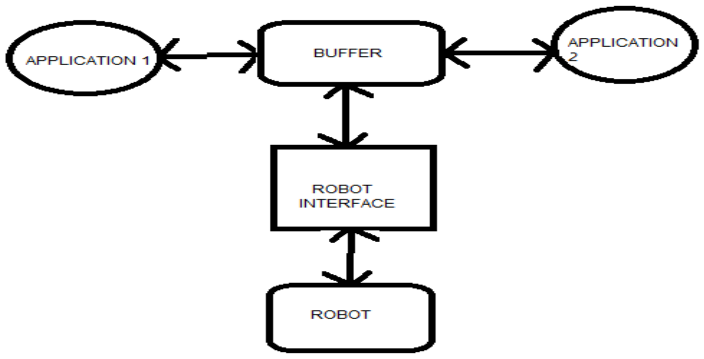


Fig. 4. Interaction between different components of the architecture.

Right now, we are focusing on 2-3 components to connect. We are taking player/stage [3] and Carmen [6] to integrate. In Carmen [6], we take navigation functionality, and the player wanders functionality. Navigation functionality is used to make the robot navigate to the desired set goal in the least distance path, and it also has avoided algorithm, which is used to avoid collision between moving objects and robots during its movement to the desired goal. You can also drop this functionality to make a robot push things that come on the way to the robot's goal position. Using player wander functionality, you can make the robot wander on a map avoiding collisions with walls and other things.

These components provide translational and rotational velocity to make the robot move according to its algorithm. So just by getting the rotational and translational velocity, you can make the robot wander. We have developed an interface that, according to a previously set position of the robot and map in which the robot has to wander, generates robot position data, velocity data, and laser data. An application that needs to use that functionality just has to read the file through its interface and can make the robot move according to another application. The below diagram in Fig. 5 represents data exchange between CARMEN and PLAYER through BUFFER SPACE (Wander Algorithm) Robot interacts with BUFFER SPACE, which stores the Wander algorithm of Player and Navigation algorithm of CARMENT through ROBOT INTERFACE.

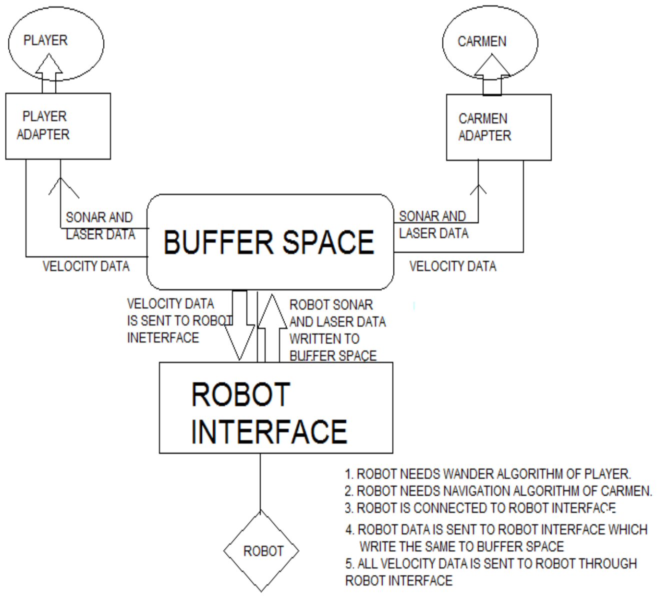

PLAYER
PLAYER ADAPTER
CARMEN
CARMEN ADAPTER
SONAR AND LASER DATA
VELOCITY DATA
BUFFER SPACE
SONAR AND LASER DATA
VELOCITY DATA
VELOCITY DATA IS SENT TO ROBOT INETERFACE
ROBOT SONAR AND LASER DATA WRITTEN TO BUFFER SPACE
ROBOT INTERFACE
ROBOT
1. ROBOT NEEDS WANDER ALGORITHM OF PLAYER.
2. ROBOT NEEDS NAVIGATION ALGORITHM OF CARMEN.
3. ROBOT IS CONNECTED TO ROBOT INTERFACE
4. ROBOT DATA IS SENT TO ROBOT INTERFACE WHICH WRITE THE SAME TO BUFFER SPACE
5. ALL VELOCITY DATA IS SENT TO ROBOT THROUGH ROBOT INTERFACE

Fig. 5. Architecture implementing Wander Algorithm.

We can interact components using the wander algorithm of PLAYER, CARMEN using Buffer space. The middleware complexity of asynchronous communication, Inter robots communication, and data broadcasting to different robots in a network has been identified by designing a middleware whose architecture is shown in Fig. 6.

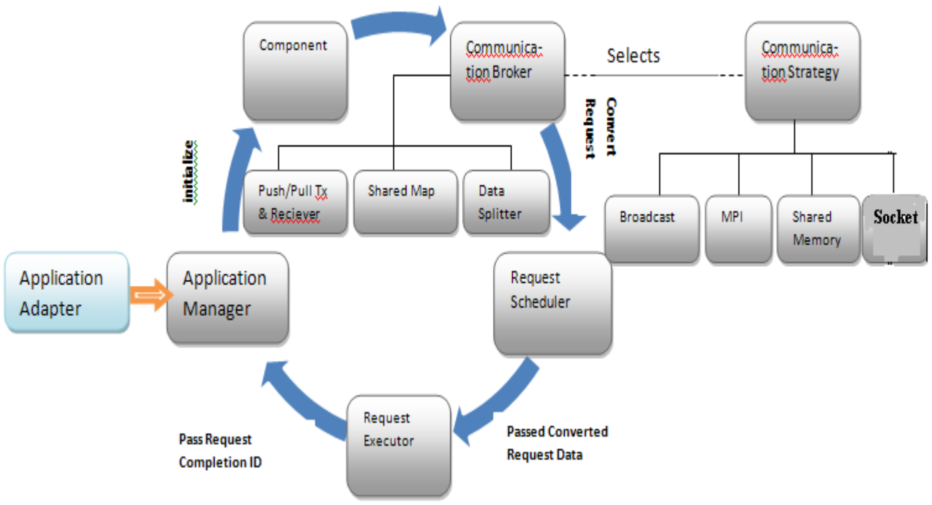


Fig. 6. Detailed architecture for component interaction (low-level diagram).

5. Implementation Methodology

Our Architecture integrates loosely coupled components based on SOA. An application needs to be developed using the functionalities provided by different components. A single component may provide more than one function that may or may not be interrelated, such as audio and video processing, collision avoidance, and path planning. Further, it's possible that these packages are not present on the same machine, creating a distributed environment. At the highest level of abstraction, the proposed architecture will look as illustrated in Fig. 7.

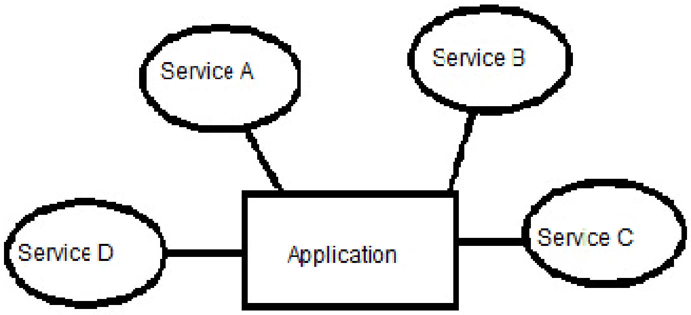


Fig. 7. High-level abstraction of application.

The proposed architecture can be understood from the following two perspectives:
5.1 Abstraction layers
5.2 Component framework

5.1. *Abstraction layers*

To provide multiple levels of abstraction, three layers are defined in the proposed architecture [11]. Each layer provides a certain set of functions to the layer above it. The three layers are illustrated in Fig. 8

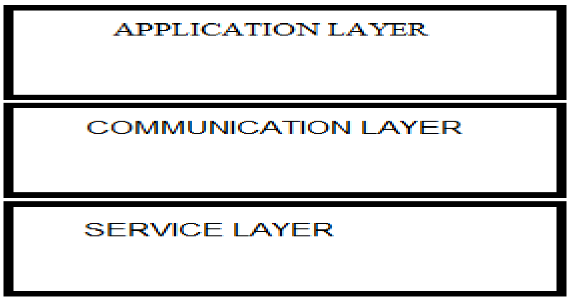


Fig. 8. Different layers of architecture.

5.1.1. *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 that needs to be developed. Each service is nothing but the functionality that a component provides [12]. All the components integrated are an integral part of the Service Layer. These components may provide much functionality. These packages may be heterogeneous, have built-in different programming languages, use different communication protocols, and may present on different machines. Hence in SOA's terminology Service layer is a collection of Service Providers.

5.1.2. *Communication layer*

This layer is present above the 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 published/subscribe and request/reply [13].

5.1.3. *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.

5.2. *Component framework*

The proposed architecture is composed of different building blocks. These blocks are services, interfaces, bus, broker, and application. These building blocks are defined below:

5.2.1. *Service*

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

5.2.2. *Interfaces*

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

5.2.2.1. *Component communication unit (CCU)*

This unit communicates with the component, more precisely with the service in which an application is interested. CMU supports both the 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.

5.2.2.2. *Data conversion unit (DCU)*

Robotics packages are data-specific. They require the data in a 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 to provide latency.

5.2.2.3. *Bus communication unit (BCU)*

It is responsible for communication between interface and bus and vice versa. Fig. 9 below shows the component together with the internal interface working.

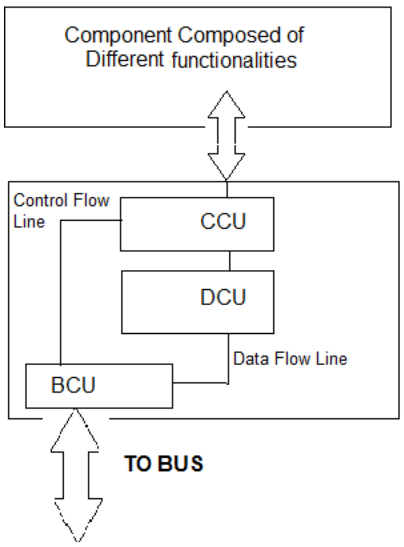


Fig. 9. Interface.

5.2.3. *Bus*

The bus is a messaging infrastructure to allow different systems to communicate through a shared set of interfaces [11]. This is analogous to a bus in a computer system which serves as the focal point for communication between the CPU, Main memory, and peripherals. Messages may be data messages or control messages. Control messages enable managing services, while data messages are service-specific input and output data of services.

5.2.4. *Broker*

It synchronizes different services. It's the duty of the broker to discover the services required for the application to work. This record of services is maintained for rapid service discovery. This also involves invoking the service in case the service is not running.

6. Experimental Results

In order to address the challenges associated with concurrency while running two different services Wall Follower and Space Wanderer and to validate the effective implementation of concurrency using DBUS and SOA, the experimental setup was designed. There exists two or more than two robots (N robots), equipped with sonar, position 2D and blob finder sensors. These robots are asynchronously controlled by a single JVM. There are the services which are running on these individual robots. It is required to establish a coordination mechanism between the robots accessing the services concurrently. This coordination is achieved by using subsumption architecture, where robots subsume their services to run another service based on detection of colors on their paths. The three distinguished coordination among the robots are identified. This coordination is based on group switching, selective switching and individual robot rotation. In group switching, the group of robots switches their services in groups, in selective switching the selected robots in a group switch their services whereas rotation service is specifically designed for individual robot rotation. To implement this, it is assumed that each robot has its own local state. Table 2 summarizes, the services running based on color detection and robots local state.

Table 2. Salient features of setup for comparison.

Local State	Color	Service
0	Blue	Wall Fallower
1	Red	Space Wanderer
2	Green	Rotate Service
3	Gray	Space wanderer and hold service in other robot

For getting information regarding wall directions sonar data is required. It is assumed in the service that sonar data is available. Using this data turns in wall is detected and robot is made to turn as wall turns. So it starts following the wall in the left direction.

The calculated values can be sent to robot. To any application which can provide sonar data can use this Wall follower Service. Figure 10 depicts the flow chart of running services.

The calculated values can be sent to the robot. Any application which can provide sonar data can use this Wall follower Service**.** Fig. 10 depicts the flow chart of running services.

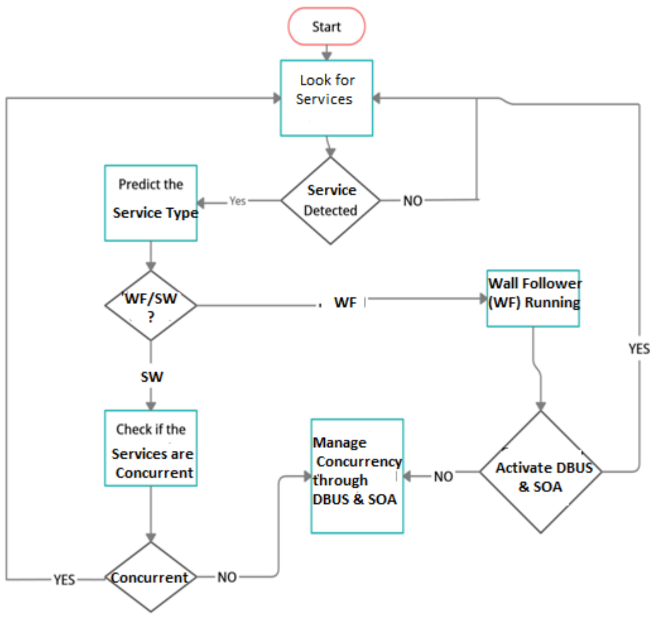
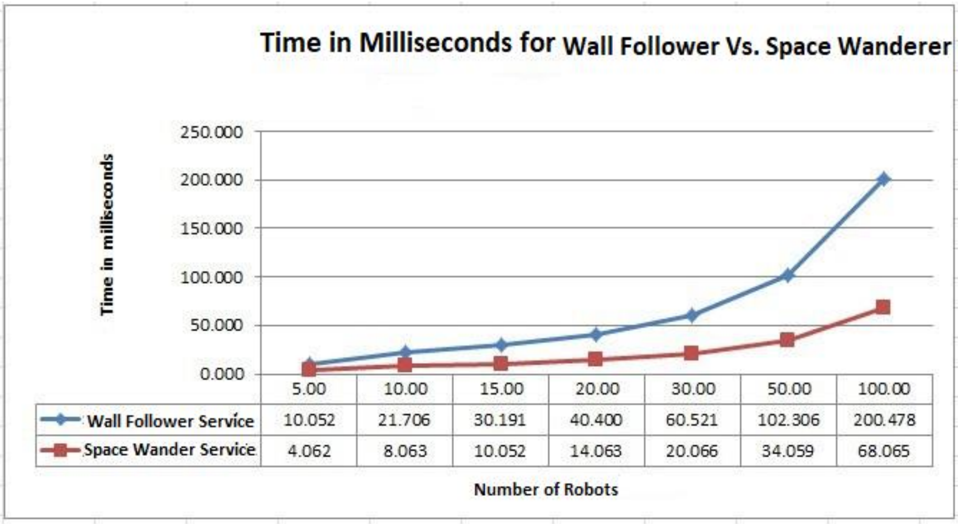


Fig. 10. Flow chart of running services.

All the services that are required need to be connected to the BUS so that application can connect to these services. We are using DBUS as the messaging BUS. We are registering all the services to the DBUS using the following code snippet:
register_service (connection, bus_proxy, DBUS_SERVICE);

Whenever the service is required message to the service is passed using the common BUS between the application and the service. All the other issues, like synchronization between two services, need to be handled by the application.
VoidWallFollower (DBusGProxy *,int *,double *ver_x,double *vel_thetha)
Void wander (DBusGProxy *,int *,double *ver_x,double *vel_thetha)
Application is tested on pioneer robot with sonars. The performance comparison of running different robots using space wanderer service and Wall Follower services is shown in the below Fig. 11. The experiment was run for 5, 10, 20, 30, 50 and 100 robots for both the cases independently. As shown in the Fig. 12, the Wall Follower case takes significantly more response time in comparison to space wanderer case as expected.



	5.00	10.00	15.00	20.00	30.00	50.00	100.00
◆ Wall Follower Service	10.052	21.706	30.191	40.400	60.521	102.306	200.478
■ Space Wander Service	4.062	8.063	10.052	14.063	20.066	34.059	68.065

Fig. 11. Time taken in running services.

In all the cases with different number of robots response time is reduced by a factor of approximately 2.5 to 3 in asynchronous case as shown in Fig. 12.

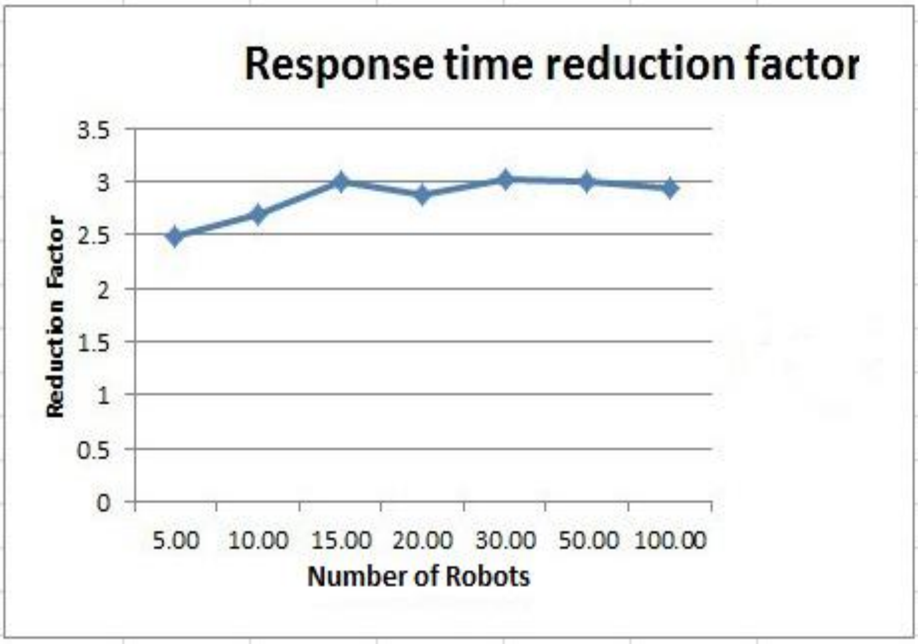


Fig. 12. Average time taken by robots.

7. Conclusion

Service Oriented Architecture seems a good approach for connecting different heterogeneous robotics packages which are built independently. Moreover hot plugging the services is one of the most preferred attribute of the built architecture. By hot plugging we mean the services can be integrated without obsoleting the previously built architecture. Thus new services can be added whenever it is required. Moreover we know that services keep on updating. So updated service can be integrated easily. Thus Architecture is dynamic in the sense of integration. The percentage saving in response time by using DBUS and SOA ranges from minimum 59.59 % to maximum 66.8 % in comparison to other communication. Thus response time is reduced by a factor of approximately 2.5 to 3.0 in space wanderer service case. Therefore, DBUS with SOA support message communication effectively in robotics application. The conventional message passing approach of concurrent services implementation produces a well-known issue of Callback Hell. DBUS and SOA based approaches solves this issue. The Services switching time during concurrency is reduced by a factor of approximately 1.04 to 1.208.

References

1. R. A. Brooks, Science **253,** 1227 (1991). https://doi.org/10.1126/science.253.5025.1227
2. W. Hongxing, D. Xinming, L. Shiyi, T. Guofeng, and W Tianmiao, A Component Based Design Framework for Robot Software Architecture – *Proc. of the 2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems* (IEEE Press, Piscataway, NJ, 2009) pp. 3429-3434.
3. X. Yan, W. Li, and D. Chen, IEEE Int. Conf. on Robotics and Biomimetics, 750 (2006).
4. C. Cote, Y. Brosseau, D. Letourneau, C. Raievsky, and F. Michaud, Int. J. Adv. Robotic Syst. 1729 (2006). https://doi.org/10.5772/5758
5. M. Montemerlo, N. Roy, and S. Thrun, Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit, Int. Conf. on Intelligent Robots and Syst. Las Vegas, **3**, 2436 (2003).
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1994).
7. W. R. Zwicky, PhD thesis, University of Illinois at Urbana-Champaign (2008).
8. R. K. Karmani, A. Shali, and G. Agha, Actor Frameworks for the JVM Platform: A Comparative Analysis - *7th Int. Conf. on Principles and Practice of Programming in Java*, 11 (2009). https://doi.org/10.1145/1596655.1596658
9. A. Elkady and T. Sobh, J. Robotics 959013 (2012). https://doi.org/10.1155/2012/959013
10. D. Létourneau, C. Côté, C. Raïevsky, Y. Brosseau, and F. Michaud, Springer Tracts on Adv. Robotics, 221 (2006).
11. D. Spinellis, Another Level of Indirection, in Beautiful Code: Leading Programmers Explain How They Think, Edited by A. Oram and G. Wilson (O'Reilly Media, Inc. 2007), pp. 279–291.
12. Software as a Service: Strategic Backgrounder, Software & Information Industry Association, (2001).
13. Messaging Pattern in Service oriented Architecture, http://msdn.microsoft.com/en-us/library/aa480027.aspx, retrieved September 2010.
14. Enterprise Message Bus, http://en.wikipedia.org/wiki/Enterprise_service_bus, retrieved September 2010.
15. R. P. Srivastava and G. C. Nandi, Controlling Multi Thread Execution using Single Thread Event Loop – *Int. Conf. on Innovations in Control, Communication and Information Systems (ICICCI)* (2017) pp. 1-7. https://doi.org/10.1109/ICICCIS.2017.8660809

16. R. P. Srivastava and G. C. Nandi, Integration of Robotics Components and Verification using Petri Net – *Int. Conf. on Innovations in Control, Communication and Information Systems (ICICCI)* (2017) pp. 1-7. https://doi.org/10.1109/ICICCIS.2017.8660756
17. T. K. Kaiser, C. Lang, F. A. Marwitz, C. Charles, S. Dreier, J. Petzold, M. F. Hannawald, M. J. Begemann, and H. Hamann, Distributed Autonomous Robotic Systems 190 (2021). https://doi.org/10.1007/978-3-030-92790-5_15
18. M. Mohammadi and M. Mukhtar, Service-Oriented Architecture and Process Modeling - *Int. Conf. on Information Technologies (InfoTech)* (2018) pp. 1-4. https://doi.org/10.1109/InfoTech.2018.8510730
19. R. Srivastav, G. Nandi, R. Shukla, and H. Verma, IAES Int. J. Robotics Automat. (IJRA) **8**, 217 (2019). https://doi.org/10.11591/ijra.v8i4.pp217-244
20. R. K. Megalingam, S. Tantravahi, H. S. S. K. Tammana, N. Thokala, H. S. R. Puram, and N. Samudrala, Robot Operating System Integrated robot control through Secure Shell(SSH) - *Int. Conf. on Recent Developments in Control, Automation & Power Engineering (RDCAPE)* (2019) pp. 569-573. https://doi.org/10.1109/RDCAPE47089.2019.8979113
21. W. S. Cheong, S. F. Kamarulzaman, and M. A. Rahman, Implementation of Robot Operating System in Smart Garbage Bin Robot with Obstacle Avoidance System, Emerging Technology in Computing, Communication and Electronics (ETCCE) (2020) pp. 1-6. https://doi.org/10.1109/ETCCE51779.2020.9350912,
22. P. Leger, H. Fukuda, and I. Figueroa, J. Universal Comput. Sci. **27**, 955 (2021). https://doi.org/10.3897/jucs.72205