Communication Architecture for Robotic Applications

Gabriel Tamashiro[1], Kelen C. T. Vivaldini[1], José Martins Junior[2], and Marcelo Becker[1*]

[1]Mechatronics Group - Mobile Robotics Lab.
EESC-USP
Av. Trabalhador Sao Carlense, 400. São Carlos-SP, Brazil

[2]Computer Science Dept
EEP-FUMEP
Av Monsenhor Martinho Salgot, 560, Piracicaba-SP, Brazil

ABSTRACT

In robotic applications, the communication protocol is one of the main features that dictate scalability and network topology. The communication architecture must consider heterogeneity (i.e. network, hardware, operational system and programming language) and provide programming abstraction to simplify its development. There are several middlewares and frameworks that can be applied in robotic applications, differing considerably in complexity, programming languages and approach. In this context, this paper presents a communication architecture that fulfils such requirements and ensures information exchange through the network. It was evolved from a previous study, providing more flexibility and easily to adapt to other applications. An interface definition language (IDL) was conceived that enables users to define and deploy services, and also adjustable constraints (service request timeout, message size, maximum number of connected nodes) that restrict provided functionalities. The middleware is based on a multi-threaded service-oriented hybrid peer-to-peer architecture that uses concepts of object-oriented programming in a layered structure to provide flexibility for the communication implementation, minimizing code changes when ported to other robotic systems. Tests of availability and network response time were performed to evaluate its time constraints. The middleware applicability was proven when implemented in an AGV distributed system, designed to operate an intelligent warehouse.

1. INTRODUCTION

In distributed applications, communication protocol used among nodes is one of the main features that dictate scalability and topology. It's important to choose the communication architecture that best suits each application purpose. An interesting way to deal with communication is to adopt a middleware to handle the information exchange among nodes. A middleware is a software layer that implements communication abstractions, which simplify the application integration in a target platform containing the operating system, network protocols and hardware. It provides a programming model that masks the underlying layers and spares application developers from dealing with low-level and platform-dependent code, such as socket network programming [1-2]. The middleware also provides high-level network-oriented abstraction and services, which simplify the development of distributed systems. Therefore, it allows developers to produce much more reusable codes and focus on problems concerning the application details.

Regarding robotic applications, such as sensors and actuators network [3-4], AGV (Automated Guided Vehicle) systems [5] and multi-agent systems [6], there are a reasonable number of frameworks and middlewares (CORBA, Java RMI, ROS, YARP, etc) that can provide support on this type of application. They vary considerably in complexity, available services, programming languages and operational systems. Regarding robotic frameworks, besides providing a communication architecture, they also provide algorithms and drivers to solve specific problems concerning robotic applications. By adopting a specific middleware or framework to handle the communication in a determined application, it could be adding unnecessary complexity or it could be imposing restrictions in the communication mechanisms. For example, the use of CORBA implementations usually requires additional elements in the system to its execution and implementation. Regarding the YARP framework, it provides a flexible and easy to implement communication architecture that allows multiple nodes to exchange information, but relies only on indirect communication, thus requiring a server node to mediate the communication. Therefore, it is desirable a communication architecture that is adapted to maximize the performance of a determined application based on its requirements.

* Corresponding author: Tel.: (55) 3373-9431; Fax: (55) 3373-9402; E-mail: becker@sc.usp.br

In the adoption of a middleware, some characteristics are important for distributed applications [1][7]. The middleware should provide integration of client applications in an efficient, flexible and transparent way and provide a standardized model to simplify the process of implementing new functionalities. These characteristics allow generation of reusable code that reduces the lifecycle cost of the application development.

This paper presents a service-oriented hybrid peer-to-peer communication middleware architecture designed to provide a useful and versatile tool for the development of distributed applications, especially in mobile robotics. It provides set of services, such as the discovery and registry services, which grants scalability, availability and reliability to the middleware, and allows both synchronous and asynchronous communication. Since its main objective is to provide services in the network, an interface definition language (IDL) has also been conceived to both define and request services. Section 2 describes the middleware architecture, section 3 reports on the availability and response time tests performed to evaluate the architecture. Section 4 discusses the previous sections and future work and concludes the paper.

2. COMMUNICATION ARCHITECTURE

The proposed communication middleware is still under development and initiated on a previous work presented by Vivaldini et al. [8]. Many aspects of the architecture have been changed to improve its flexibility and applicability, resulting in the structure presented in this paper. The middleware was structured using a layered approach to simplify the development process. This approach reduces the maintenance time as layers can be implemented separately without readapting the whole structure to each modification. The structure is subdivided into three layers [1]: host infrastructure, distribution and common services. The descriptions of each layer as well as a partial class diagram of the middleware architecture are shown in Fig. 1.

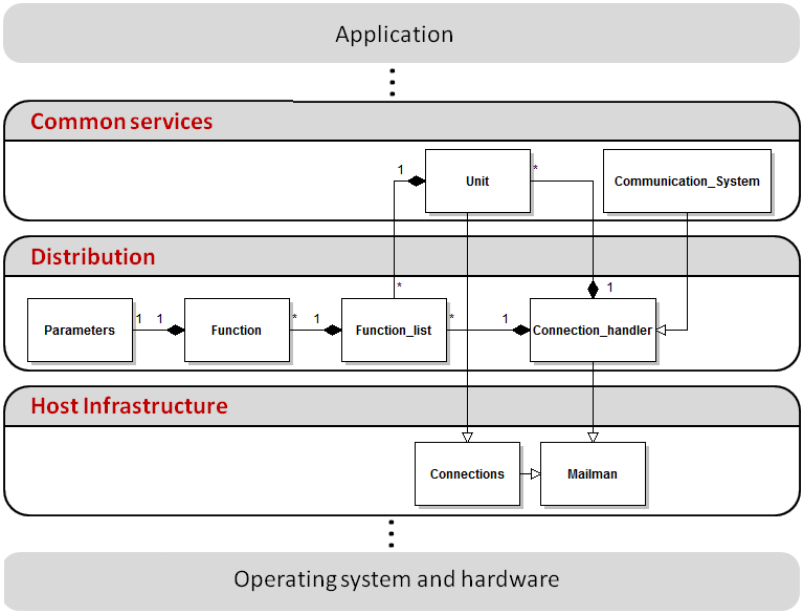

Figure 1. Layers and the partial class diagram of the middleware.

The host infrastructure layer is responsible for abstracting native OS communication and concurrency mechanisms. It deals with low-level OS programming APIs, such as sockets and POSIX pthreads to provide the middleware with a set of functionalities related to message delivery, message monitoring and connection establishment. As shown in Fig. 1, the host infrastructure layer is implemented by the Connections and Mailman classes. Messages are exchanged through the Internet Protocol (IP), thus the Mailman class provides connection establishment and message exchange services for both the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). On the other hand, the Connection class manages and monitors the connections established and forwards incoming messages to higher layers. Both Connections and Mailman classes implement concurrency mechanisms to provide synchronous and asynchronous communication.

The distribution layer uses services offered by the underlying layer and provides high-level distributed communication functionalities. It also defines protocols that enable the interaction of distributed nodes regardless of their heterogeneity (i.e. network, hardware, operating system and programming language). Parameters, Function, Function_list and Connection_handler classes (Fig. 1) are responsible for implementing this layer. The first three classes define the interface definition language (IDL), which provides clients with a language-independent model to define application services and arguments. Basically, the Parameters class allows developers to define variable types (int, float or string) and sizes passed as arguments in the functions defined through the Function and Function_list classes. The IDL usage will be exemplified in the code example presented later in this section. Another important aspect of such middleware is the discovery service implemented by the Connection_handler class. This service searches and establishes connections with unknown nodes and exchanges available services list information with then. It works like the Address Resolution Protocol [9], in which one node broadcasts a message containing its identifier in the network and each node that receives it may send a response message back to establish a connection. The Connection_handler class is also a container that manages and classifies all information related to the nodes, such as address, identifiers and services.

Finally, the common services layer enhances the middleware distributed "aspect" by providing high-level services and models, which allow developers to focus on the application construction logic rather than on the details of the communication or message encapsulation. Some of the functionalities provided in this layer are service definition and synchronous and asynchronous requesting and publishing and are implemented by Unit and Communication_System classes. The main important difference between them is the number of nodes in which they can connect. While the Communication_System class enables the creation of objects (named m_unit) that connect to multiple nodes, the Unit class produces objects (named s_unit) that connect to only a single node. These classes allow the development of distributed applications with both pure or hybrid peer-to-peer topology (see Fig. 2), providing more flexibility to the model.

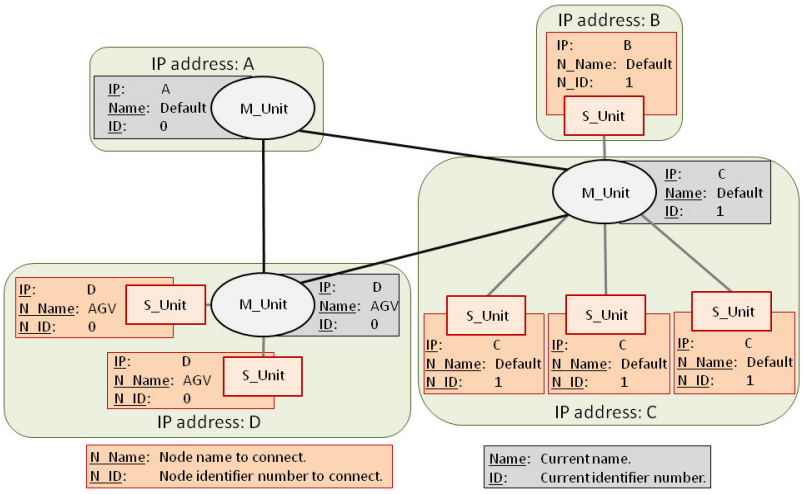

Figure 2. Possible topology provided by objects of the Communication_System and Unit classes.

As shown in Fig. 2, the IP address, name and ID number are the three properties that distinguish each node and are used by the discovery service. The m_unit type holds its own properties, whereas the s_unit type holds the manually set properties (name and ID number) of the m_unit object they shall connect.

When requesting a service from another node, each middleware layer adds its relevant information to the request message and sends it to the layer below, until the full message is ready to be delivered. When receiving a service request message, it performs the opposite direction: receives it at the lowest layer and sends it to the layers above, until the message is completely processed and the proper service starts being executed. The message format is illustrated in Fig. 3. The middleware was also designed to work under some constraints that can be manually set: time out of synchronous service requests, which will return an error when the time waiting for a service return exceeds the time out defined; time between service requests sent to a given node, which limits the frequency of requests; maximum number of connected nodes; maximum message size, which will limit the number and size of arguments.

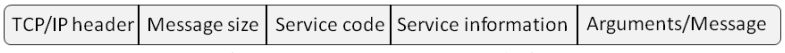

Figure 3. Message encapsulation.

An example code of a simple AGV system manager (m_unit) connecting and interacting with an AGV (s_unit) is provided below to exemplify how the middleware is used. The available services mentioned are supposed to be properly implemented and published in the network.

Example code of an AGV (m_unit) system manager that establishes the connection with an AGV (s_unit), requests services from the AGV and publishes a service.

```
1     Communication_System<> AGV_manager;
2     AGV_manager.start_thread();
3     int s_unit_number = AGV_manager.get_number_s_units();
4     while ( s_unit_number < 1 )
5     {
6       AGV_manager.broadcast();
7       sleep(2);
8       s_unit_number = AGV_manager.get_number_s_units();
9     }
10    AGV_manager.s_units[0]->their_function_list->show_functions();
11    Parameters pos;
12    pos.add_int("position", 2);
13    AGV_manager.s_units[0]->block_request_their_function("pos_info", pos);
14    int coordinates[2];
15    pos.get_variable("position", coordinates);
16    coordinates [0] = 1;
17    coordinates [1] = 2;
18    pos.set_value("position", coordinates);
19    AGV_manager.s_units[0]->request_their_function("go_to", pos);
20    Parameters arguments;
21    //arguments variables defined here.
22   AGV_manager.my_function_list->add_function("call_mechanic", arguments, &function_pointer);
23    AGV_manager.s_units[0]->my_function_list->publish_my_function_list();
24    AGV_manager.my_function_list->publish_my_function_list();
```

Lines 1 and 2 create the m_unit object node and search for other nodes in the network, spawning threads to establish and maintain the connections. Lines 3 to 9 check the number of s_unit nodes connected and ensure that at least one s_unit node (the AGV) will connect before proceeding with the rest of the code. Line 10 shows all available services offered by the AGV (index 0 since it is the only node connected). At this point, if the manager wishes to know the position of the AGV, he could request the "pos_info" service (supposed to be defined at the AGV routine) from the AGV using the IDL shown in lines 11 to 13: it creates a Parameters object (line 11), adds an int array with size 2 (for x and y coordinates of the AGV) named “position” (line 12) and performs a synchronous request to the AGV (line 13). The request will update the “pos” object as it receives a response from the AGV, and the “position” variable within the “pos” object is retrieved in lines 14 and 15. It is important to mention that when performing a service request, if the Parameters object passed does not have the arguments necessary for the function being called (i.e. name, type and size), the call returns an error.

If the manager wishes the AGV to move to the position (x,y)=(1,2), he could use the "go_to" service as shown in lines 16 to 19: lines 16 to 18 set the “position” variable within the “pos” object and then line 19 performs an asynchronous request, meaning that the manager would request the AGV to go to the position (x,y)=(1,2) without waiting for a return message (“pos” object will not be updated).

Finally, if the manger wishes to provide the AGV with the "call_mechanic" service, he can define and publish it as shown in code lines 20 to 24. When defining the function where the service is implemented, it’s necessary that both the return and argument (single argument) of the function is set to Parameters type. It is also possible to publish services implemented in an object function, which only requires the object class to be passed as a template parameter when creating a Communication_System or Unit object. Lines 23 and 24 publish the "call_mechanic" service in the network, but the difference is that whereas line 23 publishes the new service only to the “s_units[0]” node, line 24 publishes it to all s_unit and m_unit nodes connected to the manager.

3. EXPERIMENTAL TESTS

Availability and response time tests were performed to evaluate the middleware architecture. These tests estimate the middleware intrinsic time constraints and reliability, which are important for the applications routine. The availability test consisted of sending an asynchronous service request to a given node several times with a determined frequency. For each frequency, it was measured how many times the service being requested was actually executed in that node. The service defined for this test consisted of a requested node accepting a string argument and then performing a print command, which does not significantly increase the execution time of the communication. The availability test was measured for different sets of frequencies and string sizes. Regarding the response time test, it was performed by synchronously requesting a service from a given node several times with a constant time of two seconds between requests. The service defined, in this case, consisted of a requested node receiving a string variable as an argument, performing a print command and then returning the same string to the requesting node. For each string size, the RTT (Round Trip Time), i.e., the time waited by the requesting node to receive a service return message once it has sent the request, was measured.

The two tests described were performed in two different node distributions (1 and 2 from Figs. 4 and 5), but for both cases the measurements were performed by a m_unit (main) node requesting services to another m_unit node (passive). In distribution 1, it consisted of an isolated scenario in which only the main and passive nodes are presented. In distribution 2, it consisted of a more dynamic scenario, where five additional s_unit nodes were introduced to increase the load in the network. These additional nodes request synchronously the same service hosted by the passive node with a random request interval between 1 and 20 ms, and a message size of 1000 characters. For each configuration (node distribution, string size and request frequency of the main node) a thousand service requests were performed for the availability test (measured in sequence groups of a hundred requests), and a hundred service requests were performed for the response time test. The node distributions and the results are shown in Figs. 4, 5, 6 and 7. It is also important to note that before the tests started, the constant time between requests, previously mentioned as one of the middleware constraints, was set to zero.

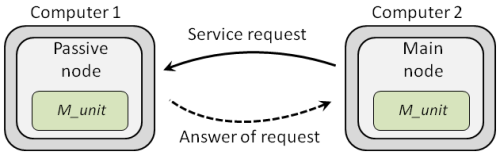

Figure 4. Node distributions 1 used in the availability and response time tests.

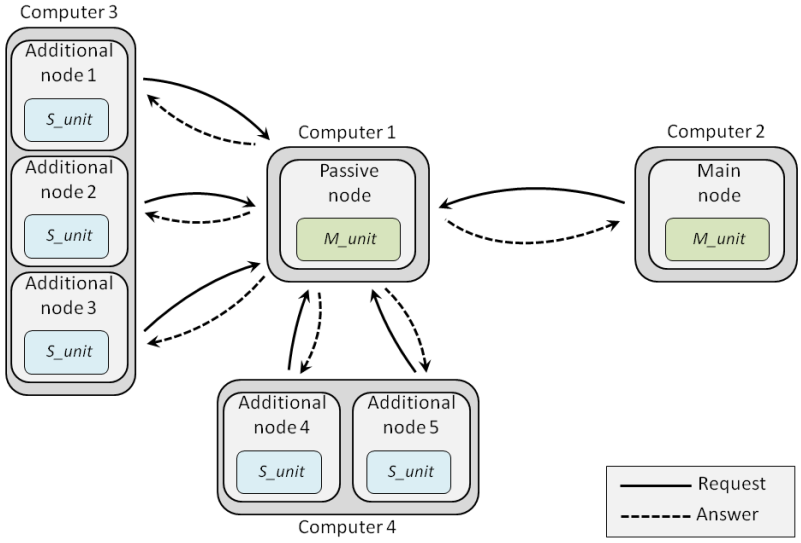

Figure 5. Node distributions 2 used in the availability and response time tests.

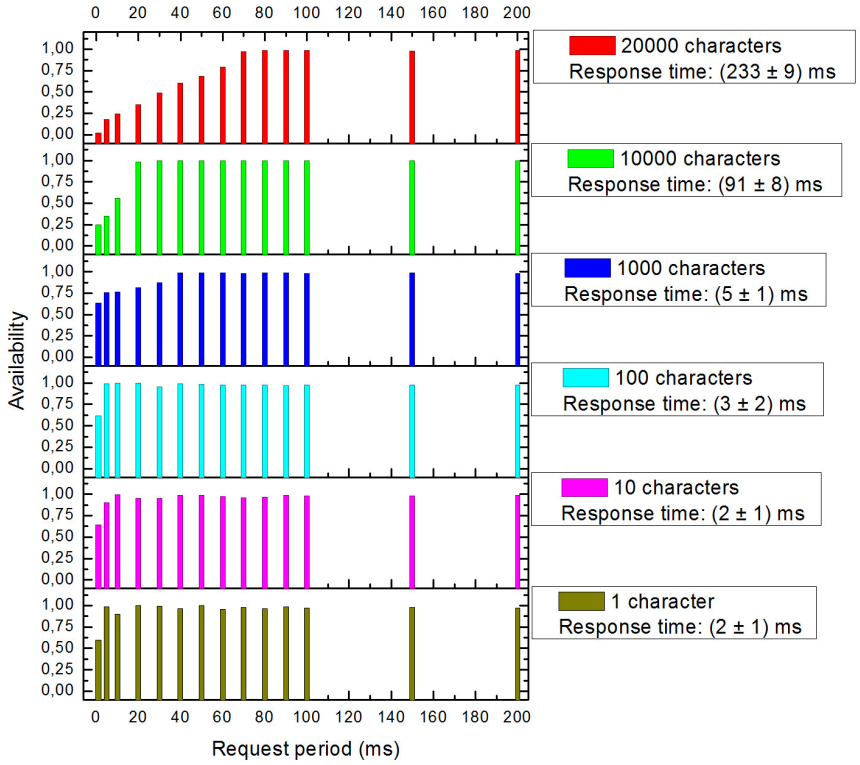

Figure 6. Availability and response times measured for distribution 1.

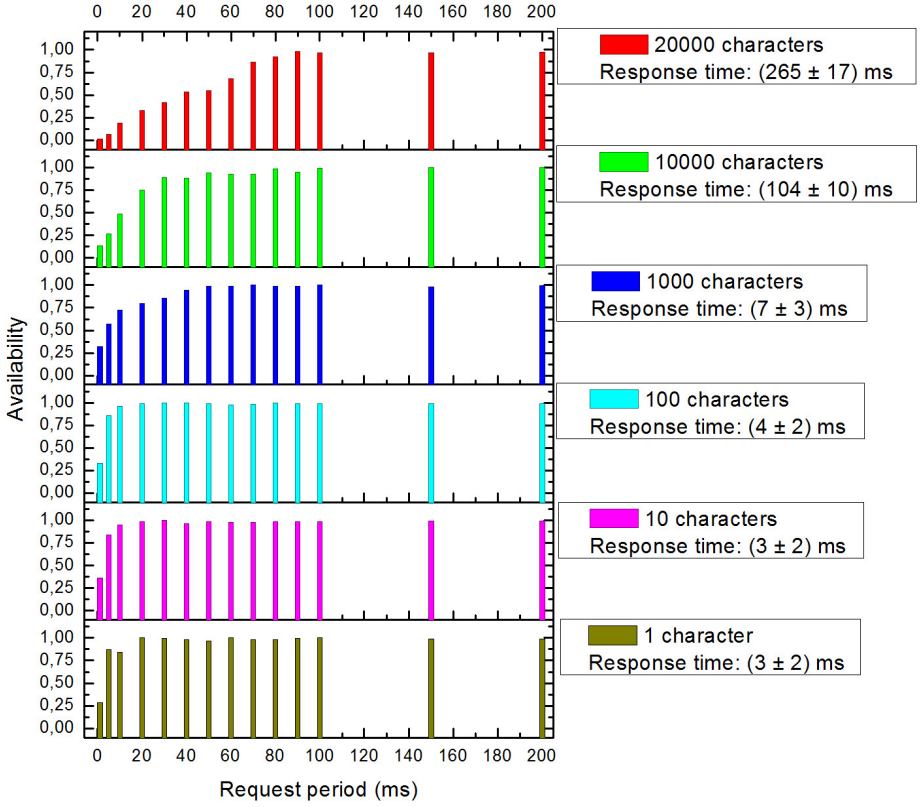

Figure 7. Availability and response times measured for distribution 2.

As shown in Figs. 6 and 7, the response time measured for string sizes up to 1000 characters presented the same order of magnitude, increasing considerably for larger sizes. An internal analysis for the string size of 20000 characters, for the distribution 1, showed that the process time required for the generation and processing of the messages exchanged consisted of about 72% (11% for generating and 61% for processing messages) of the response time measured. Therefore, the mechanisms used in the generation and processing of messages must be improved to increase the performance of the proposed architecture.

For the availability test presented in Fig. 6, it can be seen that for string sizes up to 10000 characters, the availability demonstrated to be closer to 1 (minimum of 0.81) for request periods larger than 20 ms. For string sizes of 20000 characters, the availability was closer to 1 for request periods larger than 70 ms (minimum of 0.97). As expected, the introduction of the additional nodes shown in Fig. 7 provided a negative impact on the tests measured. The mean response times increased, as can be seen clearly for string sizes larger than 10000 characters. The availability measured decreased considerably for request periods shorter than 20 ms for string sizes up to 10000 characters. For the string size of 20000 characters, it could be seen a reduction for request periods shorter than 80 ms (Fig. 8).

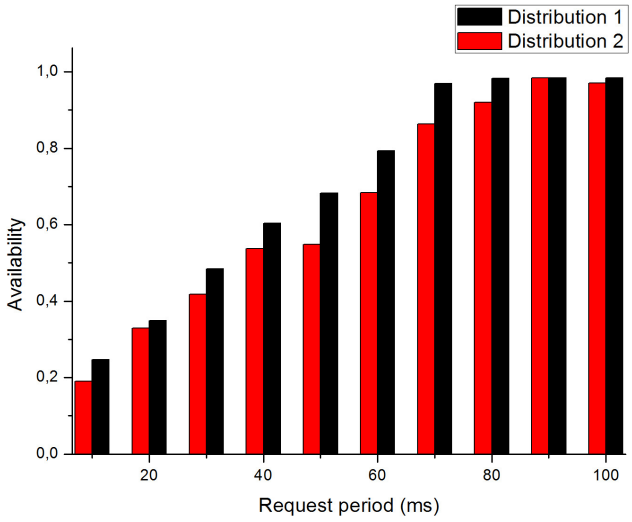

Figure 8. Availability test measured for distributions 1 and 2 for the string size of 20000 characters.

Regarding the response time test, approximately 3% of the RTT data collected were discarded for being up to 3 orders of magnitude higher than the average. This behavior was observed in several distributed applications, and is discussed by Dumitraş [10]. In this case, it might occur because the transport mechanism implemented (TCP and UDP) does not ensure communication latency for every message exchanged. Nonetheless, the response time observed in both distributions remained in the same order of magnitude, quickly increasing for string sizes higher than 10 000 characters.

4. CONCLUSIONS

In this paper, the advantages of adopting a communication architecture when developing distributed applications were briefly described, and a communication middleware architecture was proposed. The paper describes the services and internal structure of the proposed architecture and provides a usage example code of the middleware. Also, an IDL was conceived to provide developers with a language-independent model to define application services and its arguments. As can be seen in the example code, the middleware has shown to be a useful and flexible tool for the development of distributed robotic applications, allowing nodes to easily find, connect and interact with other nodes in the network. To evaluate the architecture, availability and response time tests were performed over two different node distributions. The tests showed that although the transport mechanism implemented does not support real-time capabilities, it provided acceptable results of response time and availability depending on the request period and string size. Even with the introduction of additional nodes to increase the load in the network, the performance measured was still acceptable for the node distribution used. Better results can be achieved if the mechanisms used in the generation and processing of messages are improved, as could be seen that they have a considerable impact on the response time of the architecture. The middleware operation constraints may be adjusted so that a balanced level of performance and reliability can be achieved for each node distribution and usual message size required by developers.

The layered structure adopted has considerably contributed to the middleware development process. In future studies, services regarding priority request queues, scheduling and node election will be considered to support a larger range of application requirements. Other transport mechanisms may also be considered to improve the middleware predictability and reliability, and the mechanisms involved in the marshalling and unmarshalling of data exchanged in the messages will be further analyzed to achieve a better performance.

ACKNOWLEDGEMENTS

The authors would like to acknowledge CNPq (Grant 142184/2010-1 and 130220/2012-4) for the financial support provided to this research.

REFERENCES

[1] *R.E. Schantz, and D.C., Schmidt: "Research Advances in Middleware for Distributed Systems: State of the Art", IFIP World Computer Congress, Montreal, Canada. 2002.*

[2] *G. Coulouris, J. Dollimore and T. Kindberg: Distributed Systems Concepts and Design. Addison-Wesley, Boston, 5th edition, 2011*

[3] *J.A. Stankovic, et al.: "Real-time communication and coordination in embedded sensor networks".Proceedings of the IEEE, Vol. 91, Issue 7, pp.1002–1022, 2003.*

[4] *J. Chen, M. Díaz and J. M. Troya:. "PS-QUASAR: A publish/subscribe QoS aware middleware for Wireless Sensor and Actor Networks". Journal of Systems and Software, Vol. 86, No. 6, pp.1650–1662, 2013.*

[5] *K. C. T. Vivaldini, et al.: "Automatic Routing System for Intelligent Warehouses". IEEE International Conference on Robotics And Automation, pp. 93–98, 2010.*

[6] *P. Iñigo-Blasco et al.:"Robotics software frameworks for multi-agent robotic systems development". Robotics and Autonomous Systems, Vol. 60, No. 6, pp.803-821, 2012.*

[7] *J. Al-Jaroodi and N. Mohamed: "Service-oriented middleware: A survey". Journal of Network and Computer Applications, Vol. 35, No.1, pp.211–220, 2012.*

[8] *K. C. T. Vivaldini, et al.: "Communication Infrastructure in the Centralized Management System for Intelligent Warehouses". In Proceedings of the 23rd International Conference Flexible Automation and Intelligent Manufacturing – FAIM2013, Porto, Portugal, 2013.*

[9] *A.S Tanenbaum and M. Van Steen: Distributed Systems: Principles and Paradigms. Pearson Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2007.*

[10] *T. Dumitraş and P. Narasimhan: "A study of unpredictability in fault-tolerant middleware". Computer Networks,Vol. 57, No. 3, pp. 682–698, 2013.*