J Intell Robot Syst (2013) 72:5–19
DOI 10.1007/s10846-012-9798-y

A Structured Approach for Modular Design in Robotics and Automation Environments

Ayssam Elkady · Jovin Joy · Tarek Sobh · Kimon Valavanis

Received: 11 May 2012 / Accepted: 10 December 2012 / Published online: 15 January 2013

Abstract In this paper, the *RISCWare* framework is proposed as a robotic middleware for the modular design of sensory modules, actuation platforms, and task descriptions. This framework will be used to customize robotic platforms by simply defining the available sensing devices, actuation platforms and required tasks. In addition, this framework will significantly increase the capability of robotic industries in the analysis, design, and development of autonomous mobile platforms. *RISCWare* is comprised of three modules. The first module encapsulates the sensors, which gather information about the remote or local environment. The second module defines the platforms, manipulators, and actuation methods. The last module describes the tasks that the robotic platforms will perform such as teleoperation, navigation, obstacle avoidance, manipulation, 3-D reconstruction, and map building. The objective is to design a middleware framework to allow a user to plug in new sensors, tasks or actuation hardware, resulting in a fully functional operational system. Furthermore, the user is able to install and uninstall hardware/software components through system lifetime with ease and modularity. In addition, when hardware devices are plugged into the framework, they are automatically detected by the middleware layer, which loads the appropriate software and avails the device for applications usage. This automatic detection and configuration of devices make it efficient and seamless for end users to add and use new devices and software applications. Several experiments, performed on the *RISCbot II* mobile robot, are implemented to evaluate the *RISCWare* framework with respect to applicability and resource utilization.

Keywords *RISCWare* · Middleware · Plug and play · Teleoperation · Face detection · Face recognition · Modular design · Architecture · *RISCbot II*

A. Elkady (✉) · J. Joy · T. Sobh
School of Engineering, University of Bridgeport, Denver, CO 80208, USA
e-mail: ayssam.elkady@gmail.com

J. Joy
e-mail: jovinj@gmail.com

T. Sobh
e-mail: sobh@bridgeport.edu

K. Valavanis
Electrical and Computer Engineering, University of Denver Tampa, FL 33620, USA
e-mail: kimon.valavanis@du.edu

1 Introduction

Robotic middleware is a software architecture represented by an Application Programming Interface (API), which is a set of specifications that

developers should follow while writing their own code to guarantee software portability between different vendor platforms. Robotic middleware, which is an abstraction layer residing between the operation system and the software applications, is designed to manage the heterogeneity of the hardware, improve software application quality, simplify software design, and reduce development costs.

RISCWare is developed to support different sensory devices, actuation platforms, and software applications; in addition, it is available for different programming languages. It is designed to be a modular, efficient, flexible, platform-independent middleware platform. Furthermore, the components (software and hardware) can be replaced before or during runtime. The flexible property of *RISCWare* enables a third party to design self-configuring modules, and the Plug and Play (PnP) property of the *RISCWare* gives the robot's user the ability to plug in a component, which the *RISCWare* then recognizes. One of the most desirable characteristics of *RISCWare* is the resource-efficient transmission schemes that achieve the best utilization of available networking resources. The unitization of the network is a very important issue because the sensory devices generate many small messages on a regular basis.

RISCWare is implemented as a messaging system used to communicate between any component (sensory device, actuation platform and software application) at any desired time. *RISCWare* allows two or more components to exchange information in the form of messages. Using messaging, services can be invoked from software applications and hardware devices. We chose to implement *RISCWare* as a messaging system because messaging provides a high degree of decoupling between components, so it plays a key role in the integration of heterogeneous systems. The components can be abstracted and decoupled in such a way that they can be replaced with little or no knowledge by the other components. In addition, messaging offers the ability to process requests asynchronously to increase the performance of the system and reduce system bottlenecks. *RISCWare* ensures that messages are properly distributed among applications. Furthermore, it provides fault tolerance, load balancing, scalability, and transactional support for applications that need to reliably exchange large quantities of messages. Parallel processing in *RISCWare* is achieved by introducing multiple message receivers that can process different messages simultaneously.

We are proposing the implementation of a framework (*RISCWare*) for modular design, which is comprised of three modules. The first module encapsulates the sensors that gather information about the remote or local environment. The second module defines the platforms and actuation methods. The last module describes the tasks that the platforms will perform, such as teleoperation, navigation, obstacle avoidance, manipulation, 3-D reconstruction and map building.

Fig. 1 A prototype of the RISCbot II

This modular design allows for easy customization of applications that require different sensor technologies, resolutions and spectral sensitivities.

We have built a platform (*RISCbot II*, shown in Fig. 1), which consists of a comprehensive sensor suite. *RISCWare* is designed to allow for modular capabilities, including a variety of sensory modules (software and hardware), methods to define tasks in a seamless manner, and finally, the capability to define various actuation and manipulation platforms.

Section 2 introduces prior work, Section 3 presents the features of *RISCWare*, and then the architecture and some case studies of *RISCWare* are presented in Sections 4 and 5. *RISCWare* components are described in Section 6. Section 7 evaluates the *RISCWare* framework with respect to "A greeting Application". Finally, Section 8 presents a summary of the work and draws some conclusions.

2 Related Work

There are several other robotics software platforms available, such as Robot Operating System (ROS) [1], CLARAty [2], Player [3], Miro [4], Webots [5], RT-Middleware (RTM) [6] SmartSoft [7], ERSP [8], Orca [9], OPRoS [10], and Microsoft Robotic Studio [11]. A survey of robot development environments (RDEs) by Kramer and Scheutz [12] described and evaluated nine open source, freely available RDEs for mobile robots. In [13, 14], an overview study of robot middleware is provided. Finally in [15], some freely available middleware frameworks for robotics were addressed, including their technologies within the field of multi-robot systems.

In [16], we outlined the architecture and some important attributes, with the comprehensive set of appropriate bibliographic references that are classified based on middleware attributes for most of the existing robotic middleware, such as Player, CLARAty, ORCA, MIRO, and etc. Furthermore in [16], we present a literature survey and attribute-based bibliography of the current state-of-the art in robotic middleware design.

3 Features

RISCWare is designed to provide the following features:

1 ***Modularity***
2 ***Open Source***
3 ***Real-time System***
4 ***Plug and Play, and Dynamic Wiring***: The software and hardware components can be downloaded, installed and configured at or before run-time.
5 ***Easy to Use***
6 ***Authentication, Authorization and Secure Communication***: Authentication is the mechanism to provide a way of identifying the user of the systems (maybe by having the user enter a valid user name and password). After the authentication process, authorization is required to determine what types or resources, services and access level a particular authenticated user is permitted. The secure communication is performed by encrypting the messages to prevent eavesdroppers from spying on the messages.
7 ***Hardware Abstraction***: Hides the low-level, device-specific details of the device in order to give the developers more convenient, standardized hardware APIs.
8 ***Algorithm and Platform Independence***: It should be possible to develop each algorithm in isolation, that is, in a separate module, and switch the algorithm being used by selecting some configuration values.
9 ***Scalability***: The framework can be upgradable as its components grow and take full advantage of it. The components should be open for extension and closed for modification (i.e., the components should be easy to extended but without modifying the existing codes).
10 ***Flexible Architecture***: The framework has no restrictions on the architecture of the control software.
11 ***Support for Parallelism***: *RISCWare* can perform a number of processes simultaneously.

12 ***Fault Detection and Recovery Capabilities***: The faults in the robot framework should be detected and localized and also, the robot should be able to complete its mission or at least proceed to a safe mode.

13 ***Asynchronous Communication***: The sender is not required to wait for the message to be handled by the receivers.

4 Architecture of the *RISCWare*

The architecture of the middleware has six primary layers for managing the heterogeneity of the hardware and the software, and creating behaviors that will be used by many applications. As shown in Fig. 2, each layer is cleanly separated. Each layer is cleanly separated from the other layers. Each layer is described in [17].

As shown in Fig. 3, the lower layer of robot middleware is the OS & Hardware Layer (OSHL), which consists of a variety of sensors (such as Laser RangeFinder, Sonar, Infrared sensors, etc.), actuators, as well as the operating system (i.e., UNIX). The hardware drivers used in the second layer, Hardware Abstraction Layer (HAL), are designed to hide the details of communication with the devices, transform the data to match units and conventions used by the rest of the system, and publish a message to an appropriate message channel while having no further knowledge of that message's destination. The hardware driver sends a message by creating the message in the appropriate format (based on the definition of the message format). For example, a message might contain information on sender, receiver, time stamp, and sensory data. Then, it places the message into the communications system. The hardware driver also receives a message from the communications system, and then parses the message into its control information and data.

In the third layer, *Messaging Layer* (ML), message channels provide a very basic form of routing capabilities. Furthermore, a message translator is used to translate the sensor-specific (private) messages to canonical (public) messages, for example, to transform a message generated by a SONAR1 sonar-type message to a generic sonar message.

In the *Actions Abstraction Layer* (AAL), channel adapters are used to send messages to or receive them from actions in the AL, but one instance does not do both. An adapter is channel-specific, so a single action would use multiple adapters to interface with multiple channels. A message adapter takes that command or data, converts it into a message, and sends it on a

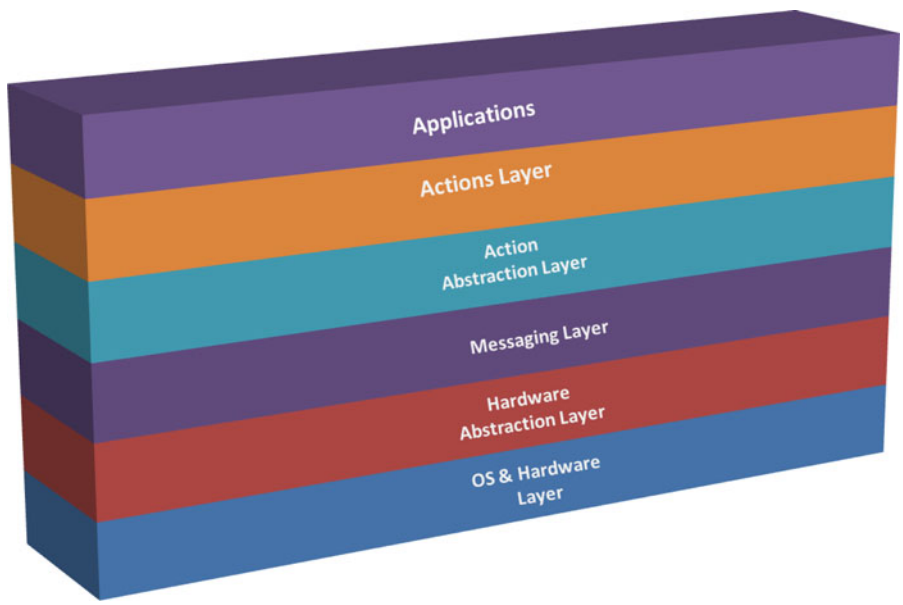

Fig. 2 The architecture of *RISCWare* middleware

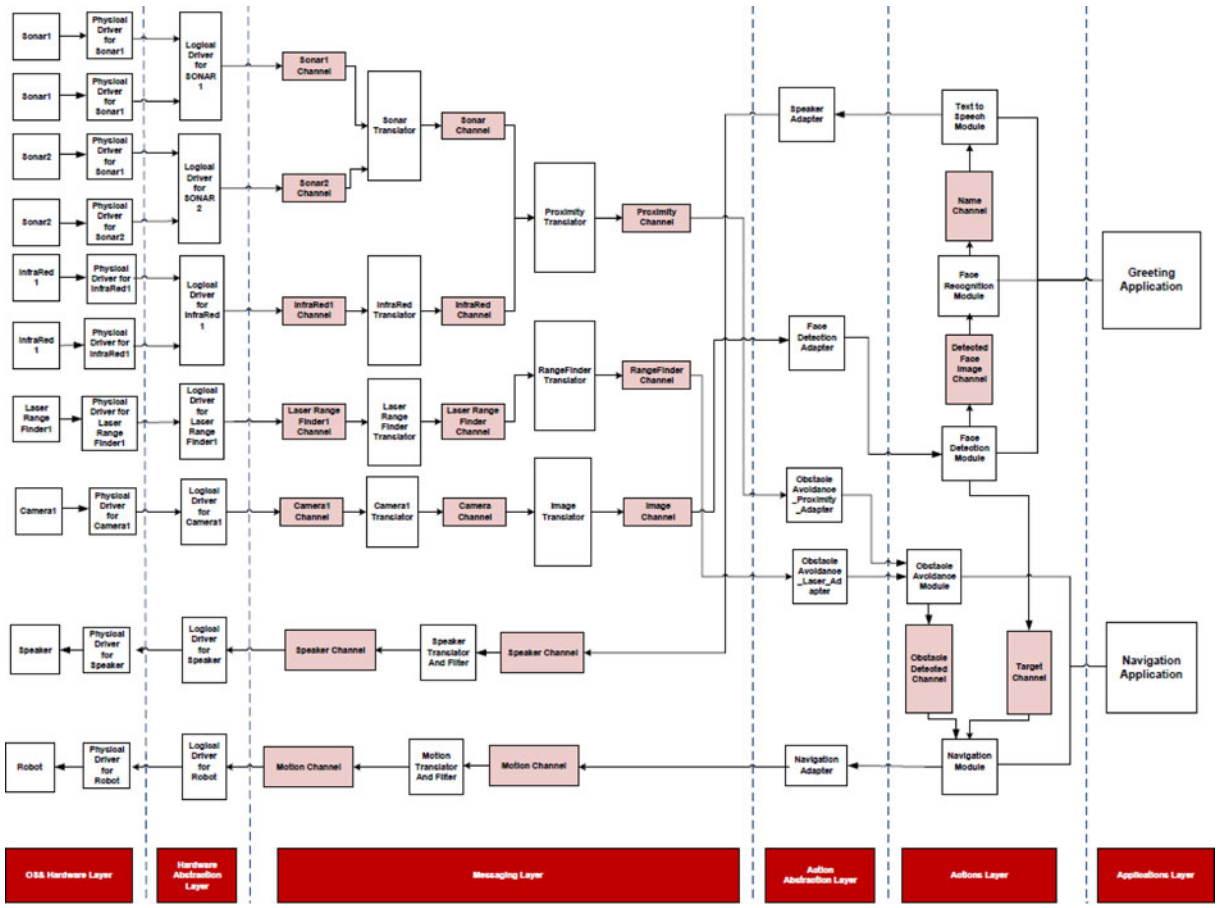

Fig. 3 An overview of the *RISCWare* middleware

particular messaging channel. Alternately, it receives a message, extracts the contents, and gives them to the application in a meaningful way. A channel filter can be used before the channel adapter.

The *Actions Layer* (AL) consists of the software services to perform a number of specific tasks used for sensing, decision-making, and autonomous action, such as obstacle avoidance, navigation, face detection, etc. These software modules communicate together using message channels.

The *Applications Layer* consists of a set of tasks that work together such as SLAM, Navigation, Vision, etc. Each application consists of a set of software modules (implemented in the *AL*). This layer provides the required Application Program Interface (API) to the application layer, such as installing/uninstalling a robot application, starting/stopping it, registering/unregistering a logical device, etc.

The RISCWare is a middleware which consists of two main parts: the core logic, which contains the main functionalities, and the wrapper, which provides the interface of the logic core to the external components. SWIG, which is a free software development tool, is used to connect between the RISCWare core (written in C++) and the other software components (written in other languages such as Python and Perl). SWIG supports different types of target languages, including scripting languages such as Perl, PHP, Python, Tcl and Ruby, and non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI),

D, Go language, Java, Lua, Modula-3, OCAML, Octave, and R [18].

5 Case Studies

5.1 Plug in a New Sensor

New modules (software and hardware) are easily added to *RISCWare* without any change to the existing software because of *RISCWare*'s *loosely coupled* feature. For example, in a practical robot, several proximity sensors may exist, but the software applications do not know and do not need to know which proximity sensor (such as sonar and infrared) is being used at any specific time. If a new proximity sensor is installed, its corresponding logical driver can be added to the framework without any change to the rest of the system.

5.1.1 Install a Predefined Sensor

The user only needs to specify the location and orientation of the sensor in the robot, the sensor name, etc., in its hardware configuration XML file. For example, when inserting a new sensor of type SONAR1 to the system, as shown in Fig. 4, *RISCWare* will detect this sensor and automatically publish its messages to the SONAR1 channel. The obstacle avoidance module, for example, will not be affected (there is no need to restart or stop any application) and it will take full advantage of it.

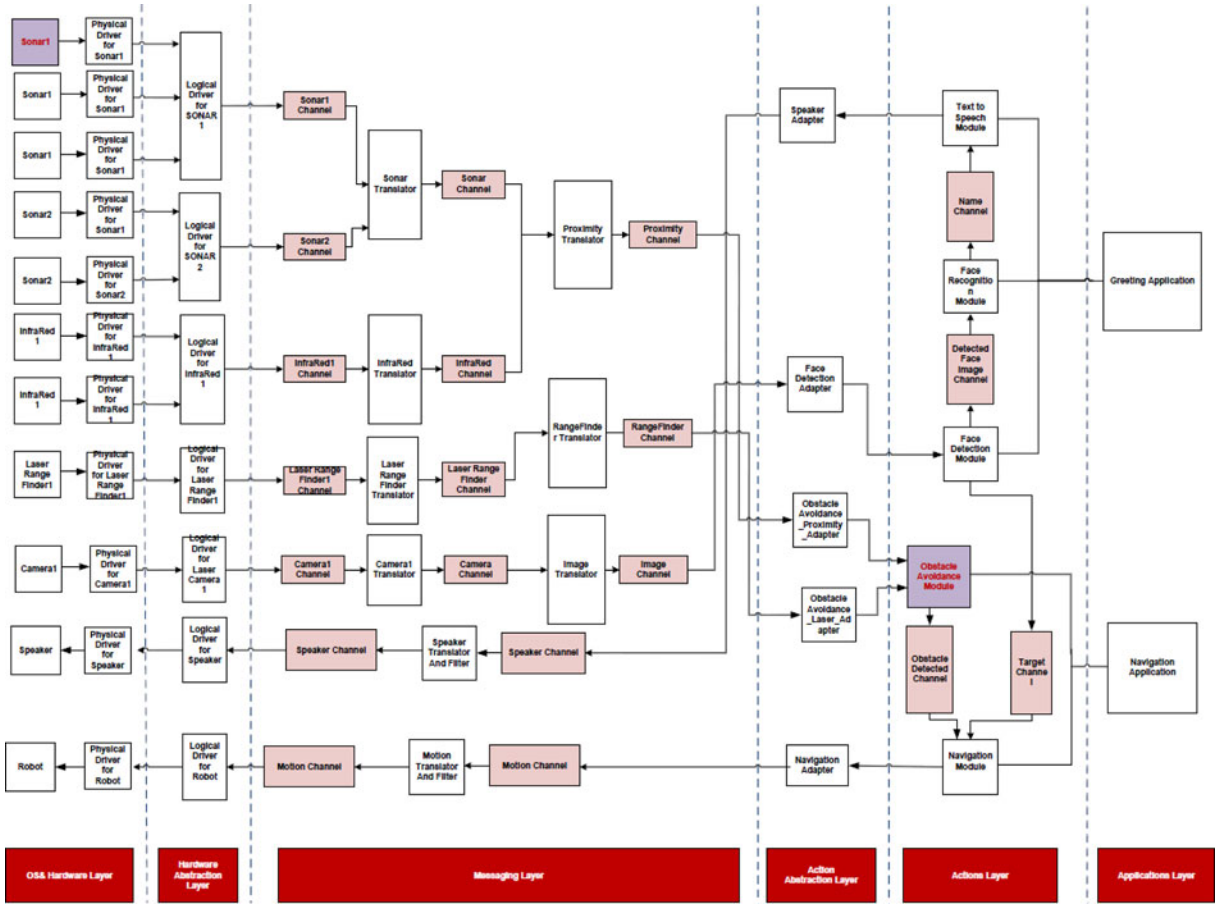

Fig. 4 Installing a SONAR1 sensor

5.1.2 Install a Similar Sensor

For example, before installing a SONAR3 sensor, using the Driver Writing Tool and the SONAR template, the non-expert developer can develop a logical driver for SONAR3. Then this process will exactly follow the installation of a predefined sensor. This feature represents the *Interchangeability* property of *RISCWare*.

5.1.3 Install a Completely New Sensor

When installing a completely new kind of sensor, such as an IMU sensor, an expert developer should write the logical driver for this sensor using the Driver Writing Tool with the basic hardware device template. This feature represents the *Extensibility* property of *RISCWare*. Furthermore, using the logical hardware driver, a sensor can be used in different tasks, such as the stereo camera, can be used to produce 3D range points and also raw images, depending on its logical driver.

5.2 Unplug or Stop a Module/Module Failure

If a failure occurs in a module (software or hardware), or a user decides to stop, then the logical hardware driver, in the case of hardware (or the channel adapter in case of software), will detect the failure or receive the stop command and then publish an error message to the *System Manager*, indicating that this module is stopped. The *System Manager* will send an event message to all interested applications. Then each application determines whether this sensor is critical; if so, this application will be stopped or if it is not critical, then it can continue. If the application stops, it will broadcast an event message through its event port, while notifying all of the applications that it has stopped.

The *System Manager* monitors the flow of data, manages the flow of messages through the system, makes sure that all applications and components are available, tracks the quality of service (e.g. response times) of an external service, and reports error conditions. The System Manager hides the low level details such as the thread, resources and state management. The System Manager plays an important role for loading, opening, closing, and disposing of upon completion of all modules. Furthermore, it is used to manage the lifecycle and states of components, and connect between them.

5.3 Change the Robot

The whole *RISCWare* framework can be ported into a different robot; the developer just needs to change its parameters through the robot configuration file and its logical driver. A robot configuration file is an XML file containing the list of modules to be loaded, a description of their interconnections (property links) and the values of their properties.

6 RISCWare Components

A *RISCWare*-component is the basic functional module of the *RISCWare* framework, such as: a motor module, a sensor module, and an image processing module. Each component has an XML configuration file to customize the behavior of components and a profile file to describe the module name, version, authors, description, static, maximum number of instances, and the programming language used to write this component. The component name, which is registered in the Namespace, is automatically generated by combining the module name with the instance number. For the SONAR module, the names of the instated components are SONAR1, SONAR2, etc. The Namespace provides a map between a logical name of sensor (used by the developer such as SONAR1) and the physical sensor.

The module's version is recommend to be in the following form: (Major version).(Minor Version).(Patch Number).(Build number) such as (0001.0100.1120.00165). The static component (Singleton) is instantiated only once and does not be destroyed. This type of component is generally used with binding to the hardware devices.

Each component should—upon system initialization—announce its presence, register its services, etc., in a coherent way. Furthermore, the component has a PnP capability, which requires that components can be added and removed during system operation (at runtime) without system reboot. Furthermore, the component

should be reliable in the face of hardware faults, network issues, and software errors and exceptions. The failure of a component should not affect the other components running on the system.

The architecture of the *RISCWare* component is shown in Fig. 5. A *RISCWare* component should be a technology transparent component (different technologies such as programming languages are hidden from the user) and a concurrency transparent component (user should be unware that the components are shared by others).

The *RISCWare* component consists of the following objects:

1. ***Core***: The main process unit. It executes commands, events and software scripts in the component.
2. ***Input/Output data stream ports***: Based on the publisher/subscriber model, the data port is used for exchanging data. An output port (publisher) sends data to all registered subscribers (input ports) through message channels. A data message consists of two basic parts: the header (which describes the data being transmitted, its origin, its data type, and so on) and the body (data).
3. ***Properties***: The module's properties are configurable at or before runtime via its component profile (XML descriptor).
4. ***Finite State Machine (FSM)***: As shown in Fig. 6, the *RISCWare* component has eight states: Initializing, Active, Error, Starting, Running, Aborting, Stopping and Exit. Figure 6 shows the state transition diagram of *RISCWare*'s FSM. The FSM of the component is managed by the Component Manager. The manager keeps track of the component's state and binds it to the Directory, which tracks all the loaded components. The Initializing State is to create and initialize the *RISCWare* component by reading the XML configuration's file. The Active State means the component is ready to enter the Running State. The Starting State is a transient state before being run. The Running State is the main state to process the component's task. The Stopping State is a transient state between the Running and Active States. The component is in a Aborting

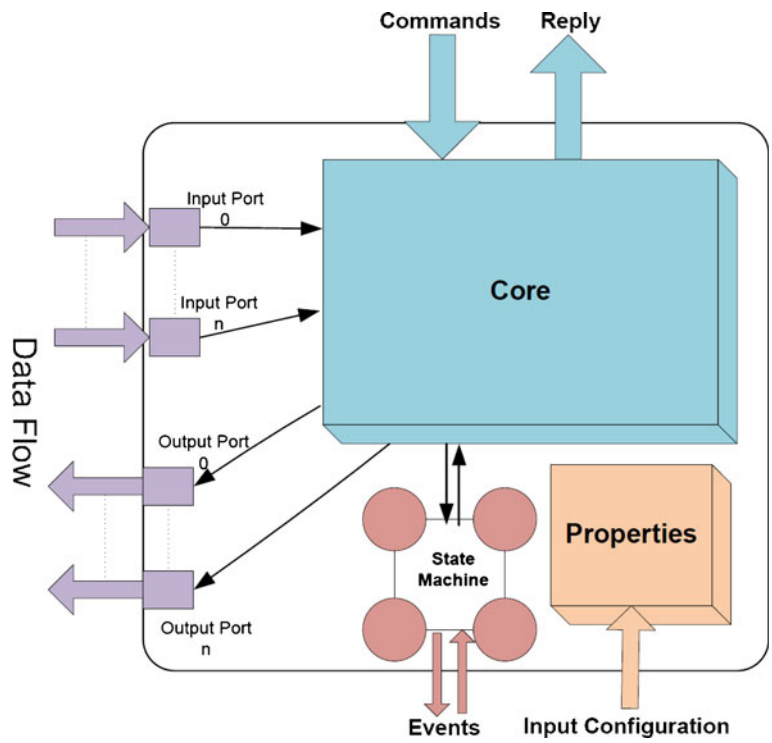

Fig. 5 Architecture of the *RISCWare*-component

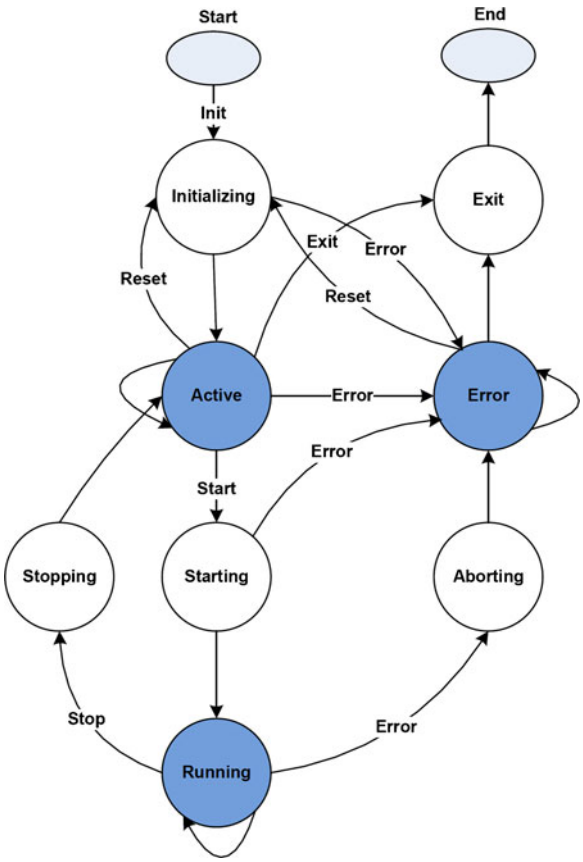

Fig. 6 FSM of the *RISCWare* component

State if an error happens in the Running State but the component is in Error State if any error happens in any state. To resume after an error, a reset command should be executed. Exist State is used to finalize and deallocate the resources used by this component.

5. ***Commands/Reply port***: This port is used to receive the "Command Messages" from the *RISCWare*. The Command message is used to invoke a procedure in another application. When the component receives a command, it should be processed immediately (not like the data in the data port). The command/reply ports transmit execution flow information in a synchronous mode.
6. ***Event port***: The Event message is used to notify another application of a change in this application. This port is used to receive "Event Messages" to update the FSM of the component by changing the current state to the next state of the FSM, based on the received event. Furthermore, once an event is detected, the automatic notification service will alert the interested component that the event has occurred. Finally, this port is used to send "Event Messages" when the FSM of the component is updated.

6.1 Logical Hardware Driver

The logical hardware driver is a high level interface, different from the physical device interface, that represents the entities that the developers like to work with, such as sonar and laser rangefinders. The logical hardware driver is an abstraction of the same or similar sensors. The feature of this abstraction is to replace or upgrade a component at run time without affecting the other parts of the system. Often, the physical device interface, usually written in C++, is device dependent (not portable), but the logical device interface is device independent and portable between the similar devices. For example, the logical sonar driver will return the distance (in cm or inches) between the sensor and the detected object, while the positional encoder will return its position in radians.

Figure 7 shows the hardware hierarchy of *RISCWare*. There are four different types of components: sensors (such as sonar), actuators (to control external equipments such as steering angle and brakes), robot arms, and robot vehicles. For example, using a Proximity Driver, it is possible to exchange the hardware with all supported proximity devices. In this way, sensors can be replaced by similar yet not identical sensors (i.e. a sensor of one type can be replaced by another one). This is one of *RISCWare*'s features. For example, a satellite GPS can be replaced with an odometry sensor if the satellite signals are occluded. A logical driver has parameters describing their function stored in the *Hardware Configuration File*. For a sensor, this may include sampling rate,

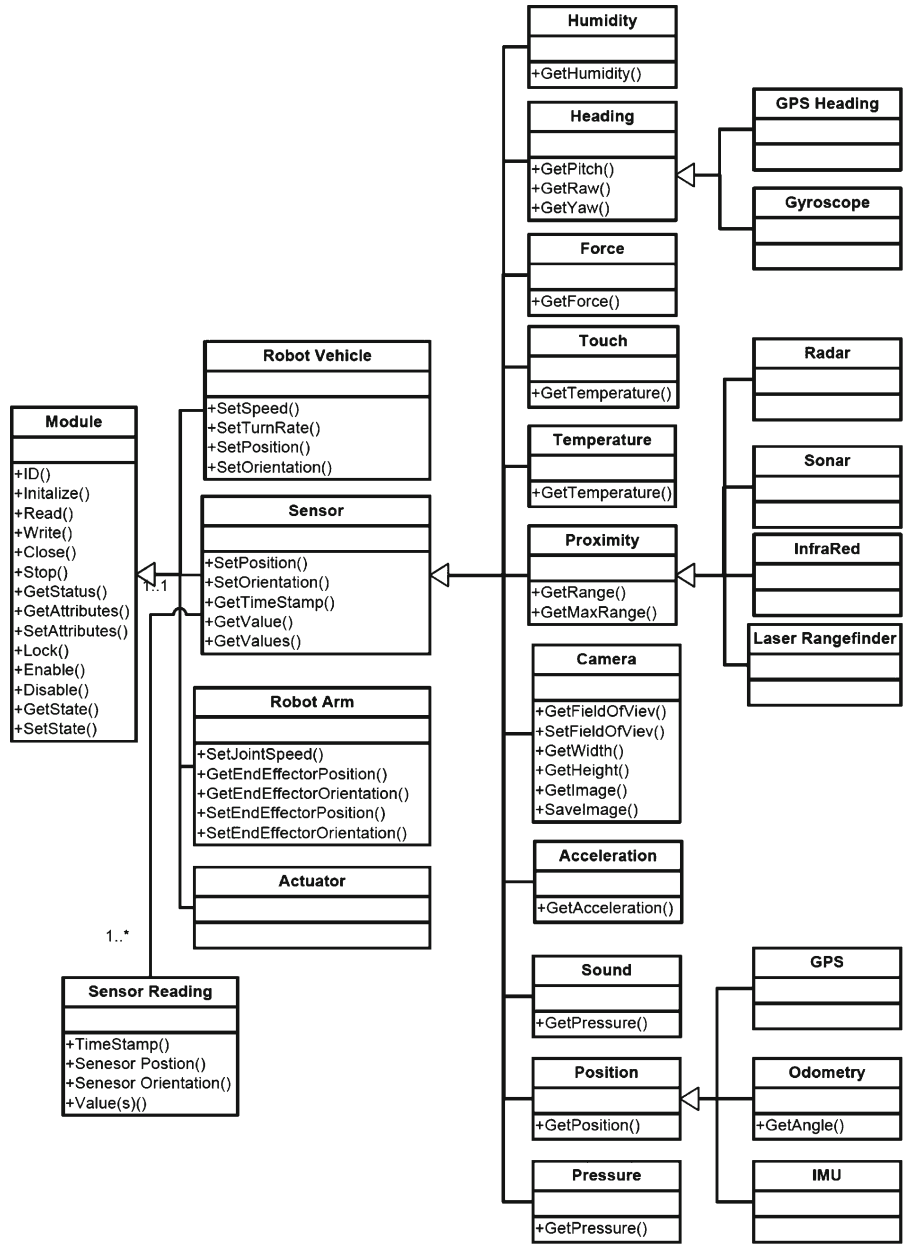

Fig. 7 *RISCWare* hardware hierarchy

network address, position and orientation. Furthermore, different robots may use different types and configurations of sensors for producing identical or similar information. For instance, both a 3D laser scanner and a stereo camera can be used to produce 3D point clouds and obstacle avoidance can be implemented as camera-based or sonar-based. By using an appropriate, multi-level device abstraction hierarchy, these sensor devices could be made exchangeable. As shown in Fig. 7, the proximity sensor class is the general abstraction of all proximity sensor devices, such as sonar and infrared. The sonar class represents the sonar and provides an interface to ultrasonic sensors such as $LV-MaxSonar^{®}-EZ0^{TM}$ and $EZ1^{TM}$. The infrared class represents the infrared sensors such as *Sharp GP20A21YK*. The robot consists of actuators and sensors and contains fields for name,

vendor and model. The actuator includes properties such as vendor, model and driver.

Each hardware driver should declare its API Interface, which is a set of specifications and instructions (syntax and semantics of all messages that can be exchanged) to allow interaction with the sensory devices, actuation platforms and software algorithms. For example, the Hokuyo UHG-08LX Scanning Laser Rangefinder interface defines the format of range readings. This physical driver communicates with the Hokuyo UHG-08LX Scanning Laser Rangefinder over USB 2.0 and retrieves range data from it. Then the logical driver translates the retrieved data to make it conform with the data structure defined in the interface. In this case, because the logical drivers for all the laser range finders use the same interface, the software application does not depend on a certain vendor-type of laser range finder.

6.2 Messaging Bridge

A messaging bridge is used to connect between robot agents that run the *RISCWare* system by replicating messages between systems. "The messaging bridge is a set of channel adapters, and each pair of adapters connects a pair of corresponding channels. The bridge acts as a map from one set of channels to another, and also transforms the message format of one robot to another" [19]. A separate namespace is defined for each robot to give the robots the ability to address each other. A naming service maps names to addresses. The namespace used in *RISCWare* is modeled as a hierarchical namespace, presented as a directed graph, as shown in Fig. 8. The namespace service is used to support multirobot control.

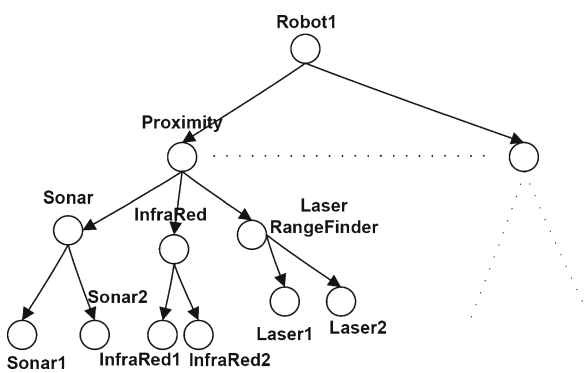

Fig. 8 A hierarchical namespace used in *RISCWare*

7 Experiment and Results

The robot application, "greeting a person", is implemented to evaluate the *RISCWare* framework, with respect to applicability. In Fig. 3, a robot application of "greeting a person" is composed of Face Detection, Face Recognition and Text to Speech Modules. The face detection module processes images. When it detects the face of a human, the robot approaches the person, detects collisions with obstacles, and sends the image to the face recognition module in an attempt to recognize the person. If this person is recognized, his name will be sent to the Text to Speech Modules. Messages from both the Face Detection and the Obstacle Avoided modules are fused by the navigation module to control the movement of the robot. The obstacle avoidance task is used to have the robot avoid the moving and stationary obstacles, and it uses the proximity sensors (such as the sonar and infra red) and the range finder. Usually, it is used with the other tasks such as Navigation. Both the messages from the Face Detection and Obstacle Avoidance modules are fused by the navigation module to control the movement of the robot. Microsft Xbox Kinect is used to capture images at a rate of 30 fps. Raw image data is transformed into the OpenCV Image format and sent to the Face Detection module, using message channels. The algorithms used for Face Detection and Recognition modules are described in detail in [20].

Face Detection and Recognition modules must be tested simultaneously with the input of the Detection Program fed into the Recognition Program. The Face Detection Program on an average detects up to five faces in real time (30 frames/second), running on a Dual Core Intel Processor, therefore bringing the total to 150 images/second. Figure 9 shows a sample output from the Detection Program. The Recognition Module, on the other hand, can take each of the detected faces and search the database to find possible matches. The initialization process of the Face

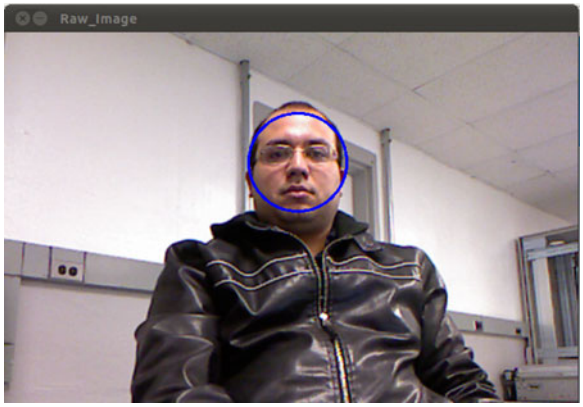

Fig. 9 Sample output of the face detection module

Recognition database is found to be a processor hog, hence plans to recompute the database values at run time had to be abandoned. Another bottleneck is the total memory requirement for the database, which increases due to storing the feature vectors in uncompressed formats in system memory.

In order to evaluate the performance of *RISCWare*, a series of stress tests have been performed, testing different message sizes (16, 32, 64, 128, 256, and 512 Bytes); the system was run for about an hour and measured the end-to-end data packet latency. The specifications of the used laptop are as follows:

- Processor: 2nd generation Intel(R) Dual Core(TM) i7-2620M (2.7 GHz, 4 MB L3 Cache), with Turbo Boost up to 3.4 GHz
- Graphics: Intel(R) HD Graphics 3000
- RAM: 8 GB DDR3

Figures 10, 11, 12, 13, 14, 15 and 16 show that the latency of different message sizes (16, 32, 64, 128, 256, and 512 Bytes) is about 4.59 micro seconds on average.

8 Conclusions

Autonomous robots are complex systems which require the interaction between numerous heterogeneous components (software and hardware). Because of the increase in complexity of robotic applications and also the diverse range of hardware, *RISCWare* is designed to manage the complexity and heterogeneity of hardware and applications, promote the integration of new technologies, simplify software design, hide the complexity of low-level communication and the heterogeneity of components, improve software quality, reuse of robotic software infrastructure across multiple research efforts, and reduce production costs.

In this paper, the *RISCWare* framework is proposed as a robotic middleware for the modular design of sensory modules, actuation platforms, and task descriptions. *RISCWare* consists of three modules. The first module encapsulates the sensors that gather information about the remote or local environment. The second module defines the platforms, manipulators and actuation methods.

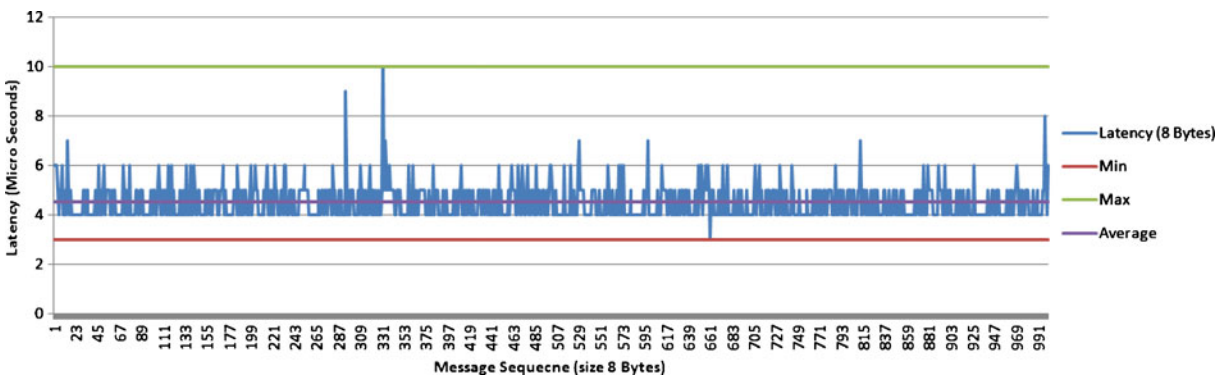

Fig. 10 Latency of the *RISCWare* tested using a message size of 8 bytes

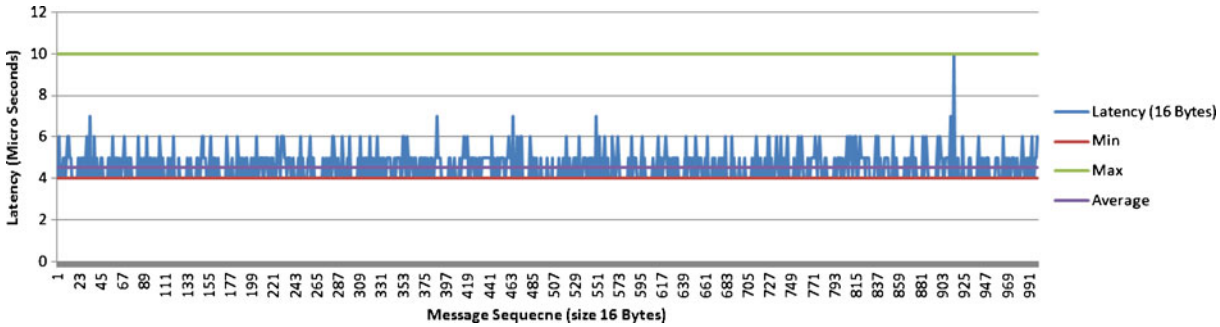

Fig. 11 Latency of the *RISCWare* tested using a message size of 16 bytes

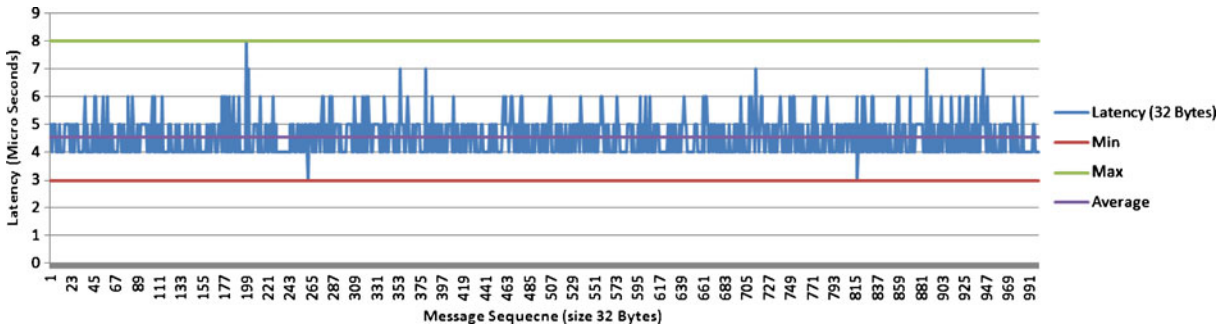

Fig. 12 Latency of the *RISCWare* tested using a message size of 32 bytes

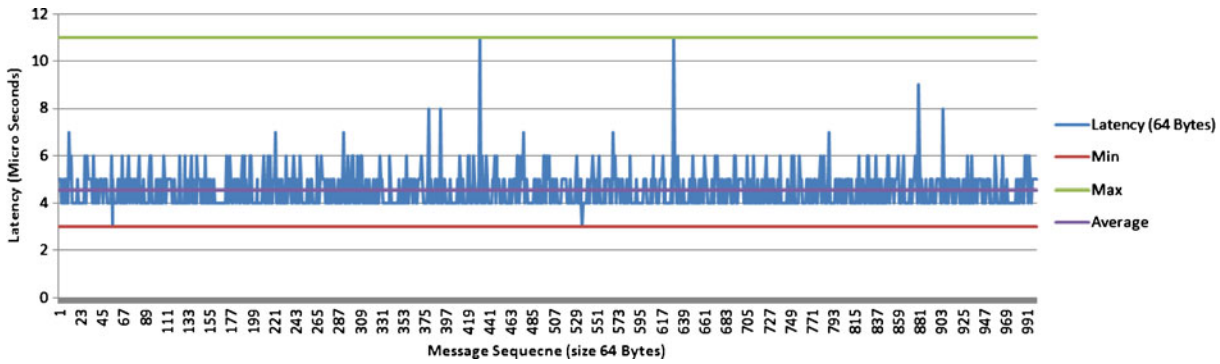

Fig. 13 Latency of the *RISCWare* tested using message size of 64 bytes

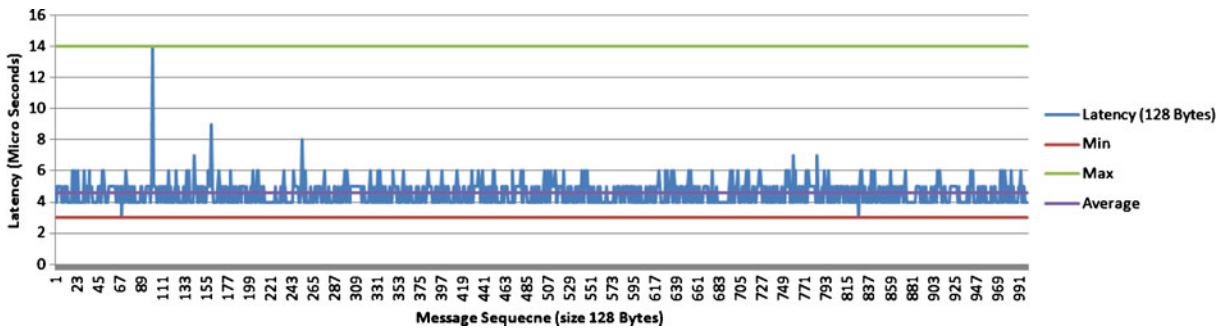

Fig. 14 Latency of the *RISCWare* tested using a message size of 128 bytes

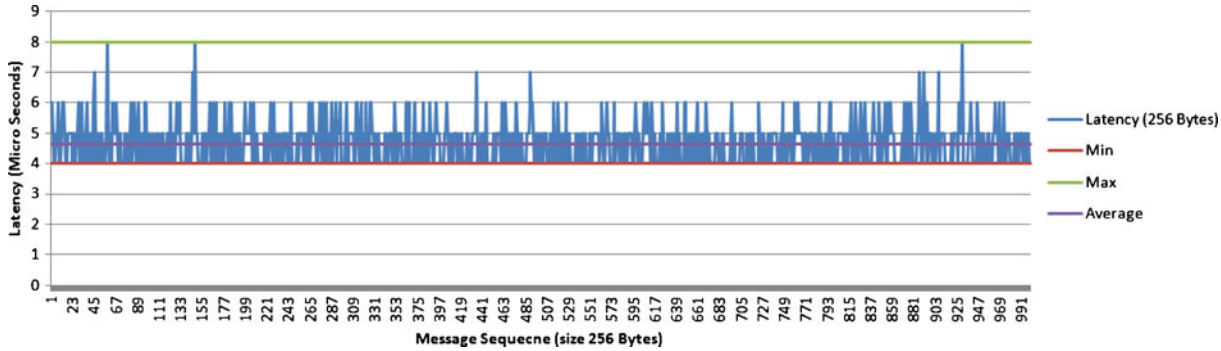

Fig. 15 Latency of the *RISCWare* tested using a message size of 256 bytes

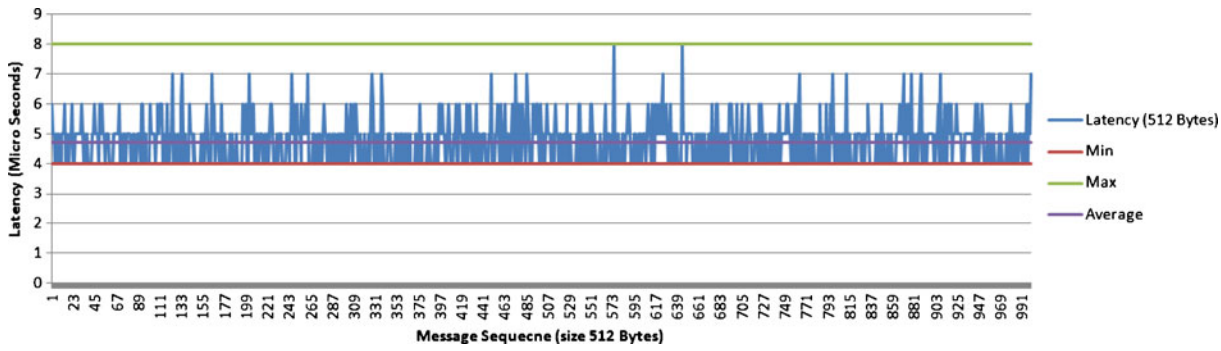

Fig. 16 Latency of the *RISCWare* tested using a message size of 512 bytes

The last module describes the tasks that the robotic platforms will perform, such as: navigation, obstacle avoidance and map building. Finally, some experiments, performed on the *RISCbot II* mobile robot, are described to evaluate the *RISCWare* middleware.

RISCWare framework provides Plug and Play, and Dynamic Wiring: The software and hardware components can be downloaded, installed and configured at or before run-time.

The main advantage of the RISCWare is that it is a modular framework whose hardware and software work together to automatically use available sensing devices and assign resources to perform the required task. Furthermore, RISCWare allows the reconfiguration of the sensors to be used in different tasks such as the stereo camera, which can be used to produce 3D range points and also raw images. A demo of the behavior of the *RISCWare* is uploaded to Youtube at the following link: http://www.youtube.com/watch?v=8uYlMB1eMwc.

References

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
2. Nesnas, I.: The claraty project: coping with hardware and software heterogeneity. In: Brugali, D. (ed.) Software Engineering for Experimental Robotics. ser. Springer Tracts in Advanced Robotics, vol. 30, ch. 3, pp. 31–70. Springer, Berlin, Heidelberg (2007)
3. Collett, T.H., MacDonald, B.A., Gerkey, B.P.: Player 2.0: toward a practical robot programming framework. In: Proc. of the Australasian Conf. on Robotics and Automation (ACRA). Sydney, Australia (2005)

4. Utz, H., Sablatnog, S., Enderle, S., Kraetzschmar, G.: Miro-middleware for mobile robot applications. IEEE Trans. Robot. Autom. **18**(4), 493–497 (2002)
5. Michel, O.: Webots: professional mobile robot simulation. J. Adv. Robot. Syst. **1**, 39–42 (2004)
6. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.-K.: RT-component object model in RT-middleware–distributed component middleware for RT (Robot Technology). In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005, (IROS 2005), vol. 2–6, pp. 3933–3938 (2005)
7. Schlegel, C., Hassler, T., Lotz, A., Steck, A.: Robotic software systems: from code-driven to model-driven designs. In: International Conference on Advanced Robotics, 2009, ICAR 2009, vol. 22–26, pp. 1–8 (2009)
8. ERSP 3.1 software development kit: Online: http://www.evolution.com/products/ersp/ (2010)
9. Alexei Makarenko, A.B., Kaupp, T.: On the benefits of making robotic software frameworks thin. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07), 29 Oct.–02 Nov. 2007. San Diego CA, USA (2007)
10. Jang, C., Lee, S.-I., Jung, S.-W., Song, B., Kim, R., Kim, S., Lee, C.-H.: OPROS: a new component-based robot software platform. ETRI J. **32**, 646–656 (2010)
11. Jackson, J.: Microsoft robotics studio: a technical introduction. IEEE Robot. Autom. Mag. **14**(4), 82–87 (2007)
12. Kramer, J., Scheutz, M.: Development environments for autonomous mobile robots: a survey. Autonomous Robots **22**(2), 101–132 (2007)
13. Mohamed, N., Al-Jaroodi, J., Jawhar, I.: Middleware for robotics: a survey. In: 2008 IEEE Conference on Robotics, Automation and Mechatronics, 21–24 Sept., pp. 736–742 (2008)
14. Mohamed, N., Al-Jaroodi, J., Jawhar, I.: A review of middleware for networked robots. Intl. Journal of Computer Science and Network Security **9**(5), 139–148 (2009)
15. Namoshe, M., Tlale, N., Kumile, C., Bright, G.: Open middleware for robotics. In: 15th International Conference on Mechatronics and Machine Vision in Practice, 2008. M2VIP 2008, 2–4 Dec., 189–194 (2008)
16. Elkady, A., Sobh, T.: Robotics middleware: a comprehensive literature survey and attribute-based bibliography. J. Robot. **2012**, 15 (2012). doi:10.1155/2012/959013
17. Elkady, A., Joy, J., Sobh, T.: A plug and play middleware for sensory modules, actuation platforms and task descriptions in robotic manipulation platforms. In: Submitted to Proc. 2011 ASME International Design Engineering Technical Conf. and Computers and Information in Engineering Conf. (IDETC/CIE '11) (2011)
18. Swig: Website. http://www.swig.org/ (2011)
19. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2003)
20. Elkady, A., Babariya, V., Joy, J., Sobh, T.: Modular design and implementation for a sensory-driven mobile manipulation framework. J. Intell. Robot. Syst. 1–27 (2010). doi:10.1007/s10846-010-9454-3