Design of an API for Integrating Robotic Software Frameworks

Min Ho Lee[1], Ho Seok Ahn[2], Bruce A. MacDonald[3]
Department of Electrical and Computer Engineering, CARES, University of Auckland
Auckland, New Zealand
[1]mlee242@aucklanduni.ac.nz, [2, 3]{hs.ahn, b.macdonald}@auckland.ac.nz

Abstract

While there are a number of good software frameworks for robotics applications, these changes over time and it is not so easy to create applications that use multiple frameworks. In this paper, we present the design of an Application Programming Interface (API) for our UoA Robotic Software Framework, a heterogeneous robot development framework that allows interoperation between existing component-based frameworks such as ROS, ROCOS, and OpenRTM, and also reduces the impact of changing frameworks on development of robotic applications. In the development of this API, four key design concepts are: interoperability, compatibility, support for heterogeneous applications, and dynamic monitoring and control. The API can be used for developing different applications with different frameworks, using the same robot hardware platforms and components, with minimal development of additional procedures. To evaluate the effectiveness of our framework, we developed two case studies, which are both healthcare applications with different programming languages and frameworks, and then applied them to two robot platforms with the same specifications.

1 Introduction

Development of robot systems has always been a complex challenge because integration and development of complex components requires significant time and effort [Ahn *et al.*, 2008]. However with requirements for robotic systems growing more complex, researchers have developed component-based software frameworks [Herman *et al.*, 2013] such as ROS [Quigley *et al.*, 2009], Open-RTM [Ando *et al.*, 2008], OPROS [Jang *et al.*, 2010], and ROCOS [Jayawardena *et al.*, 2012]. These component-based frameworks provide libraries and distributed communications abilities, enhance the opportunities for developers to reuse code and components, and provide help for development, integration and deployment of robotic software systems.

Many component-based frameworks are open source frameworks, so researchers develop intelligent components, such as image recognition, speech recognition, and navigation, and share them with other researchers. But there are several limitations. Some operate on specific operating systems, for example, ROS runs mainly in a Linux environment, and ROCOS mainly in a Windows environment, while others, such as Open-RTM, OPROS, and Orocos are supported by both Windows and Linux. ROS has a Windows version WinROS, but its packages are different from the ones in ROS. Overall the frameworks are dependent on a specific development environment, and researchers and developers must re-write intelligent components and applications in different ways to transfer them from one framework to another. This can be difficult and time-consuming. In addition when a new version of a framework is created, or a framework becomes obsolete and another new one becomes popular, applications must be updated.

Recently, there have been some efforts to allow interoperation between two different frameworks, such as ROS-OpenRTM [Biggs *et al.*, 2010] and ROS-OPROS [Jang *et al.*, 2012] Most of implementations were done by translating the messages and protocol through some kind of bridging software to channel between the two frameworks. Although they allow researchers to use components of two different frameworks at the same time, they only provide one-to-one interaction of the specified frameworks, and hence as frameworks are developed there will be a need for more bridges to be created. Therefore we should develop these kinds of frameworks repeatedly when we need to combine different combinations for our robots.

Researchers and developers are faced with a few key different frameworks to choose from and the prospect of future changes as one framework becomes obsolete and others become popular. Rather than creating a new framework for our needs, we designed the UoA robotic software framework, which gives developers access to multiple existing software frameworks, including components, packages, libraries and distributed services from multiple combinations of existing frameworks. We can relatively easily add new frameworks including those developed in the future, as well as delete old frameworks. Our framework also supports the use of different versions of the same framework; it isolates the components in one version from those in another. For using various components, we designed a robot manager, which manages the connections between applications and

components of frameworks. We designed the API of the robot manager for using various components of frameworks, so researchers can develop various applications without modifying those components. In this paper, we focus on the API design of the robot manager.

This paper is organised as follows. We describe concepts of our robotic software framework in Section 2, and introduce the design of our robotic software framework in Section 3. We present an experimental robot platform HealthBot and two case studies in Section 4. Finally, we conclude this paper in Section 5.

2 Concept of UoA Robotic SW Framework

2.1 Limitations

Like many other researchers, in our research laboratory we have many robots, with several robotics frameworks, which are used in various robot applications. Like others we seek to reuse software for efficiency, and to integrate applications with different robots working together. They are operated very well, but there are several limitations. Firstly, code used for any framework is often restricted only to the specific framework the code was implemented in. To migrate a component from one framework to another framework, the component must be rewritten to the API of the new framework. This is especially a problem as there are risks of frameworks being discontinued, rendering the old code useless until a solution for compatibility is implemented.

Some frameworks such as ROS have software restrictions on applications such as the limited language support and platform restrictions, and all have a defined API. Hence, in order to migrate a component from one framework to another, the component needs to be modified to be compatible with these requirements such as the operating system, in some cases creating large efforts for the developers.

Some components are only designed for use within a single framework, such as sensor drivers. In order to use an implementation of a component using a different framework, new software has to be developed to create functionality for the target framework. In some cases efforts are made to make drivers and other key components portable across frameworks, such as Gearbox [Makarenko *et al.*, 2007], which argues for a thin framework with refactored code components at the lower levels. In this paper, we take a different approach, providing integration at a higher level.

Usually for each different kind of robot, there is one robotic framework used to develop software. To complicate the development process, there is a large learning curve associated with each robot framework. Hence, in order to migrate an application to another robot system, the developer may be required to learn both of the frameworks. Overall, a considerable reimplementation effort may be required to port an old application suite to a new framework, during which time the operational ability may be significantly reduced.

2.2 Key Concepts

The main focus of our design is to create a robotic software framework that allows interoperation among various existing component-based frameworks, and is extensible to future frameworks with a minimum of effort. Thus we can accommodate new frameworks in our development roadmap with a minimum of effort at any one time. An important part of our design is to design our API to control robots' hardware and use components, packages, and libraries available in any framework. In addition, it is also important to define methods in the robot manager's API to dynamically identify, classify and monitor the connected robot frameworks that are available for requests from applications. For this, our design includes the following functions.

1) Interoperability: for any robot the developer of an application should be allowed to choose the components and packages from any of the frameworks available. If there is a service that requires use of multiple components across different frameworks, the robot manager should be able to determine whether or not these components are compatible with each other.

2) Compatibility: use of the UoA framework should not be restricted to a specific platform or environment but rather should function with frameworks in various platforms or environments.

3) Support for heterogeneous applications: the UoA framework should be accessible by applications from heterogeneous environments. The communication protocol should be easily implemented in heterogeneous applications, which can be executed in various environments and use different frameworks.

4) Dynamic monitoring and control: the API should provide a method for controlling and managing the status of each data flow connection.

3 Design of UoA Robotic SW Framework

We designed the UoA robotic software framework based on the four key concepts. It allows researchers to develop different applications with different frameworks by using the API of the robot manager, which manages the connections to several frameworks.

3.1 Overall Architecture

The UoA robotic software framework consists of three layers as shown in Fig. 1: an application layer, a robot manager layer, and a component layer. We consulted Bruyninckx's principle of separation of concern in robotics design [Herman *et al.*, 2013], which separates the functionalities in levels of coordination, configuration, computation and communication coupled into composition of the features. The application layer comprises the user applications and communication interface to define the application logic of the developers by sending and receiving data from the components. The application is able to interact with the robot manager layer using the predefined API, using a combination of WebSocket connections and JSON messages through the communication interface. The communication interface has the role of channeling between the user application and the robot software framework, and any type of platform can be used as the application layer. Therefore, the client application has an interface defining the input and output methods as well as communication functions for sending the requests from interactions with users to

the robot manager and obtaining the results. The main communication between the client and the robot manager must work consistently across multiple platforms, so we use web sockets [Lubbers *et al.*, 2010], which provide full duplex communication over a single TCP connection and allow the client to be independent of any platform, making it scalable. Then the application makes a request call for services using the API via the robot manager layer.

The robot manager layer is responsible for monitoring, mapping and channeling the communication among the components of different frameworks. The robot manager consists of two modules: a robot manager core that provides status monitoring, and a configuration file that includes priority and dependency of the API. The component layer includes various robotic frameworks, such as ROS and OpenRTM, and their components that are connected to the robot manager. The components are able to provide interaction with the robot manager layer through bridges, which provide translations of the internal communication of the framework to a generic format and vice versa, for intercommunication between the frameworks.

3.2 Robot Manager

There are three main roles of the robot manager:

1) managing the status of each connected framework;
2) mapping the higher level API to the framework functions;
3) transferring the data from framework to framework.

To perform these roles, the robot manager uses two modules: a robot manager core and a configuration file. The configuration file is responsible for mapping the API. It contains a table of the available robotic components for each robotic framework, which specifies priorities, required input arguments and outputs to each component which are mapped to the high level API to the clients. Hence, when the robot manager layer receives a request from the application layer to use a certain function, it will check the configuration table to see if that call is valid and translate the higher level API calls to the component specific API.

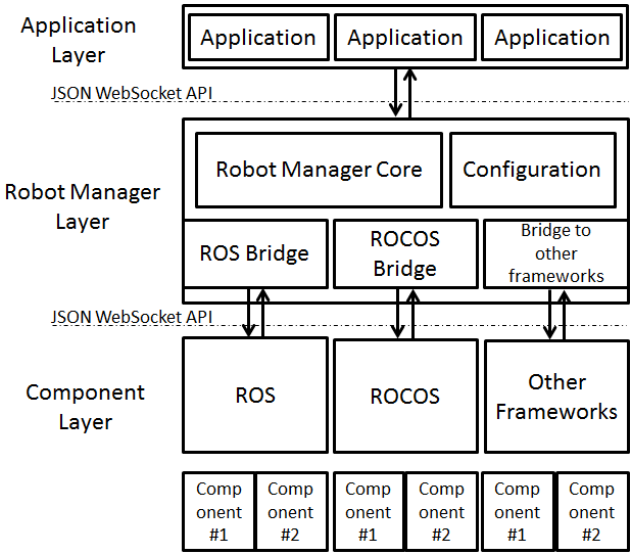

Fig. 1. Architecture of the UoA robotic software framework, which consists of three layers: application, robot manager, and component.

Fig. 2 shows the procedure for API calls based on the configuration file. When the robot manager recevics a call from the application to call a *faceRecognition* function, the robot manager searches for the details of the face recognition function from the configuration file. Then it selects the component that has the highest priority, and checks dependencies. If the selected component has the required dependency, it generates the list of dependencies required to start the *faceRecognition* function. This is then passed to the robot manager core to be validated regarding whether the operation will work.

The robot manager core, which is responsible for managing the status of each connection, sets up the communication link and forwards the request from the application to the appropriate frameworks. It also registers the connections with the list of the current running maps. The list is used to manage the various types of connections by keeping the track of the run-time status of each bridge and the data flow with the frameworks that have been registered. The messaging format in the robot manager layer consists of JSON format as a generic form of communication between the frameworks. In order for the components to operate, the robot manager must share common data types, which are message types communicating among components [Arndt *et al.*, 2013]. By JSON format, the robot manager can transfer the message among frameworks and allow the message to be transformed into the message class of the native framework in a light-weight communication. As previously mentioned, we validate the interoperability after generating the list by comparing the message structure that is transformed in JSON format. The bridge is responsible for translating the messages of the individual format into generic JSON format.

3.3 API Design

In order to design the API of our robot manager, we analysed API structures and protocols of existing

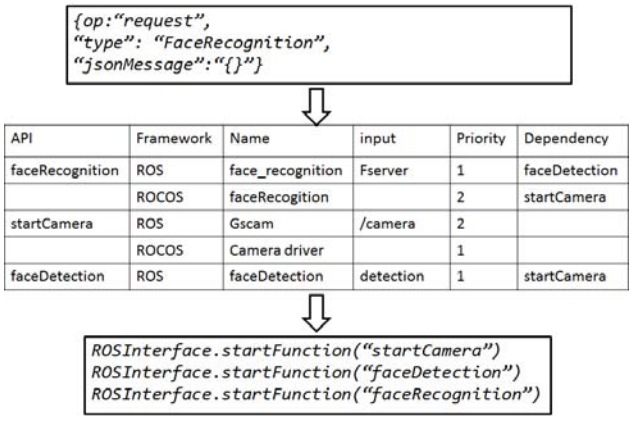

Fig. 2. API calling procedure based on the configuration file in the robot manager.

component-based frameworks. In particular, we chose three different types of existing robotic framework for analysis: ROS, ROCOS, and OpenRTM, which are the most used robotics frameworks. We analysed the required elements to establish a data port and the types of messages required for input and output. We have chosen these three types of middleware since each has its own representative model of communication; ROS using a channel model, OpenRTM using CORBA and ROCOS using plain messaging.

3.3.1 ROS

ROS is an open-source framework developed by Willow Garage. Each ROS component known as a "node" can communicate with other nodes by establishing a communication channel. Each of ROS's communications channels comprises two parts: "topic" and message types. The topic is a string that defines the name of the communication channel which is managed in the ROS master node, a nameserver containing a list of currently established nodes and topics within the ROS environment. Furthermore each topic is strongly typed by message types that only allows either publishing or subscribing to messages of that type. Furthermore in ROS there are 2 types of communication, which use 1) a publisher and subscriber model for continuous data flow and 2) ROS services, ROS's implementation of remote procedure call request and reply interactions.

3.3.2 ROCOS

ROCOS is a proprietary middleware developed by Yujin Robot, Korea, that provides application contents for the company's robots, such as iRobi and Charlie. ROCOS uses an API encoded in XML. While it is proprietary software with a hidden implementation, it is able to support both data flow and service typed messages. The input and output structures of ROCOS vary from API to API. For the robot manager context, we have implemented a class for each different message according to the name of the each API.

3.3.3 OpenRTM

OpenRTM is a middleware implementation of RT-middleware developed by AIST, Japan. OpenRTM's component is known as an RT-Component or RTC, which uses a finite state machine model to control its operations to send or receive data flows. Each state has its own characteristic ports to its interface, sending or receiving different kinds of messages depending on the state. The states are also controlled dynamically over runtime depending on the execution defined by the user. The previous version of OpenRTM used an omniORB implementation of CORBA to communicate and it therefore is available in a variety of languages for multiple platforms [Ortiz *et al.*, 2014]. The new version of OpenRTM uses ICE instead of CORBA [Morckos *et al.*, 2014].

Table 1. UoA robotics Software Software Framwork Messaging Structure.

Op	The operation instruction.
type (client to RM)	Function type to be requested or used.
type(RM to Bridge)	A list of strings which are required to communicate with components. In ROS these are names of topic and message types, and in OpenRTM this is the name of the message types and execution period.
jsonMeesage	The context data formatted in JSON. This applies to both input and output.

3.3.4 UoA Robotic Software Framework

The API design is a critical task in order to fulfill the four key cocepts as it requires a generic form of communication and the ability to connect to the other frameworks. Generally, communication of existing frameworks consists of message-based communication in two forms: a data port model that requires continuous connection maintenance for input or output data flow, and a service model that is a remote procedure call to provide results, which does not require the connection to be sustained once the results have returned. However in addition to data flow information, more information is needed in order to further handle the data flow, such as forwarding the dependant stream as input of another component. By wrapping the transmitted data with lightweight headers, we have created control messages that are structured in a way to identify components and the framework. Table 1 shows the messaging structure of the robot manager. The client is able to control the connection by specifying the *op* field using either a *Request* or *Kill* command to establish or terminate the dataflow connection respectively. In order to request for a connection to a component, the control message must specify the functionality which is defined in the *type* field of the message. The *jsonMessage* field is the message that contains the data from or to the component in lightweight JSON format.

3.4 Control Procedures

In order to request a component connection, the application is required to make the *Request* command for some kind of functionality using the high level API. The high level API will be defined in the configuration file which will be mapped to a framework specific components. When a service is requested through request messages, the robot manager will refer to the table in the configuration file and check the required dependencies. The dependencies can be satisfied if the message structure of the output is same as the message structure of the input component. Hence, the robot manager makes an XML-RPC [Allman, 2003] request to get the message structure of the component to the bridge layer, which contains a database of the message structures.

For example in Fig. 2, in order for face recognition to work, it requires a face detection component that respectively also requires a camera component. Fig. 3 shows the overall procedure of the *Request* command. When the robot manager receives a request message for the face recognition function from the application, it will generate a dependency list with dependent components; face detection and camera. If the input of the dependent component matches with the output of the depended component, the robot manager checks dependency is satisfied. Hence the message structure will be checked between a) the output of the camera function and the input of the face detection function, then b) the output of the face detection function and the face recognition function,

which together verifies the components are able to talk with each other. If all of the dependencies are satisfied, the robot manager forwards the request call to the bridge, and registers the connection to a status map.

On receving the request messages, the bridge will open a new commnuication channel that can be used to communicate with that specific component. The bridge also registers connections to a map such that, on receiving any messages from the component, it knows which communication channel it must send back to. On invocation of the *Kill* command, the robot manager will first forward the connection to the bridge and delete the registered information and close down the communication channel. Then on receiving the command, the bridge will also shut down the communication of the component.

4 Experiment: Case Study

We developed two HealthBots with healthcare applications based on the robot manager framework. The applications of two HealthBots are developed with different programming languages and frameworks, with the same functionalities, using the APIs of the UoA robotic software framework. To evaluate our framework, we developed a general case study with the two HealthBots.

4.1 System Overview

Fig. 4 shows the HealthBot, which is based on a kiosk type robot platform from Yujin Robot in South Korea and is originally designed as a serving robot in cafes and restaurants, and for assisting teachers in schools; it has a friendly appearance and tray for carrying items. The HealthBot is a differential drive mobile robot 1.2 meters high, powered by a 24V Li-Polymer battery, and consists of a camera, a rotatable touch screen, speakers, microphones, ultrasonic sensors, bumper sensors, a laser range finder and two single board computers. User responses were received via the touch screen and HealthBot responds to participants with synthesized speech, visual output on the screen, and movements. The touch screen helps the older people who have hearing or speaking difficulties by showing messages or pictures. HealthBot's synthetic speech is generated through a diphone concatenation type synthesis implemented with the Festival speech synthesis system [Black *et al.*, 2012] and used a New Zealand accented diphone voice developed at the University of Auckland [Watson *et al.*, 2009]. Expression was added to the synthetic speech through an intonation modeling technique [Igic *et al.*, 2009] called 'Say Emotional'.

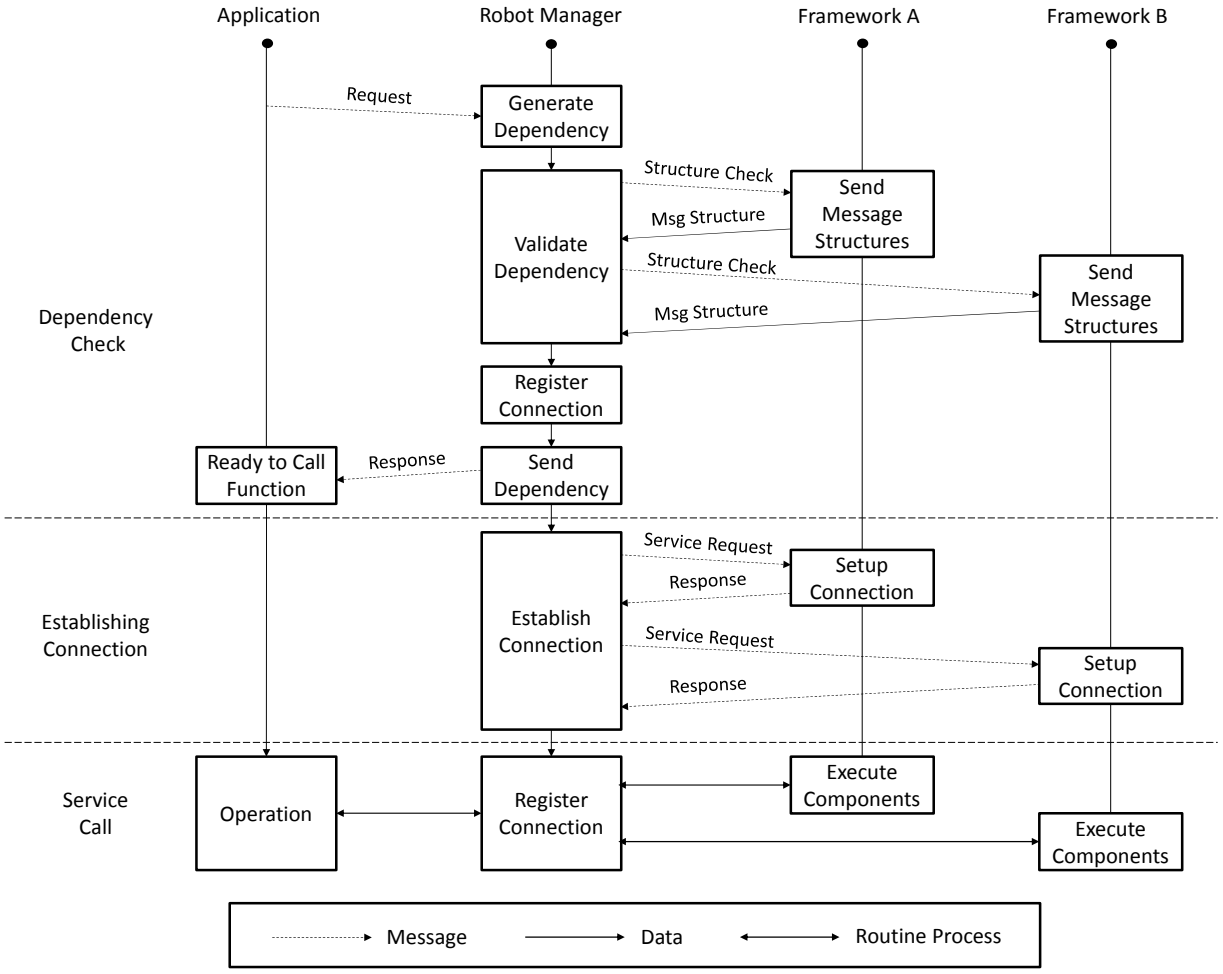

Fig. 3. Overall procedure of *Request* command.

4.2 Healthcare Application

The application of the HealthBot is originally developed by our HealthBot project [Robinson *et al.*, 2012; Robinson *et al.*, 2013; Datta *et al.*, 2013; Stafford *et al.*, 2014]. It was deployed in a general practice of hospital to capture vital signs information, in order to save nurses' time. HealthBot measures patients' vital signs, such as blood pressure, pulse rate, and the blood oxygen saturation level, and transfers the data over webservices to RoboGen, which is our medical server. We developed this application using Flex/ActionScript 3.0, which is becoming an outdated technology especially with growing mobile markets and its limited support in Linux environments.

Therefore, we developed another application, which has the same functionalities as our Flex/ActionScript-based application, using pure HTML and Javascript. Actually developing new applications with different programming languages and tools requires time-consumming effort on developing similar systems. However, we did not need to develop all of them again, because we used our new robotic software framework for the new application. Fig. 5 shows the system diagram of our two different applications. We developed only the top level user interface as a new component, and used other parts in the robot manager layer and the component layer of Fig. 1.

We used the API of the robot manager to reuse the existing components. HealthBot uses components from the ROCOS and OpenRTM frameworks, both of which are installed on the robots. We used a face detection component of OpenRTM for detecting patients and starting the robot application workflow. We used a speech generation component and two vital sign measurement sensor components, which are for blood pressure, pulse rate, and the blood oxygen saturation level, of ROCOS.

4.3 Case Study

We installed the two healthcare applications with different programming languages, one in Flex/ActionScript 3.0, and the other in HTML5 and Javascript, on our HealthBot to evaluate the UoA robotics software framework. We evaluated the performance of our system focusing on the four design concepts of the UoA robotic software framework.

Our case study operated successfully in a simple test and each component ran correctly, and the workflow ran correctly. We checked the latency of each functional process including dependency checking and the communication between two different frameworks, and it took less than 1 millisecond. Further tests are required; however this does show that the design goals have been met. Future tests will evaluate the effectiveness of our framework in future development and deployment of real applications.

4.3.1 Interoperability

In this case study, we used two existing robotics frameworks: OpenRTM and ROCOS. HealthBots detected human faces using the face detection component of OpenRTM, and spoke some sentences from the speech generation component of ROCOS. HealthBots measured three vital signs of humans using the vital sign measurement sensors components of ROCOS. From this result, we confirmed that the UoA robotics software framework satisfies the first design concept, interoperability.

4.3.2 Compatibility

We developed two different applications, which are operated in different environments; the Flex application is executed independently, and the HTML application is executed in a web browser. Both applications can be attached to the robot manager and communicate with various components of two frameworks in the Windows environment by calling the API of the robot manager. From this result, we confirmed that the UoA robotics software framework satisfies the second design concept, compatibility in one machine. But we did not evaluate of other frameworks operated in different environments such

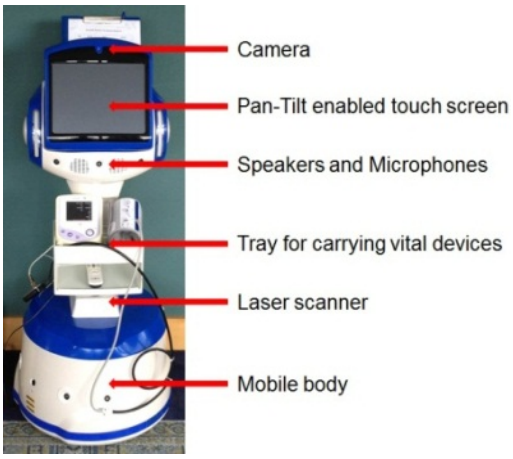

Fig. 4. Charlie, which is used for the HealthBot platform, consists of a camera, a Pan-Tilt enabled touch screen, speakers, microphones, ultrasonic sensors, bumper sensors, a laser scanner, two single board computers and a 24V Li-Polymer battery [Ahn et al., 2014].

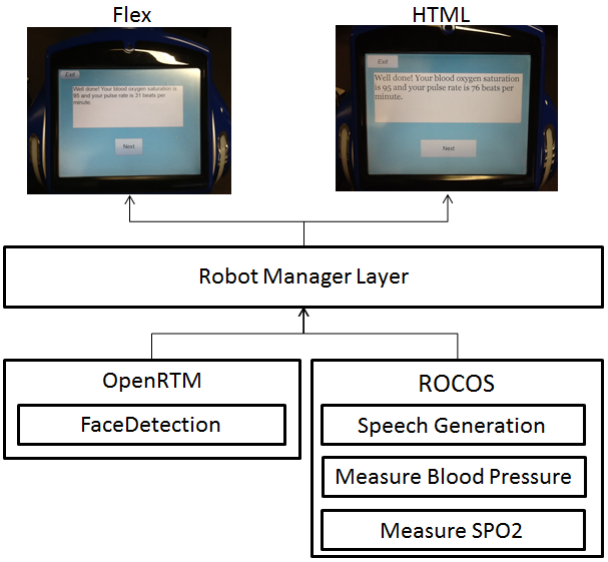

Fig. 5. System diagram of HealthBot. Two healthcare applications, which used different programming languages, can be applied to the same system.

as the Linux environment. So we will do more experiments with different platforms with other environments, in the future.

4.3.3 Support for heterogeneous applications

We developed two different applications with different programming languages. Both applications were applied on the same robot platform usng the same components. Both applications were able to use components from both tasks including detecting people's faces, from OpenRTM's face detection component, and using speech functions and healthcare devices, from ROCOS. The effort required to use these components in both applications was not difficult as they can both be accessed through use of the same API. The greatest advantage of using our framework is that it hides the framework-specific procedures of receiving the data while giving freedom to access the data from components of different frameworks. From these results, we confirmed that the UoA robotics software framework satisfies the third design concept, support for heterogeneous applications.

4.3.4 Dynamic monitoring and control

Our API allowed two applications to control the status of the component connections. The HealthBots also could check the connection and notify the result to applications if any of the connections are terminated, and automatically issue a *Kill* operation removing the maps. From these results, we confirmed that the UoA robotics software framework satisfies the last design concept, dynamic monitoring and control.

5 Conclusions

We designed and developed the UoA robotic software framework that allows using various combinations of components, packages, and libraries from the various existing frameworks. Through our API design of, we could benefit from the key functionalities of our software framework including interoperability, compatibility, support for heterogeneous applications, and dynamic monitoring and control of the communication. We have performed a case study where we created a healthcare assistant robot system using two applications developed in different programming languages and used components from two different frameworks. For future work, we plan to extend our framework by allowing dynamic configuration to eliminate the effort of predefining communication by the user. We also aim to evaluate our system in other application domains than healthcare.

References

[Ahn et al., 2008] Ho Seok Ahn, Young Min Baek, In-Kyu Sa, Jin Hee Na, Woo-Sung Kang, and Jin Young Choi. Design of Reconfigurable Heterogeneous Modular Architecture for Service Robot. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, pages 1313-1318, 2008.

[Ahn et al., 2014] Ho Seok Ahn, I-Han Kuo, Elizabeth Broadbent, Ngaire Kerse, Kathy Peri, Chandan Datta, Rebecca Stafford, and Bruce A. MacDonald. Design of a Kiosk Type Healthcare Robot System for Older People in Private and Public Places. *Proceedings of the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, page 578-589, 2014.

[Allman, 2003] M. Allman. An evaluation of XML-RPC. *ACM Sigmetrics Performance Evaluation Review*, 30(4):2-11, 2003.

[Ando et al., 2008] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87-98. 2008.

[Arndt et al., 2013] M. Arndt, M. Reichardt, J. Hirth, amd K. Berns. Requirements for interoperability and seamless integration of different robotic frameworks. *Proceedings of the Workshop on Software Development and Integration in Robotics*, Page 38-40, 2013.

[Biggs et al., 2010] G. Biggs, N. Ando, T. Kotoku. Native Robot Software Framework Inter-operation. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 180-191, 2010.

[Black et al., 2012] A. W. Black, P. Taylor, and R. Caley. The festival speech synthesis system. http://www.cstr.ed.ac.uk/projects/festival, 2012.

[Datta et al., 2013] Chandan Datta, Hong Yul Yang, I-Han Kuo, Elizabeth Broadbent, and Bruce A MacDonald. Software platform design for personal service robots in healthcare. *Proceedings of the IEEE International Conference on Robotics, Automation and Mechatronics*, page 156-161, 2013.

[Herman et al., 2013] B. Herman, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmarthemis, L. Gherardi, and D. Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. *Proceedings of the Annual ACM Symposium on Applied Computing*, pages 1758-1764, 2013.

[Igic et al., 2009] Igic A, Watson CI, Teutenberg J, Broadbent E, Tamagawa R, and B. A. MacDonald. Towards a flexible platform for voice accent and expression selection on a healthcare robot. *Proceedings of the Australasian Language Technology Association Workshop*, 2009.

[Jang et al., 2010] Choulsoo Jang, Seung-Ik Lee, Seung-Woog Jung, Byoungyoul Song, Rockwon Kim, Sunghoon Kim, and Cheol-Hoon Lee. OPRoS: A New Component-Based Robot Software Platform. *ETRI Journal*, 32(5):646-656, 2010.

[Jang et al., 2012] C. Jang, B. Song, S. Jung and S. Kim. A heterogeneous coupling scheme of OPRoS component framework with ROS. *Proceedings of the RAS/EMBS International Conference on Ubiquitous Robots and Ambient Intelligence*, page 298-301, 2012.

[Jayawardena et al., 2012] C. Jayawardena, I. Kuo, C. Datta, R. Q. Stafford, E. Broadbent, and Bruce. A. MacDonald. Design, implementation and field tests of a socially assistive robot for the elderly: HealthBot Version 2. *Proceedings of the RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*, page 1837-1842, 2012.

[Lubbers et al., 2010] Peter Lubbers, and Frank Greco. HTML 5 web sockets: A quantum leap in scalability for the web. *SOA World Magazine*, 2010. [Ortiz et al., 2014] Francisco J. Ortiz, Carlos C. Insaurralde, Diego Alonso, Francisco Sánchez, and Yvan R. Petillot. Model-driven analysis and design for software development of autonomous underwater vehicles.

Robotica, pages 1-20, 2014.

[Makarenko et al., 2007] A. Makarenko, A. Brooks, and T. Kaupp. On the Benefits of Making Robotic Software Frameworks Thin. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, 2007.

[Morckos et al., 2014] Michael Morckos, and Fakhreddine Karray. Axon: A Middleware for Robotic Platforms in an Experimental Environment. *Advances in Intelligent Systems and Computing: Robot Intelligence Technology and Applications 2*, 274:911-926, 2014.

[Quigley et al., 2009] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. *Proceedings of the ICRA workshop on open source software*, 2009.

[Robinson et al., 2012] Hayley Robinson, Bruce A. MacDonald, Ngaire Kerse, and Elizabeth Broadbent. Suitability of Healthcare Robots for a Dementia Unit and Suggested Improvements. *Journal of the American Medical Directors Association*, 14(1):34-40, 2012.

[Robinson et al., 2013] Hayley Robinson, Bruce A. MacDonald, Ngaire Kerse, and Elizabeth Broadbent. The Psychosocial Effects of a Companion Robot: A Randomized Controlled Trial. *Journal of the American Medical Directors Association*, 14(9):661–667, 2013.

[Stafford et al., 2014] Rebecca Q. Stafford, Bruce A. MacDonald, Chandimal Jayawardena, Daniel M. Wegner, and Elizabeth Broadbent. Does the Robot Have a Mind? Mind Perception and Attitudes Towards Robots Predict Use of an Eldercare Robot. *International Journal of Social Robotics*, 6(1):17-32, 2014.

[Watson et al., 2009] C. I. Watson, J. Teutenberg, L. Thompson, S. Roehling, and A. Igic. How to build a New Zealand voice. *Proceedings of the New Zealand Linguistic Society Conference*, 2009.