The Robot Software Communications Architecture (RSCA): Embedded Middleware for Networked Service Robots

Seongsoo Hong[1], Jaesoo Lee[1], Hyeonsang Eom[2], and Gwangil Jeon[3]
[1]*Real-Time Operating Systems Laboratory, School of Electrical Engineering and Computer Science,*
Seoul National University, Seoul 151-744, Korea
{sshong, jslee }@redwood.snu.ac.kr
[2]*Distributed Information Processing Laboratory, School of Computer Science and Computer Engineering,*
Seoul National University, Seoul 151-744, Korea
hseom@cse.snu.ac.kr
[3]*Department of Computer Engineering*
Korea Polytechnic University, 2121 Jungwang-Dong, Siheung-Si, Gyunggi-Do 429-793, Korea
gijeon@kpu.ac.kr

Abstract

In this paper, we present a robot middleware technology named Robot Software Communications Architecture (RSCA) for its use in networked home service robots. The RSCA provides a standard operating environment for the robot applications together with a framework that expedites the development of such applications. The operating environment is comprised of a real-time operating system, a communication middleware, and a deployment middleware. Particularly, the deployment middleware supports the reconfiguration of component-based robot applications including installation, creation, start, stop, tear-down, and un-installation. In designing RSCA, we have adopted a middleware called SCA from the software defined radio domain and extend it since the original SCA lacks the real-time guarantees and appropriate event services. We have fully implemented RSCA and performed measurements to quantify its run-time performance. Our implementation clearly shows the viability of RSCA.

1. Introduction

Recently, the Ubiquitous Robotic Companion (URC) project has been launched in Korea with an aim of putting networked service robots into a practical use in residential environments by overcoming technical challenges of the conventional home service robots. While the usefulness of an intelligent service robot has been evident for a long time, its emergence as a common household device has been painfully slow. This is due in part to the variety of technologies involved in creating a cost-effective robot. A modern service robot often makes a self-contained distributed system, typically composed of a number of embedded processors, hardware devices, and communication buses. The logistics behind integrating these devices are dauntingly complex, especially if the robot is to interface with other household devices. The ever-falling prices of high performance CPUs and the evolution of communication technologies have made the realization of robots' potential closer than ever. What is left is to address the complexity of the robotic technology convergence.

At the head of this effort is the URC robot project. Under development since 2004, the URC has been conceived as "a robot friend to help people anywhere, anytime." The project's aim is to improve the robot technology and to facilitate the spread of the robot use by making the robots more cost-effective and practical. Specifically to that end, it has been proposed that the robot's most complex calculations are handled by a high-performance remote server, which is connected via a broadband communication network. For example, the vision or navigation systems that need a high performance MPU or DSP would be implemented on a remote server, and the robot itself would act as a thin client, making it cheaper and more lightweight.

Clearly, this type of system demands a very sophisticated software platform which makes the logistics of such a robot system manageable. Furthermore, beyond simply creating such a platform, the desired goal is to create a standard that could serve the robotics community at large. Recently, there has been a great deal of research activity in this area, and yet there is still no current standard that has garnered international approval. In this paper, we thus propose a new middleware architecture for networked service robots by adopting an existing middleware

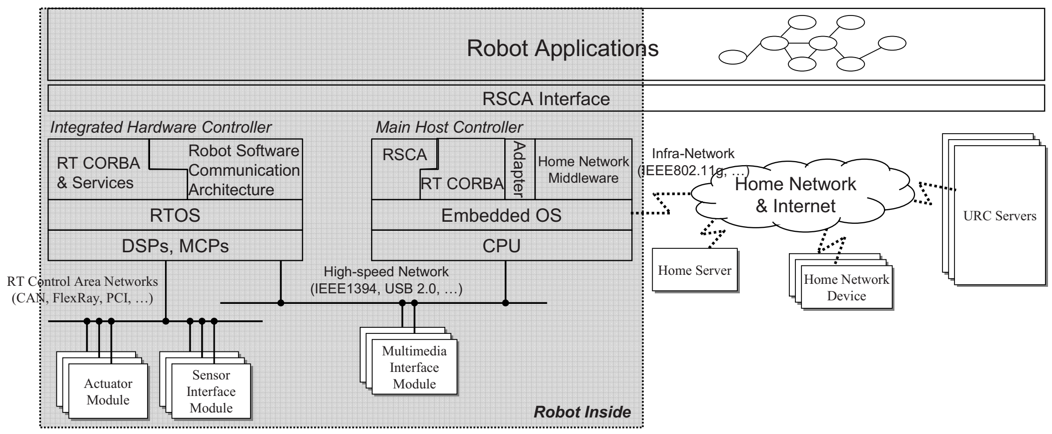


Figure 1. Structure of hardware and software of URC robot.

technology from the Software Defined Radio (SDR) domain. It is called the Software Communications Architecture (SCA) [1]. We extend it for the use in the URC robots. The SCA was defined by Joint Tactical Radio Systems (JTRS) and has become a de facto standard middleware currently adopted by the SDR forum. It is now widely accepted as a viable solution to reconfigurable component-based distributed computing for adaptive wireless radio terminals and base stations. In spite of its numerous strengths as an embedded middleware, it cannot be directly applied to our URC robots since it lacks some features necessary for the URC robot applications including real-time and QoS capabilities and appropriate event services. Thus, we have significantly extended it to incorporate these features and we have named the end result Robot Software Communications Architecture (RSCA). The RSCA provides a standard operating environment for robot applications together with a framework that expedites the development of such applications. We have fully implemented RSCA and performed measurements to quantify its run-time performance. Our implementation clearly shows the viability of RSCA.

2. URC Robot Hardware Platform and Software Requirements

Figure 1 depicts the hardware and software structure of the URC robot. Two of the most essential properties of the URC robot are (1) that it should be able to utilize a high-performance remote server called a URC server provided by a URC service provider and (2) that it should be able to interface with various smart home appliances and sensor networks that are connected to a larger home network. Thus, the URC robot is inherently a part of an overall distributed system including the URC servers and various home network appliances. In this section, we first look into the hardware structure of a URC robot and describe the properties that the system software of a URC robot must have to support the hardware structure.

2.1. Hardware Structure of URC Robot

A URC robot itself is a self-contained distributed system, composed of a number of embedded processors, hardware devices, and communication busses. More specifically, as shown in Figure 1, the main hardware components in a URC robot are a Main Host Controller (MHC) and one or more Integrated Hardware Controllers (IHC). An IHC provides accesses to the sensors and the actuators for other components such as other IHCs and an MHC. The MHC acts as an interface to the robot from the outside world; it provides a GUI for the interactions with the robot users and it routes messages from an inner component to the URC server and the home network appliances and vice versa.

For communication among the IHCs and MHC, a high bandwidth medium such as Giga-bit Ethernet, USB 2.0, or IEEE1394 is used. It allows a huge amount of data streams such as MPEG4 video frames to be exchanged inside the robot. Also, a controller area network such as CAN or FlexRay is used for communication among the IHCs, sensors, and actuators. Note that it is important to provide timing guarantees for this type of communication.

2.2. Requirements for URC Robot Software

As previously mentioned, the URC robot is not only a self-contained distributed system by itself but also part of an overall distributed system including remote URC servers and various home network appliances. Therefore, its application software must be developed according to the special requirements of a distributed robotic system. We argue that reconfigurability, flexibility, and reusability are the key requirements, among many others, for the URC robot software. To achieve these, developers should construct the application software according to the component-based software model, and the system software of the URC robot should support this model along with the reconfigurable distributed computing.

More specifically, the system software should provide (1) a framework in which programs can be executed in a distributed environment, (2) a dynamic deployment mechanism by which a program can be loaded, reconfigured, and run, (3) real-time capabilities that allow the robot software to meet hard deadlines, (4) QoS capabilities which can support the robotic vision and voice processing, and (5) a management capability for limited resources and heterogeneous hardware inherent in the URC robot.

3. Overall Structure of RSCA

The RSCA is specified in terms of a set of common interfaces for the robot applications as the SCA is. These interfaces are grouped into two classes: (1) the standard operating environment (OE) interfaces and (2) the standard application component interfaces. The former defines APIs that developers use to dynamically deploy and control applications and to exploit services from underlying platforms. The latter defines interfaces that an application component should implement in order to exploit the component-based software model supported by the underlying platforms.

As shown in Figure 1, the RSCA's operating environment consists of a real-time operating system (RTOS), a communication middleware, and a deployment middleware called core framework (CF). Since RSCA exploits COTS software for the RTOS and communication middleware layers, most of the RSCA specification is devoted to the CF. More specifically, RSCA defines the RTOS to be compliant to the PSE52 class of the IEEE POSIX.13 Real-Time Controller System profile [6], and the communication middleware to be compliant to minimum CORBA [3] and RT-CORBA v.1.1 [2]. The CF is defined in terms of a set of standard interfaces, called CF interfaces, and a set of XML descriptors, called domain profiles, as will be explained subsequently in Section 4.

The RTOS provides a basic abstraction layer that makes the robot applications both portable and reusable on diverse hardware platforms. Specifically, a POSIX compliant RTOS in RSCA defines standard interfaces for multi-tasking, file system, clock, timer, scheduling, task synchronization, message passing, and I/O to name a few.

The communication middleware is an essential layer that makes it possible to construct distributed and component-based software. Specifically, the RT-CORBA compliant middleware provides (1) a standard way of message communication, (2) a standard way of using various services, and (3) real-time capabilities. First, the (minimum) CORBA ORB in RSCA provides a standard way of message communication between components in a manner transparent to heterogeneities existing in hardware,

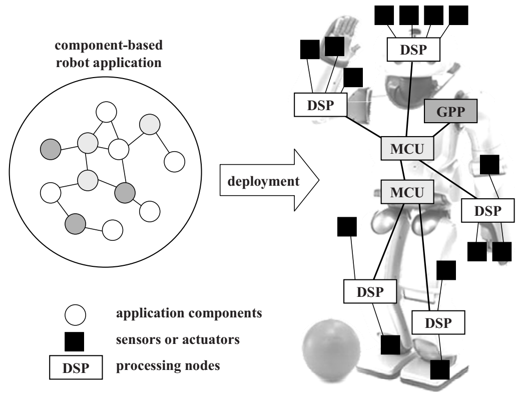


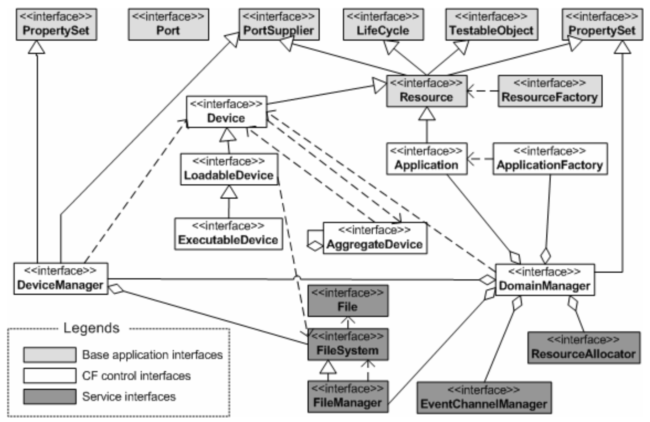
Figure 2. Deployment of component-based robot applications.

operating systems, network media, communication protocols, and programming languages. Second, the RSCA communication middleware provides a standard way of using various services. Among others, naming, logging, and event services are the key services that RSCA specifies as the mandatory services. Finally, the RT-CORBA in RSCA provides real-time capabilities including static and dynamic priority scheduling disciplines and prioritized communications in addition to the features provided by CORBA. Robot application developers are free to exploit these real-time capabilities to meet their applications' hard deadlines. A robot application developer, for example, can assign a higher priority to the processing of an emergency-stop event than to the processing of other lesser important events, thereby avoiding the deadline misses of the emergency-stop processing that could otherwise occur. Note that the original SCA's communication middleware does not support real-time capabilities since it recommends using minimum CORBA instead of RT-CORBA.

The deployment middleware layer provides a dynamic deployment mechanism by which robot applications can be loaded, reconfigured, and run. A URC robot application consists of application components that are connected to and cooperate with each other as illustrated in Figure 2. Consequently, the deployment entails a series of tasks that include determining a particular processing node to load each component, connecting the loaded components, enabling them to communicate with each other, and starting or stopping the whole URC robot software.

4. RSCA Core Framework

Before getting into the details of the RSCA CF, we begin with a brief explanation about the structural elements that the RSCA CF uses to model a robot system and the relationship between these elements. In the RSCA, a robot system is modeled as a domain that distinguishes each robot system uniquely. In a domain, there exist multiple



<<interface>> PropertySet
<<interface>> Port
<<interface>> PortSupplier
<<interface>> LifeCycle
<<interface>> TestableObject
<<interface>> PropertySet
<<interface>> Resource
<<interface>> ResourceFactory
<<interface>> Device
<<interface>> Application
<<interface>> ApplicationFactory
<<interface>> LoadableDevice
<<interface>> ExecutableDevice
<<interface>> AggregateDevice
<<interface>> DeviceManager
<<interface>> DomainManager
<<interface>> File
<<interface>> FileSystem
<<interface>> FileManager
<<interface>> ResourceAllocator
<<interface>> EventChannelManager
Legends
Base application interfaces
CF control interfaces
Service interfaces

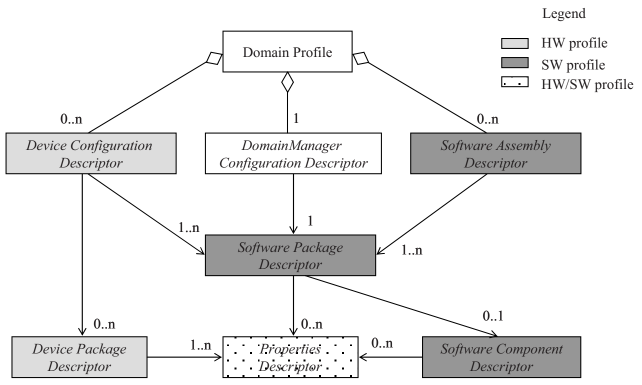
Figure 3. Relationships among CF interfaces.

processing nodes and multiple applications. The nodes and applications respectively serve as units of hardware and software reconfigurability. Hardware reconfigurability is achieved by attaching or detaching a node to or from the domain. A node may have multiple logical devices, which act as device drivers for real hardware devices such as Field Programmable Gate Arrays (FPGA), Digital Signal Processors (DSP), General Purpose Processors (GPP), or other proprietary devices. On the other hand, software reconfigurability is achieved by creating an instance of an application in a domain or removing the instance from the domain. An application consists of components, each of which is called a resource. A resource in turn exposes ports that are used for the communication to or from other resources. For communication between two components, a port of one component should be connected to a port of the other where the former port is called a *uses* port and the latter port is called a *provides* port. For the ease of communication between the components and the logical devices, the logical devices are modeled as a specialized form of a resource. Configurations of each of the nodes and applications are described in a set of XML files called domain profiles.

In this section, we explain the structure of the RSCA core framework in detail and the functionalities it provides. Note that what is described in this section is common with the SCA while the QoS capabilities and event service which will be described subsequently in Section 5 are unique to the RSCA.

4.1. Structure of RSCA Core Framework

The RSCA CF is defined in terms of a set of interfaces called CF interfaces and a set of XML files called domain profiles. As shown in Figure 3, the CF interfaces consist of three groups of APIs: the base application interfaces, the CF control interfaces, and the service interfaces. Each of these interfaces is defined for the application components, domain management, and services, respectively. The



Legend
HW profile
SW profile
HW/SW profile
Domain Profile
0..n
1
0..n
Device Configuration Descriptor
DomainManager Configuration Descriptor
Software Assembly Descriptor
1..n
1
1..n
Software Package Descriptor
0..n
0..n
0..1
Device Package Descriptor
1..n
Properties Descriptor
0..n
Software Component Descriptor

Figure 4. Relationships among domain profiles.

deployment middleware is therefore the implementation of the domain management and service part of the RSCA CF interfaces.

Specifically, (1) the base application interfaces are the interfaces that the deployment middleware uses to control each of the components comprising an application. Thus, every application component should implement these interfaces. These interfaces include the functionalities of starting/stopping a resource, configuring the resource, and connecting a port of the resource to a port of another resource. (2) The CF control interfaces are the interfaces provided to control the robot system. Controlling the robot system includes activities such as installing/uninstalling a robot application, starting/stopping it, registering/unregistering a logical device, tearing up/down a node, etc. (3) The service interfaces are the common interfaces that are used by both the deployment middleware and the applications. Currently, three services are provided: distributed file system, event, and QoS.

The domain profiles are a set of XML descriptors describing the configurations and the properties of hardware and software in a domain. They consist of seven types of XML descriptors as shown in Figure 4. (1) The *Device Configuration Descriptor* (DCD) describes a hardware configuration, and (2) the *Software Assembly Descriptor* (SAD) describes a software configuration and the connections among components. (3) These descriptors consist of one or more *Software Package Descriptors* (SPD), each of which describes a software component (*Resource*) or a hardware device (*Device*). (4) The *Properties Descriptor File* (PRF) describes optional reconfigurable properties, initial values, and executable parameters that are referenced by other domain profiles. (5) The *domainmanager configuration descriptor* (DMD) describes the *DomainManager* component and services used. (6) The *Software Component Descriptor* (SCD) describes the interfaces that a component provides or uses. Finally, (7) the *Device Package Descriptor* (DPD) describes a hardware device and identifies the class of the device.

4.2. Functionalities of RSCA Core Framework

Primarily, the RSCA core framework provides (1) dynamic system reconfiguration, (2) QoS and real-time guarantees, (3) heterogeneous distributed computing, and (4) heterogeneous resource management.

Dynamic system reconfiguration. In RSCA, the system reconfiguration is supported at three different levels: component level, application level and deployment level. For reconfiguration at the individual component level, the RSCA deployment middleware provides a way to specify and dynamically configure reconfigurable parameters of components. For the application-level reconfiguration, the RSCA deployment middleware provides a way to describe an application in various possible configurations (structures and parameters), each for different application requirements and constraints. The deployment-level reconfiguration indicates that the RSCA deployment middleware should choose one of possible assemblies as it is appropriate to the current resource availabilities.

QoS and real-time guarantees. As will be explained subsequently in Section 5, the deployment middleware supports application-level QoS guarantees while the RTOS and the communication middleware support real-time guarantees for individual components.

Heterogeneous distributed computing. The RSCA deployment middleware hides the distributed nature of hardware platforms from the applications by making distributed nodes be seen as a single virtual system or a domain. Robot applications need not consider how many processing nodes the domain consists of or which communication medium they use.

Heterogeneous resource management. The RSCA deployment middleware supports heterogeneous resource management via the *Device* interface. The *Device* interface provides interfaces to allocate and de-allocate a certain amount of resource such as memory, CPU, and network bandwidth. The *Device* interface also supports the synchronization of accesses to a resource by providing the resource usage and management status. A developer should, of course, choose and implement how resources are allocated and synchronized based on the efficiency of resource usage.

5. QoS and Event Support in RSCA Core Framework

QoS and event services are the newly added services that mostly delineate RSCA from the original SCA. In this section, we explain how these services are provided by the RSCA core framework in detail.

5.1. QoS Support in RSCA Core Framework

While home service robots are heavily involved in real-time signal processing such as vision and voice processing, the original SCA lacks QoS capabilities in terms of both QoS specification and enforcement. Thus, we have significantly extended the SCA for the QoS support in defining RSCA [4]. Specifically, we have (1) extended domain profiles to allow for resource and QoS requirements specification, (2) added services providing admission control and resource allocation to the RSCA core framework, and (3) extended the software communication bus based on the real-time ORB following the RT-CORBA v.1.1 specification. All of these extensions are made while maintaining backward compatibility so that URC robot developers can use existing SCA tools.

Using our RSCA core framework, the robot application developers can achieve their desired QoS by simply specifying their requirements in the domain profiles. In doing so, the application developers are responsible for describing their application structure and participating components in a dedicated XML descriptor called the Software Assembly Descriptor (SAD) described in Figure 4. Since a legacy SCA SAD only describes connections or flows of messages between components, we extend various fields in the SAD to specify QoS-related information such as the sampling periods and the maximum latencies.

The robot application component developers should specify in the extended fields of Software Package Descriptor (SPD) resource demands in terms of dependencies on the hardware and the expected computational resource requirements for data processing. Along with this, application component developers should implement a predefined set of configurable property operations that the RSCA core framework invokes to deliver the results of resource allocation. For the implementation of configurable property operations, RSCA provides a skeleton component implementation from which QoS-aware components will be derived.

In order to guarantee the desired QoS, described in the domain profiles, a certain amount of resources needs to be allocated to each application based on the current resource availability, and this must be enforced throughout the lifetime of the application. This involves admission control, resource allocation, and resource enforcement. For the admission control and the resource allocation, we add the *ResourceAllocator* component as shown in Figure 3. On the other hand, for the resource enforcement, we rely on the COTS layer of the RSCA operating environment following the design philosophy of SCA. To aid in understanding how the desired QoS is guaranteed, we explain the modified application creation process in RSCA.

An application in an RSCA domain is created by the *ApplicationFactory* component, which belongs to the RSCA domain management part and is in charge of instantiating a specified type of application. When

ApplicationFactory instantiates an application in RSCA, it ascertains its QoS requirements from the domain profile and then passes the information to the *ResourceAllocator*. If the application is admissible, the *ResourceAllocator* generates the resource allocation plan for the application based on the current resource availability. The *ApplicationFactory* component performs the resource allocation plan generated by *ResourceAllocator* in the following steps: it deploys all components onto the loadable/executable devices as designated in the plan, and then it delivers scheduling parameters to each component. On receiving the scheduling parameters, each component should set the RT-CORBA scheduling policy with the given scheduling parameters, and ascertain that those are enforced throughout its lifetime.

5.2. Event Support in RSCA Core Framework

In SCA, a CORBA event service is specified as a mandatory service. Although the CORBA event service provides a standardized way of producing or subscribing to an event to and from a certain event channel, there are three critical problems in using it for robotic applications. First, the reusability of components is seriously damaged since event channel names should be hard-coded within the components. Consequently, developers cannot deploy the components that communicate via the event channels without recompiling them. More seriously, the naming conflicts may occur among the events channels used by the irrelevant components. Second, it is very difficult for non-CORBA expert programmers to use the CORBA event service since a significant amount of manual coding is required for retrieving the proxy suppliers and consumers of the event channels. Finally, developers should manage the life cycle of the CORBA event channels manually. Specifically, developers have to assure that an event channel is launched before applications begin to use it. Developers also have to assure that the event channel is destroyed when applications do not use it any more by monitoring the usage status of the event channel. This is essential to avoid the waste of memory that the event channel occupies.

Thus, instead of directly using the CORBA event service as in the SCA, we have defined our own. To do so, we have (1) extended domain profiles of the original SCA to allow for describing connections using CORBA event channels, (2) introduced interfaces to the RSCA CF allowing application components to easily use the event channels, (3) added to the RSCA CF a service providing the life-cycle management of event channels, and (4) modified the application instantiation and torn-down process to automatically manage connections between applications and event channels.

When using our RSCA core framework, robot application developers can describe a connection between

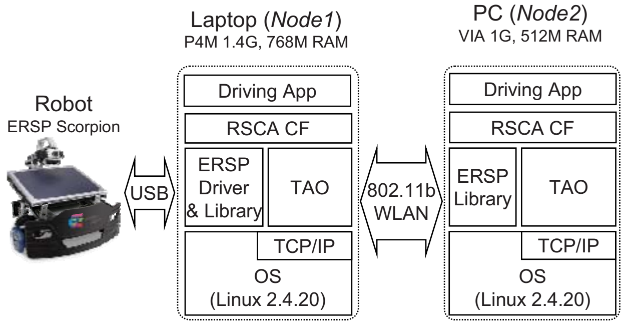


Figure 5. Experimental hardware and software configurations.

components via an event channel in our extended software assembly descriptor. The associated event channel is identified with its unique name. When *ApplicationFactory* creates an application, it locates the CORBA event channel associated with the designated name and pass the channel to the application component. In doing so, the *EventChannelManager* shown in Figure 3 provides interfaces to locate the event channel for the *ApplicationFactory*. The *EventChannelManager* also manages the lifecycle of event channels: creates or destroys event channels dynamically on needs.

6. Experimental Evaluation

In evaluating RSCA, it is important to quantify its run-time performance since it is built upon the COTS software layer containing the RT-CORBA ORB. Note that the RSCA core framework does not affect run-time performance at all since it only participates in the deployments of the robot applications. To quantify the run-time performance of RSCA, we have completely implemented the RSCA and constructed a simple robot application. In this section, we report on our experimental evaluation of RSCA.

6.1. Experimental Setup

As shown in Figure 5, our RSCA was implemented on a hardware platform consisting of an ERSP Scorpion robot from Evolution Robotics [7] and two processing nodes, a desktop computer and a laptop computer. The laptop computer is mounted on top of the Scorpion robot and connected to the robot via USB 2.0. The desktop and laptop computers are connected via 802.11b Wireless LAN. Although this configuration is not as complicated as the actual URC robots, it has all the components required to measure the performance of RSCA without incurring various side effects that could otherwise be seen.

Our RSCA core framework was implemented on top of Linux v.2.4.20 and TAO [5] real-time ORB v.1.3.1. The RSCA CF runs on both of the desktop and the laptop

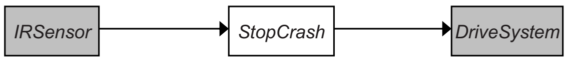


Figure 6. Structure of *RangeStop* application used for the experiments.

computers. An application, named *RangeStop*, was constructed using the ERSP library [7] and RSCA components and interfaces. Specifically, the application is constructed with two RSCA devices *IRSensor* and *DriveSystem* as shown in Figure 7. These devices abstract a set of range sensors and a set of motor actuators, respectively. The application moves the robot in the forward direction to the wall while it periodically reads the distance to the wall using the *IRSensor* device. And it stops the robot if it detects that the wall is within 60cm ahead of the robot.

To compare the run-time performance of the application against those that do not use RSCA, we also constructed the same application in two other versions making use of TCP/IP and the ERSP's legacy message passing, respectively, for communication among the components. Note that TCP/IP is widely used as a legacy transport for the communication among the components spanning multiple distributed nodes. Also, an application version making use of the ERSP's legacy message passing is constructed as a single monolithic binary, and thus it can be executed only on a single node.

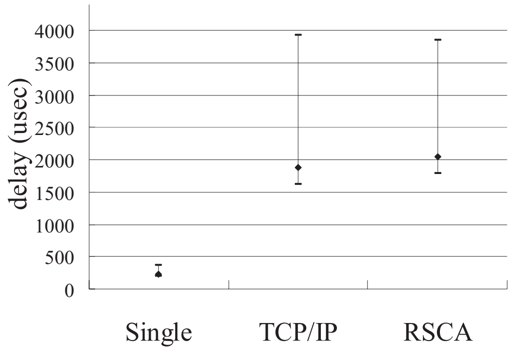
6.2. Performance Evaluation of RSCA

In order to quantify the run-time performance, we used two metrics: the communication delay and the distance from the wall. The delay incurred by transferring messages between two application components is measured to quantify the overhead incurred by using COTS software layer of the RSCA. The distance from the wall when the robot stops completely is measured to quantify the resultant effects of the COTS layer on the robot's behavior. The results are presented in comparisons among the three cases: single node, TCP/IP, and RSCA cases. Note that, in the cases of TCP/IP and RSCA, the components are deployed with spanning the two nodes: *IRSensor* and *DriveSystem* on the *Node1*, and *StopCrash* on the *Node2*.

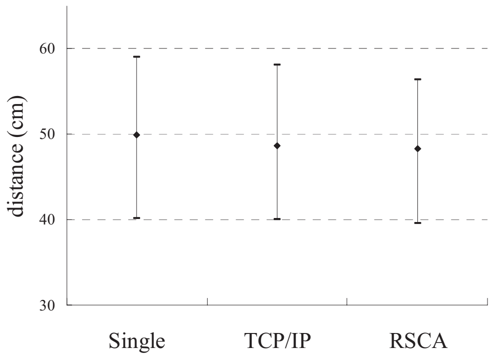
Figure 8 (a) depicts the message propagation delay measured from the *RangeStop* application. As shown, the message propagation delays of the TCP/IP and RSCA cases are 8 to 9 times larger than those of the single node case, while the message propagation delay of the RSCA case is slightly larger than that of the TCP/IP case. The average latencies in the single node, the TCP/IP, and the RSCA cases are 222.2us, 1872.6us, and 2045.9us, respectively. Thus, the overhead incurred by distributed communication is almost 900% while the communication using RT-CORBA incurs less than 10% of the additional delay compared to the TCP/IP communication.

It is worthwhile to emphasize that the distributed communication mechanisms such as TCP/IP, UDP/IP, and UNIX domain sockets would be used if the robot application components have to be collaborated on a distributed hardware inherent in the most modern robot systems. Even though RT-CORBA incurs a small additional overhead compared to the legacy communication mechanisms, it seems that the flexibility of RT-CORBA is enough to compensate the overhead. Note that RT-CORBA ORB selects the communication medium flexibly at run-time without changing the implementations of the application components. If properly configured, for example, TAO RT-CORBA ORB automatically selects shared memory for the communication between the components collocated on the same node.

Figure 8 (b) depicts the distance from the wall to the robot when it stops completely. As shown, there are no significant differences among those three cases. The average distances for each case of single node, TCP/IP, and RSCA are 49.97cm, 48.67cm, and 48.35cm, respectively, meaning that the robot advanced by 10.03cm, 11.33cm, and 11.65cm, respectively after the robot detects the wall within



(a) Message propagation delay



(b) Distance from the wall

Figure 7. Results from *StopCrash* application.

60 cm. Compared to the differences in the message propagation delay, this is not a large difference of less than 4%. Thus, we can conclude that the effects of using the RT-CORBA on the overall behavior of the robot are less than 4% for the *StopCrash* application.

7. Related Work

Traditionally, research into robot software architectures has been mainly focused on an application software framework [7][8][9][10] with an aim of helping developers programming their robot applications. However, as the robot hardware becomes distributed and heterogeneous, the robot software architectures are requested to address software complexity arising during both the management of complex distributed robot applications and the development of such applications. Recently, several approaches have been proposed based on middleware technologies to overcome ever-increasing software complexity, thereby significantly reducing the time-to-market. DROS [11] and Connexis [12] are the examples that use RPC-level custom middleware, and MIRO [13] and OCP [14] are the examples that utilize CORBA and RT-CORBA, respectively.

Unfortunately, these middleware systems fail to meet all of the software requirements presented in Section 2.2. Specifically, they do not support dynamic deployment, dynamic reconfiguration, and resource managements even though the real-time and the QoS capabilities and component-based distributed computing are partially supported in OCP. As already explained in the paper, the RSCA effectively fulfills those requirements.

8. Conclusions

In this paper, we have presented the Robot Software Communication Architecture (RSCA) we have developed to address the complexity inherent in networked home service robots. The RSCA provides a standard operating environment for the robot applications together with a framework that expedites the development of such applications. The operating environment is comprised of a real-time operating system, a communication middleware, and a deployment middleware, which collectively form a hierarchical structure. Particularly, the deployment middleware called the RSCA core framework provides (1) a framework in which programs can be executed in a distributed environment, (2) a dynamic deployment mechanism by which a program can be loaded, reconfigured, and run, (3) real-time capabilities that allow robot software to meet hard deadlines, (4) QoS capabilities which can support robotic vision and voice processing, and (5) a management capability for limited resources and heterogeneous hardware inherent in the URC robot. As a result, the RSCA solves many of important problems arising in creating an application performing complex tasks in the URC robot composed of the heterogeneous and distributed hardware.

We have completely implemented the RSCA and performed extensive measurements to analyze the effects of the RSCA's COTS software layer on the performance and the robot behaviors. The results are promising: less than 10% of an additional delay to the legacy communication and less than 4% of an effect on the overall robot compared to the case where the RSCA is not used. This outcome clearly demonstrates the viability of the RSCA. The RSCA is currently in an adoption process as a Korean domestic standard and is waiting for the industry approval.

References

[1] Joint Tactical Radio Systems. "Software Communications Architecture Specification V.3.0," August, 2004.

[2] Object Management Group. " Real-Time CORBA Specification Revision 1.1." OMG document formal/02-08-02 (August 2002).

[3] Object Management Group. " The Common Object Request Broker Architecture: Core Specification Revision 3.0." Dec. 2002.

[4] J. Lee, J. Park, S. Han, and S. Hong, "Extending Software Communications Architecture for QoS Support in SDR Signal Processing," 11th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications, 2005.

[5] F. Kuhns, D. D. Schmidt, et al. "The Design and Performance of a Real-Time Object Request Broker." IEEE Real-Time/Embedded Technology and Applications Symposium, May 2000.

[6] Institute for Electrical and Electronic Engineers. "Information Technology- Standardized Application Environment Profile- POSIX Realtime Application Support (AEP)." *IEEE Std 1003.13*, Feb. 2000.

[7] Evolution Robotics. "ERSP 3.0 Users Guide." http://www.evolution.com, 2004.

[8] Peter Soetens. "The Complete OROCOS Software Guide." http://www.orocos.org.

[9] ORiN Forum. "Specification of ORiN(Ver. 0.5)." http://jara.jp/e/orin/En_ORiN.pdf.

[10] David J. Miller and R. Charleene Lennox. "An Object-Oriented Environment for Robot System Architectures." IEEE Intl. Conf. on Robotics and Automation, Cincinnati, Ohaio, Aug. 13-16, 1990.

[11] David Austin. "Dave's Operating System." http://www.dros.org.

[12] IMB Rational Software Corporation. "Rational Rose Real-Time Connexis User Guide: Revision 2003.06.00." 2003.

[13] H. Utz, and et. el. "Miro - middleware for mobile robot applications." *IEEE Transactions on Robotics and Automation*, Volume: 18, Issue: 4 , pp:493 – 497, Aug. 2002.

[14] James L. Paunicha, Brian R. Mendel, and David E. Corman. "The OCP – An Open Middleware Solution for Embedded Systems." *Proceedings of the American Control Conference*, Arlington, VA, June 25-27, 2001.