18

Reference Middleware Architecture for Real-Time and Embedded Systems: A Case for Networked Service Robots

Saehwa Kim
Seoul National University

Seongsoo Hong
Seoul National University

18.1 Introduction

Convergence between computers and communications technologies has brought about strong commercial trends that have led to widespread deployment of distributed computing systems. These trends include the advent of e-commerce, integrated corporate information infrastructure, and web-based entertainment, just to name a few. Middleware technology has been extremely successful in these areas since it has solved many difficulties that developers faced in constructing distributed computing systems.

Middleware technology is ever expanding its frontier based on its success in the information infrastructure domain. It is thus not so strange to see middleware technology being adopted even in rather extraordinary domains such as automobile and robot industries. Unfortunately, the straightforward adoption of middleware technology to real-time and embedded systems may well lead to a failure since they are subject to severe nonfunctional constraints such as real-time guarantee, resource limitation, and fault-tolerance. In practice, most current middleware products are of only limited use in real-time and embedded systems [3].

A networked service robot is representative of contemporary real-time and embedded systems. It 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 information appliances [5]. Thus, middleware should play an essential role in the robotics domain.

To develop commercially viable middleware for a networked service robot, it is very important to explicitly elicit the unique requirements of real-time and embedded middleware and then come up with technologies that address them. These pose difficult technical challenges to middleware researchers. In this chapter, we attempt to propose a reference middleware architecture for real-time and embedded systems using robotics as a domain of discussion. To do so, we first identify the general requirements of the robot middleware and then present a specific middleware architecture called the robot software communications architecture (RSCA) that satisfies these requirements.

Many technical challenges still remain unresolved in real-time and embedded middleware research. These challenges have something to do with new performance optimization metrics, effective data streaming support, and domain-specific abstractions. For instance, real-time and embedded middleware is subject to different types of performance optimization criteria since a simple performance metric alone cannot capture a wide variety of requirements of commercial embedded systems. Instead, performance–power–cost product should be considered. Also, real-time and embedded middleware should provide right abstractions for developers. Inappropriate middleware abstraction may hurt both system performance and modeling capability.

While real-time and embedded systems industry desperately wants to adopt middleware technology, current middleware technology fail to fulfill this demand mainly because it ignores important practical concerns such as those mentioned above. We address them as future challenges before we conclude this chapter.

18.2 Robot Middleware Requirements

The emergence of an intelligent service robot has been painfully slow even though its utility has been evident for a long time. This is due in part to the variety of technologies involved in creating a cost-effective robot. However, the ever-falling prices of high-performance CPUs and the evolution of communication technologies have made the realization of robots' potential closer than ever. It is now important to address the software complexity of the robot. The aim of the robot industry is to improve the robot technology and to facilitate the spread of robots' use by making the robot more cost-effective and practical. Specifically to that end, it has been proposed that the robot's most complex calculations be handled by a high-performance remote server, which is connected via a broadband communication network to the robot. For example, a robot's vision or navigation system that needs a high-performance micro-processor unit (MPU) or digital signal processor (DSP) would be implemented on a remote server. The robot would then act as a thin client, which makes it cheaper and more lightweight.

Clearly, this type of distributed system demands a very highly sophisticated middleware to make the logistics of such a robot system manageable. There has been a great deal of research activity in robot middleware area and yet there is still no current middleware that has garnered wide industrial approval. This gives rise to specific requirements that are yet to be fulfilled.

In this section, we investigate the requirements that the middleware developers must satisfy to successfully design and deploy middleware in networked service robots. These requirements are (1) provision of device abstraction and component models, (2) dynamic reconfigurability, (3) resource frugality for scalability, and (4) real-time and quality of service (QoS) capabilities.

18.2.1 Device Abstraction and Component Models

A robot, like an automotive and avionic system, is an electromechanical system that is composed of a wide variety of hardware and software modules that have distinct idiosyncrasies. These modules work closely

with each other and have complex interaction patterns. For example, vision, navigation, and motion control modules are the most essential modules in the robot. A motion control module consists of a motor, an encoder, and a DSP. A motion controller task running on the DSP reads in position data from the encoder, runs a motion control algorithm, and then produces an actuation command to the motor. A navigation module reads in map data from the vision module, computes its route, and then sends a motion control command to the motion control module. The vision module reads in raw vision data from a robot's stereo camera sensors and then process the data to update the map.

In contrast, unlike an automotive and avionic system, a robot system makes an open platform that has to accommodate independently developed applications and execute them to enrich its features for improved user services. Such robot applications, even though they rely on services provided by the lower-level robot modules, should be made unaware of low-level devices inside the robot. As such, the robot middleware must be able to offer a sufficient level of device encapsulation to robot application developers by ensuring that the robot is device-agnostic. Specifically, the robot middleware is required to provide a device abstraction model that can effectively encapsulate and parameterize wide variety of devices used inside the robot. Currently, only few middleware products offer such a device abstraction.

As previously mentioned, a networked service robot is not only a self-contained distributed system by itself but also a part of an overall distributed system including remote servers and various home network appliances. For example, as explained before, computation-intensive modules such as vision and navigation modules could be strategically allocated to a remote server and a robot would collaborate with the remote server, a local cache, and even a residential gateway to obtain motion control commands from the remote navigation module. In such an environment, application software needs to be designed to reflect the distributed nature of a robot's operating environment (OE).

The sheer complexity of the robot software in a distributed environment requires a middleware-supported component technology as well as procedural and object-oriented communication middleware. While the communication middleware based on remote procedure calls and remote method invocations are effective in hiding heterogeneity in network protocols, operating systems, and implementation languages in distributed computing, it cannot support the plug and play of software modules. Recently, advanced software component models and their associated middleware products facilitate the easy integration of independently developed software modules and their life-cycle management. Thus, robot application software should be constructed according to a component-based software model and the middleware of the robot should support this model along with the reconfigurable distributed computing.

The middleware-supported component technology offers another benefit in the networked service robot. It serves as seamless glue between the device abstraction layer and any upper-level software layers. The device abstraction model of the robot yields logical devices that play the role of device drivers for real hardware devices. These logical devices can be transparently abstracted into components using a component wrapper. Therefore, they can be smoothly incorporated into component-based robot software. Surely, there are several differences between logical device components and pure software components. First, logical device components are preinstalled while the robot is manufactured. Second, logical devices that abstract processing units such as DSP, field-programmable gate array (FPGA), and general purpose processor (GPP) may load and execute other software components. Third, logical device components directly interface with their underlying hardware devices just like device drivers.

18.2.2 Dynamic Reconfigurability

A networked service robot should be able to adapt to various needs of users and varying operating conditions throughout its lifetime. A recent industry study shows that the robot needs a new set of features every three weeks since users are quickly accustomed to existing applications of the robot [4]. Also, the networked service robot is susceptible to network connectivity problems since part of its intelligence is implemented through the Internet. If its remote navigation server is temporarily unreachable, the robot should be able to reconfigure its software system to utilize its local navigation module with lesser precision

and get the system back later when the server is available again. Thus, dynamic reconfigurability becomes a very important requirement of robot middleware.

Robot middleware can support dynamic reconfigurability at three levels: system-level, application level, and component level. System-level reconfigurability is the ability to dynamically configure preinstalled applications and currently running applications without any interruption to the robot operation. With the system-level dynamic reconfigurability, the robot middleware can install/uninstall, instantiate/destroy, and start/stop diverse robot applications while the robot is operating. Application-level reconfigurability is the ability to dynamically configure the properties and structures of an application. Such a reconfiguration can be used to increase the system reliability by replicating components and dynamically replacing faulty components. Finally, component-level configurability denotes the ability to configure the properties of individual components and dynamically select one of many alternative implementations of a component.

18.2.3 Resource Frugality for Scalability

While the current middleware technology offers significant benefits to developers, real-time and embedded systems still cannot afford to enjoy all of these benefits if the thick layers of middleware incur too much runtime overhead and require too much hardware resource. From time to time, a new middleware standard comes out into the world after it solves most of the requests from the involved parties. This seems unavoidable since it is the role of middleware to embrace heterogeneity in distributed computing.

However, it is quite often the case that such middleware is ignored by real-time and embedded systems industry. In automotive industry, for instance, 16-bit microcontrollers with 64 KB memory are still easily found in practice. Thus, to warrant the success of middleware in real-time and embedded systems industry, it is inevitable to design middleware specifically customized to a given application domain. One noticeable example is AUTOSAR (automotive open system architecture) from the automotive industry [2]. AUTOSAR is the industry-led middleware standard for automotive electronic control units (ECU). It has already been widely adopted by many industry leaders. One of the reasons for its success lies in a very compact and efficient runtime system that is obtained through purely static binding of components. Unlike many other middleware architectures, AUTOSAR does not support dynamic binding of components.

18.2.4 Real-Time and QoS Capabilities

A networked service robot is heavily involved in real-time signal processing such as real-time vision and voice processing. Furthermore, most of the services provided by a networked service robot incorporate critical missions with strict deadlines. It is difficult to meet the real-time constraints of a robot application solely by application-level techniques without support from the underlying middleware and operating system. Particularly, the robot middleware has to provide support for real-time guarantees and QoS so that robot application developers can exploit these services.

The most essential real-time and QoS capabilities that the robot middleware should provide include abstractions for static and dynamic priority distributed scheduling, prioritized communications, admission control, resource reservation, and resource enforcement.

18.3 Reference Robot Middleware Architecture

We have identified the requirements of the robot middleware to use them in deriving a reference middleware architecture for a networked service robot. In this section, we present one specific example of such middleware called the RSCA. We argue that the RSCA demonstrates how typical robot middleware is organized. To help readers understand it better, we first show the typical internal hardware structure of a networked service robot on which it is running, focusing on its heterogeneity and distributed nature. We then show the overall structure of the RSCA and explain in detail its component management layer that is called a core framework (CF). After that, we show how the RSCA meets the requirements identified in the previous section.

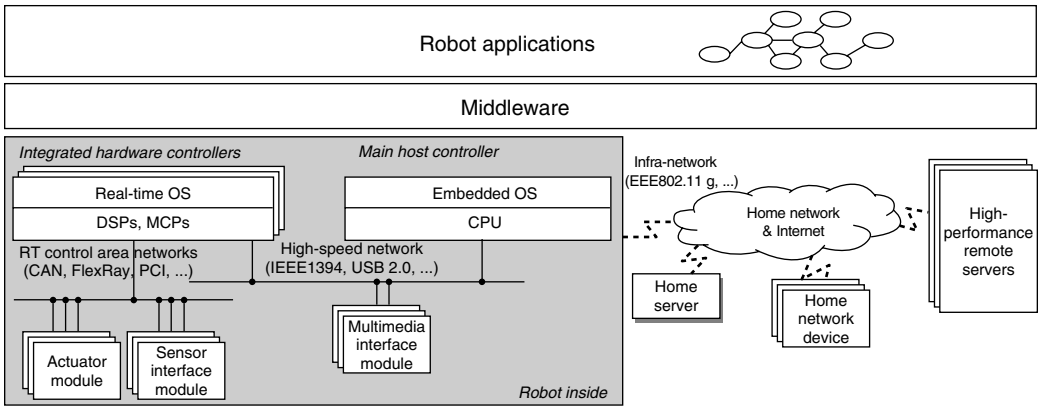

FIGURE 18.1 Reference structure of hardware and software of a networked service robot.

18.3.1 General Robot Hardware Platform

Figure 18.1 depicts the internal structure of hardware and software of the networked service robot, mainly concentrating on the hardware aspect. Two of the most essential properties of the networked service robot are (1) that it should be able to utilize a high-performance remote server provided by a 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. Clearly, this type of robot is inherently a part of an overall distributed system including remote servers and various home network appliances.

Apparently from Figure 18.1, the networked service robot itself is a self-contained distributed system as well; it is composed of a number of embedded processors, hardware devices, and communication buses. More specifically, as shown in Figure 18.1, main hardware components in the robot are main host controller (MHC) and one or more integrated hardware controllers (IHC). An IHC provides access to sensors and 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 graphical user interface for interactions with the robot users and it routes messages from an inner component to the remote server or 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.

18.3.2 Motivations for RSCA

We believe that the hardware structure shown in Figure 18.1 will be typical of future robot platforms. Simultaneously, it will pose a tremendous amount of software complexity to robot software. To overcome this challenge, robot application developers should maximize interoperability, reusability, and reconfigurability of robot software. Also, platform-, model-, and component-based methods should be incorporated into robot software development. To that end, we have developed the RSCA by adopting a widely accepted middleware technology from the software radio (SDR) domain. It is called software communications architecture (SCA) [7]. We have extended it for use in a networked service robot. Beyond simply creating the RSCA, the desired goal is to create a standard that could serve the robotics community at large.

18.3.3 Overall Structure of RSCA

The RSCA is specified in terms of a set of common interfaces for robot applications. These interfaces are grouped into two classes: (1) standard OE interfaces and (2) standard application component interfaces.

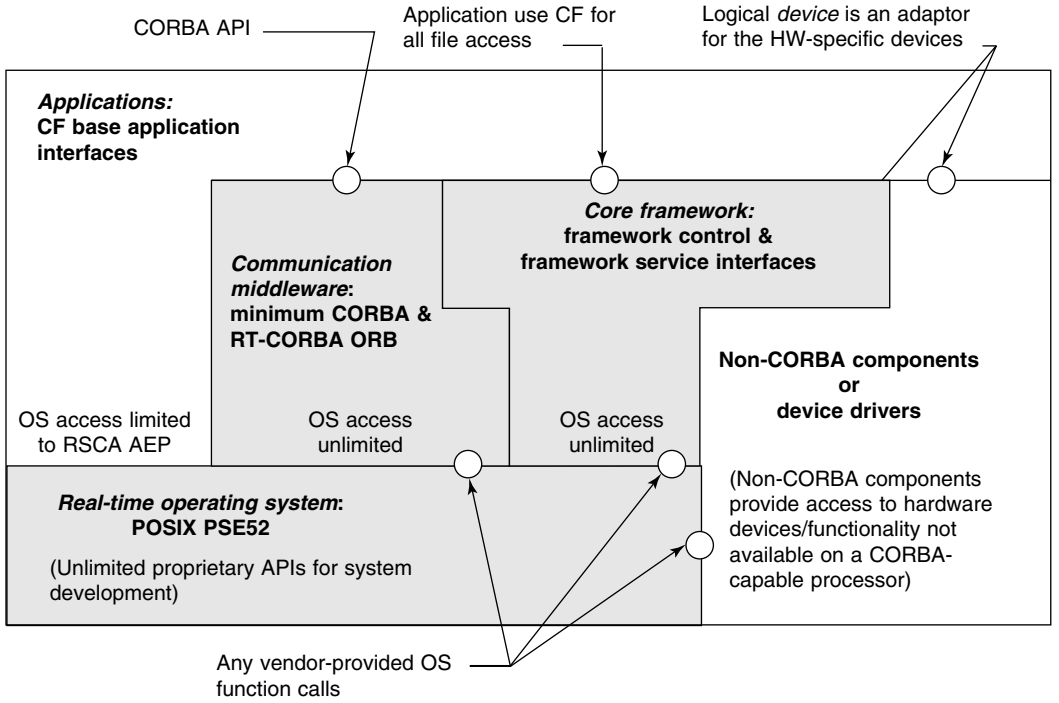

FIGURE 18.2 Overview of the operating environment of RSCA.

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 to exploit the component-based software model supported by the underlying platforms.

As shown in Figure 18.2, the RSCA's OE consists of a real-time operating system (RTOS), a communication middleware, and a deployment middleware called core framework (CF). Since the RSCA exploits COTS software for RTOS and communication middleware layers, most of the RSCA specification is devoted to the CF. More specifically, it 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 [13] and RT-CORBA v.2.0 [9,14]. 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 18.3.4.

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

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 the RSCA provides a standard way of message communication between components in a manner that is transparent to heterogeneities existing in hardware, 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 key services that the RSCA specifies as mandatory services. Finally, the RT-CORBA in the RSCA provides real-time capabilities including static and dynamic priority scheduling disciplines and prioritized communications in addition to the features provided by CORBA.

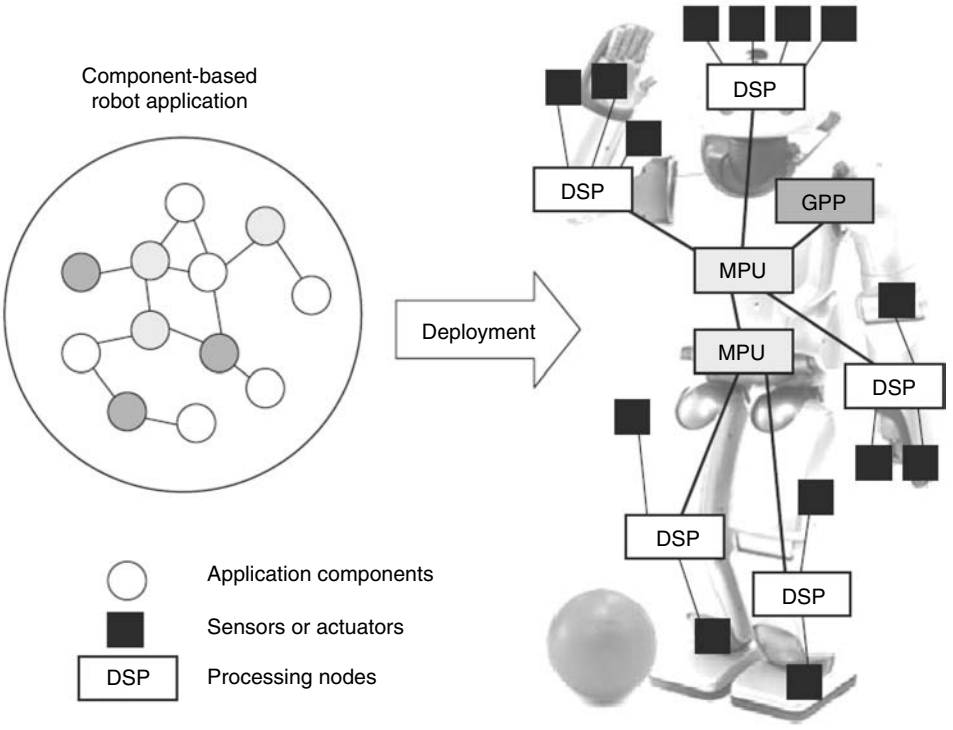

FIGURE 18.3 Deployment of component-based robot applications.

The deployment middleware layer provides a dynamic deployment mechanism by which robot applications can be loaded, reconfigured, and run. A robot application consists of application components that are connected to and cooperate with each other as illustrated in Figure 18.3. 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 robot software. Subsequently in the next subsection, we present the structure of the RSCA's deployment middleware in detail.

18.3.4 RSCA Core Framework

Before getting into the details of the RSCA CF, we begin with a brief explanation about 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 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 (FPGAs), DSPs, general-purpose processors, or other proprietary devices. In contrast, 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. As depicted in Figure 18.4, a resource in turn exposes ports that are used for communication to or from other resources [8]. 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 ease of communication between the components and the logical devices, the logical devices are modeled as a specialized form of a resource.

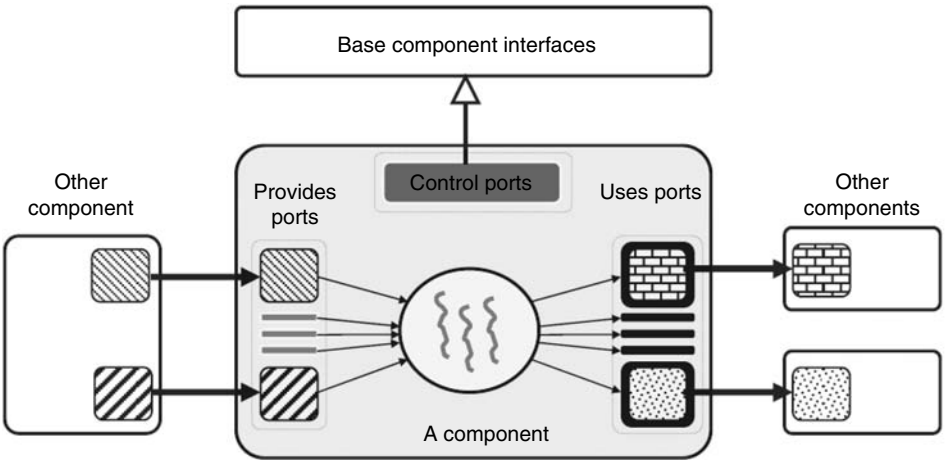

FIGURE 18.4 RSCA component model.

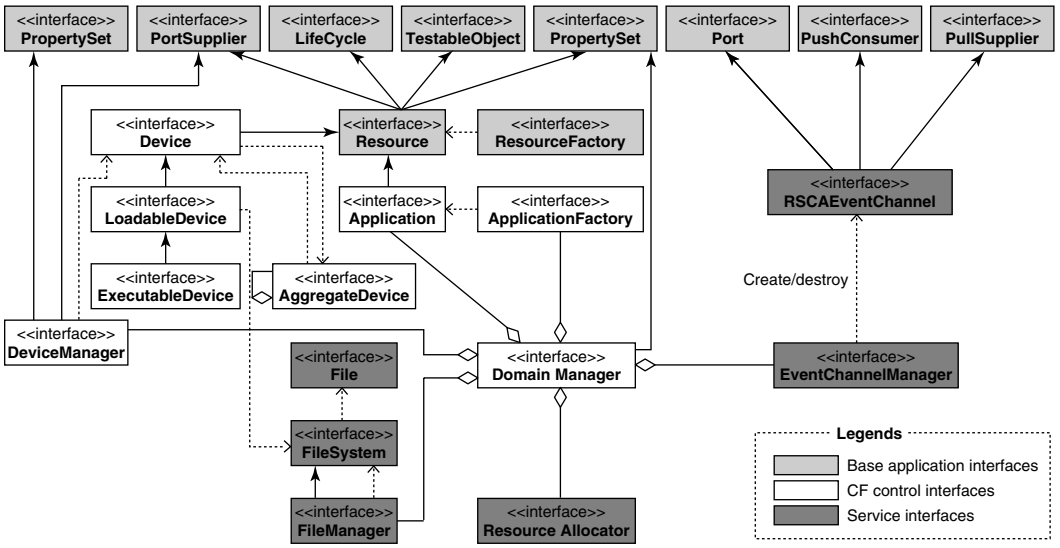

FIGURE 18.5 Relationships among CF interfaces.

Configurations of each of the nodes and applications are described in a set of XML files called domain profiles. We now explain the CF interfaces and the domain profiles in detail.

18.3.4.1 Core Framework Interfaces

As shown in Figure 18.5, CF interfaces consist of three groups of APIs: base application interfaces, CF control interfaces, and service interfaces. Each of the interface group is defined as application components, domain management, and services, respectively. The deployment middleware is therefore the implementation of the domain management and service part of the RSCA CF interfaces.

Specifically (1) base application interfaces are interfaces that the deployment middleware uses to control each of the components comprising an application. Thus, every application component should implement these interfaces as depicted in Figure 18.5. These interfaces include the functionalities of starting and stopping the resource, configuring the resource, and connecting a port of the resource to a port of another

resource. (2) The CF control interfaces are interfaces provided to control the robot system. Controlling the robot system includes activities such as installing/uninstalling a robot application, starting/stopping it, registering/deregistering a logical device, tearing-up/tearing-down a node, etc. (3) Service interfaces are common interfaces that are used by both deployment middleware and applications. Currently, three services are provided: distributed file system, distributed event, and QoS.

Among these interfaces, *ResourceAllocator* and *EventChannelManager* are interfaces defined for QoS and distributed event services, respectively. The *ResourceAllocator* together with domain profiles allow applications to achieve desired QoS guarantees by simply specifying their requirements in the domain profiles [10]. To guarantee the desired QoS described in the domain profiles, *ResourceAllocator* allocates a certain amount of resources based on current resource availability and these allocated resources are guaranteed to be enforced throughout the lifetime of an application relying on the COTS layer of the OE. In contrast, the *EventChannelManager* service together with domain profiles allow applications to make connections via CORBA event channels by simply specifying their connections in domain profiles. The *RSCAEventChannel* interface represents an event channel created by the *EventChannelManager*.

18.3.4.2 Domain Profiles

Domain profiles are a set of XML descriptors describing configurations and properties of hardware and software in a domain. They consist of seven types of XML descriptors as shown in Figure 18.6. (1) The *device configuration descriptor* (DCD) describes hardware configurations. (2) The *software assembly descriptor* (SAD) describes software configurations and the connections among components. (3) These descriptors consist of one or more *software package descriptors* (SPDs), 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 interfaces that a component provides or uses. Finally, (7) the *device package descriptor* (DPD) describes the hardware device and identifies the class of the device.

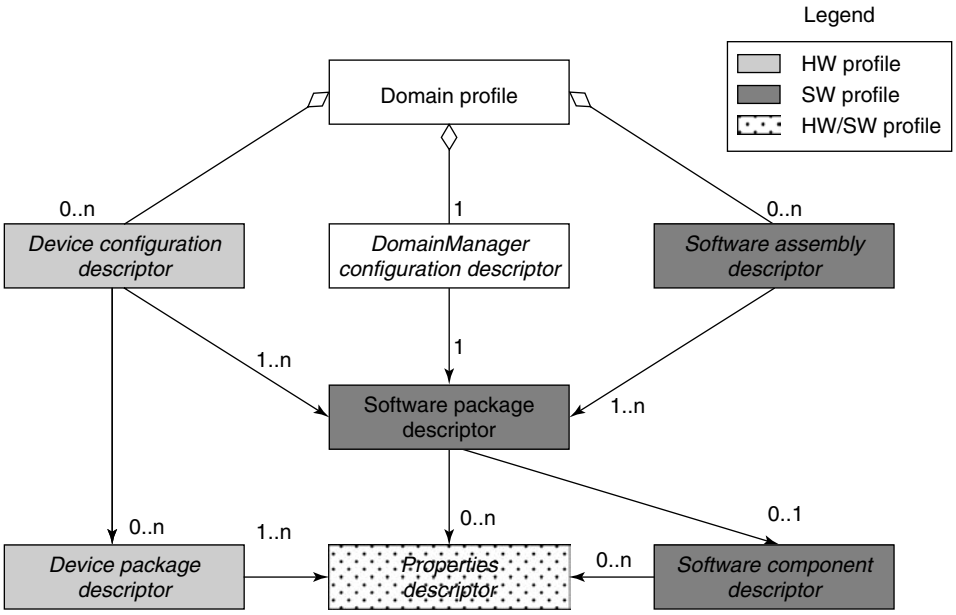

FIGURE 18.6 Relationships among domain profiles.

18.3.5 Assessment of RSCA

We now show how the RSCA meets the requirements identified in Section 18.2. First, the RSCA provides device abstraction via the *Device* interface. The *Device* interface provides interfaces to allocate and deallocate a certain amount of resources such as memory, CPU, and network bandwidth. The *Device* interface also supports synchronization of accesses to a resource by providing resource usage and management status. An application developer should, of course, specify how resources are allocated and synchronized based on efficiency of the resource usage. In contrast, the *Resource* interface provides interfaces for each component. The *Device* interface is also a specialized interface of the *Resource* interface, which enables logical devices to be transparently abstracted into components. The *Resource* interface provides interfaces for connecting components and managing component life cycle by inheriting the *PortSupplier* and *LifeCycle* interfaces, respectively. Along with these, XML descriptors that describe RSCA components and their assembly make possible the dynamic plug and play of RSCA components.

Second, the RSCA fully supports the dynamic reconfigurability of the robot system, an application, and a component. For the system-level reconfiguration, the RSCA provides a way to dynamically change the installed and running applications. For the application-level reconfiguration, the RSCA provides a way to describe an application in various possible configurations (structures and parameters), each for different application requirements and constraints. The RSCA should choose the most appropriate assembly among all possible assemblies according to the current resource availability at the deployment time. The *ApplicationFactory* is the component in the RSCA CF responsible for searching for the most suitable processing units (devices) that can satisfy the system dependency and resource requirements of a given component. However, for the time being, the current RSCA does not support runtime reconfiguration such as replicating components and replacing faulty components. Finally, for reconfiguration at the individual component level, the RSCA provides a way to specify and dynamically configure reconfigurable parameters of components. Among the RSCA CF interfaces, *PropertySet* provides interfaces that allow applications to query and configure the reconfigurable parameters at runtime.

Third, the RSCA supports resource frugality by employing a compact and efficient communication middleware and deployment middleware. The communication middleware of the RSCA is minimum CORBA that supports the pure static interface invocation of objects. It prohibits objects with unknown interfaces from communicating with each other. Also, the CF of the RSCA (the deployment middleware) does not support any configurable built-in library for component life-cycle management. Such a library is usually called a container. A container is supported by most traditional deployment middleware such as CORBA component model (CCM) [12], Enterprise JavaBeans [15], and Distributed Component Object Model (DCOM) [11]. The adoption of a container into these middleware products has been very successful in the information infrastructure domain. However, such a full-featured configurable built-in library unavoidably causes a large execution overhead and thus is not appropriate for networked service robots. Instead, the RSCA CF allows developers to manage component life cycle through the *ResourceFactory* interface, which is implemented by developers.

Finally, the RSCA supports application-level QoS guarantees. The RTOS and the communication middleware support real-time guarantees for individual components. Application developers can achieve their desired QoS by simply specifying their requirements in domain profiles. To guarantee the QoS described in domain profiles, the *ResourceAllocator* provides mechanisms for resource management, admission control, resource allocation, and resource enforcement.

18.4 Future Challenges of Robot Middleware

To develop commercially viable middleware for real-time and embedded systems, researchers must address technical challenges that arise owing to future trends in embedded systems design. Particularly, two trends are important for the discussion of future real-time and embedded middleware. First, an embedded system will become a heterogeneous distributed system as it requires extremely high performance in

both computing and communication. For instance, even a small SDR handset consists of a number of heterogeneous processors connected with diverse communication buses. Moreover, the advent of the MPSoC (multiprocessor system on a chip) technology makes chip supercomputers available for embedded systems. Secondly, applications that used to run only on high-performance mainframe computers will be deployed on embedded systems for enhanced user interfaces. However, such performance hogging applications should be rewritten to adapt to the unique OE of embedded systems.

Under these circumstances, middleware should play a crucial role for future real-time and embedded systems as it has done in other domains. In this section, we show three of the future technical challenges of real-time and embedded middleware, they are (1) performance optimization via new metrics, (2) streaming support for high-volume data, and (3) provision of domain-specific abstractions.

18.4.1 Performance Optimization via New Metrics

With the proliferation of inexpensive high-performance embedded microprocessors, future real-time and embedded systems are expected to be supercomputers that provide massive computational performance. They will make embedded devices much easier to use through human-centric interfaces that integrate real-time vision processing, speech recognition, and multimedia transmission. Clearly, middleware will be an enabling technology for realizing these functionalities [1].

However, to achieve embedded supercomputing, we need to change the way a system is designed and optimized and reflect this change into the future middleware. For instance, let us consider the case of the Internet-implemented robot intelligence. Historically, the most critical hurdle that robot industry has been facing in commercializing robots is the unacceptable cost of a robot compared to the level of intelligence that it can deliver. The Internet-implemented robot intelligence is a result of cost-conscious system redesign and optimization since it amortizes the cost of high-performance computing over a number of client robots. This leads to affordable intelligence to each robot. As we have seen earlier, the component-based middleware is a key to such an innovation.

Embedded supercomputers are mostly mobile systems whose massive computational performance is derived from the power in a battery. Thus, reduction in the power consumption of embedded systems is another critical design aspect. Consequently, embedded system developers should judge their systems via a new metric considering performance per unit power consumption and eventually via their performance–power–cost product [1]. Research into real-time and embedded middleware should deliver software architectures and mechanisms that considers this new metric.

18.4.2 Streaming Support for High-Volume Data

A majority of important applications running on real-time and embedded systems involve streaming of extremely large volume of data. For instance, a networked service robot hosts applications for real-time vision, speech recognition, and even multimedia presentation. However, current middleware products for real-time and embedded systems only partially address problems related to the data streaming. They provide neither a streaming data model as a first-class entity nor a data synchronization model for stream dataflows. Clearly, the future real-time and embedded middleware should support these mechanisms.

18.4.3 Domain-Specific Abstractions

A straightforward adoption of existing middleware products to real-time and embedded systems leads to a failure for the lack of domain-specific abstractions. Many robot developers find it very hard to use them in developing robots since their programming abstractions are different from those of existing robot software frameworks that have been used in robot industry for decades. Robot software frameworks specialize in modeling sensor and actuator devices and executable devices such as DSPs and MPUs. They also provide abstraction models that capture robot behavior architectures. To expedite the successful adoption of robot middleware, it is thus desirable to integrate existing robot software frameworks into robot middleware.

With this, robot developers can not only design robot applications using conventional robot software frameworks but also enjoy component-based software deployment at runtime.

18.5 Conclusions

While middleware technology has been successfully utilized in the enterprise computing domain, its adoption into commercial real-time and embedded systems is slow due to their extra requirements related to resource limitation, reliability, and cost. In this chapter, we presented a reference middleware architecture that specifically addresses the requirements of real-time and embedded systems. In doing so, we took robotics as a domain of discussion since a networked service robot, a representative real-time and embedded system, draws a great deal of interest. The middleware is called the RSCA.

The RSCA provides a standard OE for robot applications along with a framework that expedites the development of such applications. The OE is composed of an RTOS, a communication middleware, and a deployment middleware, which collectively forms a hierarchical structure that meets the four requirements we have identified for the networked service robot. The RSCA meets these requirements by providing the following capabilities. First, the RSCA provides a device abstraction model as well as a component model. This enables the plug and play of software modules allowing robot applications to be developed separately from the robot hardware in a device-agnostic manner. Second, the RSCA supports the dynamic reconfigurability of a robot system, an application, and a component. This enables robots to be adapted to various needs of users and varying operating conditions. Third, the RSCA supports resource frugality by employing the pure static interface invocation of objects dismissing a full-featured configurable built-in library for the component life-cycle management. This allows the RSCA to be used for robots even under stringent resource constraints. Finally, the RSCA supports real-time and QoS capabilities. As such, the RSCA solves many of the important problems arising when creating an application performing complex tasks for networked service robots.

The RSCA only partially addresses the technical challenges of real-time and embedded middleware since new important requirements arise with new trends in future real-time and embedded computing. These challenges have something to do with new performance optimization metrics, effective data streaming support, and domain-specific abstractions. Middleware research should deliver software architectures and mechanisms to cope with them.

Acknowledgment

This work was supported in part by Korea Ministry of Information and Communication (MIC) and Institute of Information Technology Assessment (IITA) through IT Leading R&D Support Project.

References

1. T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabati, and W. Wolf, Mobile supercomputers, *Communications of the ACM*, vol. 37, no. 5, pp. 81–83, May 2004.
2. AUTOSAR, AUTomotive Open System ARchitecture, http://www.autosar.org.
3. W. Emmerich, Software engineering and middleware: a roadmap, in *The Future of Software Engineering,* A. Finkelstein ed., pp. 76–90, ACM Press, New York, 2000.
4. Y. Fujita, Research and development of personal robot in NEC, NEC response to the Robotics Request for Information, http://www.omg.org/cgi-bin/doc?robotics/06-02-02.
5. S. Hong, J. Lee, H. Eom, and G. Jeon, The robot software communications architecture (RSCA): embedded middleware for networked service robots, in *Proceedings of the IEEE International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 2006.

6. Institute for Electrical and Electronic Engineers (IEEE), Information technology—standardized application environment profile—POSIX realtime application support (AEP), IEEE Std 1003.13, Feburary 2000.
7. Joint Tactical Radio Systems, Software Communications Architecture Specification V.3.0, August 2004.
8. S. Kim, J. Masse, and S. Hong, Dynamic deployment of software defined radio components for mobile wireless internet applications, in *Proceedings of International Human.Society@Internet Conference (HSI)*, June 2003.
9. F. Kuhns, D. C. Schmidt, and D. L. Levine, The performance of a real-time I/O subsystem for QoS-enabled ORB middleware, in *Proceedings of the International Symposium on Distributed Objects and Applications*, pp. 120–129, September 1999.
10. J. Lee, S. Kim, J. Park, and S. Hong, Q-SCA: incorporating QoS support into software communications architecture for SDR waveform processing, *Journal of Real-Time Systems*, vol. 34, no. 1, pp. 19–35, September 2006.
11. Microsoft Corporation, The distributed component object model (DCOM), http://www.microsoft.com/com/tech/DCOM.asp.
12. Object Management Group, CORBA component model version 3.0, June 2002.
13. Object Management Group, The common object request broker architecture: core specification revision 3.0, December 2002.
14. Object Management Group, Real-time CORBA specification revision 1.1, August 2002.
15. Sun Microsystems Inc., Enterprise JavaBeans technology, http://java.sun.com/products/ejb/.