Journal of Reliable Intelligent Environments (2018) 4:89–105
https://doi.org/10.1007/s40860-018-0060-7

ORIGINAL ARTICLE

A software framework for robotic mediators in smart environments

Young-Ho Suh[1] · Kang-Woo Lee[1] · Eun-Sun Cho[2]

Received: 31 January 2018 / Accepted: 7 April 2018 / Published online: 19 April 2018

Abstract
The experience of full automation without explicit user direction may induce anxiety among smart space users. The use of explicit mediators between users and fully automated systems may help to mitigate users' anxiety. While robots mediators are one possible solution, several issues remain, including high complexity and limited collaboration between robots and smart space platforms, reducing overall system reliability. This paper proposes the Integrated Control Architecture for Robotic mediator in Smart environments (ICARS) as a solution to improve the integration and reliability of robot mediators within automated smart spaces. Assuming relatively thin network robots as robotic mediators to enable a wide distribution with less cost, ICARS provides a well-organized software framework consisting of three layers to integrate robots and smart spaces: a flexible communication/device model, an adaptive service model for the integrated robot control architecture, and a behavior-based high-level collaboration model. In this paper, we also present details of the design, implementation, and an application scenario conducted with ICARS. The results show that ICARS enables flexible integration of the diverse devices associated with robots and environments, adaptive service provision for collaborative services, and easier development of high-level collaborative applications with decent performance.

Keywords Smart environment · Robotic mediator · Collaboration · Behavior · Control architecture

1 Introduction

Smart spaces are ordinary environments equipped with pervasive sensory and actuating devices that can perceive and react to the daily activities of their occupants. Realizing this vision requires situation-aware mediators that are able to capture users' intents and perform pertinent services proactively by organizing available resources in the spaces. According to the vision put forth by Weiser [1] for situation-aware mediators, the mediator was often seamlessly embedded and invisible to the user in early implementations. However, De Carolis and Cozzolongo [2] reported that 80% of subjects in an experiment they conducted felt uncomfortable interacting with an invisible presence and without explicit control over services, when they experience full automation without explicit delivery of their own intention. In addition, humans tend to express concerns that complicated but deficient technological products may harm them [36]. Such user concerns are a barrier to the widespread real-world deployment of smart space technology even when based on successful laboratory-based smart home projects. Recently, the concept of robotic mediators has been proposed as a novel alternative to new mediating interfaces in smart spaces [4,6–8]. The proposal is based on the idea that robots can be employed as personal assistants acting as mediators between the user and environment services using their robotic features such as mobility and multimodal interfaces. More specifically, the robotic mediators act as mobile and intelligent interfaces to the environment and embody the role of friendly companions to users. However, developing a robotic mediator system is not straightforward because it must be able to communicate and collaborate with diverse heterogeneous devices and systems in the smart spaces. In addition, an ideal robotic mediator should be lightweight to save resources like memory and batteries reducing expenses for users. Thus, we

✉ Eun-Sun Cho
eschough@cnu.ac.kr

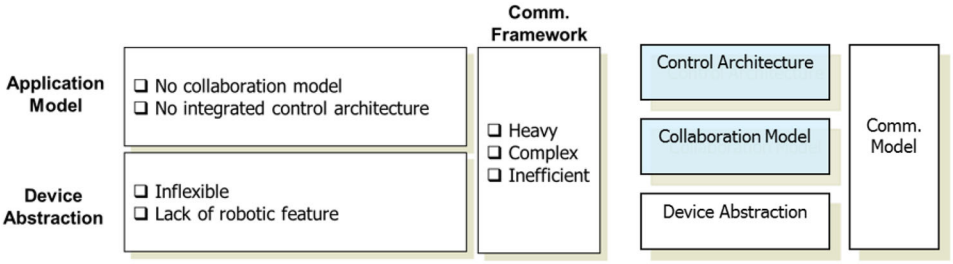
Young-Ho Suh
yhsuh@etri.re.kr

Kang-Woo Lee
kwlee@etri.re.kr

[1] Spatial Information Research Section, Electronics and Telecommunication Research Institute (ETRI), Daejeon, Republic of Korea

[2] Department of Computer Science and Engineering, Chungnam National University, Daejeon, Republic of Korea

Fig. 1 Architecture comparison



(a) Typical architecture of the existing approaches **(b)** Proposed architecture of ICARS

identify two major goals/challenges for successful robotic mediator systems:

- The developers of robotic mediator systems should be able to employ a well-designed software framework that interweaves devices and services in an efficient manner.
- Some rich capabilities of a robotic mediator may not be supported directly by the robotic mediator itself, but by the other resource-rich components of the robotic mediator systems (e.g., servers). However, complicated and even varying real-world application development that combines capabilities across devices is prone to extra faults; thus boilerplate codes should be hidden from the developers.

Existing software platforms for networked robots [9] are immediate candidates for software frameworks for robotic mediator systems. Their typical architecture commonly provides three abstraction layers: a communication model, a device model, and an application model (Fig. 1a). The communication model hides the complexity of low-level protocols and provides uniform communication interfaces. The device model encapsulates low-level device-specific details and provides high-level device abstraction. The application model provides a programming and execution environment for applications built by device components. However, we argue that existing robot software platforms are not suitable for robotic mediator systems for the following reasons. First, communication models such as CORBA [10], UPnP [11], and SOAP [12] used in existing platforms are for general purpose and thus inherently heavy and complex. Second, in terms of utilizing performance and functionalities of robotic features, device models based on the general-purpose communication models are also inefficient—e.g., support for streaming, event delivery, exception handling, and Asynchronous Method Invocation (AMI). Finally, and most importantly, we found that the component-oriented application models in existing robot software platforms provide neither an explicit collaboration model at build time nor integrated control architecture at runtime, each of which is crucial to enable various types of collaboration among devices and systems in robotic mediator systems.

To tackle these problems, we have developed a new software framework for robotic mediator systems, called the Integrated Control Architecture for Robotic mediator in Smart environments (ICARS), supporting four abstract models as shown in Fig. 1b. We presented a series of interim results of work-in-progress in previous papers [13–15]. In Suh et al. [13], we introduced a new communication middleware for robotic systems, called PLANET, which presented our design rationales and performance evaluation for the prototype implementation for processing primitive data types. In Suh et al. [14], we presented a conceptual overview of ICARS's system architecture and experimental results obtained in a proof-of-concept scenario. However, the system architecture was not yet cleanly organized into the four abstraction layers shown in Fig. 1a in Suh et al. [14], and thus the collaboration among experimental systems was virtually hard-wired without a general collaborative framework. In Suh et al. [15], we conducted performance evaluations of PLANET, focusing on processing BLOB messages with respect to simultaneous throughput, bandwidth occupancy, and forward congestion.

The contribution of this paper is the complete and thorough technical description of the design, implementation, and evaluation of ICARS by extending and developing the results of our previous work. The focus of our work is twofold. First, we developed a new, efficient, and lightweight communication middleware and a flexible device abstraction model that can effectively support a broad spectrum of data types, control mechanisms, abstraction levels, etc., provided from low-level sensors to high-level legacy systems in robotic mediator systems. Second, we try to provide smart space application developers with full-stack-like abstraction by devising the two-layered application model, a collaboration model and control architecture. In addition to abstracting out the resource limitation of thin robot mediators, this approach enables more organized and reusable abstract components such that programmers of robotic mediator systems can develop a variety of situation-aware applications that collaborate with diverse devices and systems equipped in smart spaces.

The remainder of this paper is organized as follows: Sect. 2 introduces related work. Section 3 presents a high-level

Table 1 Comparison among major software frameworks for robotic mediator systems [15]

Solution	Communication model	Device abstraction	Application model
RT-Middleware [16]	CORBA	RT component	Component composition
MARIE [17]	ACE [18]	MARIE component	
OROCOS [19]	CORBA	OROCOS component	
Orca [20]	Ice [21]	Orca component	
Miro [22]	CORBA	CORBA IDL	
ROS [23]	RPC services	ROS IDL	
Player [24]	Proxy object	Device repository	Service composition
RobAIR [25]	UPnP	OSGi [26] component	
UPnP Robot Middleware [27]	UPnP	UPnP device	
Semantic URS [28]	SOAP	Web service	
PEIS ecology [29,30]	Tuple space	PEIS component	Task planning and self-configuration
ICARS (proposed)	PLANET	PLANET IDL	State transition among composite behavior trees

overview of ICARS' architecture and the functional roles of each layer. Section 4 gives a detailed description of each layer of ICARS. Section 5 presents experimental results associated with the implementation of ICARS. Finally, Sect. 6 concludes this paper.

2 Related works

As illustrated in Table 1, this section covers several categories of abstraction models which might have been adopted for robotic mediator system software frameworks.

2.1 Middleware-based compoment platforms

RT-Middleware, MARIE, OROCOS, Orca, Miro and ROS are all middleware-based component platforms, supporting general collaboration of robots and the environments. Since every constituent device is mapped to a reusable component, application programmers can compose such components to build more complex and composite components. The communication model among the components is based on general-purpose middleware such as CORBA, ACE, and Ice.

Although this approach seems sufficient to serve for robotic mediator system software frameworks in the sense of abstraction, it has some limitations: first, sufficient flexibility or extensibility is not provided for the robotic mediator systems, because any devices in the engaging smart environment must follow the component model, which is strictly dedicated to a specific robot software platform. Moreover, because existing robot software components are provided at levels that are relatively too low, modeling higher-level collaborations just by composing them entails high levels of complexity.

2.2 Service-oriented device abstraction frameworks

Service-oriented device abstraction frameworks like Player, RobAIR, UPnP Robot Middleware, and Semantic URS might be the next candidates for robotic mediator system software frameworks. They adopt communication middleware tailored for their service models and provide device abstraction models based on common object-oriented service interfaces. In these approaches, robots can be regarded as multi-capable service containers, rather than as collaborating counterparts having their own control architectures.

However, these approaches lack collaboration models or robot control architectures, which might be serious limitations as robotic mediator system software frameworks; application programmers in this case have to develop everything they need to collaborate. Especially low level integration of heterogeneous devices would become serious burden to application programmers.

In addition, general purpose service-oriented communication protocols (like SOAP and RESTful interaction) between robots and middleware are not appropriate for robot mediators at all, because a thin and lightweight robot mediator, often emulating their capabilities located in remote servers, might handle different kinds of data from traditional service-oriented systems; it should send and receive continuous sensory event streams and asynchronous controls at once. Similarly, we found that modern publish/subscribing inter-

action between typical devices (like MQTT) dedicated to message queues for quick, but relatively unreliable conversations are not appropriate either for robot mediator system software framework, because such protocols are not robust enough to support sequential information on controls while handling sensory event streams.

2.3 Task planning frameworks

Task planning frameworks like PEIS ecology differ from other service-oriented approaches in that they support task planning and self-configuration. Namely, to achieve a high-level goal, a component generates an action plan to perform the task and dynamically composes components to execute each action sequence by generating a suitable configuration, i.e., a set of connection between components. Thus, a task planning framework is adaptable to highly dynamic environments in which available components are self-configured to cooperate with others to perform a sequence of actions whose control flow can be formally defined such as *fetch and carry* tasks.

When it comes to robotic mediator system software frameworks, however, it is not applicable to the tasks that have complex and ad hoc control flows such as a number of human–robot interactions and composites of sequential and/or concurrent executions. In addition, PEIS does not provide any task selection mechanism that is prerequisite for robotic mediators to mediate situation-aware services between users and smart environments.

2.4 IoT frameworks

Recently, AWS IoT [31], Google Brillo [32], Allseen/Alljoin [33], Eclipse SmartHomes [34] and more software frameworks for Internet of Things and ubiquitous systems [37] have been suggested for practical usage. These trendy IoT frameworks could be deployed for robotic mediator system software frameworks, considering a robotic mediator just as a *Thing*. However, with highly different goals, they have limitations when used as robotic mediator system software frameworks. Most of all, their major target applications depend on sensors rather than actuators; yet, it would be a big burden to the application developers to implement complex behaviors of robotic mediators like moving around subjects and doing missions collaborating with subjects and frameworks at once.

Some of the vendors above also support separate robotic APIs. However, such APIs are dedicated to specific devices (that is, robots) and aim for the device-specific applications like robot arms, but there have been no noticeable attempts of the vendors to link them tightly to IoT frameworks yet. For instance, AWS Robotics [35] is dedicated to a specific device and aiming for the device-specific applications like robot arms, but without any noticeable attempts of the vendors to link them to IoT frameworks yet.

Model-driven approaches for ubiquitous applications [37] introduce well-defined middleware which hides complex details like network supports from developers of ubiquitous system-based applications. However, having a different goal from ours, this paper does not seem to have a device model or separate models like ICARS. It does not have to concentrate on handling moving objects like robots either; thus problems including network disconnection by moving objects do not have to be considered to solve in their works.

3 System overview

Figure 2, a detailed version of Fig. 1a, shows the system architecture of ICARS. ICARS consists of three layers, each of which communicates with adjacent layers based on the communication middleware, called PLANET. While the bottom layer (device framework) is for various devices equipped in the robot and smart environments, the two upper layers (collaboration model and control architecture) are for the mediating server. In the figure, an arrow denotes the direction of an event, data, or a control flow. Note that there is no direct communication among devices including a robot in the physical environment, but instead, all the sensory events are delivered to a smart space server, which controls devices by invoking their remote interfaces. This means that the server can be installed on a separate platform, such as a home server, and can operate remotely through the communication middleware.

In this section, we present a high-level overview of the system and the functional roles of each layer with an introductory robotic mediator system scenario. In this scenario, we assume that there are four devices in the environment and a robotic mediator, as shown in Fig. 3. Utilizing the capabilities provided by the devices, the role of the robotic mediator is to perceive two modes of the user's situation, i.e., *rest mode* and *sleep mode*. In the case of rest mode, the smart space server controls (turns on/off, changes channels, and volume up/down, etc.) the TV and/or plays music according to the voice commands from the users. When a user is located in the bedroom and it is time for the user to sleep, it should turn off the interior lightings, play calm music for a while, and patrol the house by getting around some waypoints.

3.1 Communication framework: PLANET

We developed a new TCP-based remote object invocation framework that enables a robotic mediator to communicate with the various devices in the target environment. For example, in Fig. 3, by invoking a remote object's method, the mediator can receive the user's voice commands through

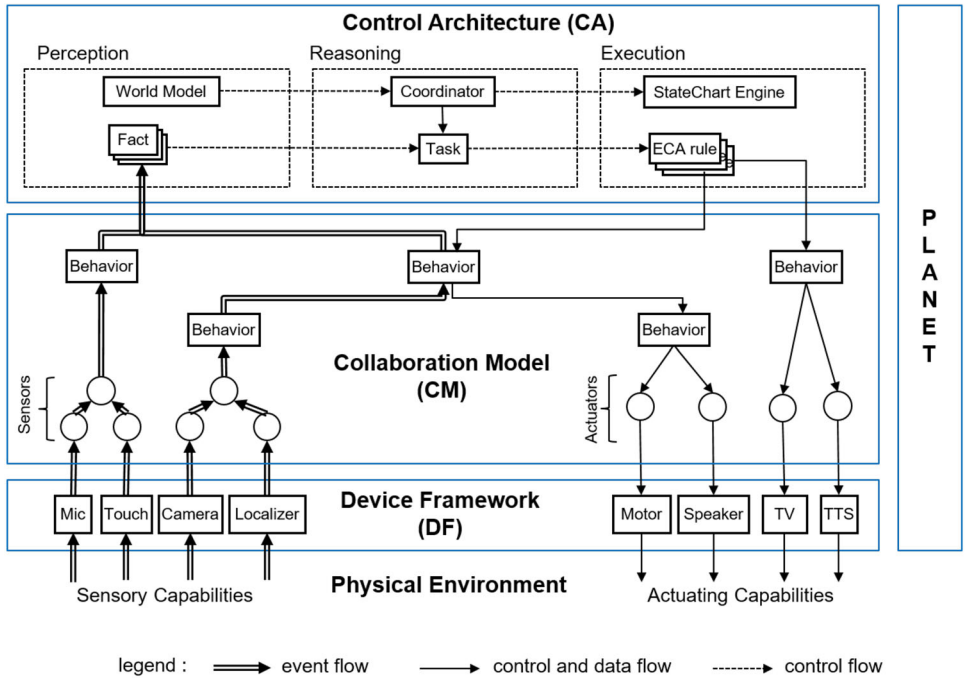


Fig. 2 The overall system architecture of ICARS

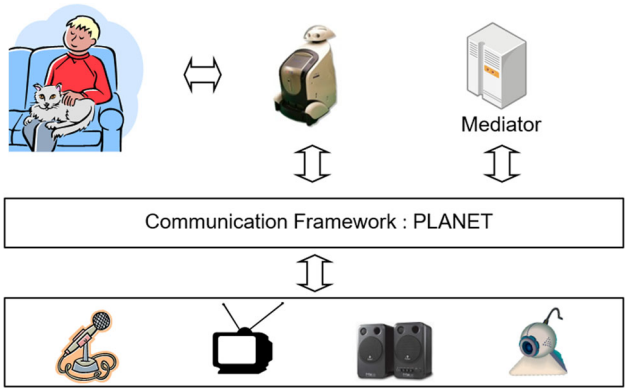


Fig. 3 System configuration of a simple service scenario

the microphone device and control the television device according to the commands. The four types of objects in the ICARS architecture—*Device, Service, Behavior, and Task*—collaborate with each other using PLANET (The four types are explained in more details in later sections.).

3.2 Device Framework (DF)

To enhance the flexibility and extensibility of device interoperability, we separate the process of device interoperation into two phases: device abstraction and device adaptation. Device Framework (DF) provides a device abstraction model to interoperate with heterogeneous devices by extracting the common capabilities of each device's type. The rectangles inside device framework in Fig. 4 represent *Device* objects, each of which is the abstraction for the device's capabilities. Each device in the physical environment exports its capa-

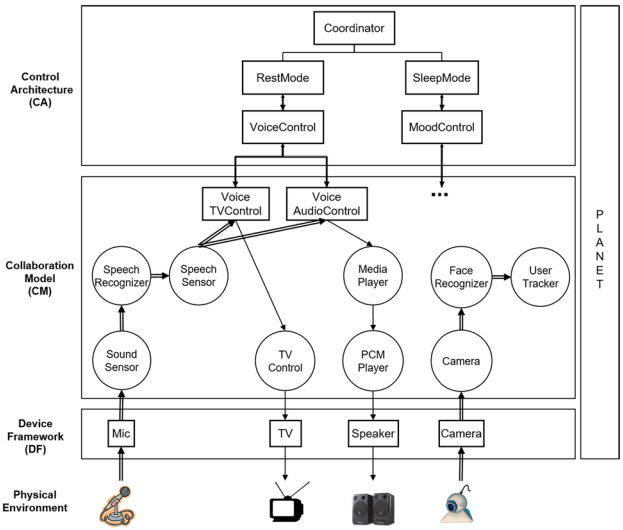


Fig. 4 Objects in each framework of ICARS' implementation of a simple service scenario

bilities to the smart space server by registering its reference to a service registry in the Collaboration Model (CM) layer. During the registration process, the reference of each device object is adapted to a *Service* object, which results in logical pairing between the two. For example, in Fig. 4, the *Mic* device has sensory capabilities for recording sounds. Whenever a microphone detects sounds in a physical environment it delivers the recorded data in the form of a sound event to the *SoundSensor* service object in CM. Because the *TV* device exports its capabilities for controlling a television, the *TVControl* service object in CM can remotely control a television in a smart environment.

3.3 Collaboration Model (CM)

Between device abstraction in DF and actual application development in CA, we let an intermediate layer called Collaboration Model (CM) bridge them, supporting *Service* objects as the circles inside CM in Fig. 4, and a behavior model to support collaboration among the multiple *Service* objects represented. CM consists of two sub-layers, a Service Framework (SF) and a behavior abstraction model. SF provides an adaptive service model and a unified service container for service providers.

The adaptive service model adapts the registered low-level device capabilities into more complex high-level services. The service container provides the upper layer with a uniform service model by integrating and managing the adapted services. For example, in Fig. 4, the *SoundSensor* service detects speech segmentation, records it into PCM data, and publishes a sound event; then, the *SpeechRecognizer* service receives speech data as input and recognizes the speech. In this case, the two services can be combined into a new and more complex service type called *SpeechSensor*, which detects speech segmentations, recognizes the speech, and publishes a speech event containing a symbol of the recognized speech, e.g., "turn on the TV".

However, developing collaboration applications by directly exploiting SF is still non-trivial because application programmers have to understand all the semantics of the low-level service interfaces and events. Moreover, even when they know the semantics, the applications tend, in general, to be complex and non-modular, which results in poor reusability. To address these problems, we introduce an additional layer that encapsulates the low-level details of the service layer and provides high-level abstraction for the collaborative services. A *Behavior*, the smallest unit of collaboration, has its own service logic by which it discovers required service objects and coordinates their execution. For example, in Fig. 4, a *VoiceTVControl Behavior* receives speech events ("turn on/off the TV", "changes channels", "volume up/down", etc.) from a *SpeechSensor Service* object and invokes remote methods ("setPower", "setChannel", "setVolume", etc.) of a *TVControl Service* object accordingly.

3.4 Control Architecture (CA)

Control Architecture (CA) helps application programmers to develop a variety of applications collaborating with diverse services of the robot and smart environments more easily.

Task framework provides a situation-aware application model. A *Task* represents a top-level collaborative application that the robotic mediator should execute in a certain situation. It is modeled as a composite of *Behaviors*, as shown in Fig. 4. The tree structure of a *Task* is organized in such a manner as to achieve its unique goal in the given situation. For example, in Fig. 4, a *RestMode* task has the goal to "provide users with relaxation". In the execution of the *RestMode* task, the *VoiceControl* behavior is executed as a child behavior and subsequently executes two child behaviors, the *VoiceTVControl* behavior and the *VoiceAudioControl* behavior, in a nested manner.

Task Execution Engine provides statechart-based environments for task selection and task execution in which a variety of *Task* objects are loaded and executed according to their own collaboration policy. A *Coordinator* coordinates a number of *Task* objects to be selected and scheduled appropriately according to the current situation and the intent of the user. For example, in Fig. 4, the *Coordinator* will switch the current task, *RestMode*, to another task, for instance the *SleepMode*, considering the user location (bedroom) and the current time.

4 Software framework

In this section, we describe each layer of ICARS with more details.

4.1 PLANET

Considering the lessons learned from field studies and experience in the robotic mediator domain, we argue that a communication framework for robotic mediators in smart environments should support the following robot-specific features:

- *Lightweight* The entities communicating with each other in a robotic mediator system include not only general computing systems, but also embedded systems whose computing resources are relatively limited. Therefore, a lightweight protocol is needed to marshal/unmarshal messages to/from compact binary format to minimize the size of the message payload.
- *Platform independence* The devices in the robotic mediator system can be implemented in various programming languages over the different kinds of OS platforms. Therefore, a communication framework that is not bound to a specific platform and programming language is needed. Specifically, support for heterogeneity with the operating systems and programming languages is required.
- *Disconnected operation* The sensors and actuators in the robot platform may resort to the wireless network because of their mobility. Because wireless communication channels are inherently unreliable and intermittent, the framework should not only support disconnected operations but also provide application developers with a transparent reconnection mechanism.

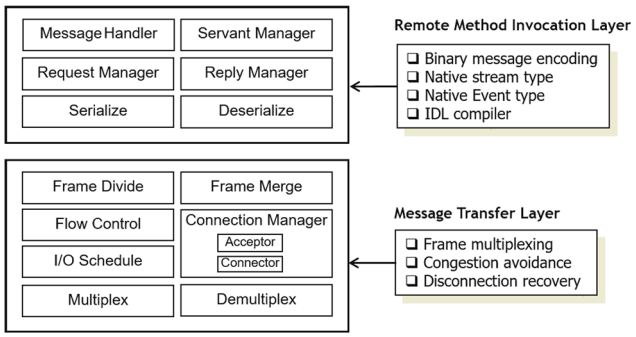


Fig. 5 The two-layered architecture of PLANET

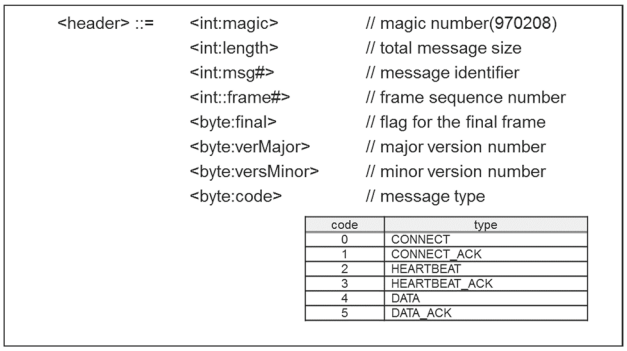


Fig. 6 Encoding protocol of a message header in the message transfer layer

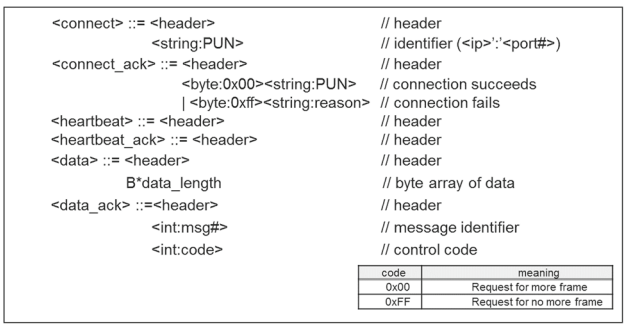


Fig. 7 Encoding protocol of six types of messages in the message transfer layer

- *Support for binary, stream, and event data* In the robotic mediator system domain, a typical message payload transmitted among devices varies from small-size control data to large-size multimedia binary data. Thus, the communication framework needs to consider efficient means of transmitting or streaming large-size multimedia data such as voice and image. Moreover, the framework should support event delivery of complex sensory data from diverse sensors in the environment.
- *Support for thin client/server* Smart environments consist of a broad spectrum of device platforms that have diverse computing power. The framework needs to provide a flow control mechanism to regulate the performance gap between the client device and the server device.
- *Support for Asynchronous Method Invocation (AMI)* Typically, a robot may perform operations of a long duration (multi-seconds) to complete a mission, such as robot navigation and speech synthesis. Thus, it is required that a client should be able to handle these operations asynchronously so that the need to block it until operations are completed does not arise.

Considering these requirements, therefore, we propose a new communication framework, i.e., PLANET. PLANET is a general-purpose communication middleware that is based on the object-oriented remote object's method invocation. However, it is distinguished from existing middleware in that it is tailored to accommodate robot-specific features for a robotic mediator collaborating with smart environments. Figure 5 shows the layered architecture of the framework and the key components of each layer.

4.1.1 Message transfer layer

The role of this layer is to support reliably the connection with multiple communication peers and efficiently exchange messages bi-directionally through the connections. Figure 6 shows the encoding structure of a message header in the message transfer layer. The *magic* field is a four byte integer value to discriminate PLANET messages from others. The *length* field is the total message size, including the header itself. The *msg*# a unique identifier for a PLANET message that is needed to identify frames in the same message. The *frame*# a frame sequence number divided from a message. The *final* field signifies whether the frame is the final frame of a message. Followed by version numbers, the code indicates the type of a message. There are six types of messages in the transport layer: CONNECT, CONNECT_ACK, HEARTBEAT, HEARTBEAT_ACK, DATA, and DATA_ACK.

Figure 7 shows the encoding structure of messages for each of the six message types in the message transfer layer. The *Connection Manager* is in charge of connecting/accepting connection requests to/from servers/clients. A client sends a CONNECT message to a server to connect with it, and the server sends back a CONNECT_ACK message to the client as a means of handshaking. A ⟨*connect*⟩ is a message from a client to a server to request establishment of a PLANET connection. The *PUN* field is the unique identifier of the client, which corresponds to a string composed of ⟨*ip*⟩':'⟨ *port*#⟩ ⟨*connect_ack*⟩ is a message from the server that receives a ⟨*connect*⟩ message from a client in reply. After receiving a ⟨ *connect*⟩ message, the server can accept or deny the request for connection. In the case of acceptance, the server sends the byte code *0x00* followed by

Table 2 Encoding the protocol of message payloads for data types supported in PLANET

Type name	Encoding (Backus–Naur Form)	
Byte	⟨byte⟩::= b8	# 8 bit
Short	⟨short⟩::= b16 b8	# 16 bit, big-endian
Int	⟨int⟩::= b32 b24 b16 b8	# 32 bit, big-endian
Long	⟨long⟩::= b64 b56 b8	# 64 bit, big-endian
Float	⟨float⟩::= b32 b24 b16 b8	# floating number
Double	⟨double⟩::= b64 b56 b8	# floating number
Boolean	⟨boolean⟩::= 0x01 | 0x02	# (true | false)
String	⟨string⟩::=⟨int:length⟩ b*length	# UTF−8 encoded
Binary	⟨binary⟩::=⟨int:length⟩ b*length	# BLOB
Enum	⟨enum⟩::=⟨int:ordinal⟩	
Valued	⟨Valued⟩::= {⟨field⟩}	
	⟨field⟩ ::= 0x00	
	| 0x16 ⟨short:ref_number⟩	# to other field
	|⟨typed⟩	# type
Remote	⟨remote⟩::=⟨string:identifier⟩	# identifier
	⟨string:path⟩	# identifier
Stream	⟨stream⟩::=⟨int:channelId⟩	
Exception	⟨exception⟩::=⟨string:exception_typecode⟩⟨string:details⟩	
event	⟨event⟩::=⟨Sequence⟨string⟩:event_type_class_names ⟩	
	⟨int:event_property_count⟩	
	(⟨string:event_property_namei⟩⟨typed:Event_property_value⟩)*	
	⟨Event_property_count⟩	
Sequence	⟨sequence⟩::=⟨int:count⟩ (⟨nestable:elm_data⟩)*count	

its own PUN; otherwise, it sends the code *0xff* followed by the reason for denial. Robots and devices in smart environments usually resort to unstable wireless networks, which are prone to intermittent disconnection. The *Connection Manager* periodically exchanges heartbeat messages with all the connected devices to detect unexpected disconnection. On detecting an abrupt disconnection, the Connection Manager notifies the occurrence of failure to the listeners so that they can handle the exception in their own way. Moreover, it supports graceful recovery from failures by periodically attempting to connect to the disconnected devices until they are reconnected. Finally, it also notifies the listeners about the new connection.

After establishing a connection, the two connected peers exchange data through the connection. More specifically, the client/server sends requests/replies to the server/client in the form of DATA messages. A ⟨*data*⟩ is a message exchanged between a client and a server after a connection is established by the ⟨*connect*⟩ message and the ⟨ *connect_ack*⟩ message. A ⟨*data_ack*⟩ is a message from the receiver of a ⟨*data*⟩ message to acknowledge receipt of the ⟨*data*⟩ message. In devices such as robots with both sensors and actuators, both large-size binary multimedia messages and small-size control messages are usually processed simultaneously in the same connection. In such cases, the small-size messages may suffer from congestion owing to the large-size messages. To avoid the congestion, every message is divided/merged into/from the fixed-size (16 K) frames by the *Frame Divider*/*Merger*. Subsequently, the *I*/*O scheduler* multiplexes/demultiplexes the frames based on the *frame*# in the message header to prevent exclusive use of a connection. Furthermore, the *Flow Controller* controls the flow of message transfer by applying a handshaking protocol in the course of message exchanges based on the *control code* field of the ⟨*data_ack*⟩ message.

4.1.2 Remote method invocation layer

This layer provides a typical mechanism (employed in RPC systems) for remote method invocations between remote objects: stubs and servants. A stub for a remote object acts as a client's local proxy for the remote object. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object. A servant for a remote object implements the same set of interfaces that a remote object implements. Table 2 shows the encoding protocol of message payloads for data types supported in PLANET. The *Serializer*/*Deserializer* is in charge of encoding/decoding message payloads. PLANET supports various user-defined data types as well as primitive data types. The primitive types include *void, Boolean, byte, short, int, long,*

```
<call> ::= <header>                                  // header
           <string:path>                             // servant path
           <method>                                  // method name
           <argumets>                                // arguments
<method>  ::= <string:declaring_type_name>           // object type name
              <string:method_name>                   // method name
<arguments> ::= <int:argument_count>                 // argument count
                (<typed:argumenti>)*argument_count   // arguments

<reply> ::= <header>                                 // header
            (  ( <typecode:declaring_result_type>    // return type
                 <typed:result> )                    // return value
             | 0x01 )                                // void return
<error> ::= <header>                                 // header
            <exception>                              // exception
<notify> ::= <call>
<stream> ::= <header>                                // header
```

Fig. 8 Encoding the structure of a message header in the message transfer layer

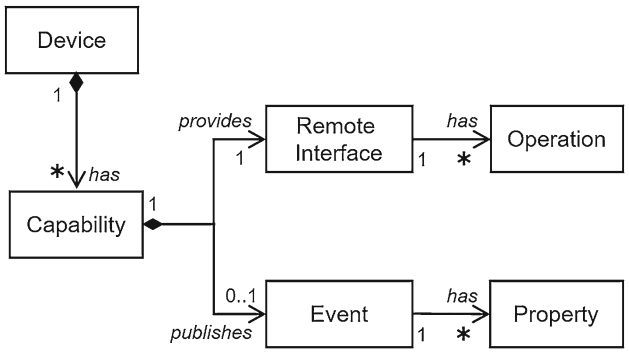


Fig. 9 Device abstraction model in DF of ICARS [14]

float, double, string, binary, and *stream* types. Sequence types for every data types are also supported. The user-defined types include *enum, valued, event, exception,* and *remote*. Among the user-defined types, *enum, valued*, and *event* types follow "pass-by-value" when they are passed as arguments or return values of a method invocation, whereas the *remote* type follows "pass-by-reference". Support for streaming is very important in smart environments as it reduces the response time for processing multimedia data. Thus, PLANET supports *stream* type as a native primitive type. Once a set of stream data is passed as an argument or the return value of a remote method invocation, a channel for a remote input stream is constructed over the network between the local caller and the remote callee. Subsequently, the receiver of the stream data can read data from the channel as much as it needs by receiving the *stream* message. Finally, events are the most efficient method of delivering sensing information from sensor devices in robots and environments. PLANET also supports *event* type. An event consists of a set of properties. Once a set of event data is passed as an argument or a return value, the receiver can get the value of the properties using the name of the property as a key.

Figure 8 shows the encoding structure of a message header in the message transfer layer. *Message Handler* encodes/decodes the header of messages in the RMI layer. There are five types of messages in the RMI layer: CALL, REPLY, ERROR, NOTIFY, and STREAM. The CALL message and the NOTIFY message correspond to request messages. Thus, they are encoded/decoded by the *Request Manager*. As depicted in Fig. 8, the request messages contain the servant path of the target remote object as well as information about the function call such as method name and argument lists. The encoding of a NOTIFY message is the same as that of a CALL message. The only difference is that the NOTIFY message is a one-way method call that does not wait for a reply to the call. In the case of CALL messages, the *Request Manager* registers the request to the pending request list using the *request_id* of the message as a key. The REPLY message and the ERROR message belong to the reply messages handled by the *Reply Manager*. The *Reply Manager* encodes/decodes the reply messages and passes them to the *Serializer/Message Handler*. The REPLY message is a return value for a remote method call and the ERROR message contains the exception object thrown by the remote method. The *Message Handler* passes the return value/exception object to the caller object by referring to the pending requests list.

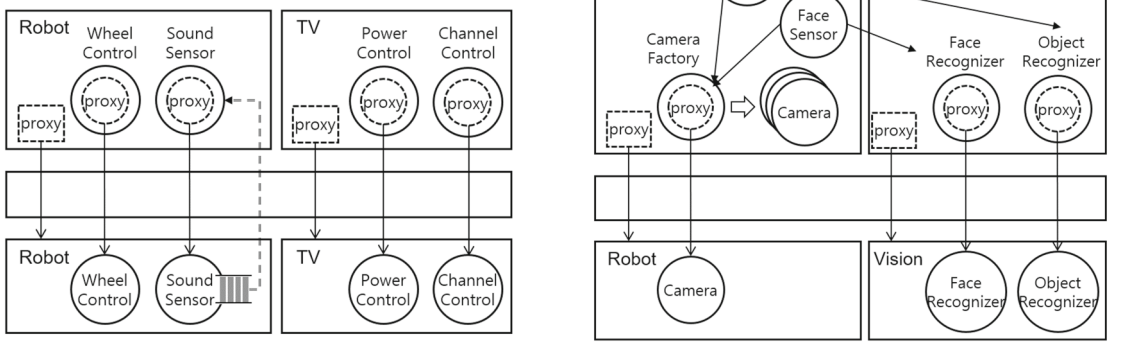
4.2 Device Framework (DF)

Device abstraction in DF enables device providers to focus on implementing the minimal primitive functionalities of each device based on common device models without considering interoperability issues. As shown in Fig. 9, a *Device* is composed of a number of *Capabilities*. A *Capability* is required to provide a *Remote Interface* that has *Operations,* each of which performs an actuating function. Optionally, a *Capability* can publish an *Event* that has *Properties,* each of which contains sensing data. Consequently, the proposed device abstraction model does not impose any restriction on data types, control methods, abstraction levels, and others, but simply specifies remote method interfaces to perform functionalities and event types to deliver sensing information. Consequently, device providers can model a broad spectrum of devices, i.e., from low-level embedded sensors to high-level legacy systems. Furthermore, the device model can exploit robot-specific features such as native support for stream and event types by using PLANET IDL.

4.3 Collaboration Model (CM)

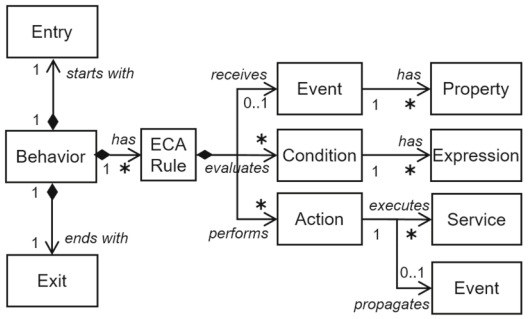
4.3.1 Service Framework (SF)

Device adaptation in SF adapts (extends/shares/composes) the device's capabilities into the more complex high-level services. Figure 10a depicts an example of device adaptation. When a device exports its capabilities to the device



(a) Examples of device adaptation **(b)** Examples of device capabilities augmentation

Fig. 10 Examples of service framework



(a) Behavior abstraction model of ICARS

```
protected void entry() {
   m_head = findService(MoveHead.class, "head");
   m_localizer = findService(SoundSensor.class, "sound");
   m_localizer.subscribe(this);
   m_localizer.start();
}
protected Event handleEvent(Event event) {
   if ( event instanceof SoundLocalized ) {
      SoundLocalized sle = (SoundLocalized)event;
      m_head.turn(sle.getSoundLocation());
   }
   return null;
}
protected void exit() {
   m_localizer.unsubscribe(this);
   m_localizer.stop();
   m_head.stop();
}
```

(b) Code fragments of *LookAtSound* behavior

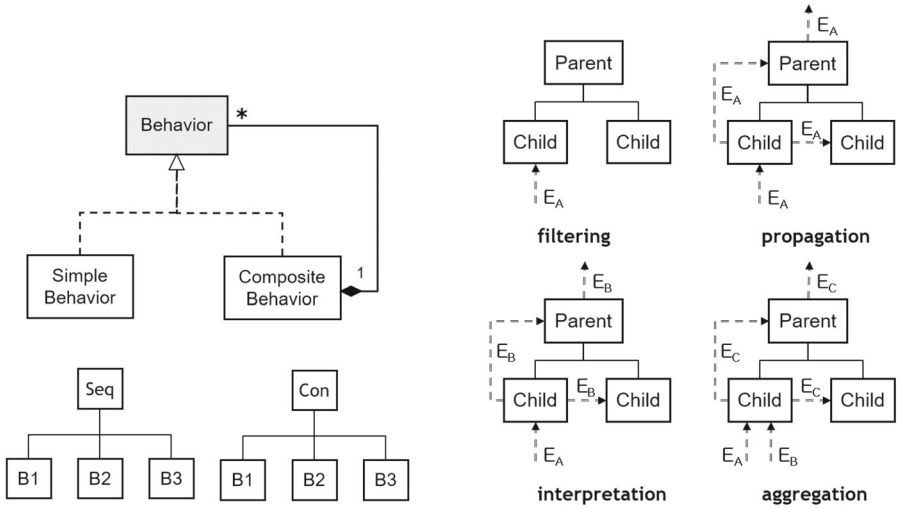
Fig. 11 Behavior abstraction model and its example

registry of SF, SF creates a service container for adaptor objects by wrapping the remote reference of each capability of the device to be inserted. In the case of sensory capabilities, the adaptor object subscribes to the event channel of the sensor to receive events. The adaptation process may involve device capabilities augmentation, as shown in Fig. 10b. A robot device provides only a *Camera* capability, which is then adapted to a *CameraFactory* service to make it sharable. Another device also provides vision-based recognition capabilities. By combining these two device capabilities, two new high-level services are augmented in the service container. *Object(Face)Sensor* captures an image from a camera, recognizes objects (faces) in the image, and publishes an event containing information about the objects (faces).

4.3.2 Behavior abstraction model

Whereas a service is a passive object that performs the operations invoked through remote interfaces, a behavior is an active object that has its own lifecycle. As shown in Fig. 11a, a behavior starts with execution of an entry function. While running, it reacts to a number of ECA rules. Specifically, when it receives an event (*Event*), it evaluates the expressions of the condition (*Condition*) whose values are updated from properties of the event. According to the evaluation result, it performs actions (*Action*) such as service invocations and event propagation. Finally, it ends with the exit function. In the entry, a behavior typically tries to discover actuator services to control and it subscribes to sensor services to receive events therefrom. In contrast to entry, a behavior releases all the resources obtained from actuator services and unsubscribes from sensor services in the exit function. For example, *LookAtSound* (see Fig. 11b) is a behavior of a robot that turns its head to the direction of a sound source. In the entry function, it discovers a *MoveHead* service and a *SoundLocalizer* service and binds them to its member variables. As the *SoundLocalizer* is a sensor, it subscribes to the *SoundLocalizer* so that it can receive sound events. In the *handleEvent* function,

Fig. 12 Composite behavior model



(a) Structure and execution semantics of a composite behavior **(b)** Event propagation among behaviors within a composite behavior [14]

the behavior reacts to a single ECA rule represented by an if clause. The rule checks whether the received event is an instance of *SoundLocalized* event or not and, if true, it turns the head to the direction of the sound source obtained from the event property. In the exit function, it unsubscribes from and releases the services.

Figure 12a depicts the behavior-based collaboration model with a composite structure. The non-terminal nodes in a tree structure are composite behaviors whose execution semantics are either sequential or concurrent. Whereas in the case of the sequential composites every child node is executed in a predefined order, all child nodes start executing in parallel in the concurrent composites. The life cycle of a child behavior is completely dependent on its parent's life cycle. Specifically, any child behavior can only be created by a parent and, when that parent is destroyed, it is automatically destroyed as well.

Figure 12b shows event propagations among nodes in a behavior tree. The general rule is as follows: "Every event that a node propagates will be sent to its parent and all the siblings". The rule implies that, in a behavior tree, every event originated from a node is propagated to ancestors and their siblings until it reaches the root node. A behavior can perform some kinds of event processing in the course of event propagation. In this way, low-level events from the bottom-level nodes are interpreted to more abstract events as they are propagated toward the top-level nodes of a tree.

- *Filtering* Does not propagate the received event.
- *Interpretation* Interprets an event, converts it to another event type, and propagates the new event.
- *Aggregation* Aggregates multiple events to another event type and propagates the new event.

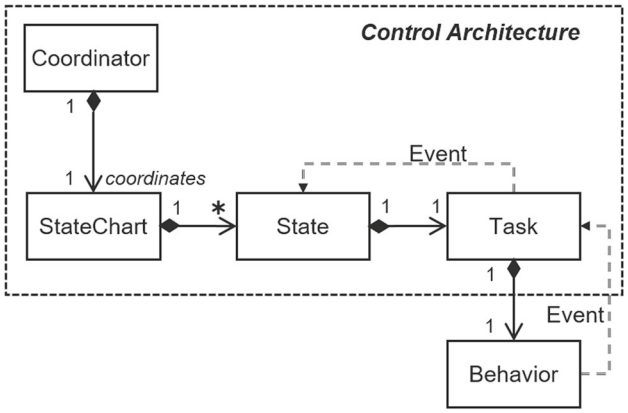


Fig. 13 Control architecture [14]

4.3.3 Control architecture

Figure 13 presents the control architecture of ICARS, in which every situation-aware application is represented as a statechart diagram. Each state in the statechart is bound to a *Task* that encapsulates a composite behavior for the robotic mediator to perform in a situation. A *Coordinator* schedules the state transition of the statechart diagram according to task events generated by interpreting the top-level events of each composite behavior, which enables the robotic mediator to cope with various situations.

Figure 14 shows the architecture of a runtime engine that executes situation-aware applications. The tree structure of a composite behavior of each task in the situation-aware application is described in a collaboration policy description file, as shown in Fig. 15. In the runtime, the Collaboration Manager loads a collaboration policy from the Policy Repository, creates a *Coordinator* instance by feeding a policy instance

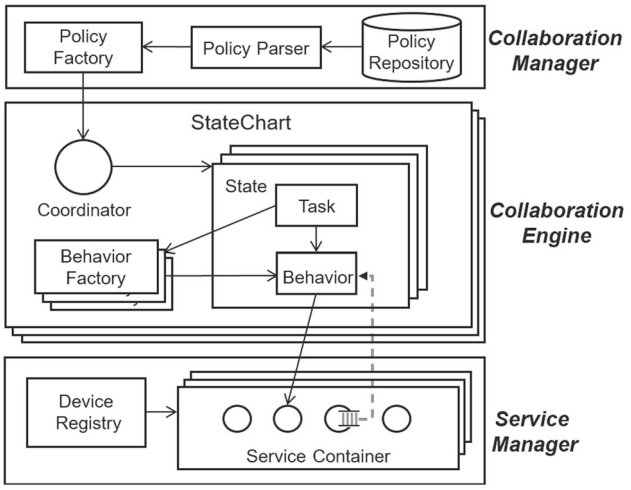


Fig. 14 Architecture of runtime task execution engine [14]

```
<coordinator id = "experiment">
  <task id="observation"/>
  <task id="rest mode"/>
    <behavior id="VoiceControl">
      <sequential>
        <behavior type="icars.GotoUser"/>
        <concurrent>
          <behavior type="icars.VoiceTVControl"/>
          <behavior type="icars.VoiceAudioControl"/>
      </concurrent>
    </sequential>
  </task>
  <task id="sleep mode">
    <behavior id="NightWatch">
    <sequential>
      <behavior type="icars.GotoUser"/>
        <sequential>
          <behavior type="icars.MoodControl">
          <behavior type="icars.PatrolHome">
        </sequential>
     </behavior>
  </task>
</coordinator>
```

Fig. 15 Example of a collaboration policy

parsed by the Policy Parser to the Policy Factory, and starts the *Coordinator* instance. The *Coordinator* instance constructs a statechart diagram by assigning a state to each *Task*. After constructing the statechart diagram, the *Coordinator* initializes the statechart to an initial state. The initial state creates an instance of the assigned *Task* and starts the *Task* instance. Once the *Task* instance has started, it again creates an instance of the top-level composite behavior and starts the behavior instance. In this way, the nested composite behavior tree will be unfolded down to terminal nodes. During the process, the behavior instances execute various collaborative services as well as propagate situation events. Accordingly, the *Coordinator* can provide users in smart environments with situation-aware collaboration services by transiting to the state appropriate for the user's intent.

5 Results and discussion

In this section, we describe the overall process of developing an experimental environment that provides the situation-aware application introduced in Sect. 3 using the ICARS framework. The environment is used to verify the functionalities and effectiveness of the proposed framework according to the following three criteria: (1) Can the diverse devices in the robot and the environment be implemented and integrated in a flexible manner? (2) Can the variety of services required to compose collaborative services be provided in an extensible manner? (3) Can the applications performing high-level collaborative services be developed with ease?

However, meeting these criteria, all together at once, is hard to measure numerically for programming models as in usual model-driven programming researches. Thus in this paper, we try to show the feasibility of our system, by including the real-world deployment example instead. We believe that this effort justifies flexibility, extensibility and structural usage of the real-world scenario.

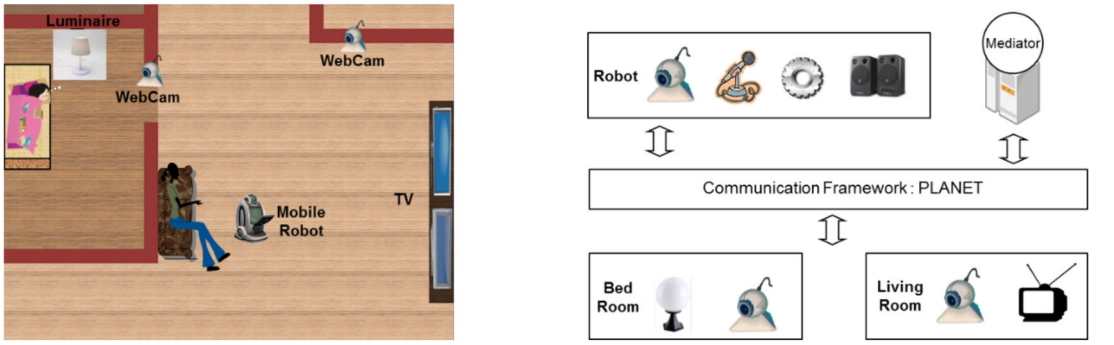
5.1 Device implementation

As shown in Fig. 16a, the experimental environment is a model of a house with one bedroom and one living room. Each room has two static devices whose positions are fixed to specific locations. The house also has a mobile platform that can move around the rooms freely, i.e., a robot, equipped with four devices. With the device abstraction tool provided by DF of ICARS, we, in the role of device providers, defined a device type for each device and implemented the capabilities of the device type. Table 3 summarizes the device types and their capabilities for the devices in the experimental house. At runtime, each device is represented by a *Device* object, which is registered to SF of the mediator through PLANET, as shown in Fig. 16b.

5.2 Service implementation

During the device abstraction phase, all the device types defined in DF are exported to the SF. We, in the role of service providers, defined and implemented a primitive service type for each exported device type by wrapping the device's capabilities as shown in Table 4.

In addition to the primitive service types, we supplement SF with additional service types that extend the functions of the other service types, as shown in Table 5. At runtime, during the device registration phase, SF creates 15 *Service* objects, 1 for each of the registered *Device* objects and the complex service types, thereby enabling *Behavior* objects to select appropriate services from the *Service* objects in the service container of SF.



(a) Bird's-eye view of the experimental setup **(b)** System configuration after device abstraction

Fig. 16 Experiment environment and system configuration

Table 3 Device types for the devices in the experimental house

Place	Device	Capability	Function
BedRoom	WebCam	ImageCapturer	Captures camera images in JPEG image format
	Luminaire	LightControl	Controls an interior lighting
LivingRoom	TV	PowerControl	Controls the TV power
		ChannelControl	Controls the TV channels
		VolumeControl	Controls the TV volume
	WebCam	ImageCapturer	Captures camera images in JPEG image format
Robot	Mic	SoundRecorder	Detects a sound, records it in PCM format, and publishes a SoundEvent
	Speaker	PCMPlayer	Plays PCM data through a speaker
	Wheel	Navigation	Navigates a robot to a target position
	WebCam	ImageCapturer	Captures camera images in JPEG image format

5.3 Behavior implementation

Based on the behavior abstraction model, we defined five primitive behavior types encapsulating collaboration among *Service* objects and implemented a behavior factory for each behavior type, as shown in Table 6.

5.4 Control architecture implementation

In the role of application programmer, we define and implement a self-contained task (*Observation*) to detect the user's current situations and two tasks with composite behavior tree structure (*RestMode* and *SleepMode*), each of which is responsible for handling its own situation, as shown in Fig. 17a. Finally, we define a state transition diagram among the three tasks as a control architecture for the experimental application, as shown in Fig. 17b. At runtime, the application is started by entering the initial state, i.e., executing the *Observation* task. The *Observation* task infers the user's current situation by analyzing his/her contexts, such as current location and daily schedule pattern.

Figure 18 illustrates the execution of the experimental application. The *Observation* task propagates a *TimeToRest* event when it detects that a user is sitting on the bed or the sofa to take rest. On receiving a *TimeToRest* event, the coordinator makes a state transition from the *Observation* task to the *RestMode* task (see Fig. 17b). Figure 18a shows snapshots of the execution of a composite behavior tree in the *RestMode* task, i.e., the *VoiceControl* behavior. Specifically, after a mobile robot arrives at the user's current location (*GoToUser*), it subsequently supports the user with watching TV (*VoiceTVControl*) or listening music (*VoiceAudioControl*) according to the user's voice commands.

In the case where the *Observation* task detects that a user is lying on his/her bed at night, it propagates a *TimeToSleep* event, which causes a state transition from the *Observation* task to the *SleepMode* task (see Fig. 17b). Figure 18b shows snapshots of the execution of a composite behavior

Table 4 Primitive service types for the exported device types

Service	Device	Function
Camera	WebCam	Extends capabilities of a WebCam to be sharable
TVControl	TV	Wraps and relays capabilities of a TV
SoundSensor	Mic.	Wraps and relays capabilities of a Mic
PCMPlayer	Speaker	Wraps and relays capabilities of a Speaker
WheelConrol	Wheel	Wraps and relays capabilities of a Wheel
LightControl	Luminaire	Wraps and relays capabilities of an Illuminator

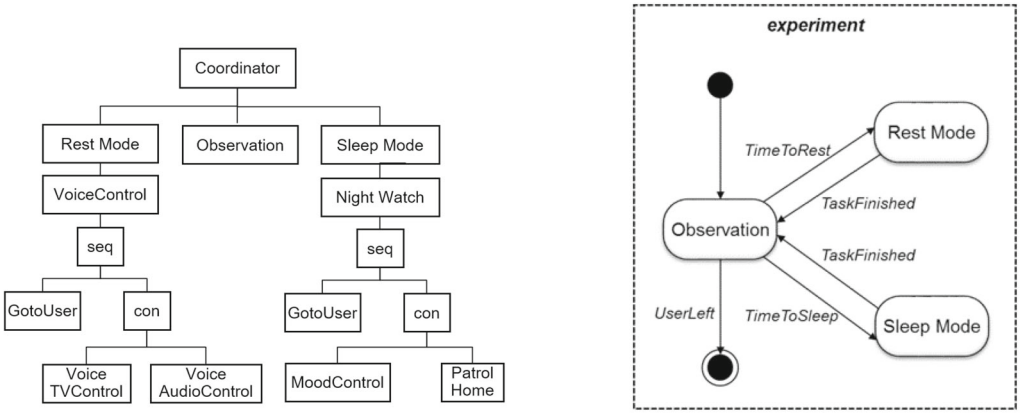
Table 5 Complex service types newly augmented in SF

Service	Device	Function
MotionSensor	Camera	Detects a motion and publishes a MotionEvent
FaceRecognizer	Camera	Recognizes identifiers of human faces in a JPEG image
UserTracker	FaceRecognizer	Keeps track of users locations with multiple cameras
SpeechRecognizer	SoundSensor	Recognizes symbols of human speech
WheelConrol	Wheel	Wraps and relays capabilities of a Wheel
LightControl	Luminaire	Wraps and relays capabilities of an Illuminator
SpeechSensor	SpeechRecognizer	Detects a speech and publishes a SpeechEvent
MediaPlayer	PCMPlayer	Plays a media file by converting it to PCM data
Navigation	WheelControl	Navigates a mobile robot to a target position

Table 6 Primitive behavior types defined in CM

Behavior	Service	Function
GoToUser	UserTracker	Navigates a mobile robot to the current position of a user
	Navigation	
VoiceTVControl	SpeechSensor	Controls a TV according to a user's voice command
	TVControl	
VoiceAudioControl	SpeechSensor	Plays a media file according to a user's voice command
	MediaPlayer	
MoodControl	LightControl	Adjusts illumination intensity of interior lightings and plays mood music according to the current situation
	MediaPlayer	
PatrolHome	MotionSensor	Patrols the house by getting around some designated waypoints
	Navigation	

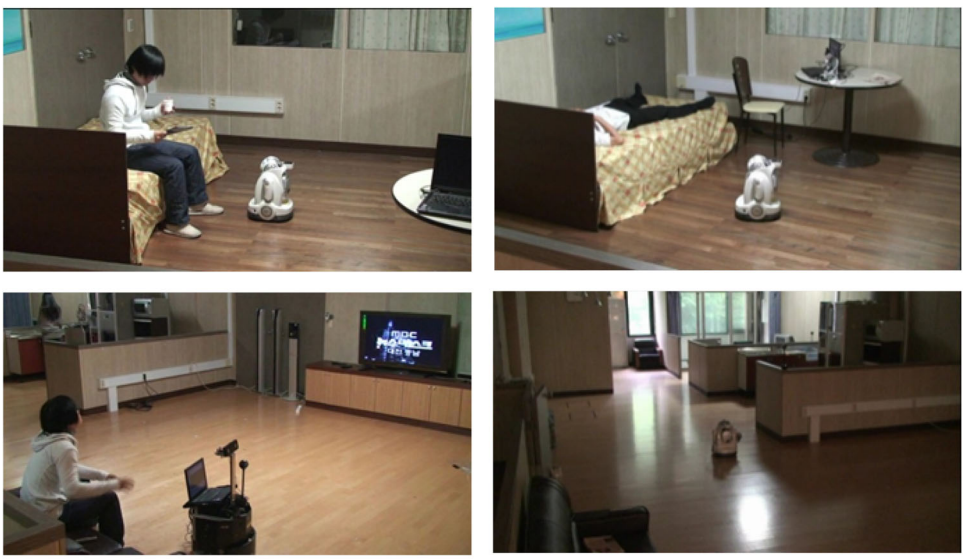
tree behavior in the *SleepMode* task, i.e., the *NightWatch* behavior. Specifically, after the mobile robot arrives at the user's current location (*GoToUser*), it subsequently turns off the interior lightings, plays calm music for a while (*MoodControl*), and then finally patrols the house by getting around some waypoints (*PatrolHome*).



(a) Composite behavior tree structure for the experimental application **(b)** Statechart diagram among the tasks of the experimental application

Fig. 17 Experimental application model

Fig. 18 Execution modes in experimental application



(a) Execution of rest mode task **(b)** Execution of sleep mode task

5.5 System deployment and integration

As described above, ICARS interacts with various sensory and/or actuating capabilities of a robotic mediator system by implementing *Device* objects for each device in the surrounding environment. Thus, as shown in Fig. 16b, the typical system configuration generally consists of a mediating server and one or more device managers when deploying ICARS to every environment such as a home and an office. A device manager is a program that manages life cycles of *Device* objects for a set of devices that need to be managed together for some administrative reasons. Therefore, integrating with legacy systems such as home network systems (e.g., OSGi), the IoT (Internet of Things) systems and commercially available robotic platforms (e.g., ROS) will be implemented on the basis of *Device* objects. Namely, device providers, as installers of ICARS to an environment, define device models and implement *Device* objects for capabilities provided by legacy systems using DF of ICARS. ICARS do not care about the internal communication protocols between *Device* objects and legacy systems, which means that *Device* objects can be integrated with legacy systems using their proprietary protocol.

In Fig. 16b, the system is configured as a centralized architecture with a single mediating server. However, it should be noted that all types of ICARS objects are distributed objects that are capable of communicating with the top of PLANET communication middleware. Thus, the system can be configured to be scaled well by distributing *Service* objects, *Behavior* objects and *Task* objects to multiple servers to form a single logical mediating server.

6 Conclusions

The experience of full automation without explicit user direction may induce anxiety among smart space users. A promising approach to address user concerns is the use of robots as personal assistants who can act as mediators between the user and their environments. However, existing frameworks bonding robots and smart space systems are unsatisfactory due to the complexity arising from interrelations between multiple components, leading to less reliable smart spaces. In this paper, we proposed a new software framework, ICARS, for robotic mediators collaborating with smart environments. Based on our experimental results, ICARS enables device providers to flexibly implement and integrate diverse devices in robots and environments, service providers to adapt a variety of services required for collaborative services in an expandable manner, and application programmers to easily develop high-level collaborative applications. The current statechart-based specification of task scheduling supports only static scheduling, which is poorly equipped to handle dynamic situations such as changes in the user's intent or exceptional failures. Thus, future research should develop adaptive task coordination based on dynamic task scheduling to support improved reliability in smart spaces.

Acknowledgements This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2015R1D1A1A09061480).

References

1. Weiser M (1991) The computer for the 21st century. Sci Am 265(3):94–104
2. De Carolis B, Cozzolongo G (2009) Interpretation of user's feedback in human-robot interaction. J Phys Agents 3(2):686–695
3. Ishii H (2008) Tangible bits: beyond pixels. In: Proceedings of the 2nd ACM international conference on tangible and embedded interaction
4. Lee D, Yamazaki T, Helal S (2009) Robotic companions for smart space interactions. IEEE Pervasive Comput 8(2):78–84
5. Linder N, Maes P (2010) LuminAR: portable robotic augmented reality interface design and prototype. In: Adjunct proceedings of the 23nd annual ACM symposium on user interface software and technology
6. Bien ZZ et al (2008) Intelligent interaction for human-friendly service robot in smart house environment. Int J Comput Intell Syst 1(1):77–93
7. Coradeschi S, Saffiotti A (2006) Symbiotic robotic systems: Humans, robots, and smart environments. IEEE Intell Syst 21(3):82–84
8. Cozzolongo G, De Carolis B, Pizzutilo S (2007) Social robots as mediators between users and smart environments. In: Proceedings of the 12th ACM international conference on intelligent user interfaces
9. Mohamed N, Al-Jaroodi J, Jawhar I (2009) A review of middleware for networked robots. Int J Comput Sci Netw Secur 9(5):139–148
10. Siegel J (1998) OMG overview: CORBA and the OMA in enterprise computing. Commun ACM 41(10):37–43
11. Ahn SC et al (2005) UPnP approach for robot middleware. In robotics and automation, 2005. In: Proceedings of the IEEE international conference on ICRA
12. Kim BK et al (2005) Web services based robot control platform for ubiquitous functions. In: Robotics and automation, in proceedings of the IEEE international conference on ICRA
13. Suh Y-H, Lee K-W, Cho E-S (2012) A communication framework for the robotic mediator collaborating with smart environments. J Inst Electron Eng Korea CI 49(2):75–82
14. Suh Y-H et al (2012) ICARS: integrated control architecture for the robotic mediator in smart environments: a software framework for the robotic mediator collaborating with smart environments. In: Proceedings of high performance computing and communication and 2012 IEEE 9th international conference on embedded software and systems (HPCC-ICESS)
15. Suh Y-H, Lee K-W, Cho E-S (2013) A device abstraction framework for the robotic mediator collaborating with smart environments. In: Proceedings of IEEE 16th international conference on computational science and engineering
16. Ando N et al (2005) RT-middleware: distributed component middleware for RT (robot technology). In: Intelligent robots and systems, in proceedings of IEEE/RSJ international conference on intelligent robots and systems
17. Cote C et al (2006) Robotic software integration using MARIE. Int J Adv Robot Syst 3(1):55–60
18. Schmidt DC (1994) ASX: an object-oriented framework for developing distributed applications. In: C++ conference
19. Bruyninckx H (2001) Open robot control software: the OROCOS project. In: Proceedings of IEEE international conference on robotics and automation
20. Brooks A et al (2007) Orca: a component model and repository. Software engineering for experimental robotics. Springer, Berlin, Heidelberg pp 231–251
21. Henning M, Spruiell M (2003) Distributed programming with ice. ZeroC Inc., Jupiter
22. Utz H et al (2002) Miro-middleware for mobile robot applications. IEEE Trans Robot Autom 18(4):493–497
23. Quigley M et al (2009) ROS: an open-source robot operating system. In: Proceedings of ICRA workshop on open source software
24. Gerkey B, Vaughan RT, Howard A (2003) The player/stage project: tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th international conference on advanced robotics
25. Calmant T et al (2012) A dynamic sca-based system for smart homes and offices. In: Proceedings of international conference on service-oriented computing
26. Alliance O (2003) Osgi service platform, release 3. IOS Press Inc, Amsterdam
27. Ahn SC et al (2006) Requirements to UPnP for robot middleware. In: Proceedings of IEEE/RSJ international conference on intelligent robots and systems
28. Yu W et al (2009) Design and implementation of a ubiquitous robotic space. IEEE Trans Autom Sci Eng 6(4):633–640
29. Saffiotti A, Broxvall M (2005) PEIS ecologies: ambient intelligence meets autonomous robotics. In: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies
30. Saffiotti A et al (2008) The PEIS-ecology project: vision and results. In: Proceedings of 2008 IEEE/RSJ international conference on intelligent robots and systems
31. What is AWS IoT?—AWS IoT. http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html. Accessed Apr 5 2018
32. Google Brillo. https://developers.google.com/brillo/. Accessed Apr 5 2018
33. Allseen alliance. https://allseenalliance.org/. Accessed Apr 5 2018

34. Eclipse SmartHome—a flexible framework for the smart home. http://www.eclipse.org/smarthome/. Accessed Apr 5 2018
35. AWS Robotics. https://github.com/awslabs/simplerobotservice. Accessed Apr 5 2018
36. Cho E-S (2017) Toward more reliable intelligent environments. In: Proceedings of 6th workshop on the reliability of intelligent environments (WoRIE)
37. Rodriguez-Dominguez C, Ruiz-Lopez T, Benghazi K, Noguera M, Luis GJ (2013) In: Proceedings of 9th international conference on, intelligent environments (IE)