Journal of Information Processing Vol. 17 119–137 (Mar. 2009)

Regular Paper

A Flexible Modeling Engine Enabling Inter-service Management

Masayuki Iwai,[†1,†2] Yoshito Tobe[†1,†2] and Hideyuki Tokuda[†3]

A large number of embedded computers, such as network appliances and sensors, have rapidly spread out to home and office environments in the last few years. These embedded computers have enough CPU power to execute the software components that can control hardware. Managing distributed components together can enhance human activity and change the real world into a "Smart Space." We name such collaboration of components "federated service" or "application." In this paper, we have developed and evaluated a novel middleware named uBlocks which enables users to build and manage applications. uBlocks, unlike other distributed application-building middleware, is distinguished by two major features. The first is a flexible communication mechanism named RT/Dragon. RT/Dragon enables the connection of heterogeneous components. The second is the universal modeling of various distributed components to support building applications by multiple users in parallel. Additionally, to enable building applications in a simple way, we provide various user interfaces (UI) for multi-modal visualization: 2D/3D User Interface, and a web interface. These features lead to reduce the cost of building and managing distributed applications by the user. This research proves that the idea of building applications by users is practical and effective.

1. Challenging in Users' side Service Management

1.1 Smart Space

A large number of embedded computers, such as networked appliances and sensors, have rapidly spread out to home and office environment in the last few years. Each embedded computer has enough CPU power to execute software components that can control hardware. Such a computing environment can be called "Smart Space."

The first feature of Smart Space is a large number of embedded computers and sensors which exist in the space. We have experimentally constructed a room called "Smart Space Lab." This room is a model of our vision towards the next generation computing environment, where many sensors and appliances are embedded. Embedded computers include PCs, digital Audio Video appliances, sensors, and PDAs. Some of the computers such as Smart Furniture [11] are invisible for users. Many projects have focused on this type of smart spaces. To name a few, Active Office [7], Active Space Project [17], Aware Home [9], Easy Living Project [1], EU House [⋆1], JEITA House [⋆2] relate to this topic.

Second feature of Smart Space is given with focus on a purpose; Smart Space is the room that helps users who are working or living in the room. Supporting the users' life cannot be achieved with a single computer. For example, we suppose a like to introduce the case of a "smart" meeting room at an office environment. Because users might be moving around from one side of the room to another, some camera-linked computers need to work together to capture both the users and whiteboards. This capturing computer should send pictures to a meeting logging server computer that logs the meeting. Even in this small meeting space, we need many embedded computers.

The third feature is devices in the Smart Space. Most of all devices in Smart Space should contain built-in network interfaces. Particularly wireless communication technology such as IEEE802.11a/b/n, Bluetooth, or ZigBee are innovative and affordable to do this. Traditional A/V appliances are currently connected with one another with composition analog cables interface, but it is no doubt that future appliances in the Smart Space will be equipped with digital network interfaces in stead of such legacy wiring technologies.

1.2 Life Span of Applications

As we can see in historical distributed applications based on the server/client model (e.g., the bank's network system for automated teller machine), each of the server and client has a clear role of communication. There is a less chance

†1 School of Science and Technology for Future Life, Tokyo Denki University
†2 "Advanced Integrated Sensing Technology" Project, JST (Japan Science and Technology Agency) CREST
†3 Faculty of Environmental Information, Keio University

⋆1 http://panasonic.co.jp/euhouse/
⋆2 http://www.eclipse-jp.com/jeita/

to replace an application after it is activated. However, when the application specification must be altered, system engineers create a new application using much time, and after the application has been tested to run correctly, system engineers finally replace the old applications with the new ones.

On the contrary, applications in a smart space have different life spans. These applications consist of various devices. Each device is replaced frequently. Users in a smart space often have the requirement to change the behaviors of applications. For example, when a new networked appliance is bought, the users have desire to replace an old networked appliance. They try to replace or modify an application within a short period. Additionally, applications in a smart space are likely unstable in comparison with applications run on servers since computers in a smart space happen to shutdown frequently. Such applications are managed by non professional users and also kept in a daily used room not in the special machine room. From our experiences in building application for the Smart Space Lab [16], we have become aware that any attractive application can soon fail and become useless. The life of applications was shorter than that we expected. Application life cycle are shorter than legacy distributed applications.

There are so many requirements to replace or update applications. Therefore, it is impossible for the system engineer to replace it whenever users want to change applications.

1.3 Hardware and Software Assumption

This paper is on the assumption that each hardware has network interface and each hardware are distributed.

In a smart space, there are various kinds of hardware: networked appliances, sensor network/IP bridge nodes, PDA, cellular phones, PCs, and computerized furniture. Such embedded computers are integrated into most of these hardware. Nowadays, it is natural to assume that embedded computers have enough processing power to run programs. Network interface chips have been miniaturized and enhanced with low power consumption. TCP/IP network is the standard protocol which is spread to all infrastructure of the internet. This paper assumes that every hardware in a smart space has a wired/wireless network interface of TCP/IP.

The devices in smart space with wireless network interface can exist almost in any place in a room. This paper assumes that every hardware are distributed and running independently. The first version of iPhone 3G already has 620 MHz arm CPU and 128 MB DRAM memory, It is natural to assume that each harware in the smart space has resource which can run Java Micro Editon or Java Standard Edition. Each device has a required Java runnable resource such as 128 Mbyte momory, 300 MHz or more over clock of CPU.

Virtual Machines (VM) technologies, such as the Java Virtual Machine and .NET Common Language Runtime (CLR), have been an important software topic in recent years. Because VM can remove differences among hardware, programmer do not necessarily consider on what architectures would run their code. The overhead of the VM runtime was big at the beginning, but has improved gratefully. Programmers can now efficiently develop software on VM.

In particular, Java has VMs for various purposes. This enables Java to run on a limited resource computer. From this feature, we assume that software on embedded computers run on a Java VM. If embedded devices lack of the resources to run a Java VM, we use a small proxy computer which connects to the device.

In this paper, hardware is represented as a Java component. We use the term "component" as a Java component on an embedded computer. A component is a set of Java classes which can handle an actuator of an appliance or can process data from sensors. Because of each hardware are distributed as mention in Section 1.3, each software component on the hardware is also distributed. We use the term of a component as a distributed Java software component.

1.4 Skill Level Assumption of Users

In a smart space, it is difficult to assume all of people having skills of both network management and distributed programming. Some users may be members of a family and they would not have the skills of computers. Users can be classified into the following three groups.

Users with Advanced Skill with Computers

The first group are people who can program software and know network protocols because they have special computer related educational background. They can re-program components and can also configure it in detail in a smart space.

Users with Average Skill with Computers

The second group are users who can read e-mail or browse web pages by themselves. They have skills of simple setting of components and can connect hardware with one another usign cables, for example, TV and Video with a composite analog cable. They have less stress using mouse or watching PC display.

Users Having Novice Skill with Computers

The third group is users who cannot use computer well. Due to handicap, some of users cannot use a mouse or cannot watch a display. There is some other people who does not have handy-cap, but refuse to touch IT devices.

The first group of users is the minority of the entire users found in home environment. The third type has been decreasing due to the rise of technologies in system automation and multi-user interface (e.g., Voice interface). We do not focus on either type of users, but instead, focus on the second type which is **Users with average skill with computers**. We will use the term of "users" in this meaning throughout this paper. Most of users who can use internet browsers, e-mail, cell phones are classified in to the second level.

1.5 Who Should Build Applications in a Smart Space?

We described the features of the short period of an application lifecycle in Section 1.2. Therefore, it is needed to rebuild applications. Rebuilding the applications can replace outdated applications with useful ones. However, there is discussion on who should reconstruct the applications? Although, the large number of studies have being conducted in a smart space, there is no research which discusses on who should be responsible to do.

There are three possibilities of "who": a component programmer, a system engineer, and user. We also classify these 3 kinds of model as a point of building person as follows.

Building by Component Programmer

Most of the distributed applications are created by highly skilled deployers who also **developed** the software components, so the user has no participation in the development of distributed application. This case is suitable for creating reliable and well adjusted systems such as enterprise systems. Therefore, applications in the model are stable.

However, as this type of application needs to prospect the end users' requirements before component programming, it is possible to mismatch the user's current requirements from the system's intention. The programmers need to rebuild distributed application at every update occasion. If rebuilding is frequently done, it will be costly to rebuild them by component programmers.

Table 1 Comparison of the person who builds applications.

Application Building Person	Monetary Cost	Stability
Component Programmer	expensive	high
System Engineer	expensive	neutral
User	**inexpensive**	inexpensive

Building by System Engineer

A system engineer does not program software components, they roughly know general parts of the components and **deploy** them to build applications. Deployment is both to configure a component and to combine with one component to another. In this model, component programmers and the deployers are different. A system engineer configures the detail of connections between components. A configuration file is, for example, as follows. *Deployment Descriptor* [22] of EJB is in conformity with this model. This model requires both application deployer and users to have sophisticated computer and software skills, to name a few, VML of VNA [13]. Building applications by system engineers also delivers the cost of rebuilding application to the users.

Building by User

Legacy audio and video equipments at home are cabled by the users themselves. Similarly, it can be said that users are the most suitable group to build distributed applications. In this model, the software components are developed by programmers of IT vender, while users choose the components and combine them. The applications build by users directly reflect their preferences. Because these users are not professional for building application, applications tend to be unstable.

1.6 Comparison of Builder

Table 1 shows monetary cost by comparing the above models. The applications, which are built by programmers or system engineer, cost more than ones built by users. As shown in **Fig. 1** left, a system engineer needs to design, implement, test, ant tune-up a application totally. When users wish to change to

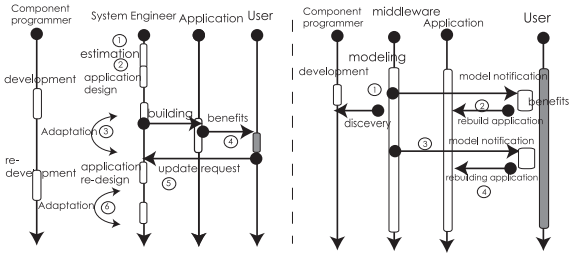

Fig. 1 Application building by system engineers (left) and users (right).

the application's behavior, the user have to tell changing request to the system engineer. Then the system engineer re-designs the application again. Monetary cost of personnel expenses for the system engineers is required without intervals. Therefore, building applications by system engineers is costly. Furthermore, users do not receive many benefits from the application due to the maintenance ant development time of the system engineer.

From the point of monetary cost, applications built by users is most suitable for a smart space. Due to re-building application are done by users in this model, there is less gap in perception between users and existing applications. Figure 1 right shows a model of application building by users. This model can provide the application's benefits in a long span. This model can also remove the delay when user requests to change application.

On the other hand, applications built by users will not be optimized and well tested. We need a mechanism which can stabilize them. Additionally, the model, that application is building by users, requires the users to be aware of networked devices and the relation between them. For instance, users must know how many RFID readers are in operation in a ubiquitous environment and which appliances are running when using an application consisting of RFID embedded appliances. These requirements will be discussed in Section 2.

1.7 Our Positions in End User Computing

Let us show an example of users' level application building. Players of LEGO block can build any size of object by joining each piece with another. Each piece has a simple uneven hole and projection on its surface, which can connect with any other blocks pieces.

The application building, which we focus, is in the same way. In our definition, **application building** means to reassign a new role to these components by composing software components which binds to the real sensors or appliances. Application building is not done by particularly configuring each component behavior, but by simply combining components with each other. This simplicity of application building, however, is difficult because we must hide complicated component communication protocols from the users.

To simplify the programming, there is a rearch are named End User Computing (EUC). Most of EUC are the programming tools for stand alone computers [10),14)]. However, in case of Smart Space near future, we have to consider totally distributed heterogeneous computing environment. All processes are running independently, we can not stop any computers to configure them.

As the point of users, most of end users are should not be exparter computer users in the home environment. They do not know the method, values, and network programming deatails of programming. Visual programming tools like Java Beans, Mac Quartz Composer, are for the computer expert people. It is needed simple connection interface but which can create powerful applications.

2. Issues in User-side Component Programming

At the previous section, we figured the merit of application building by end-users. In this section, we realize the issues which are caused by users side programming at the practical smart space.

2.1 Issue1: Lack of Components Compatibilities

In a smart space, components are developed and provided from various software/hardware vendors independently. However, in the smart space, users build component-based applications by combining such components from different vendors. For that reason it is hard for programmers of components to estimate which kinds of components communicate with each other. Such heterogeneous components are also aimed for different purposes. The lack of compatibility among components will hinder building a new application simply. Users are not so well skilled. They can neither configure the detail of component nor re-program the inside of components. So it is important to achieve compatibility among compo-

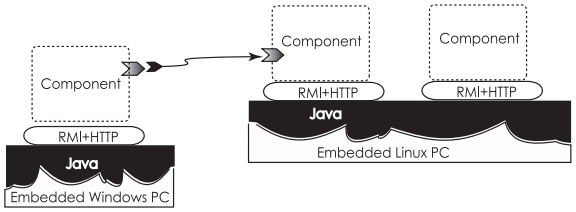

Fig. 2 Three uBlocks components are running on different Java runtimes.

nents at the layer of middleware.

To solve these compatibility issues, we propose a middleware, named uBlocks. This name originally means that users can build applications easily like building toy blocks.

First, we describe a total architecture of uBlocks briefly.

uBlocks represents each device as a software component. uBlocks components contain the following communication mechanisms: flexible communication mechanism and reliable communication mechanism. Flexible and reliable communication mechanism are named RT/Dragon. Universal modeling engine [8] is a modeling scheme upon RT/Dragon. Universal modeling engine extracts both the existence of component and relationship of components. Upon these mechanisms, users can build flexible and reliable applications.

Each component is running on a Java VM. More than two components can run on a same Java VM as shown in **Fig. 2**. Harware and software requirements are described in Section 1.3.

uBlocks utilizes Remote Method Invocation (RMI) and HTTP protocol for inter-components communication. RMI is a major communication method in Java, particularly, in component based applications like Enterprise Java Beans (EJB). As HTTP is the most popular protocol on the Internet, it is natural to implement components using it.

Figure 3 shows a basic component design of uBlocks. A basic component is the most primitive component in uBlocks.

The message to notify from component to another, like sensor value or appliance control command, is named **event** in this paper. The internal of a component consists of six software modules: Acting Gate, Sensing Gate, Event Admission

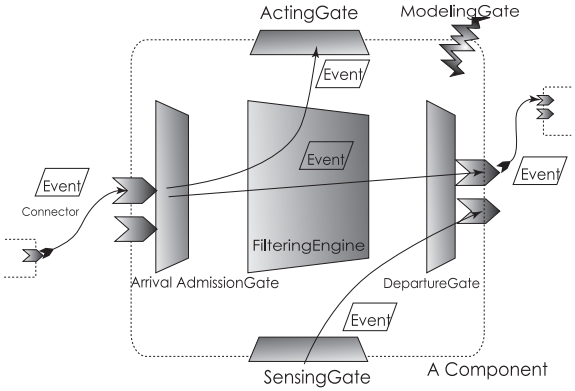

Fig. 3 A basic component design of uBlocks.

Gate, Event Departure Gate, Event Filtering Engine, and Modeling Gate. In a internal component, an event is processed by passing the gates. The functions of these modules are described as follows.

- **Acting Gate (AG)** makes an action against the real world entities, when a component receives an event. Programmers of components implements the event handler at *AG*. If some devices are connected via serial communication line, a component programmer implements a processing code of communication with devices by extending *AG*. For instance, it works like a controller of an air conditioner or a room light controller.
- **Sensing Gate (SG)** defines how to sense environmental information from real world entities. For example, at *SG*, a component programmer defines the way of obtain the temperature from the air conditioner. *SG* generate events and passes them to Event Departure Gate through Event Filtering Engine. This gate is implemented by programmers.
- **Event Arrival Admission Gate (EAG)** is an object that handles many kinds of events that contain occurred in other components. *EAG* receives events from other components and passes the event objects to Event Filtering Engine. This gate is provided as a port of middleware uBlocks.
- **Event Departure Gate (EDG)** fires events to another component in accordance with an ordering table that was created when a user configured

the relation of components. This gate is also provided by uBlocks. So it is not needed for programmers to concern about communication mechanism among components. This EDG gate is also provided as a part of middleware uBlocks.

- **Event Filtering Engine (EFE)** checks the contents of the events and decides on firing events or not. EFE is designed too generic so that programmers can implement characteristic behavior of event handling. uBlocks provide several templates filters which extends the EFE.
- **Modeling Gate (MG)** is a software module which communicates with the Modeling Engine. The Modeling Engine is an only component which recognizes the existence of every component in a smart space. MG tries to discover the Modeling Engine and also notice its existence to the Modeling Engine when a component is launched. If a relationship between components is changed, MG notices the difference to the Modeling Engine.

Events are propagated between components by following a connector. A central component has neither Sensing Gate (SG) nor Acting Gate (AG). This component, instead, mediates events through its Filtering Engine.

The model, in which each component conveys events to another component, are similar to data flow programming. This data flow programming receives much attention at the large scale dynamic web-site developing like WSFL [5] and SEDA [25].

2.2 Issue2: Complex and Unpredictable Communication between Components

As mentioned in Section 1.1, embedded computers have wireless communication in a smart space. Due to some of computers or sensors are embedded in furniture or in a wall, it is hard to recognize and to remember them. The computers, which are out of people's memories, are not used effectively. New role will not be assigned a part to forgotten components. Needless to say, it is impossible to composite these components to build applications.

Though Mark Weiser predicted an age of "calm" computing [24], users have a right to know what applications are running around us currently. Additionally, if we give up recognizing all of components, we need system engineers to build and maintenance applications in a smart space. This cost is not cheap to connect different propose of components. We need a mechanism both to adapt the relationship among different type of components.

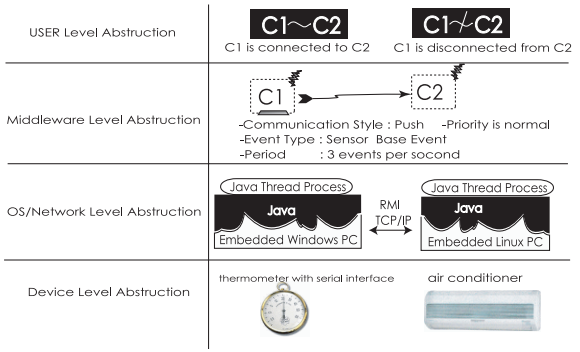

Fig. 4 Abstruction of connection among components.

Figure 4 shows difference of abstraction between user level and middleware level. At the middleware layer in Fig. 4, abstraction of connection among components includes six communication styles, priority of communication, event types, and period of firing events. For users, it is needed to hide complicated communication protocols and various type of events. Users have not to be unconscious of complex communication mechanism of the middleware. However, we want to keep the users' conscious of the connection/disconnection among components simply. So it is needed to simplify the complicated relationship of components. A mechanism of simplification and modeling of relationship among components is challenging.

To overcome the gaps of different components, we provide a methodical communication style which is independent from components. Communication style means a part of module which can send and receive some message from other components. We propose selectable communication style which can connect using a certain communication style from six styles: push, pull, callback, future-push, future-pull, and future-callback. By enabling to select communication style dynamically, this mechanism can reduce component programmers' implementation of communication with targets components. A component, named C_A, is on an embedded computer with sensors. This component sends an event every 10 s pe-

riodically. At first, C_A is select a push style. Another component, named C_B, is a component which can display information. According to the firewall of network, C_B can not get arrival events from other network domains. In this situation, C_B selects the callback communication style. Using six communication styles, a number of combination of components are extended.

2.3 Issue3: Model Collisions in Application Building from Multiple users

It is natural that multiple users are living in smart spaces. There is a possibility for different users to control and setup distributed applications at the same time. When users build an application using two components, another one might try to build a different application by using the components. In this situation, collisions of building application are occurred. It is needed users to recognize the latest relationship among components. Therefore, a synchronization mechanism which shares relationship of components is needed. When a user rebuilds an application with distributed components, the other person has to find out the change.

World Model

Figure 5 shows a scheme of modeling component relationship. Each component has information that the component communicates with others. For instance, when component C_1 is linked with C_2, C_1 has a relationship information of $C_1 \sim C_2$. (A $\sim$ B means that A is connected with B.) This whole of relationship is named **World Model**. Similarly C_2 has a relationship information of $C_2 \sim C_3$. uBlocks have a centralized Modeling Engine. The Modeling Engine gathers the relationship information from each component. Then the Modeling Engine creates a whole relationship of components in a smart space. This scheme is named Independent Modeling Scheme and is described at Section 4. This scheme can offload work from individual components. Furthermore, our modeling engine can not only recognize relationships of components but also detects both a fault of component and appearance of a new component. To detect a fault of component, Modeling Engine uses leasing mechanism which is referred to Jini Distributed Leasing Specification [21]. To discover a new component, uBlocks also refer to Jini Discovery and Join Specification [20] using multicast UDP packet.

User Interface (UI)

Both relationship recognize mechanism and discovery mechanism are implemented at Modeling Gate in Fig. 3. After the relationship information gathered from every component, the Modeling Engine creates a whole relationship graph among components. We name the integrated relationship information World Model. Figure 5 illustrates that the Modeling Engine notifies the World Model to each User interface (**UI**). In this research, the term of User Interface (UI) defines not only for the graphical visualization tools but also management middleware, which can collect information of other components. Each UI has a process to generate command to other components.

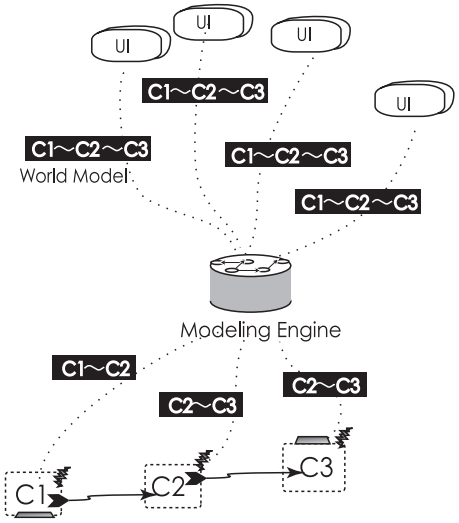

Fig. 5 The World Model, which is relationship among every components, is notified to every User-Interfaces (UI).

There is opportunity that multiple users try to build application at the same time. **Figure 6** shows the procedure of application building by user. At first, a user can recognize components running in a smart space. Second, by viewing the component information, a user builds an application. Third, when a UI received the user's request of rebuilding, the UI generates a command which is notified to components. Fourthly, after a component catches the command, the component tries to negotiate to other component. Fifthly if the command is succeeded, the component sends new relationship information to the Modeling Engine. The

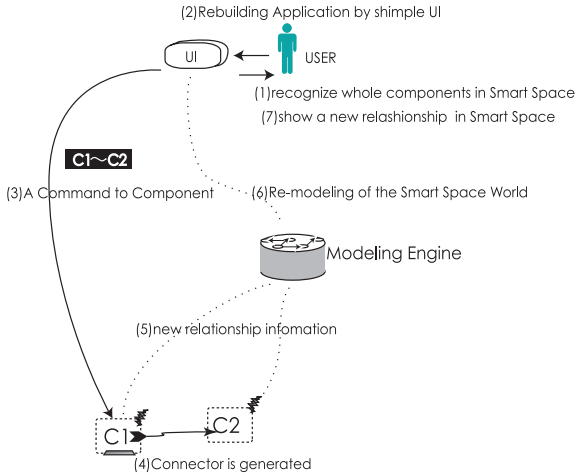

Fig. 6 Procedure of application building by user.

Modeling Engine updates world model, and then sends the model to every UI. Finally the UI updates the world model after rebuilding.

Through this process by users, there is a possibility that some other users tries to rebuild the applications. After a user change a new component relationship, each UI has a locked "world model". When the updated model is different from the locked world model, users can detect that from unlocked UIs. The collision detection mechanism of building by multiple users will be described in Section 4 in detail.

2.4 Issue4: Platform Restrictions of Application Building

Providing platforms of building applications for users are also challenging. In a smart space, there are many sorts of PCs and PDAs. Users will select different type of user interfaces (UIs) for building application according to their circumstances and preferences. Users who are outside home need use web browsers of their cellular phones or PDAs. Exparter computer users prefer to use a character-based UI just like Unix shell. For homeuse, users wish to build applications by using their own PCs. Therefore, to providing many sort of UIs is usable for users.

uBlocks provides four UIs to configure the event paths for distributed components visually. With these UIs, users can create ad-hoc distributed systems without the cost of configuration. Such UIs can also allow users to control distributed ubiquitous applications with no programming.

Users may not stay at the same place for long, therefore their circumstances change frequently. We need to provide user-interfaces which can adapt to users' various environment. We have developed various UIs. Java2D-based UI, Java3D-based UI, Web-based interface, and shell type UI.

Each UI is developed on a fundamental module named "UIBase." UIBase holds component's name, component status, and connection vector. Using UIBase, various kinds of interfaces can run simultaneously.

3. Flexible Communication

In Section 2.2, we mentioned the issue about complex and unpredictable communication between components. The middleware layer have to hide such a complex mechanism from user side. we describe the detail of flexible communication mechanism named RT/Dragon. We will mention six communication styles: push, callback, pull, future push, future callback, and future pull. Additionally, we describe about mechanism of Fintering Engine.

3.1 Inter-component Communication Style

Limited to say about the communications between two components, a certain component notifies events to components.

Events include numerical values, strings, objects, XML files, URLs, binary data, and etc. Figure 4 shows that a component sends an event to another component. The component which generated the event is defined as a source component. Sensors which can detect something from real world are usually classified to source. The component which receives the event from source component is defined as target components. Some appliances which can act to real world is categorized to target component. Applications in a smart space are constructed by linking source components and target components.

Classification by Synchronization

When a source component transmits events of to a Target component, it is able to classify the communication to two generally: synchronous communication and asynchronous communication. In this paper, we have divided style communication into two: asynchronous communication and synchronous communication. In

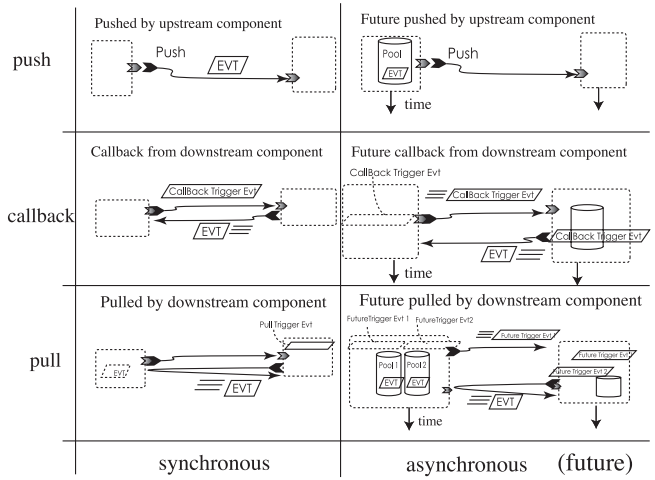

Fig. 7 Classficication of communication style.

synchronous communication, each component has no time lag to return or to send an event after the component received a request for data. As synchronous communication can realize swift transmission, this style is suitable for the component which needs to notify as soon as possible. However, with this communication style components needs to wait for a result of the event notification. While waiting, the component can not process other events.

In cases that the network condition is not fine, such that bandwidth is saturated, the synchronous event transmission might not be suitable. For such case, RT/Dragon provides components with an asynchronous communication style, called future communication style. With the future communication style, components can hold several events in certain duration and send to another stream in future.

On the other hand in asynchronous communication, which we name *future communication style*, each component holds request events in certain duration and sends the holed event will be sent to the other component in future. Future communication style can transfer events without considering the partner component by using event pool (see **Fig. 7**). A component is able to continue original processing without blocking it.

Table 2 Classification of communication styles.

	Name	trigger comes from	data cames from
Synchronous	push	N/A	source
	callback	source	target
	pull	source	upstream
Future	future push	N/A	source
	future callback	source	target
	future pull	source	upstream

With the future communication style, however, there is some delay from the actual event occurrence to the transmission, and there are possibility that transmission of events is not send perfectly.

Classification by the Initiative of Communication

In the description above, we assume that a source component initiates the event transmission. In addition to such a "push" communication style, RT/Dragon provide the following two styles: callback and pull.

In callback communication style, a target component receives a trigger event from a source component firstly. The trigger event dose not contain data or messages in it. However, it contains information of source components. Using this information from the trigger event, target components send back a new event, which contains data or messages (see **Table 2**). When a target component decides that it does not have room to send event to source, a target component can stop sending back events to source. In callback communication style, a target component is a leader of communication who can decide to send or not to send events to source components.

Final case is a **pull communication style**. In the pull communication style, a target component receives a trigger event from a source component. The trigger event includes information of a source component. Then when it is necessary to get events which contain data or message from the source component, the target component tries to pull events from source component actively using the information from the trigger event (see Table 2).

Selection of Various Communication Style

RT/Dragon has a novel flexible communication mechanism which never device the communication style of itself until a target component receives a trigger

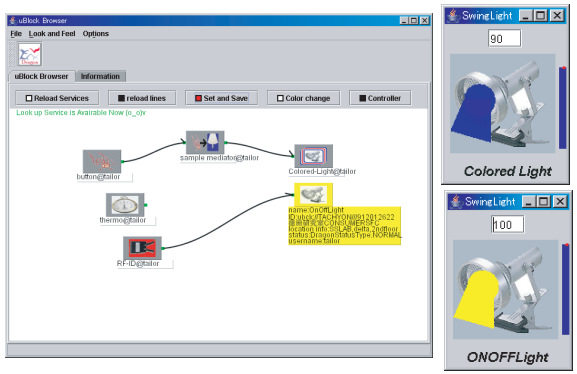

Fig. 8 Experiment with five reference components.

event. By this evaluation, each component can decide its communication style on execution time. The trigger event is sent from a source component and is describe which communication style the source component intends to. After the target receives the trigger event from the source component, it decides the communication style from six: push, pull, callback, future push, future pull, and future callback. Because each components developer can not estimate the communication style of target components beforehand, these styles are needed to communicate various kinds of components.

3.2 Experiment for Flexible Communication

This subsection explains the effectiveness of flexible communication among component with different versions.

We prepare five reference components: a colored light component, a simple light component, a button component, a light brightness controller component and a light coloring controller component. **Figure 8** shows a screen shot of the experiment.

A colored light component is a component of a light which has a function to change its color. A simple light component is also a component of a light, which only can switch on/off its power. This simple light component can not handle an event that commands to change its color.

A button component is a source component which can generate the most simple events: *DSequentialEvent*. A light brightness controller component can send a command which can change brightness of a light. A light coloring controller component can also send a command to control a light. The difference between a light coloring controller and a light brightness controller is that coloring controller can send an event to change the color of a light. Each source component is itemized as follows.

- The button component generates *DSequentialEvent*.
- The brightness controller component generates *DLightControlEvent*, which is a subclass of *DSequentialEvent*.
- The coloring controller component generates *DColoredLightControlEvent*, witch is a subclass of *DLightControlEvent*.

Each target components are itemized as follows.

- The simple light component can handle *DSequentialEvent*. When this component recerives an event of *DColoredLightControlEvent*, this component can not handle the function of coloring. However, *DColoredLightControlEvent* is a subclass of *DSequentialEvent*. So the simple light can handle this event as a *DSequentialEvent*.
- The colored light component can handle *DSequentialEvent*, *DLightControlEvent*, *DColoredLightControlEvent*. When the colored light component receive events of *DSequentialEvent* or *DLightControlEvent*, this component turns a light to a default color.

3.3 System Evaluation

We have evaluated the system performance of push, callback, and pull communication style in RT/Dragon. Each components sends events of *DSequentialEvent*, which is 968 k byte. We use Pentium 2 GHz CPU and WindowsXP OS for evaluation.

To evaluate push, callback, and pull communication style, a source component send events to target component 150 times.

Figure 9 shows a overhead of communication to finish the communication.

Gray is a ratio of trials finished within 1–9 ms. Light gray is a ratio of trials finished within 10–19 ms. White is a ratio of trials finished within 20–29 ms.

Push communication style can finish transaction within 9 ms at a 147 times while 150 trials, This communication is implemented to withstand highly periodic

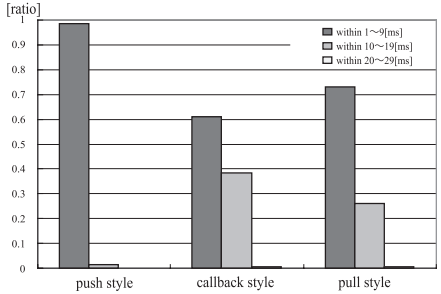

Fig. 9 Comparison with cost of Push/Callback/Pull communication style.

event transaction.

Callback communication style can finish transaction with 10–19 ms at 38% of whole trials (see Fig. 9). This result is caused by two RMI sessions. First is for transmission of a trigger event, and second is for transmission of a callback event from target component to a source component. Callback style needs comparatively high communication cost. In the case of a push communication style, only one trial take 20 ms-over to finish communication in 150 trials and over 70% trails succeed to finish communication with in 10 ms.

4. uBlocks Modeling Engine

In Section 2.3, we mentioned the issue about Model Collisions in Application Building from Multiple users. We can not assume that each component is running on a PC with rich resources. In Smart Space computing environment, some of the components are executed on embedded machines and some of them on PDAs with restricted resources. Therefore, a scheme to reduce the component-side computation is needed. We will compare three schemes: component-side modeling scheme, interface-side modeling scheme, and independent modeling scheme. And we will mention the collision detection mechanism for enabling multiple users build applications simultaneously. Finally, we will show multi-modal user interfaces which can be used by various level of users at various circumstance.

4.1 Modeling Scheme Comparison

As mentioned in Section 1, users should be able to recognize distributed devices and build applications for smart spaces by themselves. Furthermore, we need a scheme to balance of component-side computation load work.

Definition of terms

We have selected four elements in distributed component-based application: Hardware, Component, ModelingEngine, and User Interface. **Hardware** are physical devices such as a networked camera or doors with embedded sensors. Hardware is abstracted by software components. **Component (C)** is the software abstraction of distributed hardware. A component must send a relationship list of its communication parties to the ModelingEngine. **ModelingEngine (M)** gathers each distributed component relationship from the network and generates the model by merging them. ModelingEngine notifies the model to every UI when the new model is generated. **User Interface (UI)** is a client software that can deploy distributed application visually. Using these terms, we will discuss schemes for simultaneous modeling distributed components from multiple users. We will at first classify the modeling schemes into three parts. This novel classification intends to clarify up which method is most suitable when gathering distributed component relationship and holding gathered total relationships as a common world model.

Component-side Modeling Scheme

Component-side Modeling Scheme is a system where each component has a ModelingEngine and a User Interface. Therefore, each component is homogeneous from one another, but this in turn makes it hard to keep a common model among numerous ModelingEngines.

Interface-side Modeling Scheme

Interface-side Modeling Scheme is a system in which each UI has a ModelingEngine. Every component runs independently. When there is only a small number of UI, there is little load for each component to process. However, as the number of UI increases, thin components may lack resources when processing data sent from numerous ModelingEngines.

Independent Modeling Scheme

Independent Modeling Scheme is suitable for an environment where many components are found on embedded computers with limited computation resources. This scheme completely separates components, UI, and ModelingEngines. We assume at least one ModelingEngine on a relatively fast computer on this scheme.

Table 3 Parameters list (Number of transaction per second).

f	Number of fault components per second ($f \ll 1$)
d	Number of discovered components per second ($d \ll 1$)
c	Number of commands issued by users to modify the relationship of components per second
N	Total number of user interfaces (UI) and components (C)
x	Number of UIs independent from components
TrC_{in}	Number of component's transaction that is notified from other components (per second)
TrC_{out}	Number of component transaction to notify other components (per second)
TrC_{total}	Total number of transactions by a component (per second)
$TrUI_{in}$	Number of UI transactions notified from other UIs (per second)
$TrUI_{out}$	Number of UI transactions notified to ModelingEngine (per second)
$TrUI_{total}$	Total number of transactions from a UI (per second)
TrM_{in}	Number of ModelingEngine transactions notified from components and UIs (per second)
TrM_{out}	Number of ModelingEngine transaction notified to UIs (per second)
TrM_{total}	Number of total ModelingEngine transaction (per second)

Even when many user interfaces are running, a system using this scheme will have less load on its components, because most of the load processing is done on one of the ModelingEngine.

To compare the three schemes, we will explain using a mathematical expression.

Let parameters be defined as **Table 3**. In Section 2.2, we mentioned that components are unreliable in a ubiquitous computing environment. The system needs to detect which components stopped or restarted. For this reason, we define f and d parameters. c represents how many times users issue commands to change the relationship of components through a UI per second. We name the new relationship as "changed-model". c varies according to the number of both users and applications in the environment.

4.1.1 Efficiency of Component-side Modeling Scheme

As shown in **Fig. 10**, *Component-side Modeling Scheme* assumes that ModelingEngine, UI and Component are on the same node.

Let us discuss component C_i. C_i holds UI_i and M_i on the same runtime node. Each component notifies the change by the users' command to other components except themselves. The incoming transaction of C_i is the total transaction from

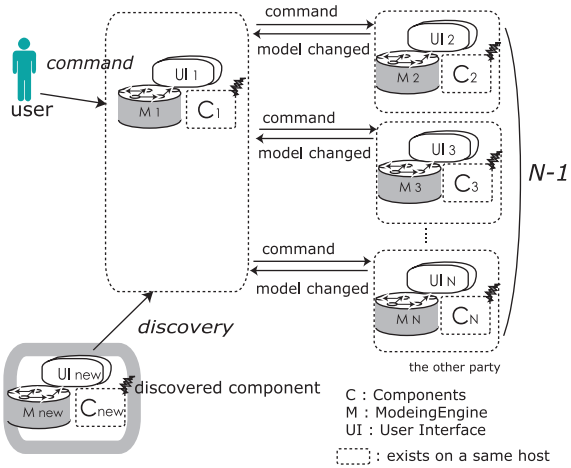

Fig. 10 Details of component-side modeling scheme.

$C_1, C_2, C_{i-1}, C_{i+1}, \ldots, C_N$. TrC_{in} and TrC_{out} can be written as:

$$TrC_{in} = \sum_{i=2}^{N}(c+f) = (c+f)(N-1)+d \quad (1)$$

$$TrC_{out} = (c+f)(N-1) \quad (2)$$

The total number of transactions for a component is the sum of every incoming transaction and outgoing transaction. TrC_{total} can be written as:

$$TrC_{total} = TrC_{in} + TrC_{out} = 2(c+f)(N-1)+d$$

4.1.2 Efficiency of Interface-side Modeling Scheme

Figure 11 shows details of *Interface-side Modeling Scheme*. In *Interface-side Modeling Scheme*, each component is independent from both UI and ModelingEngine.

x is number of user interface. UI_1 receive changed-model messages from each component when they are changed. The number of commands from one UI to one component is $\frac{c}{(N-x)}$. Therefore, $TrUI_{in}$ and $TrUI_{out}$ can be expressed as follows.

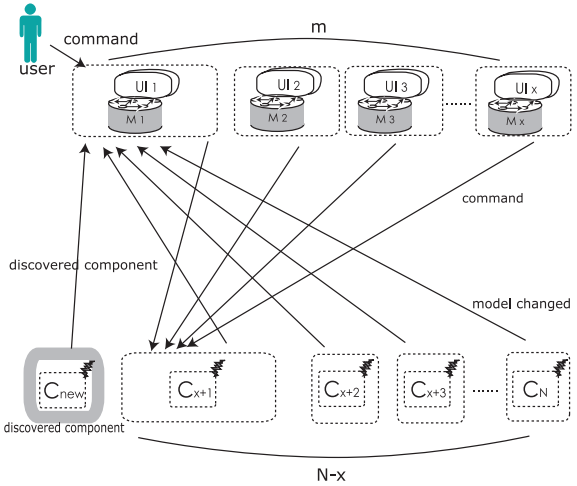

Fig. 11 Details of interface-side modeling scheme.

$$TrUI_{in} = \sum_{i=x+1}^{N} TrC_{out} + d \tag{3}$$

$$TrUI_{out} = \frac{c}{(N-x)}(N-x)d = c + d \tag{4}$$

In Fig. 11, C_{x+1} must notify the changed-model of itself to all UIs. Accordingly, TrC_{total} can be expressed following.

$$TrC_{in} = \sum_{i=1}^{x} c$$

$$TrC_{out} = \sum_{i=1}^{x}(TrC_{in} + f) = x(xc + f)$$

$$TrC_{total} = TrC_{out} + TrC_{in} = cx^2 + (f + c)x \tag{5}$$

Because the transaction of TrC_{total} is x squared order, it can be said the transaction for the components is heavy.

In the same way, $TrUI_{in}$ and $TrUI_{out}$ are x cubed order transaction.

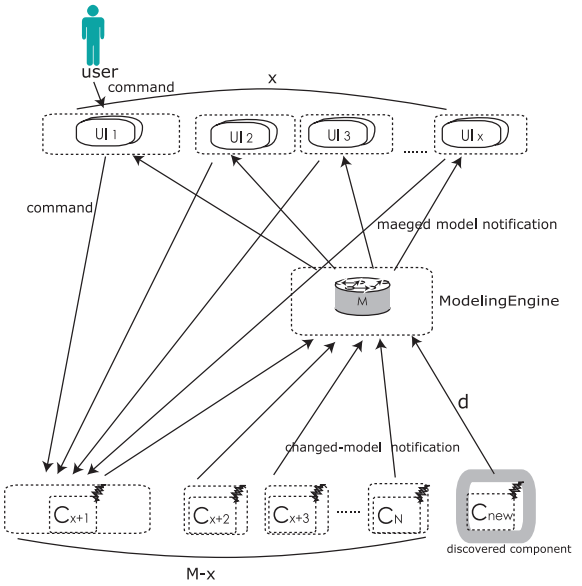

Fig. 12 Details of independent modeling scheme.

$$\begin{aligned} TrUI_{in} &= \sum_{i=x+1}^{N} TrC_{out} + d \\ &= x(N-x)(xc+f) + d \\ TrUI_{out} &= TrUI_{in} + TrUI_{out} \\ &= x(N-x)(xc+f) + d + c \end{aligned} \tag{6}$$

Efficiency of Independent Modeling Scheme

As shown in **Fig. 12**, *Independent Modeling Scheme* gathers the connection information from each component.

Since each UI command is issued by a user, $TrUI_{out}$ and $TrUI_{in}$ are as follows:

$$\begin{aligned} TrUI_{out} &= c \\ TrUI_{in} &= TrM_{out} \\ TrUI_{total} &= TrUI_{in} + TrUI_{out} \\ &= (N-x)(cx+f) + d + c \end{aligned} \tag{7}$$

Each component accepts commands from every UI.

$$
\begin{aligned}
TrC_{in} &= \sum_{i=1}^{x} TrUI_{out} = cx \\
TrC_{out} &= TrC_{in} + f = cx + f \\
TrC_{total} &= TrC_{in} + TrC_{out} = 2cx + f
\end{aligned} \tag{8}
$$

On the other hand, a ModelingEngine accepts the relations of all components, and generates and notifies a graph showing the relations to all UIs. So, TrM_{in} and TrM_{in} can be expressed as follows.

$$
\begin{aligned}
TrM_{in} &= \sum_{i=x+1}^{N} TrC_{out} + d = (N - x)(cx + f) + d \\
TrM_{out} &= \sum_{j=1}^{x} TrM_{in} = x((N - x)(cx + f) + d) \\
TrM_{total} &= TrM_{in} + TrM_{out} \\
&= c(N - x)(x^2 + 1) + (f + d)(x + 1)
\end{aligned} \tag{9}
$$

The total number of transactions for component of *Independent Modeling Scheme* is x order. This study shows that the *Independent Modeling Scheme* is suited for management of distributed components.

4.2 Platform Independency of uBlocks UI

In Section 2.4, we mentioned about platform restrictions of application building. Users may not stay in the same place for a long time; therefore their circumstances may change frequently. We need to provide user interfaces (UI) which can adapt to various users' environment. uBlocks provides four UIs which visually configure the event paths for distributed components. With these UIs, users can create ad-hoc distributed systems without increasing the cost of configuration. Such user interfaces can also allow users to control distributed ubiquitous applications with no programming.

2D User Interface

Figure 13 shows a screenshot of the 2D User Interface. Users who have no programming skills can view which type of components exists on the network and setup the event-driven system.

When a user brings the mouse cursor on top of an icon, the icon shows the component details: name, location, vendor name, and serial number. When the user clicks the icon, the icon shows the interface and the lines to connect to other components. Then, the user moves the mouse cursor to another component icon. If the interfaces match, the editor repaints the icon and lines with the color blue, then shows the details of the component. If the interfaces do not match, the editor repaints the icons with the color red.

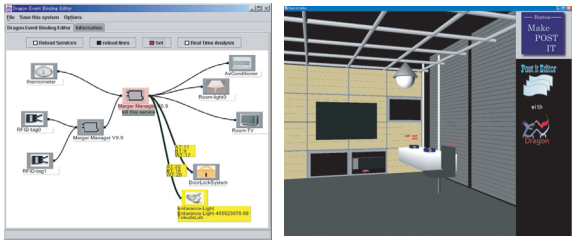

Fig. 13 A screen shot of connecting icons on 2D UI (left)/3D UI (right).

The user clicks on the blue colored icons to connect them. This action is transmitted immediately to the target distributed components. Each target component tries to register itself towards the source components in the event stream. The users can configure event path with three simple types of connections: normal connection, priority-based connection ⋆1, and disconnection. This simplicity eases users' setup during the construction of distributed components.

Web User Interface

Users are always moving from one place to another in the ubiquitous computing environment. In any circumstance, users may wish to construct distributed ubiquitous application through their cellular phone.

This web User Interface page structure is shown in **Fig. 14** right. However, It is difficult to represent rich context of components because of the limited resources of cellular phones. Accordingly, we designed these character interfaces simply. Users can control and setup distributed ubiquitous applications through cellular phone.

As shown in Fig. 14, multiple UIs can run with consistency. If a user creates an application through a cellular phone, other users who are viewing from the 2D

⋆1 Priority-based connection is mentioned in the paper [12)]

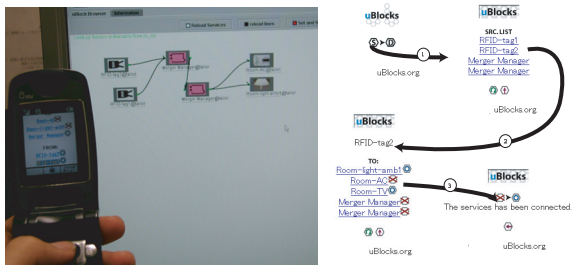

Fig. 14 WEB user interface and 2D user interface are working together with consistency.

UI can see the updated distributed application immediately. This is the result of the Independent Modeling Scheme.

4.3 System Evaluation

In this section, we measure the system performance to evaluate the stability of model transmission. In addition, we mention about two different kinds of applications to show useful scenarios using uBlocks.

Notification Time to Multiple UI When the Model is Changed

The first evaluation is based on the 2D User Interface commands components on networks to change the relationship between them. The number of UIs increased from 1 to 13. We measured the notification time to all other 2D UIs 50 times. In **Fig. 15**, we plot the average, maximum, and minimum time. UIs are on a Windows XP (Pentium4 3 GHz) host, and the Modeling Engine is on a different FreeBSD (Duron 800 M) host. Two components are running on another WindowsXP (Pentium4 2.4 G).

We can approximate the notification time t in milliseconds as a linear equation as $t = 13.477x + 75.207$ $(g \leqq 13)$. x is the number of UIs. Calculation quantity of t can be estimated as $O(x)$.

Notification Time between Different Kinds of UIs

In this measurement, we changed the combination of the notifying UI and the UI which accepts the changed-model. We evaluated the time to finish notification fifty times. **Table 4** shows the minimal time, average time, and maximal time. In this table, Web User Interface represents the time to change the model at the server-side. 2D UI is inferior to others, however we reached a conclusion that each UI has finished notification within 100 milliseconds.

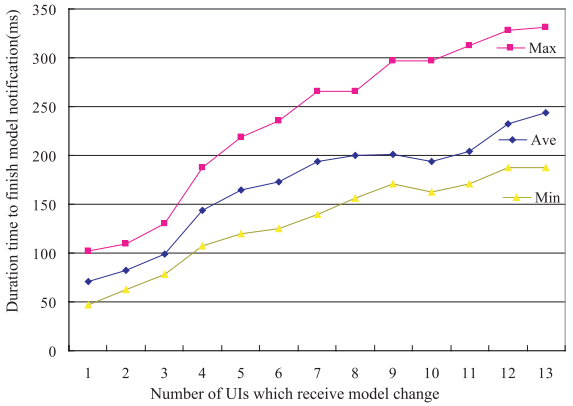

Fig. 15 Notification time of changed-model to all UIs.

Table 4 Model notification time from a UI to other (milliseconds).

Model Notifying UI	Model Receiving UI	MIN	AVG	MAX
Web UI	2D UI	62	93.7	172
Web UI	Character	46	65.9	94
2D UI	Web UI	62	74.0	93
2D UI	Shell UI	47	75.2	110

Summary of Evaluation

Increase of embedded devices will make the maintenance of such devices impossible. Wireless connection among embedded devices also prevents users from having a clear grasp of what types of applications that are running on them. As well-skilled system engineers or developers are not always available, the user's inability to keep track of the devices will make the ubiquitous computing environment completely useless.

To overcome the user's inability, we have developed *uBlocks* which enables user user-level visual composition.

The evaluation shows that uBlocks supports multiple-user construction of component distributed application at a same time. The evaluation shows that uBlocks has a suitable performance for multimodal visualization.

We have evaluated 2D UI, 3D UI and a web UI accessible through a cellular

phone. Providing these multi-modal user interfaces enables users of variable skill-levels to utilize distributed devices and building useful application and reduce the cost of management and construction of distributed applications.

5. Applications and Related Works

To ensure the effectiveness of uBlocks, we have developed several applications based upon uBlocks targeted at different areas. In addition to the home area, uBlocks is usable for other situations. We will show applications for home domain and media transcoding.

5.1 Application for Home Living Domain

Room Mode Changer is an application which controls many devices with one click through the cellular phone. Room Mode Changer is created by the user as shown in **Fig. 16**. Several networked appliances consist this application. One of networked appliance is an electronic power supply controller, CD player and ambient light. The user can drag the icon of the mode switcher to the icons of these devices. After this operation, the user can control the registered devices simultaneously.

For example, when the user clicks the "sleep mode" button on a cellular phone, all selected devices turns off. In turn, when the user clicks the "wakeup mode" button, all selected devices turn on. From this application, we confirm that it is simple for users to recognize the relationship among components with uBlocks middleware. Furthermore, we have shown that heterogeneous devices can connect with one another in uBlocks.

5.2 Application for Media Transcoding

The Ubiquitous Doorbell enables picture delivery of the entrance door to anywhere the home owner is located. At first, the user connects the entrance door camera and PC in the living room. If the entrance doorbell button is pushed, a picture captured by a USB camera will be immediately displayed on the living room PC. When the user is not home, they will want to know who is visiting their house, so the picture is delivered from the PC to their cellular phone via a web-based UI. In this case, our application can dynamically reduce the picture size and drop the colors to gray scale (**Fig. 17**).

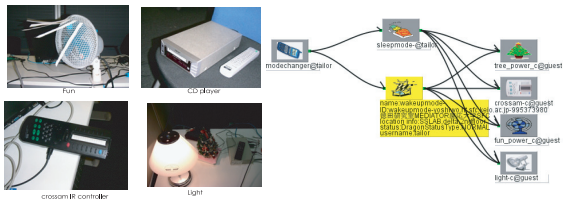

Fig. 16 Room mode changer application constructed upon uBlock UI.

5.3 Related Work on Flexible Service Building

In this section, we describe related work of platforms to construct distributed application. Then, we summarize the related work and uBlocks in **Table 5**. Then we certificate that uBlocks is a novel middleware and has features for building application by users in Smart Space.

Gaia Project [3),17)] is a CORBA-based powerful distributed middleware that has a application building tool for active space. However, this tool is not friendly for unskilled users. One of our goals is to provide user friendly simple UIs like web UI and Java UI that are controllable with mouse or cell phones. uBlocks realizes simple modeling and can reduce the cost of building application by users.

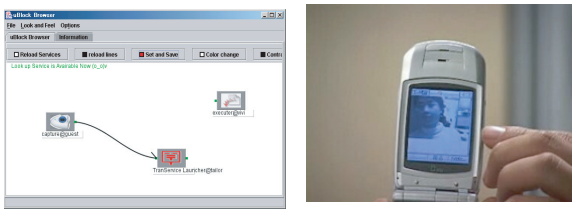

Fig. 17 Pictures of delivered to a user's cellular phone.

Table 5 Related work of application building.

name	method/who	Num. of UI	Modeling
Carp@ [4)]	mouse click/engineers	single	Independent
ICrafter [19)]	xml UI/user	multi-	User-side
SIENA [2)]	message header/user	-	event routing
COCA [18)]	mouse click/user	single	User-side
VNA [13)]	xml /engineer	multi	Comp.-side
uBlocks	multimodal-UI/user	multi	independent

A reflection based tool for observing Jini service (Carp@) [4] is a management tool for dynamic distributed Jini systems. Carp@ allows observation of services and clients, their interconnections and messages exchanged among them. The tool extracts an architectural component model based on components, ports and connectors. RT/Dragon can offer a more flexible event communication than Carp@. In addition, RT/Dragon can offer soft reliable communication mechanism. However, Carp@ is not aimed at multimodal UIs.

COCA [18] can transform heterogeneous ubiquitous computing resources through a process called classification into a conceptualized representation, which allows high-level manipulation and configuration by ubiquitous computing applications. However, COCA is neither supports multiple-users nor multi-modal UI. uBlocks provides a scheme for developing distributed application by multiple users through multimodal UIs.

i-LAND [23] is a project that enables automatic collaboration among heterogeneous devices. Our uBlocks focuses not only on Computer Supported Cooperative Work (CSCW) but collaboration among small computation power devices. Thus, at the sacrifice of automatic collaboration, we designed uBlocks to be as simple as possible.

ICrafter [19] realizes client-side dynamic GUI rendering using XML. Through a single GUI of ICrafter, a user can control multiple devices at the same time. However, ICrafter does not focus on building applications with distributed components. On the other hand, our uBlocks is aimed at creating ad-hoc ubiquitous application by connecting each distributed component directly.

Table 5 shows a comparison of application building middlware.uBlocks is the only middleware which is support building applicaitons through multimodal UI. Furthermore, uBlocks allows users to build application with less workd load for both each component and UI.

5.4 Related Work on Messaging among Distributed Components

There are some related work of messaging middleware comparing with RT/Dragon.

Common Object Request Broker Architecture (CORBA) [15] is a general-purpose, internet-scale software architecture for distributed system using the object-oriented paradigm. CORBA Event Service is one of the Common Object Services in CORBA specifications. CORBA Event Service defines a set of interfaces that provide the synchronous event communication mechanism. The interfaces support pull-style and push-style communication. Furthermore, multicast distribution among event suppliers and event consumers is accommodated with event channels. RT/Dragon has no multicast communication but comprehend 6 communication styles.

Unlike event data of CORBA Event Service, RT/Dragon has hierarchical events which can communicate with many types of components. The Real-Time CORBA [6] has a scheme of scheduling of event groups into one ORB object. RT/Dragon does not decide one object for event controlling. All the components have reliable communication mechanism.

Java Event Channel (JECho) [26] aims to support distributed group communication by offering the notions of events and event channels. Event Channel is a logical construction that links some number of endpoints to each other. JECho employs multicast transmission. Therefore, every distributed object must support and understand related and non-related event. In the case of low computation power machine like sensors and appliances, controlling multicast channel and groups cause difficult problems. Managing which service has to join which multicast channel and how to discover services users want, are unsolved problems.

SIENA [2] is middleware for distributed event service. SIENA restricts type of event's contents strictly, because SIENA depends on contents based event routing. SIENA also offers "patterns." "A Pattern" is an expression whose basic elements are filters. However, patterns have very restricted expression. On the contrary, RT/Dragon permits any type of serializable object as event object, so programmers of each component has to decide the detailed behaviors of each Event Filtering Engine. We can adapt more strict filtering models, including SIENA's Pattern model.

Digital Living Network Alliance [⋆1] is an international, the collaboration of consumer electronics, computing industry companies. DLNA is has a http-based XML messaging architecture. We can use DLNA appliances as a component module. However, DLNA does not contain sensors and user defined appliances.

⋆1 http://www.dlna.org/home

DLNA, unlike uBlocks, can not support various communication style, which supports mainly media contents streaming by http.

6. Summary and Results

To enable users to build and manage applications, we have accomplished in creating a high-level simple abstraction of the communication among components. We have designed, developed, and evaluated a novel middleware named uBlocks. uBlocks has solved two challenges found when building applications by users.

The first challenge is flexible communication mechanism. We have classified six communication styles and provided a mechanism of a lazy communication style decision. Furthermore, uBlocks has event hierarchy which can connect components widely.

The second challenge is to ensure universal modeling of relationship among various components. We have compared three modeling schemes and figured out that the *Independent Modeling Scheme* allows construction of distributed applications by multiple users with fewer loads on components. uBlocks also provides multi-modal interfaces, which allows construction of component-based distributed applications in various ways that depend on the user's situation. To show the versatility of the middleware, we have implemented and tested two applications built upon uBlocks: home domain and media transcoding.

With the features mentioned above, uBlocks can reduce the cost of building and management of distributed applications by the users themselves. This research has reached to the conclusion that the idea of building applications by users is practical and effective. In the area of computer science, this research accelerates to advance the paradigm of middleware research from targeting system engineers to users.

We plan to extend uBlocks to enterprise component-based system development. Enterprise component-based system development using Java, such as EJB, WSDL, is currently popular, however maintenance and development costs of enterprise systems are still high. The simple and visual building application method of uBlocks will contribute within this area.

Acknowledgments This research research has been done as a part of CREST/OSOITE project and KAKEN Grant-in-Aid for Young Scientist (B).

References

1) Brumitt, B., Meyers, B., Krumm, J., Kern, A. and Shafer, S.: Easyliving: Technologies for intelligent environments, *Second International Symposium on Handheld and Ubiquitous Computing, HUC 2000*, pp.12–29 (Sep. 2000).
2) Carzaniga, A., Rosenblum, D.S. and Wolf, A.L.: Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service, *Nineteenth ACM Symposium on Principles of Distributed Computing* (July 2000).
3) Hess, C.K., Roman, M. and Campbell, R.H.: Building Applications for Ubiquitous Computing Environments, *Pervasive 2002, LNCS 2414*, pp.16–29 (Aug. 2002).
4) Fahrmair, M., Salzmann, C. and Schoenmakers, M.: A Reflection Based Tool for Observing Jini Services, *Reflection and Software Engineering*, pp.209–227 (June 2000).
5) Leymann, F. and IBM Software Group: Web Services Flow Language 1.0 (May 2001).
6) Harrison, T.H., Levine, D.L. and Schmidt, D.C.: The Deighn and Performance of a Real-time CORBA Object Event Service, *OOPSLA* (Oct. 1997).
7) Harter, A. and Hopper, A.: A distributed location system for the active office, *IEEE Network Magazine*, Vol.8, No.1 (Jan. 1994).
8) Iwai, M. and Tokuda, H.: Distributed Data-centric Application Development using Multiple Mobile Devices, *Proc. 6th International Conference on Mobile Data Management* (*MDM2005*), pp.200–210 (2005).
9) Kidd, C.D., Orr, R., Abowd, G.D., Atkeson, C.G., Essa, I.A., MacIntyre, B., Mynatt, E., Starner, T.E. and Newsletter, W.: The aware home: A living laboratory for ubiquitous computing research, *Proc. Second International Workshop on Cooperative Buildings—CoBuild'99* (Oct. 1999).
10) Lawrance, J., Clarke, S., Burnett, M. and Rothermel, G.: How well do professional developers test with code coverage visualizations? An empirical study, *VLHCC '05: Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp.53–60, Washington, DC, USA, IEEE Computer Society (2005).
11) Ito, M., Iwaya, A., Saito, M., Nakanishi, K., Matsumiya, K., Nakazawa, J., Nishio, N., Takashio, K. and Tokuda, H.: Smart furniture: Improvising ubiquitous hot-spot environment, IEEE 3rd International Workshop on Smart Appliances and Wearable Computing (May 2003).
12) Iwai, M., Nakazawa, J. and Tokuda, H.: Dragon: Soft Real-Time Event Delivering Architecture for Networked Sensors and Appliances, *The 7th International Conference on Real-Time Computing System and Applications*, pp.425–432 (Dec. 2000).
13) Nakazawa, J., Tobe, Y. and Tokuda, H.: On Dynamic Service Integration in VNA Architecture, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E84-A, No.7, pp.1610–1623 (July 2001).

14) Nardi, B.A., Miller, J.R. and Wright, D.J.: Collaborative, programmable intelligent agents, *Commun. ACM*, Vol.41, No.3, pp.96–104 (1998).
15) Object Management Group: The Common Object Request Broker Architecture and Specification 2, 2ed, CORBA Event Service (Feb. 1998).
16) Okoshi, T., et al.: Smart space laboratoty project: Toward the next generation computing environment, *IWNA2001* (Feb. 2001).
17) Roman, M., Hess, C., Cerqueira, R., Renganat, A., Campbell, R.H. and Nahrstedt, K.: Gaia: A middleware infrastructure to enable active spaces, *IEEE Pervasive Computing*, pp.74–83 (Dec. 2002).
18) Schubiger-Banz, S. and Hirsbrunner, B.: A Model for Software Configuration in Ubiquitous Computing Environments, *Pervasive 2002, LNCS 2414*, pp.181–194 (Aug. 2002).
19) Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P. and Winograd, T.: ICrafter: A Service Framework for Ubiquitous Computing Environments, *UBICOMP 2001, LNCS 2201* (Oct. 2001).
20) Sun Microsystems Inc.: *Jini Discovery and Join Specification* (Oct. 2000). http://www.sun.com/jini/specs/core1_1.pdf
21) Sun Microsystems Inc.: *Jini Distributed Leasing Specification* (Oct. 2000). http://www.sun.com/jini/specs/core1_1.pdf
22) Sun Microsystems, Inc.: Enterprise Java Beans Technology (2001). http://java.sun.com/products/ejb/
23) Tandler, P.: Software Infrastructure for Ubiquitous Computing Environments Supporting Synchronous Collaboration with Multiple Single- and Multi-User Devices, *UBICOMP 2001, LNCS 2201*, pp.96–115 (Aug. 2002).
24) Weiser, M.: The Computer for the Twenty-First Century, *Scientific American*, Vol.265, No.3, pp.94–104 (Sep. 1991).
25) Welsh, M., Culler, D.E. and Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services, *Symposium on Operating Systems Principles*, pp.230–243 (2001).
26) Zhou, D., Schwan, K., Eisenhauer, G. and Chen, Y.: JECho—Interactive High Performance Computing with Java Event Channels, *International Parallel and Distributed Processing Symposium* (Apr. 2001).

(Received June 11, 2008)
(Accepted December 5, 2008)
(Released March 11, 2009)

Masayuki Iwai received Ph.D. in Media and Governance from Keio University in 2002. He is currently a Project Associate Professor of Tokyo Denki University. His research interests include distributed middleware, visual programming, wireless sensor applications, and RFID systems. He is a member of IPSJ and JSSST.

Yoshito Tobe received Ph.D. in Media and Governance from Keio University in 2000. He is currently a Professor at Tokyo Denki University. His research includes multimedia communications and sensor networks. He is a member of IEEE Communications Society, ACM, and IPSJ.

Hideyuki Tokuda received Ph.D. in Computer Science from the University of Waterloo in 1983. He is currently a Professor in the Faculty of Environmental Information, Keio University. His research interests include distributed real-time systems, multimedia systems, mobile systems, ubiquitous system, and sensor networks. He is a member of IPSJ, IEEE, ACM.