Vol. 40 No. 11 Transactions of Information Processing Society of Japan Nov. 1999

Regular Paper

Possession System: Middleware for Adaptive Collaborative Applications

Masahiro Mochizuki[†] and Hideyuki Tokuda[†,††]

In this paper, we describe the design and implementation of a middleware system named Possession System. The system is designed based on Possession Model which provides users with a consistent view of application components, networked sensors, and distributed devices by introducing two separated abstractions: Body and Soul. Possession System realizes a simple framework to deal with adaptation of distributed collaborative applications by changing relationship among Bodies and Souls. Furthermore, it enables users to interact with system components with a simple operation named possession. Distributed collaborative applications built on top of our middleware realize adaptive behavior, which is achieved by a mechanism combining system events with various behavioral changes of application components including the migration of components. The middleware is applicable to adaptive systems in mobile and ubiquitous computing environment where unpredictable changes in available computing resources, device configuration, and geographic location frequently occur. We describe how collaborative applications using our middleware adaptively behave.

1. Introduction

With the maturation of a mobile computing environment and a forthcoming wearable computing environment, the usage of collaborative applications is diversified. We have focused on a particular type of application, which can dynamically change functionality and services according to the contextual changes of activities among individuals and groups. Adaptation mechanisms for applications are required for this purpose, and we extend the concept of adaptation from the application-aware adaptation [1], which is achieved by the cooperation between system- and application-level mechanisms, to the multiuser-aware adaptation so as to accommodate the characteristics of an adaptive collaborative application.

An example scenario can be introduced as follows. Group members in a distributed environment use video conferencing applications, and send video data captured at each location. Some of the members might move to the same meeting room, and stop sending video data to each other's hosts, while keep sending to the distant hosts. One member might want to check the geographic location of all the members by displaying the area map, and some other members might want to share the map information. Distant members might use a shared text editor for realtime text communication, while the members in the same room might prefer to pass around a virtual memo object. The example indicates that the long-term use of collaborative applications in a mobile computing environment causes frequent and successive changes in types and ways of services suitable for a certain situation. Moreover, one service can be performed in relation to other services.

Therefore, adaptation mechanisms are required not only to guarantee the continuity of services provided by legacy applications but to satisfy requests from novel applications which actively use information on contextual changes to provide advanced services. The contextual information includes the changes of group members, proximity of people, location, and schedules, and a part of them was formerly addressed in the pioneering research [2]. The adaptation mechanisms are realized and reinforced by the advent of networked sensors and devices that provide information on the internal state of computing devices, physical environment surrounding users, and context of human activities. From the above-mentioned point, adaptive collaborative applications should be developed based on a common adaptation framework to achieve flexibility and extendibility.

In our research, we developed new middleware named *Possession System*, which contribute to the easy development of adaptive collaborative applications built of a collection of distributed components in Java. Adaptation supported in the middleware is briefly described as the automatic adjustment of system state and configuration responsive to dynamic envi-

† Graduate School of Media and Governance, Keio University
†† Faculty of Environmental Information, Keio University

ronmental changes. The system characteristics can be described as follows; (1) it adopts an abstraction for transparent access to distributed devices, sensors, and application components, (2) it provides a framework to relate system-level environmental event management mechanisms with application-level adaptive behavior, and (3) it enables dynamic placement and reconfiguration of distributed application components with a simple operation partially deploying object migration facilities.

The rest of this paper is organized as follows: Section 2 explains our research approach on distributed adaptive applications. Section 3 describes the Possession Model which provides basic abstraction to our middleware, and Section 4 explains the system architecture and prototype implementation. Then Section 5 describes several applications implemented on top of the prototype system, and Section 6 discusses overall consideration. Finally, Section 7 addresses related works, followed by the final section with conclusion.

2. Adaptation in Collaborative Applications

The concept of adaptation in computer science is wide-ranging and covers a variety of areas such as operating systems, computer networks (including mobile networks), user interfaces, applications, etc. The characteristics of our research can be explained in the aspect of adaptation policies and distributed communication paradigms.

2.1 Adaptation Policies

Adaptation policies can be viewed from the following aspects:

(1) **System-level vs. Application-level**
System-level adaptation is usually self-contained and hidden from application-levels. On the other hand, in application-level approach, application programmers need to design and implement adaptation mechanisms individually for themselves. System- and Application-level approaches do not inherently conflict with each other.

(2) **Hidden vs. Perceivable**
The process or result of adaptation can be hidden from applications or users to provide transparency in different environments (e.g., a stationary environment and a mobile environment). On the other hand, perceivable adaptation is needed to develop software supporting human collaborative activities because it is necessary to take human perception and response into account due to reflect them as system feedback for further adaptation process.

(3) **Coarse-grained vs. Fine-grained**
This aspect is related to the minimum unit of adaptation. Command or process execution appropriate for a certain context is an example of coarse-grained adaptation. Fine-grained adaptation is performed within an application process without restarting the process.

(4) **Symmetric vs. Asymmetric**
In the symmetric adaptation approach, updates resulting from the adaptation of a collaborative application at one host occur in the same manner in other applications at other hosts irrespective to the difference of each computing environment. In the latter approach, adaptation occurs in a variety of ways at each collaborative application in distributed hosts reflecting the differences of each environment. This implies that a process of adaptation advances through both environment-independent and environment-dependent stages.

In our research, adaptation is enhanced by providing a framework for the cooperation between system- and application-level. Moreover, multiuser-aware adaptation is supported in the framework by incorporating the Perceivable aspect. At the same time, the Fine-grained and Asymmetric adaptation approaches are appropriate for our system because users can flexibly specify that a certain service is sharable or non-sharable, while it is not only easy to customize and personalize adaptive applications but suited to relocate the application components in a distributed environment.

2.2 Distributed Communication Paradigms

Our main concern is in the placement policies and access methods to distributed resources, therefore an abstraction methodology for mobile code paradigm presented in Ref. 3) is suitable for this purpose. In the methodology, distributed applications are decomposed into four basic elements. The elements are Resource components (code, data, or physical devices), Computational components (flow of control such as process and thread), Interactions (event and information passing between two or more components), and Sites (execution environment). According to the differences of relationships among the basic elements, mobile code paradigms of distributed applications are classified into the following 5 types which consist of original four paradigms presented in Ref. 3) and one additional type provided in Ref. 4):

- Client/Server (CS): The code, data, and execution remain fixed at the server site S_B. Component A at Site A sends data to Component B at Site B. Execution is performed at S_B.
- Remote Execution (REV): Component A at Site A sends code and data to Component B at Site B.
- Code on Demand (COD): Inversion of REV. Component A requests code and data from Component B.
- Remote Code Execution (RCE): Union of REV and COD.
- Mobile Agent (MA): Computational components can migrate together with code and data from Site A to Site B.

In the Possession System, COD, REV, and MA paradigms are adopted for the purpose of run-time system management; downloading/uploading of Resource and Computational components from/to repositories or other hosts, distributed placement and configuration of the components, and update of each component. Communication among application components are based on CS paradigm. MA paradigm is usually applied to task-oriented processing, which means that a task is allocated to an agent and the agent itself is terminated soon after the task has completed. In the Possession System, however, multiple processing entities can be related to a single target task. Furthermore, the system can dynamically select whether a processing entity should migrate and perform a task at a remote site or the entity should stay and start the remote execution of the task at a local site.

3. Possession System

Possession System is designed and implemented by a component model named *Possession Model*. We will briefly explain the concept of the model and then move on to the system implementation details.

3.1 Design Goals

We have the following design goals:

Describability: This feature is achieved by the separation between the description of component's procession and composition. Applications can be composed by the description of simple configuration script.

Reusability: Components (Bodies and Souls) can be shared and reused in a distributed environment. Body instances used in one application service can be reused in other application services without reinstantiation.

Flexibility: An application made of Souls and Bodies can be dynamically and easily reconfigured to brand new application serving to different purposes.

Extendibility: Application functionalities can be upgraded on a per component basis. Each component can be replaced with a new version through the RCE paradigm.

Adaptability: Reconfiguration of applications can be automatically performed in accordance with various environmental events.

Mobility: Soul component can migrate to remote sites in order to dispatch the execution. MA paradigm is incorporated in the Possession System by this feature.

In the existing component-based middleware, the characteristics of reusability, flexibility, and extendibility are not integrated with the adaptability and mobility features in terms of the long-term execution of applications with successive environmental changes enough to facilitate the adaptive collaborative application development. In order to address the issue, middleware need to provide application programmers with a unified framework.

3.2 Possession Model

Possession Model is designed to give an abstractive view on the relationships of system components. It is based on an analogy that a soul "possesses" a body. Hence, the model provides a simple framework to see collaborative multimedia applications as a collection of components named Souls and Bodies. The separation between Souls and Bodies corresponds to the decomposition of components into Computational and Resource components.

Body Component: Body is designed as a static entity in the Possession Model, and it is activated by Soul's possession operation. Body can be classified into the following four categories:

(1) Core Bodies: System components which provide fundamental features including component instance management and communication facilities.

(2) Graphics Bodies: GUI components and drawing areas (e.g., window and animation sprite objects).

(3) Device Bodies: Wrapper objects for various sensors and devices (e.g., video capture de-

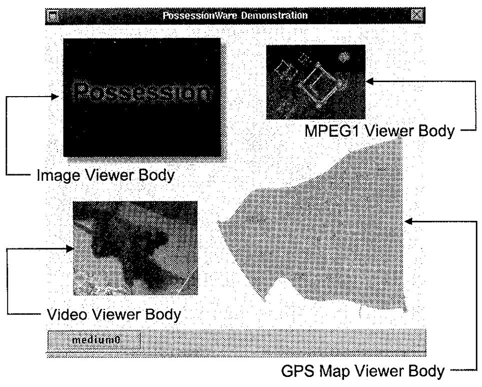


Fig. 1 Some examples of Bodies.

vice, GPS device, audio device, and information appliances).

(4) Filter Bodies: Two types of filter Bodies exist; one is used for simple filtering purposes (e.g., compression/decompression, encryption/decryption, text and image format conversion, etc.), the other is used for intelligent processing (e.g., image recognition and voice recognition).

Body is modeled as a Java object and its basic functions are defined in `BodyComponent` class. If users would like to create a new Body, they need to define a remote interface at first and the new Body has to inherit `BodyComponent` class and implement a remote interface. **Figure 1** shows sample implementations of Bodies at the time of writing.

Soul Component: Soul is an active entity in the Possession Model, and it knows how to control a Body (interface) and what to do while it is executed (thread of execution). Soul is written in Java and its basic functions are provided in `SoulComponent` class. When users create a new Soul, `SoulComponent` class has to be inherited and Body remote interface must be implemented so as to control the Body. If a Soul would like to control several different types of Bodies, the Soul must implement all corresponding Body remote interfaces.

Soul has two main roles; one is to describe contents of its execution and the other is to branch off its method invocation. As for the former role, since Body instance only provides functionality of the entity the Body is representing and its method implementation, the description of control flow has to be defined as a Soul at compile time. The latter role works when multiple Bodies are registered in a Soul. When the Soul executes its thread and invokes its methods, the invocation is branched off and methods of multiple Bodies are called. For this purpose, Soul and multiple Bodies need to implement the same remote interface. For homogeneous multiple Bodies, Soul has to implement a common remote interface, and for heterogeneous multiple Bodies, Soul has to implement multiple remote interfaces enough to control all types of Bodies.

Soul has an ability to migrate among distributed hosts. This ability enables it to incorporate the MA paradigm to the Possession System. The Soul's migration facilities basically provide a simple method to change a place to process Soul's execution. However, it can be used for other purposes by connecting the Soul with various types of Bodies.

Possession Operation: Possession operation is performed by Souls to select and register target Bodies to be controlled. In other words, it creates a dynamic binding between Resource components and Computational components. Since several different types of Bodies exist, we can consider the following possession patterns; a Soul possessing a Body (Soul-to-Body), multiple homogeneous Bodies (Soul-to-homogeneous Bodies), multiple heterogeneous Bodies (Soul-to-heterogeneous Bodies), and multiple Souls possessing a single Body (homogeneous/heterogeneous Souls-to-Body). In addition to a normal possession pattern where Souls possess Bodies, a Soul can also possess another Soul.

When a possession operation is called, class type checking and access control is internally performed. In the current implementation, Java reflection API is used for this purpose. Access control mechanism is introduced to ensure ownership of both Souls and Bodies, and achieved by an access control list mechanism.

3.3 System Architecture

The major components of the Possession System, other than Souls and Bodies, are Field, Mediums, and Possession Shell. Each component can be described as follows:

Field: Field exists on each host and provides a basic communication mechanism to send and receive system management information among remote Fields. Moreover, it accommodates name registration and lookup facilities.

Medium: Medium is a logical management unit and gives Bodies distinct name space. Bodies created in a current Medium are activated, and active groups of Bodies are switched

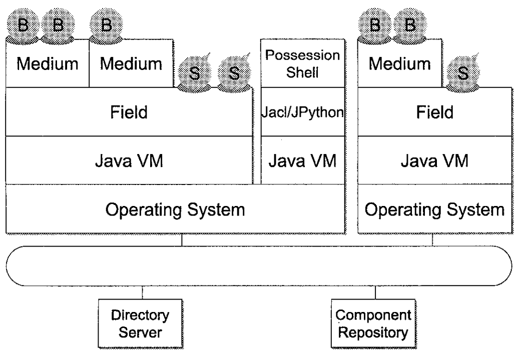


Fig. 2 Possession system architecture.

according to the selection of current Medium by users.

Possession Shell: Users can manually control the system by the Possession shell. A notable feature here is that the shell itself is implemented as a Body. Possession shell can therefore be possessed and controlled by a Soul, and appropriate script is evaluated according to event notification from Device Bodies or an event management Body in order to perform adaptation behavior of applications composed by multiple Bodies.

Entire system architecture is depicted in **Fig. 2**. It shows Possession Systems running on both Host A and Host B. A collaborative multimedia application is composed as the collection of distributed Bodies and Souls. Communication between system components are based on Java Remote Method Invocation (RMI). Users access system components with a Possession shell which is implemented with Jacl[5] or JPython[6].

3.4 Adaptation Support Mechanism

Adaptation in the Possession System is achieved by the following methods: (1) By downloading a new component from a network and replacing an old one with it, (2) By letting Soul change the attributes of possessing Body, and (3) By changing the configuration of application components. Any of the preceding methods or combination of the methods are required to be selected and applied with the analysis of information obtained from devices and sensors or system-providing environmental event manager such as Ref. 7). Adaptation support mechanisms, therefore, should be designed to fulfill those requirements. A typical adaptation support mechanism is shown in **Fig. 3**. Inside the oval shows the mechanism supporting adaptation of collaborative multimedia applications by

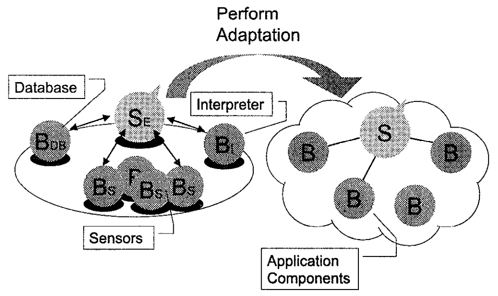


Fig. 3 Adaptation support mechanism.

```
1: m = cmd.cm()
2: s = cmd.cs('comp.SoulScc')
3: b1 = cmd.cb(m, 'comp.BodyScc')
4: b2 = cmd.cb(m, 'comp.BodyVideoUI')
5: s.possessIn(b1)
6: s.possessOut(b2)
7: s.start()
```

Fig. 4 Configuration script (vidviewer.py).

the method (3).

S_E works as a sort of environment manager. It collects information from B_S which represents Bodies as sensors, and decides an adaptation policy. An adaptation policy is described as a script or a set of scripts stored in a Database (B_{DB}). S_E downloads an appropriate script from B_{DB} and executes it by interacting with B_I which represents a script interpreter. The cloud part denotes an application which consists of a Soul and multiple Bodies. The configuration of the application components is changed gradually or drastically by the evaluation of adaptation scripts in a network transparent manner.

A simple example of the configuration script written in JPython scripting languages is presented in **Fig. 4**. The example shows the configuration of a video viewer application. Each number shown at left-side denotes a line number. From line number 1 to 4, Medium, Soul, video capture Body, and video viewer Body are respectively created. At line 5, the Soul possesses the video capture Body for input and also possesses the video viewer Body for output at line 6. Then Soul starts processing at line 7.

The example is rather iterative and straightforward. The scripting language itself, however, has expressive power enough to describe conditional behavior and time-driven behavior.

Figure 5 shows a code fragment of a simple event server. When events such as TimerEvent and LocationEvent are notified from Body Components, the server's event listener method

```
 1: public void eventPosted(EventObject event) {
 2:   if (event.toString().equals("TimerEvent")) {
 3:     Vector v = (Vector)htScripts.get("timer");
 4:     for (int i=0; i<v.size(); ++i) {
 5:       String scriptname = (String)v.elementAt(i);
 6:       String script = loadScript(scriptname);
 7:       execScript(script);
 8:       v.removeElementAt(i);
 9:     }
10:     htScript.put("timer", v);
11:   } else if (event.toString().equals("LocationEvent")) {
        ...
 n: }
```

Fig. 5 A code fragment for event handling.

(`eventPosted`) is called. Then, the server checks if any configuration script is registered in relation to the notified events. If it is registered, the corresponding configuration script is retrieved from a database (`loadScript` at line 6) and executed at an interpreter (`execScript` at line 7). For instance, a video viewer application configured by the `vidviewer.py` script will be cleaned up and a new text editor program will be launched by the execution of pre-registered `cleanup.py` and `texteditor.py` scripts respectively at a certain time specified using a scheduler application. This example presents typical integration between the adaptation support and generic event handling mechanisms. Although the adaptation example is performed by the script evaluation, it can be achieved by the simple replacement of components based on the interactive command execution between Souls and interpreter Bodies.

4. Applications

We have implemented several applications based on Possession Model including video viewer application, video conferencing application, and an example of application configuration change triggered by PC card replacement. Each of them consists of several Souls and Bodies and have ability to adapt events notified from Bodies wrapping devices.

4.1 Video Viewer Application

As an example of Possession system, we show a simple video viewer application. **Figure 6** shows the configuration of video viewer application on the host A. In the application, a Soul possesses two Bodies; one is a wrapper for a video capture device and the other works as a video viewer. The Soul (S) reads video data from B_C and draws video frames onto B_V. Arrows in the figure show the direction of data flow. Components on the host B is not related to the application at this point.

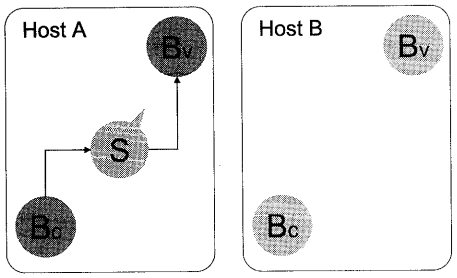


Fig. 6 A video viewer application on the host A.

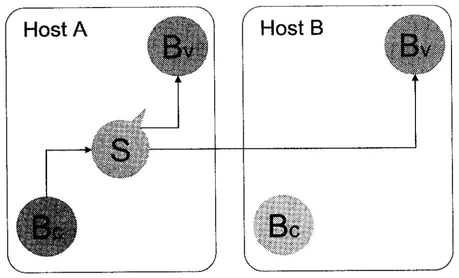


Fig. 7 Soul possessing multiple video viewer bodies.

By letting the Soul possess B_V on the host B, we can duplicate the video data flow. **Figure 7** shows the configuration that the S possesses B_V on both the host A and host B. We can deliver the video data to multiple viewers on distributed hosts in the same way.

Since Souls have ability to migrate from one Field to another, the Soul possessing the video capture device Body on the host A can migrate to the host B maintaining the connection between the Soul and the Bodies.

When a Soul migrates from one host to another, the hash table in the Field is looked up with the Soul's identifier and object instance is obtained. The identifier consists of initial hostname, field identifier, and integer value, and is assigned at the creation time and immutable during the component's lifetime. Since the hostname part includes the hierarchical domain name of Internet DNS (Domain Name System), the identifier's uniqueness is guaranteed. You can specify the Soul in the host A with the following URL convention; `rmi://hostA/hostA.field0.soul0`. The "`rmi://hostA/`" part can be omitted when you address local components.

The obtained object instance is serialized to byte codes and transmitted to the destination host, as well as the name and remote reference pair is deleted from the name service at the source host. After the byte code transmission, the Soul component is reinstantiated with the byte code, while the identifier and instance is stored in the Field's hash table and the component's name and remote reference pair is registered to the name service at the destination host. At this point, the migrated Soul component on the host B can be specified in this way; `rmi://hostB/hostA.field0.soul0`.

The Soul holds remote references to the possessing Bodies even after the migration, and keeps connection to the Body representing the video capture device on the host A and receive video data from the Body. Users can select whether the Soul should continue to reference the original Body (B_C on the host A) or it should unpossess the old Body and possess a one on the host B. The application configuration change before and after the Soul's migration is shown in **Fig. 8**. This selection should be made according to the context under which the application is used, but it is important to provide versatile options in order to increase the system flexibility.

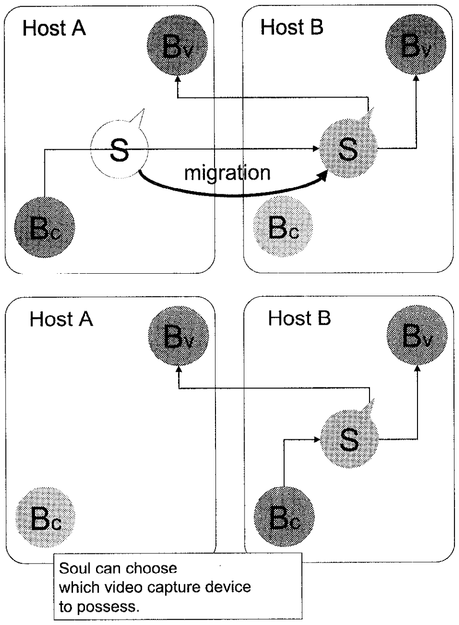
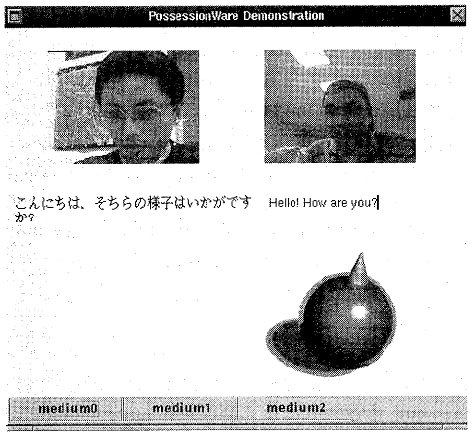


Fig. 8 Soul migration and Body selection.



こんにちは。そちらの様子はいかがですか?

Hello! How are you?

Fig. 9 Screen-shot of a video conferencing application.

4.2 Video Conferencing Application

By configuring a few additional Souls and Bodies to the video viewer application, we can realize a video conferencing application on top of Possession System. **Figure 9** shows a screen-shot of the video conferencing application and the structure is presented in **Fig. 10**.

There are two Souls and 5 Bodies in each host. S_{VA} in the host A works as a video transmitter, and it captures video data from a Body (B_{CA}) which represents video capture device and passes the data to Bodies (B_{VA} and B_{VB}) which work as video viewers. This configuration is the same as the example explained in the previous section. The other Soul in host A (S_{TA}) works as a text capturer. It possesses two Bodies representing text areas (B_{TA} on the host A and B_{TA} on the host B) and capture and transmit text data when users input text information on the Bodies. Moreover, users can add Bodies so as to share a variety of information with conference participants such as images, files, screen-shot of other applications, etc.

In this application, users can choose whether data should be sharable or non-sharable on a per Body basis. For example, if a user on the host A (user A) would not like to show a particular text information to a user on the host B (user B), user A simply let S_{TA} unpossess B_{TA} on the host B, and prohibit user B from accessing to B_{TA} on the host A by setting

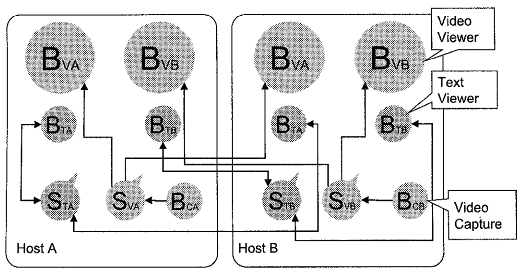


Fig. 10 Configuration of a video conferencing application.

proper ACL parameters. We can also add a video recording feature by letting a Soul drawing video data possess a newly created Body as a video data storage. When you would like to playback recorded video data, you can change a video source from a Body as a video capture device to the Body as a video data storage by possession operations.

In addition to ordinary video conferencing functionality, simple adaptation can be achieved in coordination with a Body providing locational information. For instance, when a Body as a window is shown in a large display device installed in a laboratory, the Body adapt its size according to the relative distance between users and the display device. When a user comes close to or becomes distant from the display, the Body adjust its size, smaller or larger, so as to be appropriate for the user to see the contents at the location.

4.3 Adaptation to PC Card Replacement

As an example of integration between system-level event handling mechanism and application-level adaptation mechanism, we prepared a wrapper Body that communicates with a FreeBSD standard PC card management daemon named pccardd. The wrapper Body receives the notification of insertion and removal events sent from pccardd, and the Body enables Souls to access the information of device configuration changes. **Figure 11** shows an application realizing the automatic configuration of Bodies which works as compression/decompression filters according to the exchange of network interface PC cards from a 10 Mbits Ethernet card to a 2 Mbits WaveLAN card.

5. Consideration

We evaluated basic performance of the video conferencing application. Two machines are

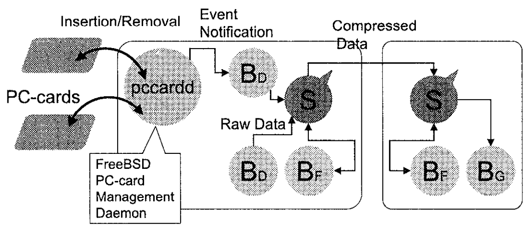


Fig. 11 Component selection based on the PC card replacement.

used for the measurement (Toshiba DynaBook SS-R590 with a 90 MHz Pentium processor and 40 MB RAM). Both are connected via 10 Mbps 3Com 3C589 Ethernet PC-cards. Captured video frame size is 160×120 and the color depth is 16 bpp. In the current implementation, drawing a frame with RMI costs 80.93 msec (12.36 fps). Moreover, it takes 91.19 msec to draw a frame (10.97 fps) from the remote host after Soul's migration. To compare local and remote execution, we implemented a normal video viewer application using Java native method. it takes 65.71 msec per frame (15.22 fps). The results indicates that drawing with RMI takes about twice as much time as drawing without RMI.

We also investigated the execution time of a possession shell command. We implemented a Tcl `null` command and measured its execution time. It costs 140.15 msec for the overall execution, and about 20% of the execution time (28.00 msec) is consumed by the internal invocation of Soul's `null` Java method and about 36% of Soul's method invocation (9.95 msec) is consumed by Body's `null` method invocation.

In the multiuser-aware adaptation, the notification on the process of adaptation through the state changes of GUI and device components is important so as to let users know and share the changes of social context and environment. Support for intellectual processing considered to be useful due to enhance system ability to deal with the contextual information including human activities. Although our middleware currently does not directly support the processing, it is possible to accommodate this by means of simple system configuration changes. For instance, you can replace a normal database with a knowledge base and substitute an interpreter Body for an inference engine, and we can incorporate AI technologies within our system framework.

As for the implementation detail of the migration scheme, we adopt the object serialization and dynamic class loading facilities provided by the Java language. It is a standard technique used in various Java-based mobile object and agent systems[8)~12)], where a thread migration technique to keep the running thread's state is not deployed because of the requirement for native code libraries which could reduce the advantages of interoperability.

With regard to the distribution of resource components, it is difficult to predict the device configuration and available resources of mobile hosts a priori. This issue is relevant to a multiuser application support because each host has to equip resource components such as GUI components, audio device, and video capture device to some extent. Therefore, the mechanism to find closest resource repository and targeted component is required with the help of mechanisms such as Service Location Protocol[13)]. Furthermore, different types of components with the same interface definition make it easy to incorporate multi-modality in a system, and it increases interoperability between hosts with different device configuration.

As a consequence of adopting Java RMI as basic communication facilities, a few drawbacks exist in communication performance and system scalability. For possible solutions to the problems, we should improve group communication performance in RMI by adopting IP multicast.

6. Related Work

Possession System has benefited from numerous prior researches in the area of distributed systems, one of which is extensive research results of process and object migration[14),15)]. In those researches, migration mechanisms are provided in both kernel- and user-level. In the kernel-level approaches[16)~19)], coarse-grained process migration mechanisms are provided, while mechanisms for the fine-grained mobility of language object are provided in the user-level language and system approaches[20)~24)].

Process migration is originally intended to be used for load-balancing, fault-tolerance, and locality of resource reference, where migration itself is invisible to users and migrating processes. From this aspect, Soul's migration functionality is incorporated and used in a more explicit manner in terms of the location-aware resource access for the continuous computation and location-specific services in a heterogeneous network environment. Under the environment, mobile and wearable computers tend to be connected/disconnected to/from the network and PDAs and information appliances could more frequently be turned on/off than ordinary computers. Therefore the requirement for the migration functionality is potentially high, although the invisibility provided in the kernel-level approaches is not necessarily required.

The user-level language and system approach realizes finer-grained object migration independent of the kernel-level process migration mechanism. It provides sufficient support for creating mobile and distributed applications, though it is rather complex to construct applications from scratch. For instance, Obliq[24)], an object-oriented scripting language with distributed scope, provides language primitives enough to create mobile agents. Application programmers, however, need to write programs using low-level language primitives being fully aware of the distributed scoping. On the contrary, Possession System provides higher-level abstraction and its own semantics based on abstraction by Soul and Body, thus application programmers need not care of the lower-level language semantics. Moreover, the Possession Model itself is independent of a specific language, and it is possible to implement a system with the equivalent functionality in the aforementioned languages.

StratOSphere[4)] provides a framework to unify distributed objects and mobile code applications. The design is based on a layered architecture, where various types of communication paradigms, CS, REV, COD, RCE, and MA, are supported at different layers. Furthermore, run-time object adaptation is achieved by the dynamic loading of new method implementations, though the method invocation syntax is not fully merged in that of Java language. Therefore it imposes additional programming efforts on application programmers.

From a conceptual viewpoint, the object model in the StratOSphere is designed mainly to encapsulate the geographical map data stored in a digital library, which is inherently suitable for MA type task-oriented procession. The object methods are a set of operations applied to modify the map data contents, and those are defined in terms of the place to deliver the map data in a certain order and the type of computation to perform at the place. On the other hand, Soul's behavior and service interface is determined in terms of what kind of applications and services a certain group of Souls and Bodies provide. Therefore, the distribution, selection, and integration of appropriate components are our main concerns. The role of MA paradigm in our system is rather equivalent to other communication paradigms than in the StratOSphere.

With regard to the security issues related to the object mobility, our current prototype prohibits direct access to the system resources by separating components into Body and Soul and doing access control at the time of the possession method invocation, as well as protecting the resources by the code verification and security manager mechanisms provided in Java[25),26)]. These mechanisms mainly deal with host integrity issues, while we have been developing the successive implementation on the Java 2 platform to cover the mobile code integrity issues by incorporating the security enhancements for a secure object[27),28)].

In the researches of adaptive applications in a mobile computing environment[7),29),30)], system-level resource management mechanisms and event delivery mechanisms are their major research concerns. In the researches, QOS adaptation of applications are mainly treated as examples of application-aware adaptation to demonstrate the system-level mechanisms. Although our research is not directly focused on a system-level resource management mechanism, we provide adaptation mechanisms at the application layer and application framework to relate the system-level mechanisms with application-level adaptive behavior.

VNC (Virtual Network Computing) formerly known as Teleporting System, supports application-level adaptation of mobile applications in the way that applications migrate according to locational information and the migrated applications adapt themselves according to the characteristics of computing environments such as display resolution, display size, supported devices, etc. The mobility of applications is achieved by the transmission of remote frame buffer data rather than the actual code mobility, thus the migration and adaptation are on a per-desktop basis and remain coarse-grained. In Refs. 31)–33), services equivalent to the Teleporting system is provided in a different way that an application is composed of a collection of mobile active objects and its migration can be performed on a per-object basis. However, they place mobile agents on their center of application coordination, therefore applications are not beyond the scope of the ones targeted in the current MA paradigm.

Although several application scenarios are presented in the related researches, what kind of services users actually require and how they apply developed technologies are usually beyond researchers expectations. With the rapid evolution and proliferation of Java and distributed object technologies, worldwide computing infrastructure for practical experiments has been improved, and our further development of practical applications remain to be done.

7. Conclusion

We have designed and developed new middleware named Possession System, which supports the adaptation of adaptive collaborative applications described in Java.

- It provided an unified framework to access distributed resource components (application components, devices, and sensors) based on an abstraction by Soul and Body.
- It presented a method to relate system-level environmental information with application-level adaptation behavior. The adaptation is achieved by the following 3 methods; (1) the definition of individual Soul's behavior, (2) the dynamic reconfiguration of relationships among components through by switching multiple configuration scripts based on event notifications, and (3) the replacement of each component implementation by RCE paradigm.
- It realized a simple method for the reconfiguration of distributed multiuser applications by means of (1) remote interface sharing and dynamic remote referencing between Soul and Body, (2) object composition support by scripting, and (3) dynamic distributed placement of Computational Resources enhanced by object mobility.

Besides the issues we have already described in the preceding sections, the issue of implementing components enough to construct a variety of actual applications is left for the future work. In addition, we are currently working on the system extension so as to apply it to the actual target domain of information appliances and conduct a further evaluation. It will include the support for the construction of Virtual Network Appliances (in short, VNA) which can be assembled by borrowing the partial functionality of ubiquitous physical information appliances in an ad hoc and context-aware manner.

Acknowledgments The authors would like to thank the members of KMSF and SMAF Project for valuable discussions and comments. We also thank the anonymous reviewers for proofreading and feedback.

References

1) Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J. and Walker, K.R.: Agile Application-Aware Adaptation for Mobility, *Proc. 16th ACM Symposium on Operating System Principles* (1997).
2) Schilit, B., Adams, N. and Want, R.: Context-Aware Computing Applications, *IEEE Workshop on Mobile Computing* (1994).
3) Carzaniga, A., Picco, G. and Vigna, G.: Designing Distributed Applications with Mobile Code Paradigms, *Proc. 19th International Conference on Software Engineering*, pp.22–32 (1997).
4) Wu, D., Agrawal, D. and Abbadi, A.E.: StratOSphere: Mobile Processing of Distributed Objects in Java, *Proc. MOBICOM'98* (1998).
5) Lam, I.K. and Smith, B.: Jacl: A Tcl Implementation in Java, *Proc. 5th Anual Tcl/Tk Workshop 1997* (1997).
6) Hugunin, J.: Python and Java – The Best of Both Worlds, *Proc. 6th International Python Conference* (1997).
7) Nakajima, T., Aizu, H., Kobayashi, M. and Shimamoto, K.: Environment Server: A System Support for Adaptive Distributed Applications, *Worldwide Computing and Its Applications (WWCA'98)* (1998).
8) Lange, D.B., Oshima, M. and Kosaka, K.: Aglets: Programming Mobile Agents in Java, *Proc. WWCA'97* (1997).
9) Tripathi, A.R., Karnik, N.M., Vora, M.K., Ahmed, T. and Singh, R.D.: Mobile Agent Programming in Ajanta, *IEEE International Conference on Distributed Computing Systems (ICDCS'99)* (1999).
10) Johansen, D., van Renesse, R. and Schneider, F.B.: An Introduction to the TACOMA Distributed System Version 1.0, Technical Report 95-23, Department of Computer Science, University of Tromsø, Norway (1995).
11) Baumann, J., Hohl, F., Rothermel, K. and Straßer, M.: Mole-Concepts of a Mobile Agent System, *World Wide Web*, Vol.1, No.3, pp.123–137 (1998).
12) Cugola, G., Ghezzi, C., Picco, G. and Vigna, G.: Analyzing Mobile Code Languages, *Mobile Object Systems: Towards the Programmable Internet*, Vitek, J. and Tschudin, C. (Eds.), LNCS, Vol.1222, pp.93–111, Springer (1997).
13) Veizades, J., Guttman, E., Perkins, C. and Kaplan, S.: Service Location Protocol, RFC 2165, Internet Engineering Task Force, Network Working Group (1997).
14) Smith, J.M.: A Survey of Process Migration Mechanisms, *ACM Operating Systems Review*, Vol.22, No.3, pp.28–40 (1988).
15) Milojičić, D., Douglis, F., Paindaveine, Y. and Wheeler, R.: Process Migration, Technical Report HPL-1999-21, Hewlett-Packard Laboratories (1999).
16) Powell, M.L. and Miller, B.P.: Process Migration in DEMOS/MP, *Proc. 9th ACM Symposium on Operating Systems Principles*, pp.110–119 (1983).
17) Almes, G.T., Black, A.P., Lazowska, E.D. and Noe, J.D.: The Eden System: A Technical Review, *IEEE Trans. Softw. Eng.*, Vol.SE-11, pp.43–59 (1985).
18) Black, A.P.: Supporting Distributed Applications: Experience with Eden, *Proc. Tenth ACM Symposium on Operating Systems Principles*, pp.181–193 (1985).
19) Cheriton: The V Distributed System, *Comm. ACM*, Vol.31, No.3, pp.314–333 (1988).
20) Jul, E., Levy, H., Hutchinson, N. and Black, A.: Fine-Grained Mobility in the Emerald System, *ACM Trans. Computer Systems*, Vol.6, No.1, pp.109–133 (1988).
21) Shapiro, M. and Gautron, P.: Persistence and Migration for C++ Objects, *Proc. European Conference on Object-Oriented Programming (ECOOP'89)* (1989).
22) Lea, R., Jacquemot, C. and Pillevesse, E.: COOL: system support for distributed programming, *Comm. ACM*, Vol.36, No.9, pp.37–46 (1993).
23) Achauer, B.: The DOWL distributed object-oriented language, *Comm. ACM*, Vol.36, No.9, pp.48–55 (1993).
24) Cardelli, L.: A language with distributed scope, *Computing Systems*, Vol.8, No.1, pp.27–59 (1995).
25) Gong, L.: Secure Java Class Loading, *IEEE Internet Computing*, Nov/Dec (1998).
26) Rubin, A.D. and Geer, D.E.: Mobile Code Security, *IEEE Internet Computing* (1998).
27) Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, *Proc. USENIX Symposium on Internet Technologies and Systems*, pp.103–112 (1997).
28) Gong, L. and Schemers, R.: Signing, Sealing, and Guarding Java Objects, *Lecture Notes in*

Computer Science (LNCS), Vol.1419, Springer-Verlag (1998).

29) Inouye, J., Cen, S., Pu, C. and Walpole, J.: System Support for Mobile Multimedia Applications, *Proc. 7th International Workshop on Network and Operating System Support for Digital Audio and Video* (*NOSSDAV'97*) (1997).

30) Welling, G. and Badrinath, B.R.: A Framework for Environment Aware Mobile Applications, *IEEE International Conference on Distributed Computing Systems* (*ICDCS'97*) (1997).

31) Bates, J., Halls, D. and Bacon, J.: A Framework to Support Mobile Users of Multimedia Applications, *ACM Mobile Networks and Nomadic Applications* (1992).

32) Bates, J., Halls, D. and Bacon, J.: Middleware Support for Mobile Multimedia Applications, *ICL Systems Journal*, Vol.11, No.4 (1997).

33) Bacon, J. and Halls, D.: Mobile Applications for Ubiquitous Environments, *ICL Systems Journal*, Vol.11, No.4 (1997).

(Received March 31, 1999)
(Accepted October 7, 1999)

Masahiro Mochizuki received his B.A. degree in Policy Management from Keio University in 1994. He received M.A. degree in Media and Governance from Keio University in 1996. He is a Ph.D. candidate at graduate school of Media and Governance, Keio University. He is currently studying mobile and adaptive middleware and applications. He is a member of ACM and JSSST.

Hideyuki Tokuda received his B.S. and M.S. degrees in electrical engineering from Keio University in 1975 and 1977, respectively; a Ph.D. degree in computer science from the University of Waterloo in 1983. He joined the School of Computer Science at Carnegie Mellon University in 1983, and is an Adjunct Associate Professor from 1994. He joined the Faculty of Environmental Information at Keio University in 1990, and is currently an Executive Vice President and a Professor in the Faculty of Environmental Information, Keio University. His current interests include distributed real-time systems, multimedia systems, mobile systems, communication protocols, massively parallel/distributed systems, and embedded systems.