Mobile Agent-based Compound Documents

Ichiro Satoh
National Institute of Informatics /
Japan Science and Technology Corporation
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Tel: +81-3-4212-2546 Fax: +81-3-3556-1916
ichiro@nii.ac.jp

ABSTRACT

This paper presents a mobile agent-based framework for building mobile compound document, which can each be dynamically composed of mobile agents and can migrate itself over a network as a whole, with all its embedded agents. The key of this framework is that it builds a hierarchical mobile agent system that enables multiple mobile agents to be combined into a single mobile agent. The framework also provides several value-added mechanisms for visually manipulating components embedded in a compound document and for sharing a window on the screen among the components. This paper describes this framework and some experiences in the implementation of a prototype system, currently using Java the both implementation language and component development language, and then illustrates several interesting applications to demonstrate the framework's utility and flexibility.

1. INTRODUCTION

The notion of compound documents is a document-centric component framework, where various visible parts, such as text, images, and video, created by different applications, can be combined into one document and independently manipulated in-place in the document. Several frameworks for software components have been developed, such as COM/OLE [4], OpenDoc [1], and CommonPoint [11], although some of these have been discontinued. Although there are few compound document frameworks available on the market today, their advent appears to be necessary and unavoidable in the long run. Nowadays, documents must be available in network environments, but there have been several problems in the few existing compound document frameworks. A compound component is typically defined by two parts: contents and codes for modifying them. Contents are stored in the component but the codes for accessing them are often not. Thus, when a user receives a document, he/she cannot view or modify its contents, which need the support of different applications, if he/she does not have the applications themselves. Moreover, compound documents are inherently designed as passive entities in the sense that they are transmitted over a network by external network systems such as electronic mail systems and workflow management systems. Therefore, it is difficult to apply network processing to documents according to their requirements or inner components. However, some documents should determine their own destinations and itineraries according to their contents, and some should be encrypted before transmitting over the network.

This paper proposes a new framework for building mobile compound documents. The key idea is to construct each document as a collection of mobile agents, because accessing compound documents over a network requires a powerful infrastructure for building and migrating, such as mobile agents. Mobile agents are self-contained and autonomous programs that can travel from computer to computer under their own control. When an agent migrates over the network, both the state and the code can be transferred to the destination. As a result, the agent can continue its processing without needing to be restarted or reinitialized, after arriving at its destination. Therefore, such a document is a self-contained entity in the sense that it can view or modify itself by using its own code. Also, such a document can deliver itself to one or more computers over a network according to its contents, independently of external network systems. Therefore, I built a framework based on a unique mobile agent system, called *MobileSpaces*, [13]. The system is constructed using Java language [2] and provides mobile agents that can move over a network, like other mobile agent systems. However, it also allows multiple mobile agents to be hierarchically assembled into a single mobile agent. Consequently, in this framework, a compound document is a hierarchical mobile agent that contains its contents and a hierarchy of mobile agents, which correspond to nested components embedded in the document. Furthermore, the framework offers several mechanisms for coordinating visible components so that they can effectively share the visual real estate on a user screen in a seamless-looking way.

This paper is organized as follow: Section 2 surveys related work and Section 3 presents the basic ideas of the compound document framework, called *MobiDoc*. Section 4 details its prototype implementation and Section 5 discusses the usability of this framework based on three practical examples. Section 6 makes some concluding remarks.

2. APPROACH

This section briefly describes the basic ideas of the framework for building mobile agent-based compound documents, called *MobiDoc*.

2.1 Software Components as Mobile Agents

As defined in [19], a compound document is constructed as a collection of visible components, such as text and images. On the other hand, a mobile agent resembles a software component in the

sense that each agent is a self-contained module holding its code and state. Moreover, each agent can migrate itself or be migrated to another computer. However, most existing mobile agent systems unfortunately lack any mechanism for structurally assembling more than one mobile agent into a single mobile agent. This is because each mobile agent is basically designed as an isolated entity that always acts and migrates independently. Since a mobile agent is characterized by its mobility, a composition of mobile agents must be designed to keep their mobility. This framework is therefore built on a mobile agent system, named MobileSpaces, [13]. The system supports the execution and migration of mobile agents and introduces two concepts: *Agent Hierarchy* and *Group Migration*.

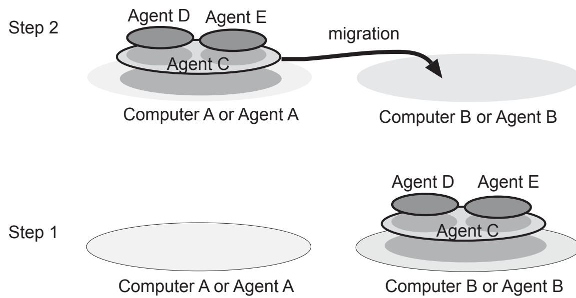

Figure 1: Agent hierarchy and group migration

- **Agent Hierarchy** means that each mobile agent can be contained within one mobile agent. It lets us assemble more than one mobile agent into a single mobile agent in a tree structure.

- **Group Migration** means that each mobile agent can migrate to another location as a whole, with all of its inner agents. This allows a group of mobile agents to be treated as a single mobile agent in their agent migration.

The first concept is needed in the development of a mobile compound document, because such a document should be able to contain other components, like OpenDoc. In an agent hierarchy, each agent is still active and mobile and thus can freely move into any computers or any agents in the same agent hierarchy except into itself or its inner agents, as long as the destination accepts the moving agent. The second concept enables a compound document to migrate itself and its components as a whole. Accordingly, a compound document is implemented as a collection of mobile components and can be treated as a mobile component. Figure 1 shows an example of group migration in an agent hierarchy.

2.2Mobile Agent-based Compound Documents

The notion of compound documents is a metaphor for organizing multiple components, where MobileSpaces can treat a mobile agent as a mobile component or a composition of multiple components and ship them across networks to other desktops. Consequently, MobileSpaces is an infrastructure for building shippable compound documents, but does not provide any document-centric mechanisms for managing components in a compound document. Therefore, we need a compound document framework for supporting mobile agent-based components, including graphical user interfaces for manipulating visible components. Also, compound documents need to have an intuitive way to be activated as active contents, displayed and edited them seamlessly in a window, and stored in a file. For example, a compound document must be able to contain various kinds of contents, text, images, and video, created by different applications, and draw them in their proper representation manners on the document. Also, switching among visual components within a document or across documents should be less intrusive than switching among conventional applications. This framework, called *MobiDoc*, is implemented as a collection of Java objects that belong to one of about 50 classes. It defines the protocols that let components embedded in a document communicate with each other. It offers an in-place editing service similar to those provided by OpenDoc and OLE. The service allows a user to immediately edit any content in-place without having to launch and execute different applications to create and assemble data. The framework should offer several mechanisms for effectively sharing the visual estate of a container among embedded components and for coordinating their use of shared resources, such as the keyboard, mouse, and windows.

3. IMPLEMENTATION

This section briefly overviews MobileSpaces and describes how to construct components and compound documents as mobile agents. The system can execute and migrate mobile agents that are incorporated using the two concepts presented in the previous section as much as possible. It has been incorporated in Java Development Kit version 1.2 and can run on any computer that has a runtime system compatible with this version.

3.1 MobileSpaces Runtime System

The MobileSpaces runtime system is a platform for executing and migrating mobile agents. It is built on a Java virtual machine and mobile agents are Java objects [2]. Each component is implemented a mobile agent in the system and the containment hierarchy of components in a document is implemented as an agent hierarchy managed by the system. The runtime system has the following functions:

Agent Hierarchy Management

The agent hierarchy is given as a tree structure in which each node contains a mobile agent and its attributes. The runtime system is assumed to be at the root node of the agent hierarchy. Agent migration in an agent hierarchy is performed just as a transformation of the tree structure of the hierarchy. In the runtime system, each agent has direct control of its inner agents. That is, a container agent can instruct its embedded agents to move to other agents or computers and can marshal and destroy them. In contrast, each agent has no direct control over its container agent, but each container offers a set of service methods that can be accessed by it embedded agents.

Agent Execution Management:

The runtime system maintains the life-cycle of agents: initialization, execution, suspension, and termination. When the life-cycle state of an agent is changed, the runtime system issues events to invoke certain methods in the agent and its containing agents. Moreover, the runtime system enforces interoperation among mobile agent-based components. It monitors the changes in components and propagates certain events to the right ones. For example, when a component is added to or removed from its container component, the system dispatches specified events to the component and the container.

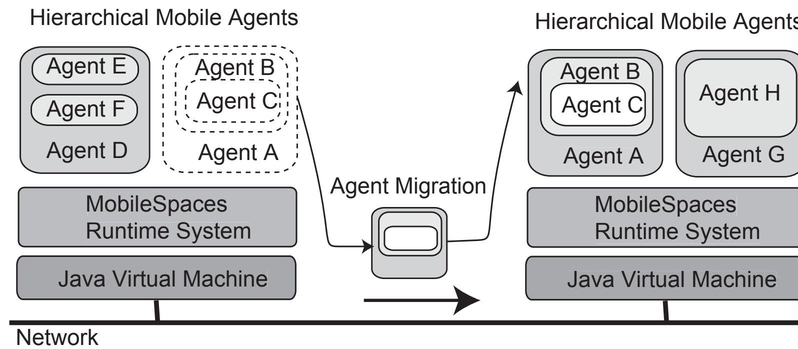

Figure 2: Agent migration between two mobilespaces runtime systems.

Agent Migration Management:
Each document is saved and transmitted as a group of mobile agents. When a component is moved inside a computer, the component and its inner components can still be running. When a component is transferred over a network, the runtime system stores the state and the code of the component, including the components embedded in it, into a bit-stream formed in Java's JAR file format that can compact the stored agent in zip-compression and support digital signatures for authentication. The system provides a built-in mechanism for transmitting the bit-stream over the network by using an extension of the HTTP protocol.[1]

The current system basically uses the Java object serialization package for marshaling components. The package does not support the capturing of stack frames of threads. Instead, when a component is serialized, the system propagates certain events to its embedded components to instruct the agent to stop its active threads. For example, each component can have one or more activities that are performed using the Java thread library, but needs to capture certain events issued to stop its own activities before it migrates over a network.

3.2 Mobile Agent-based Component

Each component, including a compound document is implemented as a mobile agent, which consists of a body program and a set of services implemented in Java language. The body program defines the behavior of the component and the set of services defines various APIs for components embedded within the component. Every agent program has to be an instance of a subclass of the abstract class `ComponentAgent`, which consists of some fundamental methods to control the mobility and life-cycle of a mobile agent-based component.

```
 1: public class ComponentAgent extends Agent {
 2:   // (un)registering services for inner agents
 3:   void addContextService(
 4:     ContextService service) { ... }
 5:   void removeContextService(
 6:     ContextService service) { ... }
 7:   ....
 8:   // (un)registering listener objects
 9:   // to hook events
10:   void addListener(
```

[1] Agent transmission and routing mechanisms of the MobileSpaces can be dynamically changed according to the requirements of moving agents by using adaptive agent migration protocols studied in another paper [15, 16].

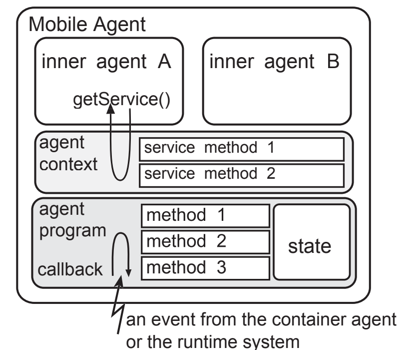

Figure 3: Structure of a mobile agent

```
11:     AgentEventListener listener) { ... }
12:   void removeListener(
13:     AgentEventListener listener) { ... }
14:   ....
15:   void getService(Service service)
16:     throws ... { ... }
17:   void go(AgentURL url)
18:     throws ... { ... }
19:   void go(AgentURL url1, AgentURL url2)
20:     throws ... { ... }
21:   byte[] create(byte[] data) throws ... { ... }
22:   byte[] serialize(AgentURL url) throws ... { ... }
23:   AgentURL deserialize(byte[] data)
24:     throws ... { ... }
25:   void destroy(AgentURL url) throws ... { ... }
26:   ....
27:   ComponentFrame getFrame() { .. }
28:   ComponentFrame getFrame(AgentURL url) { ... }
29:   ....
30: }
```

The methods used to control mobility and lifecycle defined in the `ComponentAgent` class are as follows:

- An agent can invoke public methods defined in a set of service methods offered by its container agent by invoking the `getService()` method with an instance of the `Service` class. The instance can specify the kind of service methods, which have arbitrary objects as arguments, and the deadline for timeout exception.
- The `go(AgentURL url)` method is an instruction to mi-

grate an agent to another agent or another computer. When an agent performs the `go(AgentURL url)` method, the agent migrates itself to the destination agent specified as `url`. The `go(AgentURL url1, AgentURL url2)` method instructs the inner agent specified as `url1` to move to the destination agent specified as `url2`.

- Each container agent can dispatch certain events to its inner agents and notify them when specified actions happen within their surroundings by using the `dispatchEvent()` method.

This framework provides an event mechanism based on the delegation event model introduced in the Abstract Window Toolkit of JDK 1.1 or later, like Aglets [9]. When an agent is migrated, marshaled, or destroyed, our runtime system does not automatically release all the resources, such as files, windows, and sockets, which were captured by the agent. Instead, the runtime system can issue certain events in the changes of life-cycle states. Also, a container agent can dispatch specified events to its inner mobile agent-based components at the occurrence of user-interface level actions, such as mouse clicks, keystrokes, and window activation, as well as at the occurrence of application-level actions, such as the opening and closing of documents. To hook these events, each agent can have one or more listener objects that implement specific methods invoked by the runtime system and its container component. A listener object implements a specific listener interface extended from the generic `AgentEventListener` interface that defines callback methods that should be invoked by the core system before or after the life-cycle state of the agent changes. The `ComponentEventListener` interfaces are designed for mobile components and shown as follows:

```
 1: interface ComponentEventListener
 2:   extends AgentEventListener {
 3:   // invoked after creation at url
 4:   void create(AgentURL url);
 5:   // invoked before termination
 6:   void destroy();
 7:   // invoked after accepting a child
 8:   void add(AgentURL child);
 9:   // invoked before removing a child
10:   void remove(AgentURL child);
11:   // invoked after arriving at the destination
12:   void arrive(AgentURL dst);
13:   // invoked before moving to the destination
14:   void leave(AgentURL dst);
15:   ....
16: }
```

The above interface specifies fundamental methods invoked by the runtime system, when agents are created, destroyed, persisted, and migrated to another agent. Also, each component can have listener objects to hook events in user interfaces provided by Java's libraries.

The program of every component consists of these callback methods. Since the framework provides several value-added mechanisms for managing components embedded in a compound document, a user can easily define a new component by overwriting its behaviors invoked at the specified timings on the corresponding callback methods. I have already implemented a variety of mobile agent-based components, such as text viewer/editor, image viewer for GIF and JPEG, animation viewer, drawing program, clock, document window, and so on. Each component is initially stored in a file. Also, since the runtime system provides a mechanism for saving the code of an activated component and its state in a file, a user can retrieve an active component from such a saved component and duplicate the component within a container component, without needing to initialize it.

3.3 MobiDoc Compound Document Framework

The *MobiDoc* framework is implemented as a collection of Java classes to enforce some principles of component-interoperation and a graphical user interface.

Visual Layout Management:

Each mobile agent-based component can be displayed within the estate of its container component or in a window on the screen, but it must be accessed through indirection: *frame* objects derived from the `ComponentFrame` class as shown in Figure 4.[2] Each frame object is the area of the display that represents the contents of components and is used for negotiating the use of geometric space between the frame of its container component and the frame of its component.

The frame object of each container component manages the display of the frames of the components it contains. That is, it can control the sizes, positions, and offsets of all the frames embedded within it, while the frame object of each contained component is responsible for drawing its own contents. For example, if a component invokes the `setFrameSize()` method to change the size of its frame, its frame must negotiate with the frame object of its container for its size and shape and then redraw its contents within the frame.

```
 1: public class ComponentFrame
 2:   extends java.awt.Panel {
 3:   // sets the size of the frame
 4:   void setFrameSize(java.awt.Point p);
 5:   // gets the size of the frame
 6:   java.awt.Point getFrameSize();
 7:   // sets the layout manager for
 8:   // the embedded frames
 9:   void setLayout(CompoundLayoutManager mgr) {
10:   // views the type of the component,
11:   // e.g. iconic, thumbnail, or framed,
12:   int getViewType();
13:   // gets the reference of the container's frame
14:   ComponentFrame getContainerFrame();
15:   // adds an embedded component as frame
16:   void addFrame(ComponentFrame frame);
17:   // removes an embedded component specified
18:   // as frame
19:   void removeFrame(ComponentFrame frame);
20:   // gets all the references of embedded frames
21:   ComponentFrame[] getEmbeddedFrames();
22:   // gets the offset and size of the inner frame
23:   // specified as cf
24:   java.awt.Rectangle getEmbeddedFramePosition(
25:                           ComponentFrame cf);
26:   // sets the offset and size of the inner
27:   // frame specified as cf
28:   void setEmbeddedFramePosition(
29:     ComponentFrame cf, java.awt.Rectangle rect);
30:   ....
31: }
```

When one component is active, another component is usually de-

[2]Although the `ComponentFrame` class is a subclass of the `java.awt.Panel` class, we call instances derived from the class *frame* objects because many existing compound document frameworks often call the visual space of an embedded component a *frame*.

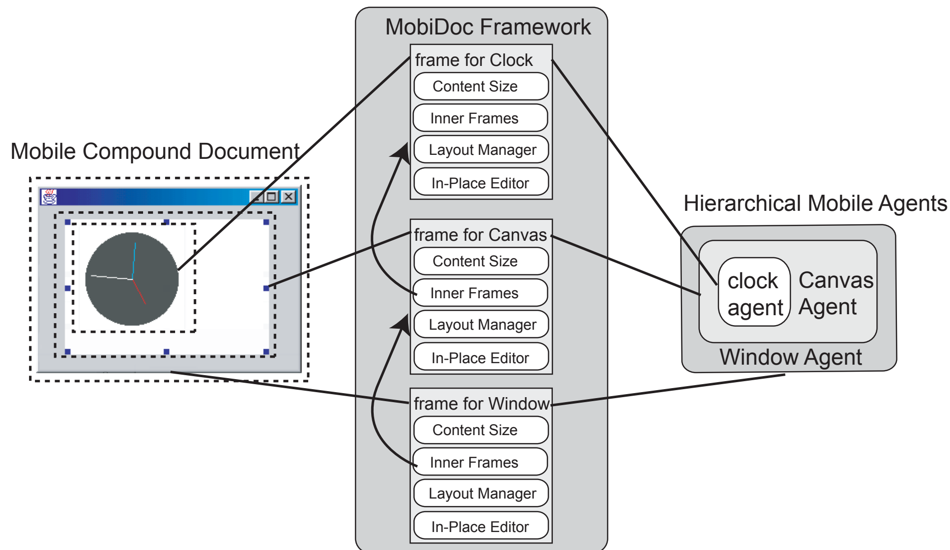

Figure 4: Components for compound document in agent hierarchy

activated but is not necessarily idle. To create a seamless application look, components embedded in a container component need to share in a coordinated manner several resources, such as keyboard, mouse, and window. Each component is restricted from directly accessing such shared resources. Instead, the frame object of one activated component is responsible for handling and dispatching user interface actions issued from most resources, and can own these resources until it sends a request to relinquish its resource.

The user can explicitly allocate and resize components in a document. Using their container as the mediator, these components cooperate to produce a seamless-looking document for the end user within their surroundings. Furthermore, this framework allows each container to explicitly define the rules of engagement that allow components to share the container's window, for example left-to-right flow arrangement like lines of text in a paragraph, and alignment in rectangular grid of cells.

In-Place Editing:

This framework supports document-wide operations, such as mouse clicks and keystrokes. It can dispatch certain events to its components to notify them when specified actions happen within their surroundings. Moreover, the framework provides each container component with a set of built-in services for switching among multiple components embedded in the container and for manipulating the borders of the frame objects of its inner components. One of these services provides graphical user interfaces for in-place editing. This mechanism allows different components in a document to share the same window and to automatically bring their editing tools to the document. Consequently, components can be immediately manipulated in-place, without the need to open a separate window and launch an application for each component.

To directly interact with a component, we need to make the component *active* by clicking the mouse within its frame. When a component is active, we can directly manipulate its contents. When the boundary of the frame is clicked, the frame becomes *selected* and displays eight angular control points for moving it around and resizing it, as shown in Figure 5. The user can easily resize and move selected components by dragging their handles.

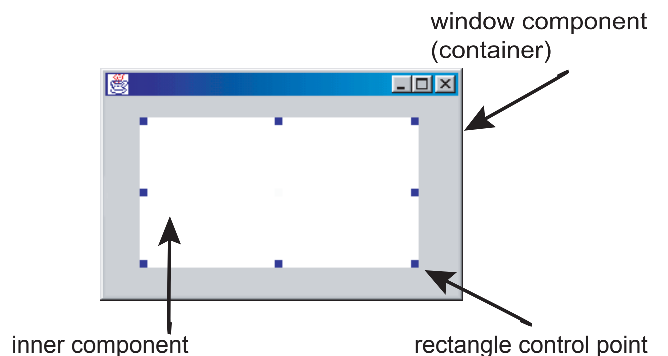

Figure 5: Selected component and its rectangle control points

Structured Storage and Migration:

When migrating over a network and being stored onto a disk, each component must be responsible for transforming its own contents and code into a stream of bytes by using the serialization facility of the runtime system. However, the frame object of each component is not stored in the component. Instead, it is dynamically created and allocated in its container's frame, when it becomes visible and restored. The framework automatically deletes frame objects of each component from the screen and stores specified attributes of the frame object in a list of values corresponding to the attributes, because other frame objects may refer to objects that are not serializable, such as several visible objects in the Java Foundation Class package. After restoring such serialized streams as components at the destination, the framework appropriately redraws the frames of the components, as accurately as possible.

Drag-and-drop Manipulation:

This framework lets the to user directly move or copy components between different containers within a computer by using drag-and-

drop manipulation. A user typically initiates a drag by first positioning the mouse pointer over some selected component and then pressing and holding down the mouse button. When the user releases the mouse button at the new location, the framework examines whether or not the destination container can accept the kind of selected component. If it can accept it, it moves the dragged component to the destination container just as in an agent migration. When copying a component, the framework makes a deep-copy of the component, in the sense that the component and all its inner components are duplicated without modifying its hierarchical structure. Also, the framework supports an intermediary buffer for copying and pasting components. The buffer corresponds to a clipboard and is implemented by a storage agent that can contain other components inside itself.

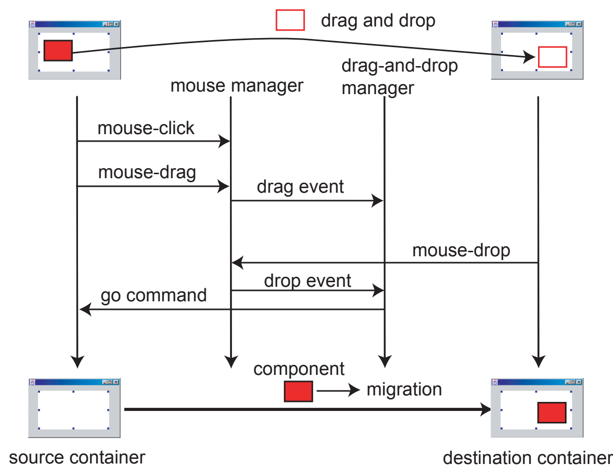

Figure 6: Scenario: drag-and-drop between two container components

Network-wide Component Assembly:
Nowadays, cut-and-paste is one of the most common manipulations for assembling visible components. However, while a cut-and-paste on the same computer is easy, the system often forces users to transfer information between computers in a very different way. Therefore, my framework offers a cut-and-paste mechanism among different computers. When a cut operation occurs at a component in one (source) container, the mechanism marshals the component and transmits the resulting byte sequence to another (destination) container at a local or remote computer by using the agent migration management of MobileSpaces. It becomes an infrastructure for providing a network-wide, direct manipulation technique, such as Pick-and-Drop, which is a kind of network-wide drag-and-drop manipulation, studied in [12].

3.4 Current Status

The MobiDoc framework has been implemented in MobileSpaces and Java language (JDK1.2 or later version), and we have developed various components for compound documents, including the examples presented in this paper. The MobileSpaces system is a general-purpose mobile agent system. Therefore, mobile agents in the system may be unwieldy as components of compound documents, but our components can inherit the powerful properties of mobile agents, including their activity and mobility.

Security becomes an essential issue in compound documents as well as mobile agents. The current system relies on the Java security manager and provides a simple mechanism for authentication of components. A container component can judge whether it accepts a new inner component or not beforehand, where the inner components can know the available methods embedded in their containers by using the class introspector mechanism of the Java language. Furthermore, since a container agent plays a role in providing resources for its inner agent, it can limit the accessibility of its inner components to resources such as window, mouse, and keyboard, by hiding events issued from these resources.

Even though our implementation was not built for performance, we have conducted a basic experiment on component migration with computers (Pentium III-800MHz with Windows2000 and SUN JDK 1.3). The cost of a component migration from a container to another container in the same hierarchy was measured to be 30 ms, including the cost to draw the visible content of the moving component and to check whether the component is permitted to enter the destination agent or not. The cost of component migration between two computers connected with Fast-Ethernet was measured to be 120 ms. The cost is the sum of the marshaling, compression, opening TCP connection, transmission, acknowledgment, decompression, security and consistency verifications, unmarshaling, layout of the visual space, and drawing of the contents. The moving component is a simple text viewer and its size (the sum of code and data) is about 4 Kbytes (zip-compressed). We believe that the latency of component migration in our framework is reasonable for a Java-based visual environment for building documents.

4. EXAMPLES

The MobiDoc compound document framework is powerful and flexible enough to support a wide range of different applications. This section shows some examples of compound documents based on the MobiDoc framework.

4.1 Electronic Mail System

One of the most illustrative examples of the MobiDoc framework is for the provision of mobile documents for communication and workflow management. We have constructed an electronic mail system based on the framework. The system consists of an inbox document and letter documents as shown in Figure 7. The inbox document provides a window that can contain two components. One of the components is a history of received mails and the other component offers a visual space for displaying the contents of mail selected from the history. The letter document corresponds to a mobile agent-based letter and can contain various components for accessing text, graphics, and animation. It also has a window for displaying its contents. It can migrate itself to its destination, but it is not a complete GUI application because it cannot display its contents without the collaboration of its container, i.e., the inbox document.

For example, to edit the text in a letter component, simply click on it, and then an editor program is invoked by the in-place editing mechanism of the MobiDoc framework. The component can deliver itself and its inner components to an inbox document at the receiver. After a moving letter has been accepted by the inbox document, if a user clicks on a letter in the list of received mail, the selected letter creates a frame object and requests the document to display the frame object within the frame of the document. The key idea of this mail system is that it composes different mobile agent-based components into a seamless-looking compound document and allows us to immediately display and access the contents

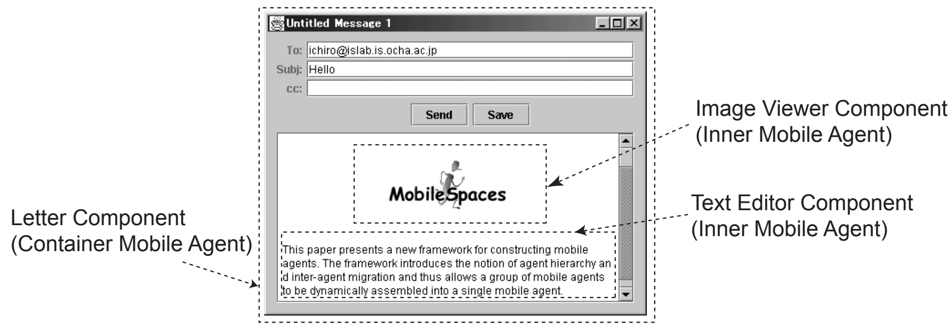

Figure 7: Structure of a letter document

of the components in-place. Since the inbox document is the root of the letter component, when the document is stored and moved, all the components embedded in the document are stored and moved with the document.

4.2 Desktop Teleporting

I constructed a compound document-based desktop system similar to the Teleporting System and the Virtual Network Computing system. Those systems are based on the X Window System and allow the running applications in the computer display to be redirected to a different computer display.

In contrast, my desktop system consists of mobile agent-based applications and thus can migrate not only the appearance of applications but also the applications themselves to another computer (Figure 8). The system consists of a window manager document and its inner applications. The manager corresponds to a desktop document at the top of the component hierarchy of applications separately displayed in their own windows on the desktop on the screen. It can be used to control the sizes, positions, and overlaps of the windows of its inner applications. When the desktop document is moved to another computer, all the components, including their windows, move to the new computer. The framework tries to keep the moving desktop and applications the same as when the user last accessed them on the previous computer, even if the previous computer and network are not running. For example, the framework can migrate a user's custom desktop and applications to another computer that the user is accessing.

4.3 Newsletter Editing System

This example is designed for editing an in-house newsletter. Each newsletter is edited by automatically compiling one or more text parts, which are written by different people, as shown in Figure 9. A newsletter is implemented as a compound document that can contain text component inside it and each text part is a mobile agent including a viewer/editor program and its own text data. When the newsletter is being edited, each text part moves from the document to the computer of its writer, and it displays a window for its editor program on the computer desktop to promote and help the user's writing. It goes back to the document after the writer finishes writing his/her text and then the document arranges the arriving components as a bound set. The document is still a mobile agent and thus can be easily duplicated and distributed to multiple locations.

5. RELATED WORK

There have been several document-centric component technologies so far. Among them OpenDoc and JavaBeans are characterized by

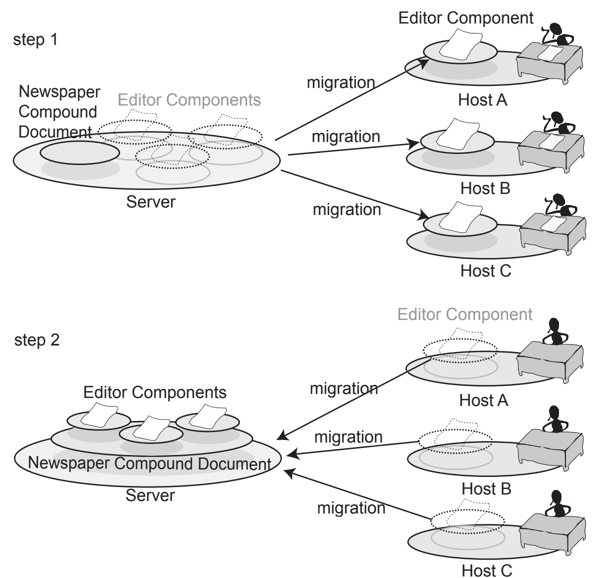

Figure 9: Newsletter editing system

allowing a component to contain a hierarchy of nested components. OpenDoc is a document-centric component framework originally developed by Apple Computer and IBM, but it was dropped as a commercial product. An OpenDoc component is a unit of visible components that can be nested. Unlike mine, an OpenDoc component cannot migrate itself over a network under its own control, although it is equipped with scripts to control itself. JavaBeans is a general framework for building reusable software components designed for the Java language. The initial release of JavaBeans (version 1.0 specified in [8]) does not contain a hierarchical or logical structure for JavaBean objects, but its latest release specified in [6] allows JavaBean objects to be organized hierarchically. However, the JavaBeans framework does not provide any higher-level document-related functions. Moreover, it is not inherently designed for mobility. Therefore, it is very difficult for a group of JavaBean objects in the containment hierarchy to migrate to another computer.

A number of other mobile agent systems have been released recently, for example Aglets [9], Mole [3], Telescript [20], and Voyager [10]. However, these agent systems unfortunately lack a mechanism for structurally assembling multiple mobile agents, unlike

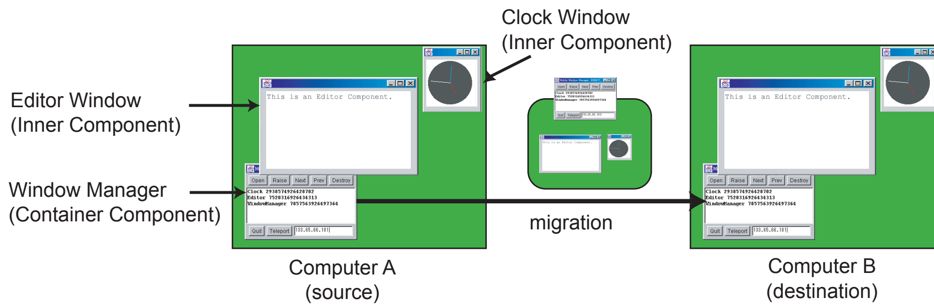

Figure 8: Desktop teleporting to another computer

component technologies. This is because each mobile agent is basically designed as an isolated entity that migrates independently. Some of them offer inter-agent communication, but they can only couple mobile agents loosely and thus cannot migrate a group of mobile agents to another computer as a whole. Telescript introduces the concept of places in addition to mobile agents. Places are agents that can contain mobile agents and places inside them, but they are not mobile. Therefore, the notion of places does not support mobile compound documents.

To solve the above problem in existing mobile agent systems, we constructed a new mobile agent system called MobileSpaces [13]. The system introduces the notion of agent hierarchy and inter-agent migration. This system allows a group of mobile agents to be dynamically assembled into a single mobile agent. Although the system itself has no mechanism for constructing compound documents, it can provide a powerful infrastructure for implementing compound documents to network computing settings. Also, I presented a compound document framework in another previous paper [14] but the previous framework was designed as just a simple example of MobileSpaces, instead of any general-purpose compound document framework. Therefore, the previous framework lacks many functionalities, which are provided by the framework presented in this paper. For example, it could deliver a compound document as a whole to another computer but not decompose a document into components nor migrate each component to another computer independently, unlike the framework presented in this paper.

ADK [7] is a framework for building mobile agents from JavaBeans. It provides an extension of Sun's visual builder tool for JavaBeans, called BeanBox, to support the visual construction of mobile agents. In contrast, I intend to construct a new framework for building mobile compound documents in which each component can be a container for components and can migrate over a network under its own control. My compound document will be able to migrate itself from one computer to another as a whole with all of its embedded components to another computer and adapt the arrangement of its inner components to the user's requirements and its environments by migrating and replacing corresponding components.

Here I should explain why a hierarchical mobile agent is essential in the development of compound documents. You might think that existing software development methodologies such as Java Beans and OpenDoc, are components that can be shipped to other computers. Indeed, in the current implementation of my system each mobile agent can be a container of Java Beans and can migrate as a whole with its inner Java Beans. However, Java Bean components are not inherently designed to be mobile components, unlike mobile agents. Therefore, it is difficult to get each Java Bean component to move over a network under its own control. On the other hand, my framework introduces a document (or a component) as an active entity that can travel from computer to computer under its own control. Therefore, my document can determine where it should go next, according to its contents. Moreover, it can dynamically change the layouts and combinations of its inner components, and it cannot be dynamically adapted to the user's requirements.

6. CONCLUSION

This paper has presented an approach for building mobile compound documents. The key idea of the approach is to build compound documents from hierarchical mobile agents in a hierarchical mobile agent system, named MobileSpaces, which allows multiple mobile agents to be dynamically assembled into a single mobile agent. This approach allows a compound document to be dynamically composed of mobile components, to deliver itself over a network as a whole with its inner components, and to adapt itself to the needs of its users and environments. I designed and implemented a Java-based framework, called MobiDoc, to demonstrate the usability and flexibility of this approach. The framework provides value-added services for coordinating mobile agent-based components embedded in a document with graphical user interfaces.

Finally, I would like to point out further issues to be resolved. In the current system, resource management and security mechanisms were incorporated relatively straightforwardly. These should now be designed for mobile compound documents. Additionally, the programming interface of the current system is not yet satisfactory. We should design a more elegant and flexible interface incorporating with existing compound document technologies. In the current implementation, each component must be written in Java, but it can run an interpreter of other programming languages inside it. I am interesting in implementing visual components equipped with such an interpreter for scripting languages, such as JavaScript, Tcl, and Lisp so that the behaviors of a component can be defined using these languages. To develop compound documents more effectively, we need a visual builder for mobile components.

Acknowledgments

I would like to thank the anonymous reviewers for their making giving a lot of significant comments about an earlier version of this paper.

7. REFERENCES

[1] Apple Computer Inc., "OpenDoc: White Paper", Apple Computer Inc., 1994.

[2] K. Arnold and J. Gosling, “The Java Programming Language”, Addison-Wesley, 1998.

[3] J. Baumann, F. Hole, K. Rothermel, and M. Strasser, “Mole - Concepts of A Mobile Agent System”, Mobility: Processes, Computers, and Agents, pp.536-554, Addison-Wesley, 1999.

[4] K. Brockschmidt, “Inside OLE 2”, Microsoft Press, 1995.

[5] L. Cardelli and A. D. Gordon, “Mobile Ambients”, Foundations of Software Science and Computational Structures, LNCS, Vol. 1378, pp. 140–155, 1998.

[6] L. Cable, “Extensible Runtime Containment and Server Protocol for JavaBeans”, Sun Microsfystems, http://java.sun.com/beans, 1997.

[7] T. Gschwind, M. Feridun, and S. Pleisch, “ADK: Building Mobile Agents for Network and System Management from Reusable Components”, in Proc. Symposium on Agent Systems and Applications / Symposium on Mobile Agents (ASA/MA’99), pp.13-21, IEEE Computer Society, 1999.

[8] G. Hamilton, “The JavaBeans Specification”, Sun Microsfystems, http://java.sun.com/beans, 1997.

[9] B. D. Lange and M. Oshima, “Programming and Deploying Java Mobile Agents with Aglets”, Addison-Wesley, 1998.

[10] ObjectSpace Inc, “ObjectSpace Voyager Technical Overview”, ObjectSpace, Inc. 1997.

[11] M. Potel and S. Cotter, “Inside Taligent Technology”, Addison-Wesley, 1995.

[12] J. Rekimoto, “Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments”, ACM Symposium on User Interface Software and Technology (UIST’97), pp.31-39, October 1997.

[13] I. Satoh, “MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System”, Proceedings of International Conference on Distributed Computing Systems (ICDCS’2000), pp.161-168, IEEE Computer Society, April, 2000.

[14] I. Satoh, “MobiDoc: A Framework for Building Mobile Compound Documents from Hierarchical Mobile Agents”, Proceedings of Symposium on Agent Systems and Applications / Symposium on Mobile Agents (ASA/MA’2000), Lecture Notes in Computer Science, Vol.1882, pp.113-125, Springer, 2000.

[15] I. Satoh, “Adaptive Protocols for Agent Migration”, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS’2001), pp.711-714, IEEE Computer Society, 2001.

[16] I. Satoh, “Network Processing of Mobile Agents, by Mobile Agents, for Mobile Agents”, Proceedings of 3rd International Workshop on Mobile Agents for Telecommunication Applications (MATA’2001), Lecture Notes in Computer Science (LNCS), Vol.2146, Springer, pp.81-92, August, 2001.

[17] I. Satoh, “Flying Emulator: Rapid Building and Testing of Networked Applications for Mobile Computers”, to appear in Proceedings of Conference on Mobile Agents (MA’2001), LNCS, Springer, 2001.

[18] Sun Microsystems, “The Bean Development Kit”, http://java.sun.com/beans/, July, 1998.

[19] C. Szyperski, “Component Software”, Addison-Wesley, 1998.

[20] J. E. White, “Telescript Technology: Mobile Agents”, General Magic, 1995.