MobiDoc: A Mobile Agent-based Framework for 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
E-mail: ichiro@is.ocha.ac.jp

Keywords: compound document, mobile agent, software component, distributed system

Edited by:

Received: **Revised:** **Accepted:**

This paper presents a mobile-agent-based framework for building mobile compound documents, called MobiDoc, where the compound document can be dynamically composed of mobile agent-based components and can migrate itself over a network as a whole, with all its embedded agents. The key idea of this framework is that it builds a hierarchical mobile agent system that enables multiple mobile agents to be combined into a single one. 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 its prototype implementation, currently using Java as the implementation language as well as a component development language, and then illustrate several interesting applications to demonstrate the utility and flexibility of this framework.

1 Introduction

Building systems from software components has already proven useful in the development of large and complex systems. Several frameworks for software components have been developed, such as COM/OLE [4], OpenDoc [1], CommonPoint [10], and JavaBeans [7]. Among them, the notion of compound documents is a document-centric component framework, where various visible parts, such as text, image, and video, created by different applications can be combined into one document and be independently manipulated in-place in the document. An example of this type of framework is CI Labs' OpenDoc [1] developed by Apple computer and IBM, although their development work on this framework has stopped. However, 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 the contents. Contents are stored inside the component but the codes for accessing them are not always. Thus, a user cannot view or modify a document whose contents need the support of different applications, if the user does not have the applications. Moreover, existing compound documents are inherently designed as passive entities in the sense that they can be transmitted over a network by external network systems such as electronic mail systems and workflow management systems and cannot determine where it should go next. We also need network-wide manipulation for building and assembling various components located in different computers into a document. Therefore, not only a whole compound document but also each of the components of the document must be able to be transmitted to another computer.

The goal of this paper is to propose a new framework for building mobile compound documents. Each document is built as a component that can be a container for components that can migrate over a network. Accessing compound

documents over a network requires a powerful infrastructure for building and migrating, such as mobile agents. Mobile agents are autonomous programs that can travel from computer to computer under their own control. When an agent migrates over a network, both the state and the codes can be transferred to the destination. However, traditional mobile agent systems cannot be composed of more than one mobile agent, unlike component technology. Therefore, we built a framework on a unique mobile agent system, called *MobileSpaces*, which was presented in an earlier paper [12]. 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 more than one mobile agent to be hierarchically assembled into a single mobile agent. Consequently, in our 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 visual real estate on a screen in a seamless-manner.

This paper is organized as follows. 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 shows the usability of our framework based on real-world examples. Section 6 makes some concluding remarks.

2 Background

Among the component technologies developed so far, OpenDoc and JavaBeans are characterized by allowing a component to contain a hierarchy of nested components. Although there are few hierarchical components available on the market today, their advent appears to be necessary and unavoidable in the long run.

OpenDoc is a document-centric component framework and has several advantages over other frameworks, but it has been discontinued. An OpenDoc component is not self-configurable, although it is equipped with scripts to control itself, so a component cannot migrate over a network under its own control. 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 [7]) did not contain a hierarchical or logical structure for JavaBean objects, but its latest release specified in [5] 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 [8], Mole [3], Telescript [17], and Voyager [9]. However, these agent systems unfortunately lack a mechanism for structurally assembling more than one mobile agent, unlike 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, in a previous paper [12]. 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 in network computing settings. Also, we presented a compound document framework as just an application of the MobileSpaces system [13]. Therefore, the previous framework lacked 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 or migrate each com-

ponent to another computer independently. As a result, the previous one could not fetch and assemble components located at different computers into a compound document.

ADK [6] 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, we 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. Our compound document will be able to migrate itself from one computer to another as a whole with all of its embedded components to the new computer and adapt the arrangement of its inner components to the user's requirements and its environments by migrating and replacing corresponding components.

We should explain why our hierarchical mobile agent is essential in the development of compound documents. The reader might think that existing software development methodologies such as Java Beans and OpenDoc, enable components to be shipped to other computers. Indeed, in the current implementation of our system each mobile agent can be a container of Java Beans and can get 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 migrate each Java Bean component over the network under its own control. On the other hand, our framework introduces a document (or a component) as an active entity that can travel from computer to computer under its own control. Therefore, our document can determine where it should go next, according to its contents. Moreover, it can dynamically adapt the layouts and combinations of its inner components to the user's requirements and the environments.

3 Approach

This section outlines the framework for building compound documents based on mobile agents called *MobiDoc*.

3.1 Mobile Agent-based Components

To create an enriched compound document, a component or document must be able to contain other components, like OpenDoc. On the other hand, each mobile agent resembles a software component in the sense that each entity is a self-contained module holding its code and state, but most existing mobile agent systems do not allow a mobile agent to be composed structurally. Furthermore, each mobile agent is characterized by its mobility. Thus, a composition of mobile agents must be designed to keep their mobility. We intend to provide such a component through a hierarchical mobile agent. Our framework is therefore built on MobileSpaces [12] which can dynamically assemble more than one mobile agent into a single mobile agent. The system supports mobile agents that are computational and itinerant entities, like other mobile agent systems. It also incorporates the following concepts:

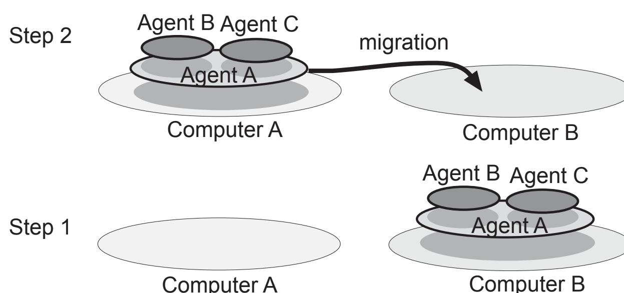

Figure 1: Agent Hierarchy and Group Migration.

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

- **Group Migration** The second concept means that each mobile agent can migrate to another agent or another computer as a whole, with all of its inner agents. It allows a group of mobile agents to be treated as a single mobile agent during their 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. The second concept enables

a compound document to migrate itself and its components as a whole. Accordingly, a compound document is given as a collection of mobile components and can be treated as a mobile component. Figure 1 shows an example of an inter-agent migration in an agent hierarchy. In an agent hierarchy, each agent is still mobile and can freely move into any computer or any agent in the same agent hierarchy except into itself or its inner agents, as long as the destination accepts the moving agent.

3.2 Compound Document Framework

MobileSpaces is a suitable infrastructure for mobile compound documents, but it does not provide any document-centric mechanisms for managing components in a compound document. We offer a compound document framework for supporting mobile agent-based components, including graphical user interfaces for manipulating visible components. This framework, called *MobiDoc*, is given 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 also deals with in-place editing services similar to those provided by OpenDoc and OLE. The framework offers several mechanisms for effectively sharing the visual estate of a container among embedded components and for coordinating their use of shared resources, such as keyboard, mouse, and window.

4 Implementation

Next, we will describe our method for using MobileSpaces to construct mobile compound documents.[1] 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.

4.1 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 given as Java objects [2]. Each component is given as a mobile agent in the system and the

[1] Details of the MobileSpaces mobile agent system can be found in our previous paper [12].

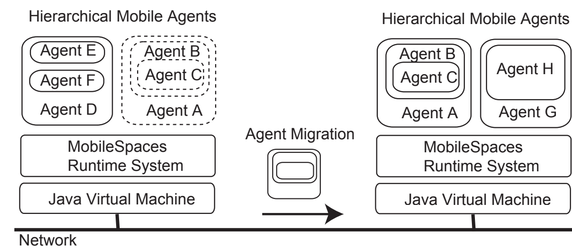

Figure 2: Agent Migration between Two MobileSpaces Runtime Systems.

containment hierarchy of components in a document is given 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 internal agents. That is, a container agent can instruct its embedded agents to move to other agents or computers, serialize them and destroy them. In contrast, an embedded agent has no direct control over its container agent. It can only access the collection of service methods offered by its container agents.

Agent Life-cycle Management

The runtime system is at the root node of the agent hierarchy and can control all the agents in the agent hierarchy. Furthermore, it 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. The runtime system monitors changes in components and propagates certain events to the right components. For example, when a component is added to or removed from its container component, the

system dispatches certain events to the component and the container.

Agent Migration Mechanism

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 codes of the component, including the components embedded in it, into a bit-stream formed in Java's JAR file format that can 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. 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.

4.2 Mobile Agent Program

In our compound document framework, each component is a group of mobile agents in MobileSpaces. They consist 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 as shown in Figure 3.

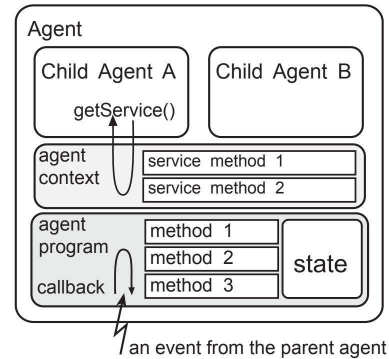

Figure 3: Structure of a Hierarchical Mobile Agent.

```
 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(
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(
29:     AgentURL url) {...}
30:   ....
32: }
```

The methods used to control mobility and life-cycle 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 by invoking the `getService()` method with an instance of the `Service` class. The instance can specify the kind of service methods, arbitrary objects as arguments, and deadline for timeout exception.
- When an agent performs the `go(AgentURL url)` method, it migrates itself to the destination agent specified by `url`. The `go(AgentURL url1, AgentURL url2)` method instructs the descendant 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 certain actions happen within their surroundings.

Our framework provides an event mechanism based on the delegation-based event model introduced in the Abstract Window Toolkit of JDK 1.1 or later, like Aglets [8]. 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 are acquired by the agent. Instead, the runtime system can issue certain events in the changes of life-cycle states. Also, a container agent can dispatch certain 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 mobile agent-based component can have one or more listener objects which implement certain methods invoked by the runtime system and its container component. For example, each component can have one or more activities that are performed using the Java thread library, but it needs to capture certain events issued before it migrates over a network and stop its own activities.

4.3 MobiDoc Compound Document Framework

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

Visual Layout Management

Each mobile agent-based component can be displayed within the estate of its container or a window on the screen, but it must be accessed through an indirection: *frame* objects derived from the `ComponentFrame` class.[2] as shown in Fig. 4. 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.

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

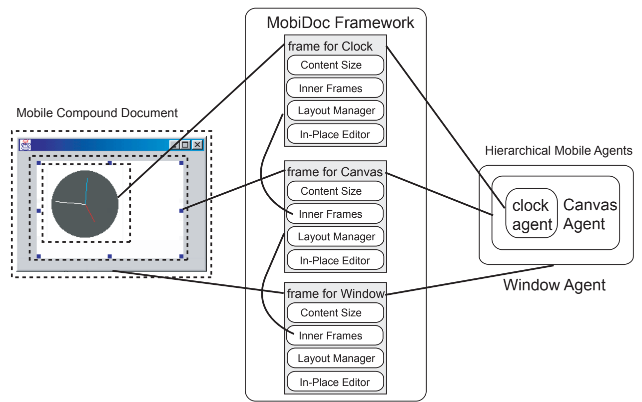

Figure 4: Components for Compound Document in Agent Hierarchy.

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 itself, while the frame object of each contained component is responsible for drawing its own contents. For example, if a component needs to change the size of its frame by calling the `setFrameSize()` method, its frame must negotiate with the frame object of its container for its size and shape and 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 specified as frame
16:   void addFrame(ComponentFrame frame);
17:   // removes an embedded component
18:   // specified 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 frame
27:   // specified as cf
28:   void setEmbeddedFramePosition(ComponentFrame cf,
29:                               java.awt.Rectangle);
30:   ....
31: }
```

When one component is activated, another component is usually deactivated but does not necessarily become 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 reserve these resources until it sends a request to relinquish them.

In-Place Editing

Our framework provides for document-wide operations, such as mouse click and keystrokes. It can dispatch certain events to its components to notify them when certain 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 offers graphical user interfaces for in-place editing. This mechanism allows different components in a document to share the same window. Consequently, components can be immediately manipulated in-place, without the need for opening a separate window 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 rectangle control points for moving it around and resizing it, as shown in Fig. 5. The user can easily resize and move the selected component by dragging its handles.

Structured Storage and Migration

While migrating over a network and being stored on a disk, each component must be responsible for transforming its own contents and codes 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.

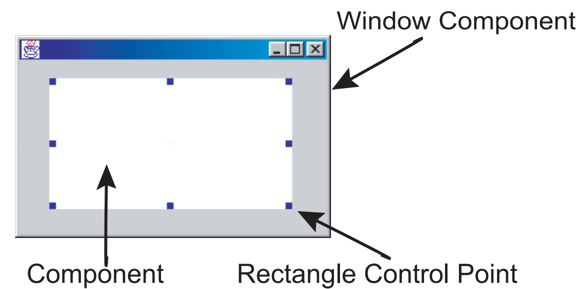

Figure 5: Selected Component and its Rectangle Control Points.

Instead, it is dynamically created and allocated in its container's frame, when it becomes visible and restored. The framework automatically removes 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.

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 a single computer is easy, the system often forces users to transfer information between computers in a very different way. Therefore, our framework offers a mechanism for cutting and pasting between 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 and direct manipulation technique, such as Pick-and-Drop that is a kind of network-wide drag-and-drop manipulations studied in [11].

4.4 Current Status

The MobiDoc framework has been implemented in the MobileSpaces system using the Java language (JDK1.2 or later version), and we have developed various components for compound documents, including the examples presented in this paper. The MobiDoc framework and the MobileSpaces System are constructed independently of the underlying system and can run on any computer with a JDK 1.2-compatible Java runtime system.

MobileSpaces 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 is essential 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 to accept a new inner component or not beforehand, while 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 1.2). The time 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. The cost of component migration between two computers connected by Fast-Ethernet was measured to be 120 ms. The cost is the sum of the marshaling, compression, opening a 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.

5 Examples

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

5.1 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 Fig. 6. 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, one simply clicks on it, and 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 a letter in the list of received mail, the selected letter creates a frame object of itself 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 combines different mobile agent-based components into a seamless-looking compound document and allows us to immediately display and access the contents of the components in-place. Since the inbox doc-

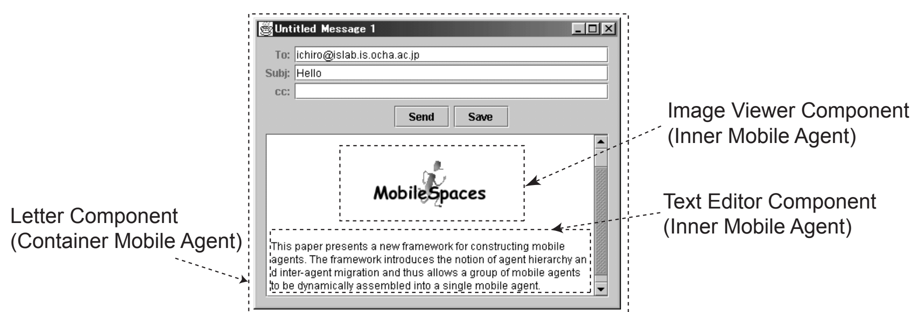

Figure 6: Structure of a Letter Document.

ument 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.

5.2 Desktop Teleporting

We constructed a mobile agent-based desktop system similar to the Teleporting System and the Virtual Network Computing system. These 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, our 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 (Fig. 7). 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 when the previous computer and network have stopped. For example, the framework can migrate a user's custom desktop and applications to another computer that the user is accessing.

6 Conclusion

We have presented an approach for building compound documents. The key idea of the approach is to build compound documents from hierarchical mobile agents in the MobileSpaces system, which allows more than one mobile agent to be dynamically assembled into a single mobile agent. Our approach allows a compound document to be dynamically composed of mobile components and to be migrated over a network as a whole with its inner components under its own control. We designed and built a 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.

Finally, we would like to point out further issues to be resolved. To develop compound documents more effectively, we need a visual builder for our mobile components. We plan to extend a visual builder tool for JavaBeans, such as the BeanBox system included in the Bean Development Kit (BDK) [15], so that can support mobile agent-based compound documents. In the current system, resource management and security mechanisms are 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 plan to design a more elegant and flexible interface incorporating existing compound document technologies.

References

[1] Apple Computer Inc. (1994) *OpenDoc: White Paper*, Apple Computer Inc.

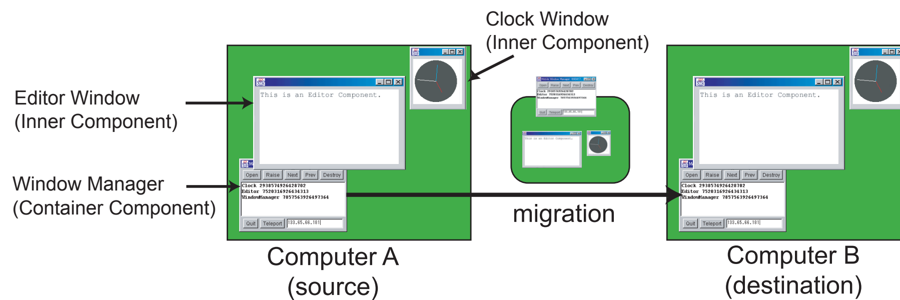

Figure 7: Desktop Teleporting to another Computer.

[2] Arnold, K. & Gosling, J. (1998) *The Java Programming Language*, Addison-Wesley.

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

[4] Brockschmidt, K. (1995) *Inside OLE 2*, Microsoft Press.

[5] Cable, L. (1997) *Extensible Runtime Containment and Server Protocol for JavaBeans*, Sun Microsfystems, http://java.sun.com/beans.

[6] Gschwind, T., Feridun, M., & Pleisch, S. (1999) *ADK: Building Mobile Agents for Network and System Management from Resuable Components*, Technical University of Vienna, TUV-1841-99-10.

[7] Hamilton G. (1997) *The JavaBeans Specification*, Sun Microsfystems, http://java.sun.com/beans.

[8] Lange, B. D., & Oshima, M. (1998) *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley.

[9] ObjectSpace Inc. (1997) *ObjectSpace Voyager Technical Overview*, ObjectSpace Inc.

[10] Potel, M., & Cotter, S. (1995) *Inside Taligent Technology*, Addison-Wesley.

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

[12] Satoh, I. (2000) 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.

[13] Satoh, I. (2000) 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.

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

[15] Sun Microsystems (1998) *The Bean Development Kit*, http://java.sun.com/beans, Sun Microsystems.

[16] Szyperski, C. (1998) *Component Software*, Addison-Wesley.

[17] White, J. E. (1995) *Telescript Technology: Mobile Agents*, General Magic.