MobiDoc: A Framework for Building Mobile Compound Documents from Hierarchical Mobile Agents

Ichiro Satoh

Department of Information Sciences, Ochanomizu University /
Japan Science and Technology Corporation
2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610 Japan
Tel: +81-3-5978-5388 Fax: +81-3-5978-5390
E-mail: ichiro@is.ocha.ac.jp

Abstract. MobiDoc is a framework for building mobile compound documents, where the compound document can 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 added-value mechanisms for visually manipulating components embedded in a compound document and for sharing a window on the screen among the components. This paper will describe the MobiDoc framework and its first implementation, currently using Java as implementation language as well as 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 [12], and JavaBeans [8]. Among them, the notion of compound documents is a document-centric component framework, where various visible parts, such as text, image, and video, that are 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 frameworks 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 often stored in the component but not the codes for accessing them. 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 themselves. Moreover, most existing frameworks assume that a user manually lays out components into a compound document. It is difficult to change a compound document autonomously. So, when a compound document arrives at a computer, the document is unable to dynamically change the layouts and combinations of its components, and it cannot be dynamically adapted to the user's requirements. A document is not designed for mobility and thus the document itself cannot determine where it should go next.

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 is able to 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 each agent migrates over 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 [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 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 these components can effectively share visual real estate on a screen in a seemless-looking way.

This paper is organized in the following sections. Section 2 surveys related work and Section 3 presents the basic ideas of the compound document framework, called *MobiDoc*. Section 4 details its first implementation and Section 5 shows the usability of our framework based on real-world examples. Section 6 gives 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 in the market today, their advent appears to be necessary and unavoidable in the long run.

OpenDoc is a document-centric components 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, and thus 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 [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 [16], and Voyager [11]. 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 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.

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, 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. The HyperNews framework [10] provides an electronic newspaper system to the WWW by using mobile agents to encapsulate and update articles. It does not offer any any general framework for building mobile compound documents, but can provide an architecture for electronic documents based on mobile agents.

3 Approach

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

3.1 Compound Documents as Mobile Agents

To create an enriched compound document, a component or document must be able to contain other components, like OpenDoc. We intend to provide such a component through a hierarchical mobile agent. Our framework is therefore built on the MobileSpaces system presented in our earlier paper [13] 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. Also, the MobileSpaces system incorporates the following concepts:

- **Agent Hierarchy:** Each mobile agent can be contained within one mobile agent.

– **Inter-agent Migration:** Each mobile agent can migrate between other mobile agents as a whole, with all of its inner agents.

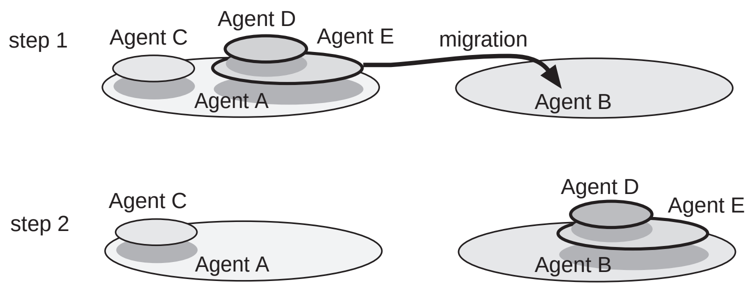


Fig. 1. Agent Hierarchy and Inter-agent Migration

The first concept enables each component to be a group of mobile agents organized hierarchically. The second concept enables a compound document to migrate itself and its components as a whole. Fig. shows an example of an inter-agent migration in an agent hierarchy. Our agent model is similar to a process calculus for modeling process migration called mobile ambients [5]. The containment hierarchy for components in a document is organized directly in the agent hierarchy in the MobileSpace system.

3.2 Compound Document Framework

The MobileSpaces system 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 40 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 a visual estate of a container among components embedded and for coordinating their use of shared resources, such as keyboard, mouse, and window.

4 Implementation

Next, we will describe our method for using the MobileSpaces system to construct mobile compound documents.[1] The system can execute and migrate mobile agents that

[1] Details of the MobileSpaces mobile agent system can be found in our previous paper [13].

are incorporated with the two concepts presented in the previous section. It has been incorporated in Java Development Kit version 1.2 and can run on any computer that has a runtime compatible with this version.

4.1 The 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 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 inner agent. That is, a container agent can instruct its embedded agents to move to other agents or computers, serialize and destroy them. In contrast, each agent has no direct control over its container agent. Instead, each container can offer a collection of service methods which can be accessed by its embedded agents.

Agent Execution 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 the changes of 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 specified events to the component and the container.

Agent Migration: 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.

Extensibility: The MobileSpaces system is characterized by offering its own facilities through mobile agents, so that these subcomponents can be dynamically added to and removed from the system by migrating and replacing the corresponding agents. Therefore, the system itself can dynamically extend and adapt its new functions, such as inter-agent communication, agent persistency, and agent migration between computers to its execution environments. For example, the system can migrate agents through unreliable, unsecured, and temporally disconnected networks, that may not have been initially supported.

4.2 Agent Model

In our compound document framework, each component is a group of mobile agents in the MobileSpaces system. 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 the life-cycle of a mobile agent-based component.

```
public class ComponentAgent extends Agent {
  // (un)registering services for inner agents
  void addContextService(ContextService service){ ... }
  void removeContextService(ContextService service){ ... }
  ....
  // (un)registering listener objects to hook events
  void addListener(AgentEventListener listener) { ... }
  void removeListener(AgentEventListener listener) { ... }
  ....
  void getService(Service service) throws ... { ... }
  void go(AgentURL url) throws ... { ... }
  void go(AgentURL url1, AgentURL url2) throws ... { ... }
  byte[] create(byte[] data) throws ... { ... }
  byte[] serialize(AgentURL url) throws ... { ... }
  AgentURL deserialize(byte[] data) throws ... { ... }
  void destroy(AgentURL url) throws ... { ... }
  ....
  ComponentFrame getFrame() { ... }
  ComponentFrame getFrame(AgentURL url) { ... }
  ....
}
```

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 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 time for timeout exception.
- 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 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 specified actions happened within their surroundings by using the `dispatchEvent()` method.

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 [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 are 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 mobile agent-based component can have one or more listener objects which implement specific methods invoked by the runtime system and its container component. For example, each component can have one or more activities which are performed by using the Java thread library, but needs to capture certain events issued before it migrates over a network and stop its own activities.

4.3 The MobiDoc Compound Document Framework

The *MobiDoc* framework is implemented as a collection of Java classes to enforce 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. 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 its container component and the frame of its component.

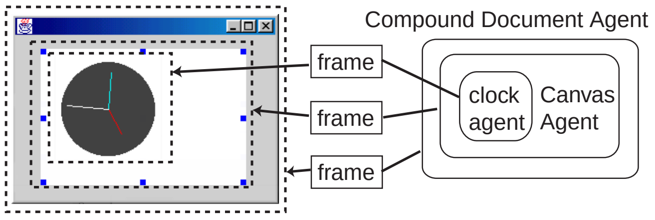


Fig. 2. Components for Compound Document in Agent Hierarchy

[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*

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 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.

```
public class ComponentFrame extends java.awt.Panel {
  // sets the size of the frame
  void setFrameSize(java.awt.Point p);
  // gets the size of the frame
  java.awt.Point getFrameSize();
  // sets the layout manager for the embedded frames
  void setLayout(CompoundLayoutManager mgr) {
  // views the type of the component, e.g. iconic, thumbnail, or framed,
  int getViewType();
  // gets the reference of the container's frame
  ComponentFrame getContainerFrame();
  // adds an embedded component specified as frame
  void addFrame(ComponentFrame frame);
  // removes an embedded component specified as frame
  void removeFrame(ComponentFrame frame);
  // gets all the references of embedded frames
  ComponentFrame[] getEmbeddedFrames();
  // gets the offset and size of the inner frame specified as cf
  java.awt.Rectangle getEmbeddedFramePosition(ComponentFrame cf);
  // sets the offset and size of the inner frame specified as cf
  void setEmbeddedFramePosition(ComponentFrame cf, java.awt.Rectangle);
  ....
}
```

When one component is activated, another component is usually deactivated but is not necessarily idle. To create a seamless application look, components embedded in a container component need to coordinately share 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.

In-Place Editing: The MobiDoc framework provides for document wide operations, such as mouse click 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 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 clicking the boundary of the frame, the frame becomes *selected* and then has eight rectangle control points for moving it around and resizing it, as shown in Fig. 3. The user can easily resize and move the selected components by dragging their handles.

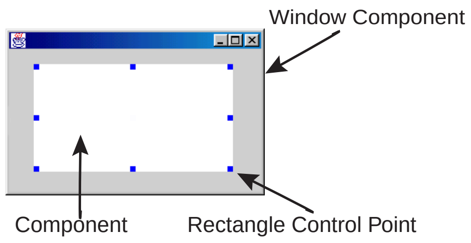


Fig. 3. Selected Component and Its Rectangle Control Points

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

4.4 The Current Status

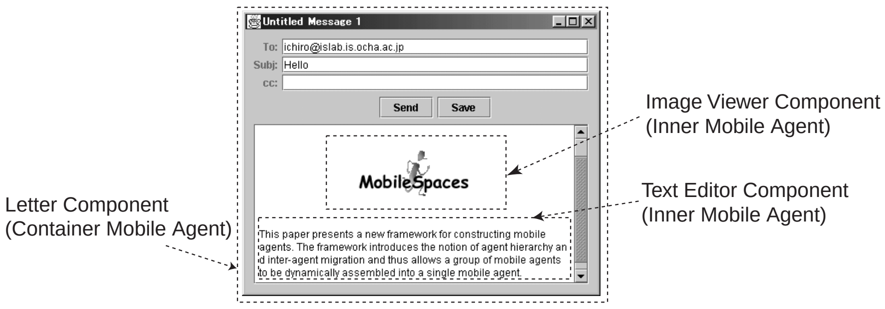
The MobiDoc framework has been implemented in the MobileSpaces system using the Java language (JDK1.2 or later version), and we have developed various components for compound documents, including the examples presented in this paper. The MobileSpaces system is a general-purpose mobile agent system. Therefore, 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 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 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.

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. 4. 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.


To: ichiro@islab.is.ocha.ac.jp
Subj: Hello
cc:

This paper presents a new framework for constructing mobile agents. The framework introduces the notion of agent hierarchy an d inter-agent migration and thus allows a group of mobile agents to be dynamically assembled into a single mobile agent.

Fig. 4. Structure of a Letter Document

For example, to edit the text in a letter component, simply click on it, and 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 is 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 it 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 seemless-looking compound document and allows us to immediately display and access the contents 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.

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 surface of applications but also the applications themselves to another computer (Fig. 5). 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 are stopped. For example, the framework can migrate a user's custom desktop and applications to another computer the user is accessing.

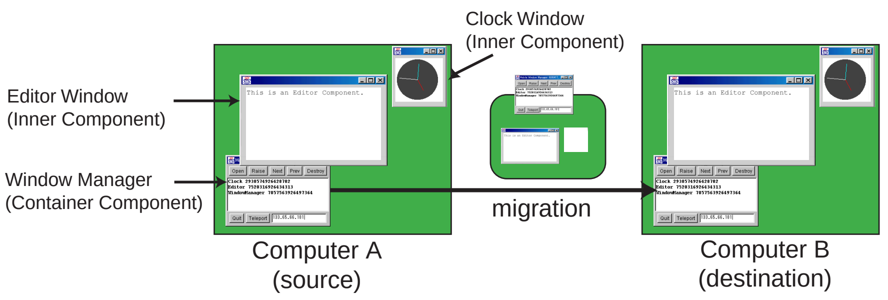


Fig. 5. A Desktop Teleporting to Another Computer

6 Conclusion

We have presented a new approach for building mobile 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. We design 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. We believe that the framework can provide a realistic and useful application of mobile agents.

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) [14], so that it has the ability to support mobile agent-based compound documents. In the current system, resource management and security mechanisms were incorporated relatively straightforwardly. These now should 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 with existing compound document technologies. The MobileSpaces system is a general-purpose mobile agent system and thus can easily be used to build the framework. However, it may be unwieldy as an infrastructure for compound documents, and thus we are interested in investigating a lightweight system, which is optimized to handle mobile compound documents.

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 Resuable Components, Technical University of Vienna, TUV-1841-99-10, 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. J. Morin, HyperNews, a Hypermedia Electronic-Newspaper Environment based on Agents, Proceedings of HICSS-31, pp.58-67, 1998.
11. ObjectSpace Inc, ObjectSpace Voyager Technical Overview, ObjectSpace, Inc. 1997.
12. M. Potel and S. Cotter, Inside Taligent Technology, Addison-Wesley, 1995.
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 Press, April, 2000.
14. Sun Microsystems, The Bean Development Kit, http://java.sun.com/beans/, July, 1998.
15. C. Szyperski, Component Software, Addison-Wesley, 1998.
16. J. E. White, Telescript Technology: Mobile Agents, General Magic, 1995.