A Document-centric Component Framework for Document Distributions

Ichiro Satoh*

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract. This paper presents a framework for building and managing compound documents in distributed systems. It enables an enriched document to be dynamically and nestedly composed of software components corresponding to various types of content, e.g., text, images, and windows. It permits the content of each component and program code to access the content inseparable inside the components so that the components can be viewed or modified without the need for any applications. It enables each component or document to migrate over a network under its own control by using mobile agent technology. Moreover, it introduces components as carriers or forwarders because it enables them to carry or transmit other components as first class objects to other locations. It offers several basic operations for network processing, e.g., forwarding, duplication, and synchronization. Since these operations are still document components, they can be dynamically deployed and customized at local or remote computers through GUI manipulations. It therefore allows an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. This paper describes the framework and its implementation, which currently uses Java as the implementation language as well as a component development language, and then illustrates several interesting applications that demonstrate its utility and flexibility.

1 Introduction

Document manipulation, such as editing, viewing, and distributing documents, is still playing a crucial role in modern information processing. Documents in distributed computing systems are always transmitted passively over a network by external systems, such as electronic mail systems and http servers. As a result, they cannot determine where, when, or how they should go next. However, there have been several applications whose network processing depends on the content of the documents that are transmitted over the network. For example, a workflow management system is required to distribute documents among employees according to the content of the documents and the company's decision-making path. As a result, end-users often want to define the network processing of documents for them to be able to accomplish their application-specific tasks. However, the customization and management of networking processing is too complex and difficult for end-users to achieve.

* e-mail: ichiro@nii.ac.jp

This paper proposes a compound document framework, called MobiDoc, as a solution to these problems. Like other existing compound document frameworks, it enables an enriched document to be composed of visual components, e.g., text and images. The framework has several unique features, which other frameworks do not have. For example, it introduces the notion of self-contained components in the sense that not only the content of each component but also its codes to view and edit the content are embedded in the component to solve various problems, including content rights management, in existing content-distribution. It also enables network protocols for documents to be implemented by a set of active documents. By using mobile agent technology, documents or components can define their own itineraries and migrate under their own control. Documents or components can transmit other documents or components as first-class objects to their destinations. The framework introduces the components for network processing as document-centric components, so that it allows an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. It provides components with a mechanism for sharing visual content on different computers.

This paper is organized as follows. We first describes the background and related work (Section 2) and outline our compound document framework (Section 3). We then present component runtime systems for executing and migrating document components (Section 4) and present our component model (Section 5). We describe its prototype implementation (Section 6) and illustrate several applications of the framework (Section 7). We conclude by providing a summary and discussing future issues (Section 8).

2 Background

Building systems from software components has already proven useful in the development of large-scale software [24]. Many frameworks for software components have been developed, e.g., COM [16] and JavaBeans [9]. These existing frameworks aim at defining the behaviors of distributed computing, i.e., server-side and client side processing, by combining software components. Therefore, these frameworks are suitable for professional developers but not end-users.

Several frameworks for compound document components have been developed, such as COM/OLE [3], OpenDoc [1], and CommonPoint [15]. They enable one document to be composed of various visible parts, such as text, image, and video, created by different applications. These frameworks assume that content is stored inside the component but it is accessed by external applications. Thus, when users receive a compound document, they cannot view or modify it, if its content needs the support of different applications, if they do not have all the applications. There have been XML-based technologies for office documents, e.g., OpenDocument formats, but these have inherited the problems of compound documents. This is because, when a computer receives XML-based documents, it may not have the viewer/editor programs for them. This becomes a serious limitation in distributing new data formats. Of these, the Bonobo framework for software components and compound documents is being developed by the GNOME project [7]. It provides several mechanisms for creating compound documents from a collection of components, but it is based on an underlying middleware

and GUI widget, i.e., CORBA and GTK+, and does not support the distribution of components, including compound documents over a network.

The framework presented in this paper, therefore, has been designed independently of existing component frameworks for distributed computing or compound documents. This is because it permits document-centric components to migrate themselves over a network and process other components as first-class objects [5], e.g., migrating or saving them to other computers or on secondary disks. These features enable our components to have direct access to novel and powerful features that existing components do not have. End-users can also easily customize their network processing of documents through user-friendly manipulations to edit visual components, and they can control their own network processing according to their content. Nevertheless, it is open to existing component frameworks. In fact, it can use typical Java-based components, e.g., Java Beans and Applets, as its components.

Several (non-component-based) attempts have been made to support active documents, e.g., Active Mail [8] and HyperNews [14], but these have aimed at particular application-specific documents, such as electronic mail and newspapers, so that they have not supported varied and complex content. The fuseONE system [26] is composed of GUI-based control panels to control appliances from active documents, i.e., GUI-based buttons and toggle switches. Like other compound document frameworks, these cannot transmit codes for viewing and editing documents. To customize distributed computing, particularly network processing between computers, several researchers have explored active networking technologies [25]. However, these existing technologies have focused on configurations for low-level network processing, e.g., routing and QoS protocols, so that they are not suitable for end-users.

As will be discussed later, this framework uses mobile agent technology [12, 6]. However, the technology assumes each mobile agent to be an isolated entity that migrates between computers independently of other agents and it does not support any document-centric approaches. We constructed a mobile agent system called MobileSpaces [17] and designed a compound document framework [18, 21] based on the system. The previous framework could not be used to define or customize any network processing, because it was only proposed as an application of the MobileSpaces system. Furthermore, there were several serious mismatches between mobile agent-based components and the requirements of document components.

3 Design Principles

This section briefly outlines the framework presented here.

3.1 Requirements

It needs to satisfy the following requirements.

Composition: Like OpenDoc and JavaBeans, our framework needs to be composed of a document or component of nested components that can display visual parts, e.g., text, images, and windows, and that enables us to edit components in-place without opening a separate window for each component.

Application-absence: Components should be distributed and operated without the need for any applications in their current computers. That is, when a computer receives a component, it must be able to view or edit it, including its inner components, even when the computer lacks applications.

Autonomy and Mobility: Each document or component is an autonomous programmable entity that can determine which components or computers it will go to according to its program code and content, and that can then migrate to that destination.

Reconfiguration: The network processing of documents, components, or plain data can be easily and naturally defined and customized as a combination of basic components by end-users through document-manipulations just like editing documents.

3.2 Component model

Let us present the basic ideas behind the framework.

Self-contained component: This framework introduces the notion of a *self-contained* component, where the content of each component and its codes are inseparable. When a component is distributed to other computers, the framework not only transmits the content of each component but also its code to the destinations. To our knowledge, no existing software component frameworks, including compound document frameworks, make the code and state of each component indivisible. This makes documents portable. This is because, when a user receives a document, he/she can view or edit it by using its code instead of any applications deployed at its current computer.

Hierarchical composition: Like OpenDoc and JavaBeans, this framework allows a hierarchy of nested components to correspond to visual parts, e.g., text, images, and windows, and conform to two notions: i) Each component can be contained by at most one component, and ii) it can dynamically migrate to other components along with all its inner components. When a component is contained by another component, the former is still an individual component so that it can be removed from the latter. Each component can move into any other component except itself or its descendant components, which may be at different computers, as long as the destination component accepts it.

Service Provider: Each container component is responsible for automatically offering its own services and resources to its inner components and controlling its inner components. When a component requires a service, it migrates itself to one of the container components that can provide the service. The framework also allows container components to process their inner components as first-class objects. As a result, each container component can migrate, save, and destroy its inner components.

Component interaction: The framework enables a component to control the size and layout of its inner components, and to invoke the service methods explicitly provided by its container (and its neighboring components through its container). In other words, a component cannot access any services supported by components other than its container component. This is important in allowing successful migration to occur. If it were

not imposed, then migrating a component could mean that the descendants of that component might suddenly find that they could no longer access services on which they had previously relied.

3.3 Network processing

This framework provides two approaches to enabling components to customize their own network processing. The first is to make components *mobile* in the sense that they can define their itinerary and travel among multiple computers along this itinerary using mobile agent technology. The second enables components to define network processing for themselves or their inner components. The framework provides three types of components as follows:

- **Visual component** stores its visual content within itself. It displays this content in the estate assigned by its container component by using its own program. When a visual component contains other components, it is responsible for managing the estates of its inner component within its own estate.
- **Carrier component** is transparent and can contain more than one component, which may contain one more component. It can carry its inner components along its own itinerary. Moreover, since it can can treat its inner components as first-class objects, it can explicitly restrict or transform its inner components.
- **Forwarder component** is allocated at a component or computer and can automatically transmit its inner components to its destinations. It can also process its visiting components as first-class objects before it forwards them.

Note that visual components can still travel between other components or computers under their control and network processing components can be assembled and operated through GUI manipulations and embedded into a document as visual components.

4 Component and Runtime System

This framework consists of two parts: runtime systems and components. The former can execute components and migrate them to/from other runtime systems, even when underlying systems, i.e., operating systems and hardware, are heterogeneous, since runtime systems and the latter are constructed with Java language.

4.1 Component

As we can see from Fig. 1, each component is a collection of Java objects wrapped in a component and has its own unique identifier and image data displayed as its icon. All the objects that each component consists of need to implement the `java.io.Serializable` interface, because they must be marshaled using Java's serialization mechanism. Each visual component needs to be defined as a subclass of either the `java.awt.Component` or `java.awt.Container` from which most of Java's visual or GUI objects are derived. To reuse existing software, we implemented an adapter to use typical Java

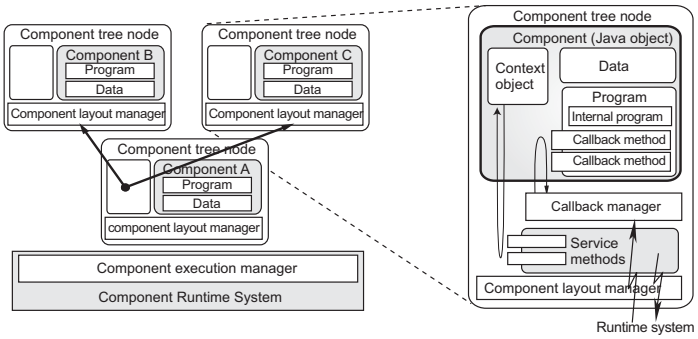

Fig. 1. Component hierarchy and structure of components.

components, e.g., Java Applets and JavaBeans, that are defined as subclasses of the `java.awt.Component` or `java.awt.Container` class within our components [1] We describe a programming model for components in the Appendix.

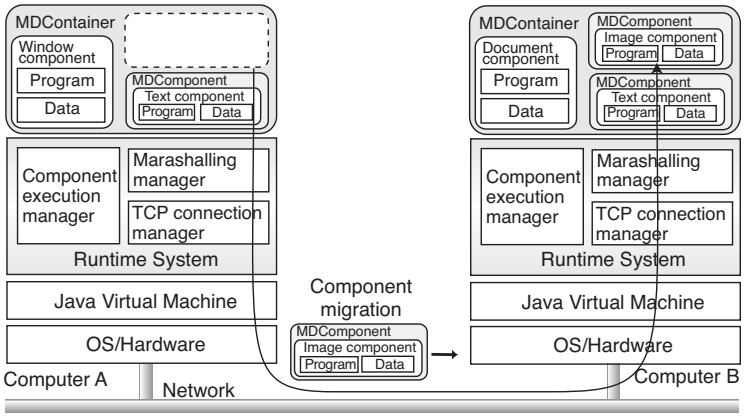

Fig. 2. Component migration between two computers.

4.2 Component runtime system

Fig. 2 outlines the basic structure of a runtime system. Each runtime system governs all the components within it and provides them with APIs for components in addition to Java's classes. It assigns one or more threads to each component and interrupts them before the component migrates, terminates, or is saved. Each component can request its current runtime system to terminate, save, and migrate itself and its inner components to the destination that it wants to migrate to. This framework provides each component with a wrapper, called a *component tree node*. Each node contains its target component, its attributes, and its containment relationship and provides interfaces between its component and the runtime system (Fig. 1). When a component is created in a runtime

[1] This is not compatible with all kinds of Applets and JavaBeans, because some of those existing manage their threads and input and output devices depreciatively.

system, it creates a component tree node for the newly created component. When a component migrates to another location or duplicates itself, the runtime system migrates its node with the component and makes a replica of the whole node.

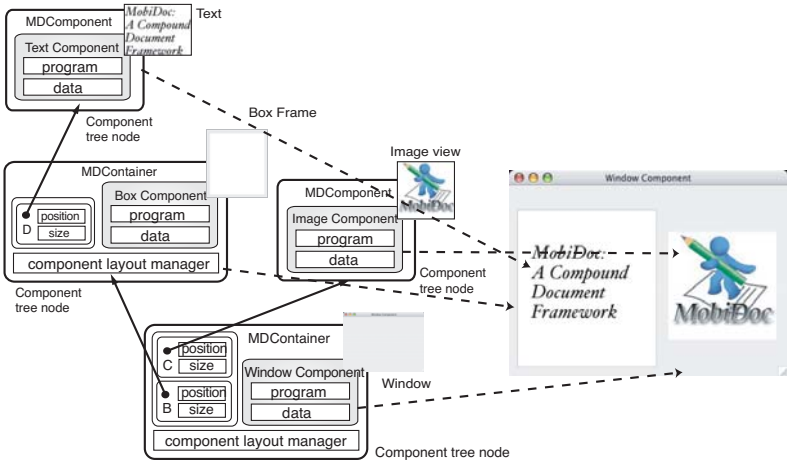

Fig. 3. Component Hierarchy

A hierarchy is maintained in the form of a tree structure of component tree nodes of the components (Fig. 3). Each node is defined as a subclass of `MDContainer` or `MDComponent`, where the first supports components, which can contain more than one component inside them and the second supports components, which cannot contain any components. For example, when a component has two other components inside it, the nodes that contains these two inner components are attached to the node that wraps the container component. Component migration in a tree is only performed as a transformation of the subtree structure of the hierarchy. When a component is moved over a network, on the other hand, the runtime system marshals the node of the component, including the nodes of its children, into a bit-stream and transmits the component and its children, and the marshaled component to the destination.

4.3 Component manipulation

Each component can display its content within the rectangular estate maintained by its container component. The node of the component, which is defined as a subclass of the `MDContainer` or `MDComponent` class, specifies attributes, e.g., its minimum size and preferable size, and the maximum size of the visible estate of its component in the estate is controlled by the node of its container component. These classes can define their new layout manager as subclasses of the `java.awt.LayoutManager` class.

This framework provides an editing environment for manipulating the components for network processing, as well as for visual components. It also provides in-place editing services similar to those provided by OpenDoc and OLE. It offers several value-added mechanisms for effectively sharing the visual estate of a container among embedded components and for coordinating their use of shared resources, such as keyboards, mice, and windows. Each component tree node can dispatch certain events to

its components to notify them when certain actions happen within their surroundings. `MDContainer` and `MDComponent` classes support built-in GUIs for manipulating components. For example, when we want to place a component on another component, including a document, we move the former component to the latter through GUI manipulations, e.g., drag-and-drop or cut-and-paste. When the boundary of the visible area of a component is clicked, the component is *selected* and displays eight rectangular control points for moving it around and resizing it (Fig. 4 (a)). The user can resize the selected component, move it to another, save it, and terminate it by dragging its handles (Fig. 4 (b)).

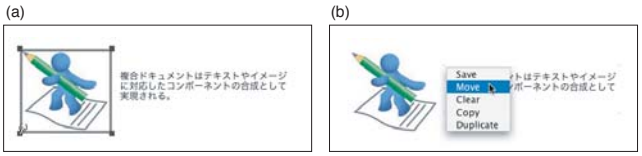

Fig. 4. Editing layout for components and popup-menu for controlling components

We developed various components, e.g., a text viewer/editor component and a JPEG or GIF viewer component (there are several example components in Fig. 9). Note that visual components allow their content to be in arbitrary as well as standard formats, because they have codes for viewing and modifying content. Most Swing and AWT GUI Widgets can be used as our components in the framework without modifications, because they have been derived from the `java.awt.Component` class.

4.4 Component distribution and duplication

The runtime system can transmit the marshaled nodes of components to their destinations through an extension of the HTTP protocol.[2] Since the runtime system transmits both the code and state of the components to the destinations, the components can continue processing, even when the destinations are disconnected from the source. Each runtime system also has a built-in mechanism for writing the marshaled component and reading it from the underlying file system, network file system, and database system without losing the component's containment structure or inner components. To duplicate components, the system marshals components into a bit-stream and then duplicates the marshaled component, because Java has no deep-copy mechanisms, which can make replicas of all objects embedded in and referred to from these components. The current implementation treats a component and its clones as independent. Components also have no problems with incompatibility even with different versions, because they contain their codes.

The current implementation uses the Java object serialization package to marshal and duplicate the states of components into a bit-stream. The package does not support the capturing of stack frames of threads. Consequently, our system cannot marshal the execution states of any thread objects. Instead, the runtime system (and the Java

[2] The current implementation can support HTTP tunneling to transfer components outside firewalls.

virtual machine) propagates certain events to components before and after marshaling and unmarshaling them. The current implementation of our system uses the standard JAR file format for passing components that can support digital signatures, allowing for authentication. If inner components embedded in a component share the same codes, the runtime system can detect and remove such redundant codes from the bit-stream corresponding to the marshaled component, including its inner components to reduce the size of the bit-stream.[3]

5 Component-based network processing

Each component for network processing is invisible and has been designed to provide its service to its inner components. A component can directly instruct its inner components to move to another location, and can transform them.

5.1 Carrier component

Carrier components are transparent, can carry other components to their destinations along their itineraries, and transform their inner components (see Fig. 5).

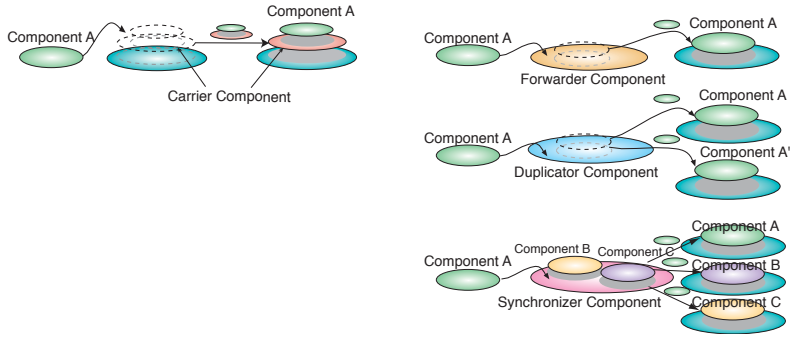

Fig. 5. Carrier component and basic forwarder components for network processing.

We developed a language enabling mobile agents to specify their own itineraries from multiple destinations [22]. The carrier components can define their itineraries with the language and migrate to other computers along their itineraries. When the movement of a component deviates from its registered itinerary, the runtime system issues an exception to the component. Moreover, carrier components can encapsulate or restrict their inner components, because they can control them while carrying them, and they can provide a secret-key-based cryptographic procedure to protect these inner components against illegal access or modification.

[3] This optimization mechanism involves a trade-off because its detection of redundant codes is not always lightweight. We intended to disable the mechanism in the evaluations presented in Section 6.

5.2 Forwarder component

Forwarder components are statically or dynamically deployed at components. When a forwarder component receives a component, it processes the visitor and then forwards it to its target component. A variety of processes for components, e.g., duplication and synchronization, can be defined in derivations from the forwarder component. The current implementation provides basic operations for component migration (Fig. 5). By combining these components, we can easily customize network processing. Since these protocols are given as Java abstract classes, we can easily define further advanced network processing by extending these basic protocols.

- **Forwarder component** can redirect its inner component to other locations. When it receives a component, it automatically transfers the visiting component to its specified destination as long as the destination is within the range that the inner component can migrate to.
- **Duplicator component** can receive another component and then create a copy of its visiting component including all instance variables. The cloned component has the same content as the original.
- **Synchronizer component** can strand its inner components until it can determine whether specified conditions can be satisfied, e.g., the number of inner components, the arrivals of specified components, and time constraints. A typical synchronizer component defines a group of moving components, as a barrier synchronization mechanism for parallel processes. It strands the visiting components inside it, until it receives all the components within the group.

We can define flows of components over a distributed system as a combination of forwarder components like the active routers (or nodes) in active network technology. The components previously described have properties that customize their processing and provide support to GUI editors like those for the property editors developed by Java Beans. The editors allow users to edit the property values of a given type. For example, forwarding components can configure their destinations in their properties and synchronization components can explicitly define their conditions. Moreover, these components can be dynamically deployed at remote hosts through document manipulations because they are still components of compound documents.

Since these components cover most basic functions to implement network protocols, end-users can rapidly and easily implement the protocols they want by combining components. Various types of network processing for components can be implemented as components. Since these protocols are given as abstract classes in the Java language, we can easily define further advanced network processing by extending these basic protocols. Fig. 6 outlines application-specific document-delivery (for workflow management systems) executed by combining basic components.

5.3 Network-wide component manipulation

This framework itself does not support any network-wide component manipulations, e.g., cut-and-paste and drag-and-drop between computers. Instead, such operations can

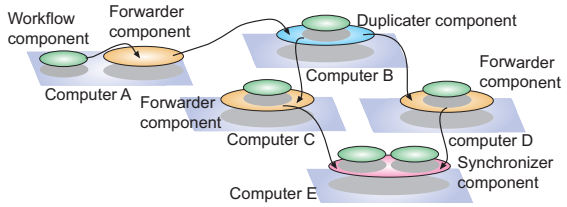

Fig. 6. Combination of basic components for network processing

easily be achieved through the *forwarding* and *duplicator* components. One can also use *forwarding* components as a mechanism to deploy network processing at remote computers. Figure 7 presents a compound document that includes several forwarding components, which automatically transfer other components to specified components at remote nodes. When a user wants to enable new network processing at remote nodes, he/she drags and drops the component that supports the processing to forwarding components corresponding to the nodes. This action transfers that component to the target of the forwarding component. Moreover, forwarding components can have property editors for their target components at the new location in addition to their own editors, allowing us to customize the properties of components deployed at remote computers by using their editors, which can be implemented as plug-in modules. Note that the user interface presented in Fig. 7 has only been implemented as components. Therefore, the interface itself can easily be distributed to other computers. That is, a component can be viewed as the only constituent of the framework. This gives users and programmers a single unified perspective of the system.

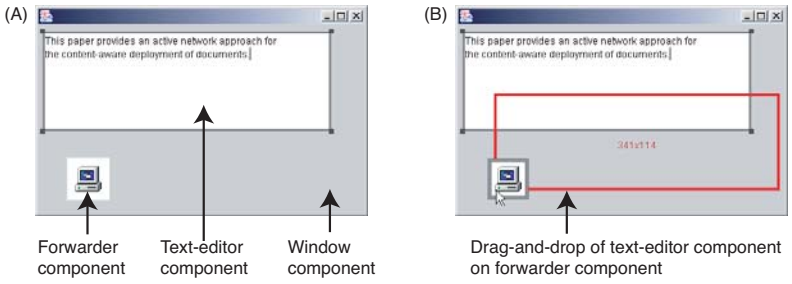

Fig. 7. (A) Screenshot of window component contains text-editor and forwarding components and (B) screenshot of window component when text-editor component is dragged and dropped on forwarding components.

Fig. 8 presents a compound document for deploying components for network processing at computers on a network. When three forwarder components contained in the document receive components, they automatically forward their visitors to their target computers. We can easily migrate a component for network processing at each of the forwarders by duplicating the component through our duplicator component. Note that the document is just a configuration for network processing and can be stored in secondary storage as a first class object just like visual components.

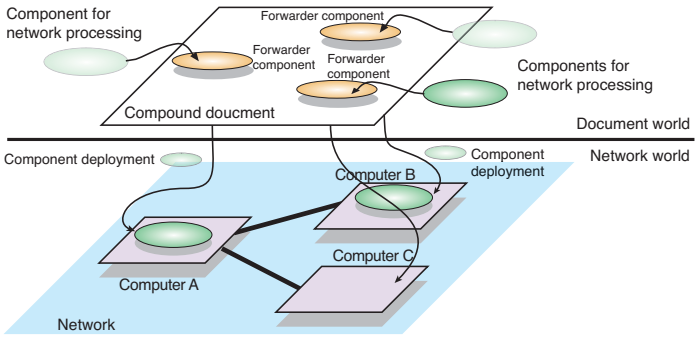

Fig. 8. Compound document for deploying components at computers.

6 Current Status

We implemented the framework using Java language (JDK 1.4 or later version), and we developed various components for compound documents and network processing. Since the Java virtual machine and libraries abstract away differences between the underlying systems, e.g., operating systems and hardware, components can migrate between and be executed on runtime systems running on different computers, whose underlying systems may be different.

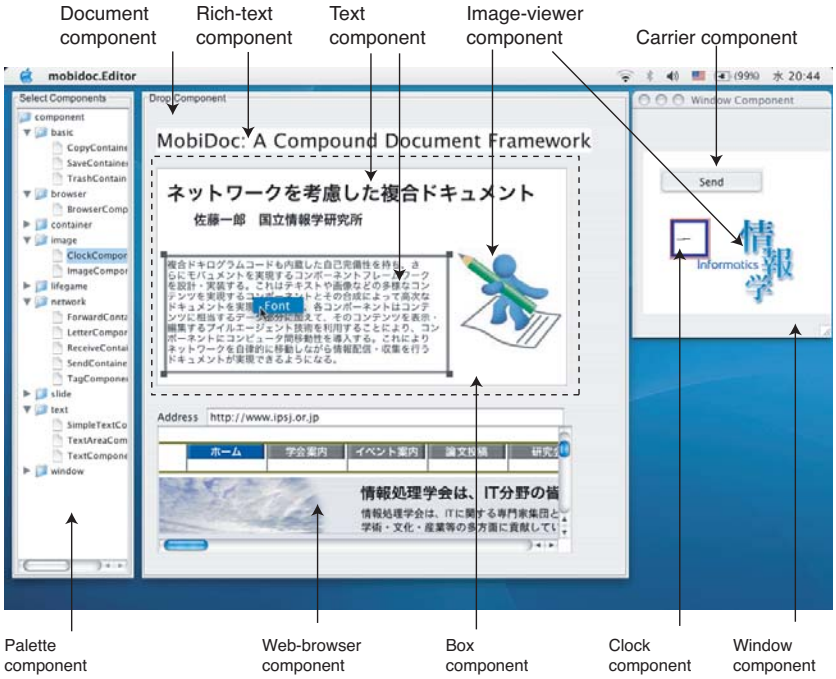

Fig. 9. Example compound documents

Fig. 9 has a screen-shot of this framework. The left window is a palette of part components and the center and right windows are compound documents contained in the components corresponding to GUI-windows. When a user wants to place a component on his/her editing compound document, he/she drags the wanted component from the palette and then drops it on the estate of the document. Since the palette itself is implemented as a container component of part components, it can migrate to another computer and be saved in secondary storage. We can register new components, which may be edited or modified, in the palette through GUI-based data-transfer operations, e.g., drag-and-drop or cut-and-paste. End-users can also define and customize their application-specific network processing by combining forwarder components through GUI manipulations in the same way as if they were editing visual components in documents.

6.1 Basic performance

Even though our current implementation was not built for performance, we evaluated some basics of the framework.

Component migration We conducted a basic experiment on component migration with computers (Pentium III 1.2-GHz with Windows XP and Sun's JDK 1.4.2). The time for component migration measured from one container to another in the same hierarchy was 10 msec., including the cost of drawing the visible content of the moving component and checking whether the component was permitted to enter the destination component. The cost of component migration measured between two computers connected through a Fast-Ethernet was measured at 64 msec.. The cost was the sum of marshaling, compressing, opening a TCP connection, transmitting, acknowledging, decompressing, verifying security and consistency, unmarshaling, laying out the visual space, and drawing the content. The moving component was a simple text viewer and its size (sum of code and data) was about 9 KB (zip-compressed). The latency of component migration was reasonable for a Java-based visual environment for exchanging compound documents between computers.

Component size Since each component in this framework not only contains its content but also program code, documents or components transmitted over a network or stored on secondary storage tend to be large. We compared the sizes of documents in this framework and the size of documents created with MS-Word, which is the most typical office application.[4] The sizes of two typical kinds of content were as follows:

- The first content was plain text. The size of the component for viewing and editing the text was 5.6 KB (0), 6.3 KB (1,000), 9.9 KB (10,000), and 19.5 KB (100,000), whereas the size of the document created with MS-Word was 19.5 KB (0), 21.0

[4] Note that the sizes of documents, which contain multimedia content, created with MS-Word may vary for no reason, so that we could not systematically compare the sizes of our documents and MS-Word documents. The results presented in this section have not always been generalized but have focused on several samples.

KB (1,000) 47.1 KB (10,000), and 306.2 KB (100,000), where the numbers in parentheses represent the length of the text where the text-content was part of this paper.

– The second was image content within the dotted box in Fig. 9. The content is composed of three components: box, text, and image-viewer, where the first contains the second and third components. The content was 68 KB, where the document corresponding to the content created with MS-Word was 24 KB.

The size of the text component was not proportional to the length of the text because our components were migrated over a network or stored in secondary storage and they were compressed in JAR-format, which was ZIP-based compression. This meant that our components were not always larger than documents created with commercial applications, e.g., MS-Word. We also found that the size of our components greatly depended on the kind of content and the complexity of their codes. When various types of components were embedded into a document, the document tended to be larger, because each type needed to embed its code for the content type into the document. Otherwise, the size was often smaller than that of corresponding documents created by existing applications, when the document contained a few component types. The size of our components was reasonable because the network bandwidth and the capacity for storage have recently increased.

7 Application

We developed a variety of components based on this framework. This section introduces several of these and their uses.

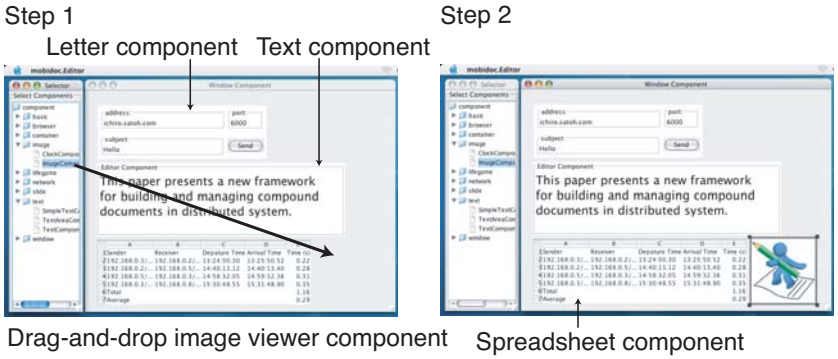

Fig. 10. Letter document

7.1 Compound document letter

Although documents are often sent to their destinations through electronic mail systems, it is difficult for these systems to send the documents to multiple destinations along specified itineraries. To solve this problem, this framework provides carrier components that can travel between multiple computers under their own control. Figure 10

shows a letter constructed as a compound document. It consist of three components: carrier, image viewer, and text components. The carrier component has a UGI corresponding to the header on an electronic mail and it can migrate with its inner components, i.e., the text and spreadsheet components, to its destination. We can easily add new components to the letter compound document by dragging them from the palette and dropping them on the document. Since not only the content but also the codes for viewing and editing the content of the components are transferred to the destination, the letter document can be viewed at the destination with no applications necessary for the content. The spread-sheet component has its own profile because it can automatically record when it has been (un)marshaled and where it has visited.

Most electronic mail systems disallow letters from traveling among multiple destinations along their own itineraries. We developed a legacy decision-making system, called *ringi*, for group decision-making, which has been widely used throughout Japan. When an employee proposes something to his/her company, he/she describes the proposition on a workflow document, called a *ringi-sho*. The document must be handed to all sections involved with the proposed issue. When the managers of the sections deem the proposal to be of worth, they stamp it with their hanko, or give rights access through its carrier component, and then resume its process as seen at the right of the computer.

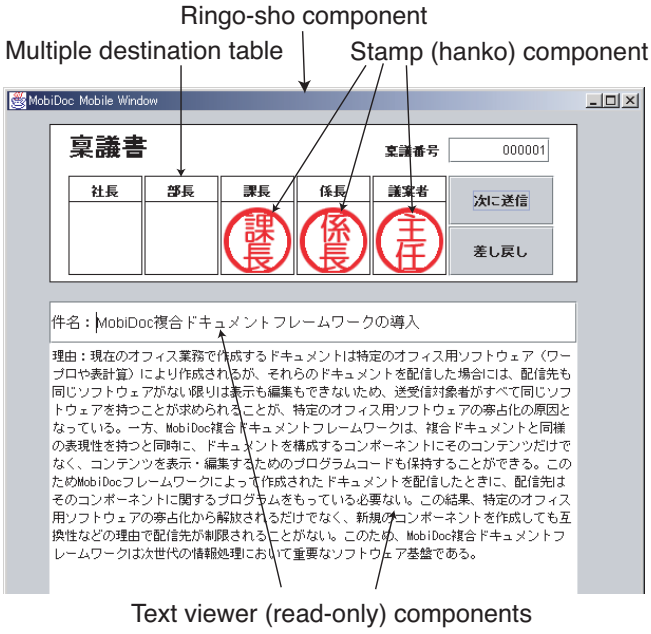

Fig. 11. Ringi-sho compound document.

7.2 Distributed Presentation System

This system is unique to other existing presentation systems because it can exchange slides or its visual parts, e.g., text and images, between different computers. Fig. 12 is a

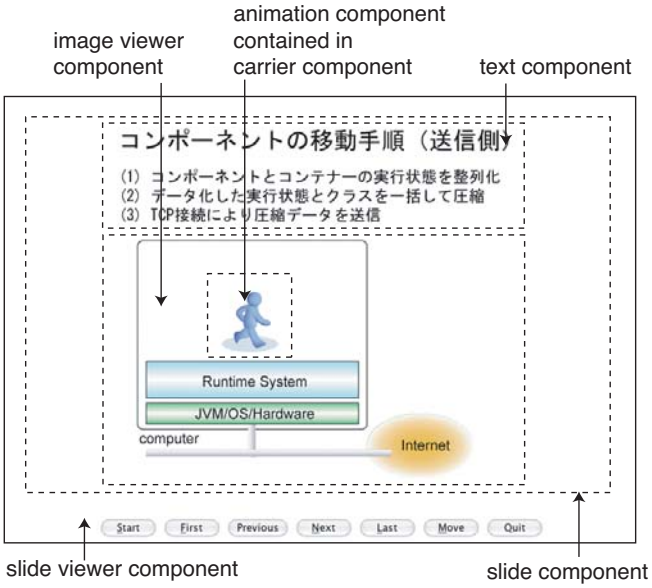

Fig. 12. Slide component.

slide-presentation compound document that can contain more than one slide component inside it, where each slide component corresponds to one slide and can contain and view one or more visual components. It stacks its slides by using the `java.awt.CardLayout` class as a layout manager. It enables us to change the order of the stack and exchange slides with other slide-presentation compound documents running on different computers through a GUI-based control panel at the bottom of its window. The center image is a GIF-animation-viewer component contained in a carrier component. When the carrier component is made active by clicking the mouse within its estate, it migrates to its specified component or computer. Figure 13 shows that the animation-viewer component has migrated between slides from the left (Macintosh) to the right computer (Windows PC) through its carrier component and has then resumed its animation at the right of the computer.

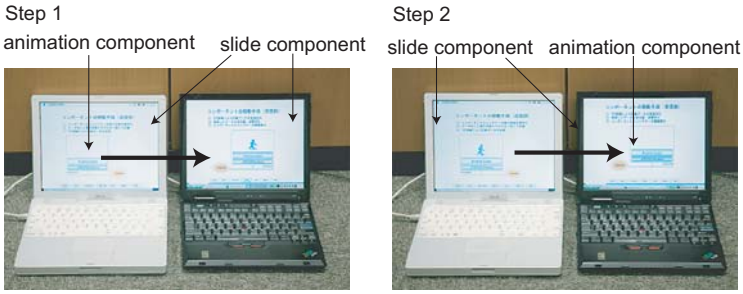

Fig. 13. Migration of components between slide components running on different computers

7.3 Application-specific document distribution

This framework enables end-users to easily define and operate CSCW applications. The third example is an editing system for an in-house newsletter. Each newsletter is edited by automatically compiling one or more text parts, which are written by different people as we can see from Figure 14. A newsletter compound document has one page component, which can contain editor components for visual content, e.g., text and images. When the newsletter is being edited, it forwards the page component to a duplicator component to make replicas of the component to match the number of writers. The duplicator component then migrates the replicas to forwarder components so that each of the page components is forwarded to a window component on its writer's computer. When it arrives at the destination, it displays a window for its editor program on the screen of the computer to assist the writer. Each writer can add his/her visual components to the page component. If a page component can contain sharing components, the components contained in the sharing components can maintain common content for all the replicas, even when one of the replicas has changed. For example, each page of the newsletter can have common text and images in its header and tail. Sharing components are located at the head and tail of each page. When a writer modifies text in text components or images in image-viewer components containing sharing components, this modification can be reflected on the heads and tails of all pages, while other writers are editing their pages. This modification goes back to the document after the writer has finished writing his/her text and then the document arranges the arriving components as a bound set. Since the newsletter document, duplicator, and forwarder components are still mobile, they can easily be deployed and coordinated according to the requirements of applications.

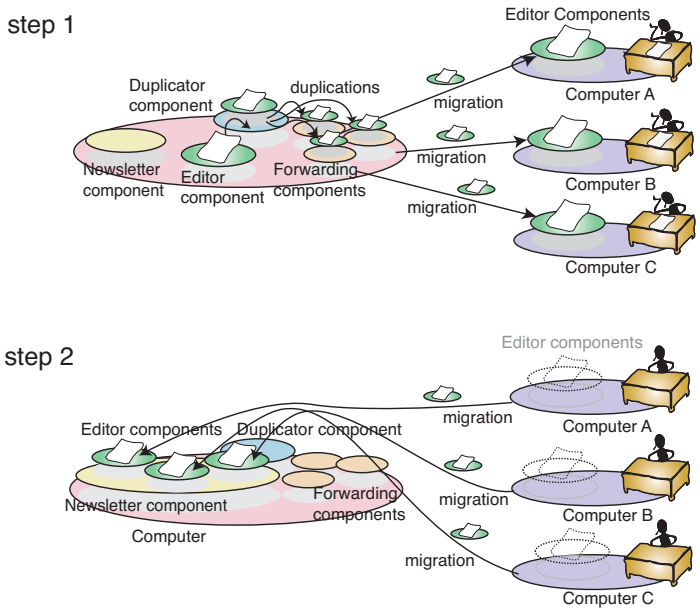

Fig. 14. Newsletter editing system.

8 Conclusion

We presented a compound document framework for document-centric network processing. The framework made three novel contributions. The first introduced the notion of a component hierarchy and mobile components. This notion enabled an enriched document to be composed of various components and to migrate between these components, which may run on different computers, under its own control. The framework provided several value-added mechanisms for visually manipulating components embedded in a compound document and for seamlessly combining multiple visible components into one. The second contribution was that it made the content of each component and its codes inseparable. It allowed us to view or modify components without the need for any applications. It was also useful in protecting content because it prevented the content of components from being accessed by external systems and it enabled us to easily insert security mechanisms in each component's codes to view or edit its content. The third contribution enabled documents to pass other documents from/to other components or computers. Components were introduced as the only constituent of our network processing for documents or components. It also offered several basic operations for network processing, e.g., carrying, forwarding, duplication, synchronization, and sharing. The operations were still document components; they could be dynamically deployed at local or remote computers through GUI-based manipulations. It therefore allowed an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. In fact, a variety of advanced network processing can be implemented as a collection of components for network processing. We constructed a prototype implementation of this infrastructure and its applications.

To conclude, we would like to point out further issues that need to be resolved. Resource management and security mechanisms in the current system were incorporated in a relatively straightforward manner. These should now be designed to incorporate compound documents. This framework does not support interoperability with other document frameworks, because these assume that each component can be viewed or modified by external applications. Nevertheless, it is open to supporting such existing document formats because the framework can provide viewers and editors components that can encapsulate such documents with their viewers and editors corresponding to their applications. When a component migrates to another component or computer, its visual resources, i.e., the size of its estate and colors in the destination, may not be the same as those in the source. It must adapt its visibility to the resources available in the current location; however, the current implementation relies on Java's layout manager. We need a more sophisticated and flexible mechanism to enable adaptation.

References

1. Apple Computer Inc. (1994) OpenDoc: White Paper, Apple Computer Inc.
2. K. Arnold, and J. Gosling, The Java Programming Language, Addison-Wesley, 1998
3. K. Brockschmidt, Inside OLE 2, Microsoft Press, 1995.
4. Cable, L. (1997) Extensible Runtime Containment and Server Protocol for JavaBeans, Sun Microsystems, http://java.sun.com/beans.

5. D. P. Friedman, M. Wand, and C. T. Haynes, Essentials of Programming Languages, MIT Press, 1992.
6. A. Fuggetta, G. P. Picco, and G. Vigna, Understanding Code Mobility, IEEE Transactions on Software Engineering, vol.24, no.5, 1998.
7. The GNOME Project, Bonobo, http://developer.gnome.org/ arch/component/ bonobo.html, 2002.
8. Y. Goldberg, M. Safran, and E. Shapiro, Active Mail - A Framework for Implementing Groupware, Proceedings of ACM CSCW'92, pp. 75-83, ACM Press, 1992.
9. Hamilton G. (1997) The JavaBeans Specification, Sun Microsystems, http://java.sun.com/beans.
10. G.H. ter Hofte, H.J. van der Lugt, CoCoDoc: a framework for collaborative compound document editing based on OpenDoc and CORBA, Proceedings of International Conference on Open Distributed Processing and Distributed Platforms, pp.15-33, 1998.
11. G. Kiczales, et al, Aspect-Oriented Programming Proceeding of European Conference on Object-Oriented Programming (ECOOP'97), LNCS, vol. 1241, Springer, 1997.
12. B. D. Lange and M. Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.
13. R. Litiu and A. Parakash, Developing adaptive groupware applications using a mobile component framework, Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW'2000) , pp.107 - 116, ACM Press, 2000.
14. J. Morin, HyperNews, a Hypermedia Electronic-Newspaper Environment based on Agents, Proceedings of HICSS-31, pp.58-67, 1998.
15. M. Potel and S. Cotter Inside Taligent Technology, Addison-Wesley, 1995.
16. D. Rogerson, Inside COM, Microsoft Press, 1997.
17. 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.
18. I. Satoh, MobiDoc: A Mobile Agent-based Framework for Compound Documents, Informatica, vol.25, no.4, pp.493-500, December 2001.
19. I. Satoh, Building Reusable Mobile Agents for Network Management, IEEE Transactions on Systems, Man and Cybernetics, vol. 33, no.3, part-C, pp.350-357, August 2003.
20. I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, vol. 7, no.1, pp.73-83, Kluwer, January 2004.
21. I. Satoh, A Compound Document Framework for Multimedia Networking, Proceedings of 1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA'2005), pp.80-87, IEEE Computer Society, February 2004.
22. I. Satoh, Selection of Mobile Agents, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2004), pp.484-493, IEEE Computer Society, March 2004.
23. I. Satoh, Network Processing of Documents, for Documents, by Documents (short position paper), to appear in Proceedings of ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'2005), Lecture Notes in Computer Science (LNCS), December 2005.
24. C. Szyperski, Component Software (2nd), Addison-Wesley, 2002
25. D. L. Tennenhouse et al., A Survey of Active Network Research, IEEE Communication Magazine, vol. 35, no. 1, 1997.
26. P. Werle, F. Kilander, M. Jonsson P. Lonqvist, C. G. Jansson, A Ubiquitous Service Environment with Active Documents for Teamwork Support, Proceedings of 3rd International Conference on Ubiquitous Computing (UBICOMP'2001), Lecture Notes in Computer Science, vol. 2201 pp.139 - 155, Springer, 2001.
27. J. E. White, Mobile Agents, in Software Agents (eds. J. M. Bradshaw), pp.437-472, AAA Press/MIT Press, 1997.

Appendix: Component programming model

Each visual component needs to be defined as a subclass of either the `java.awt.Component` or `java.awt.Container` from which most of Java's visual or GUI objects are derived.[5] When a component is moved to another location or its estate is resized, its component node invokes `repaint()`, which is a callback method to redraw the visible content of the `java.awt.Component` class. Therefore, each visual component is required to define its own visual features in addition to its application logic.

The framework also provides an event listener interface, called the `MDLifecycleListener`, which defines several callback methods to inform components about changes in their lifecycle state from the runtime system, like the Java AWT event listener interfaces. If a component implements the interface, before or after the lifecycle of the component has changed, the runtime system invokes these methods.

```
interface MDLifecycleListener
implements java.io.Serializable {
  void create(MDContext mdcx) throws IllegalAccessException;
  void destroy(MDContext mdcx) throws IllegalAccessException;
  void save(MDContext mdcx) throws IllegalAccessException;
  void suspend(MDContext mdcx) throws IllegalAccessException;
  void duplicate(MDContext mdcx) throws IllegalAccessException;
  void leave(MDContext mdcx, URL dst) throws IllegalAccessException;
  void arrive(MDContext mdcx, URL src) throws IllegalAccessException;
  void add(MDContext mdcx, URL src) throws IllegalAccessException;
  void remove(MDContext mdcx, URL src) throws IllegalAccessException;
  ....
}
```

A component, which implements the `MDLifecycleListener` interface, can define the processing it requires as the behaviors of the methods. These methods can also define processing for authentication and content rights management. Note that, even when a component does not support the interface, the Java virtual machine invokes the `writeObject()` or method of a component, before the component is marshaled or after it is unmarshaled. The `MDContext` class in the `MDLifecycleListener` interface defines service methods available in components as follows:

```
class MDContext implements java.io.Serializable {
  public void go(URL url) throws NoSuchHostException { ... }
  public URL save(URL url) throws IOException { ... }
  public ComponentID duplicate() { ... }
  ....
}
```

When a component executes `go()` or `save()`, the runtime system migrates (or stores) the component, including its inner components, to the component (or in the file) specified as `url`. The `duplicatee()` method makes a replica of the component itself.

[5] `java.awt.Container` is a subclass of `java.awt.Component`.