A Component Framework for Document-centric Network Processing

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ichiro@nii.ac.jp

Abstract

A component framework for defining content-based network processing is presented. It is unique among other existing approaches because it enables contents to naturally define their own processing and end-users to easily define network processing. By using it, we can dynamically make an enriched document as a nested composition of software components corresponding to various content, e.g., text, images, and windows. 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. It allows end-users to easily define their own content-based or application-specific network processing by assembling these components in through GUI manipulations. 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, saving, and distributing documents, still plays a crucial role in modern information processing. In fact, electronic mails exchanging documents between users through networks and web servers enable us to share documents stored in remote computers. However, existing network processing of documents, e.g., electronic mails and webs, just treat documents as data, i.e., first class objects. However, network processing of documents tends to be content-dependent and application-specific. 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 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. Some confidential documents also need to pass through their favorite secure protocols.

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 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, with 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. Furthermore, documents can transmit other documents as first-class objects to their destinations. The framework introduces 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.

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

This section briefly describes the background of this framework and its basic approaches.

2.1 Compound Document Framework

There have been many component frameworks for distributed computing, e.g., Enterprise JavaBeans (EJB) [19] and Distributed COM (DCOM) [], because building systems from software components has already proven useful in the development of large-scale software. These existing frameworks aim at defining the behaviors of distributed computing, i.e., server-side and client side processing. Therefore, these frameworks are suitable for professional developers but not end-users.

There have been a few component frameworks for building enrich documents, i.e, so called compound documents, e.g., OLE [2], OpenDoc [1], and CommonPoint [11], where various visible parts, e.g., text and images created by different applications, can be combined into one document and manipulated in-place within the document. There have been several problems with these existing compound document frameworks in distributed computing settings. A compound component is defined in two parts: content and code to modify the content. Content is stored inside the component but the code for access it not always there. Thus, when a user receives a compound document, he/she cannot view or modify it if its content needs the support of different applications, if he/she does not have all the applications. Moreover, existing compound documents are inherently designed as passive entities in the sense that their content can be transmitted over a network by external network systems, such as electronic-mail systems and workflow management systems so that they cannot determine where they should go next.

The framework presented in this paper 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 naturally enjoy novel and powerful features that existing components do not have. End-users can also easily customize network processing of documents through user-friendly manipulations to edit visual components and they can control their own network processing according to their content. The framework can also use typical Java-based components, e.g., Java Beans and Applets, as its components.

2.2 Related Work

We discuss several related works in the remainder of this section. Our framework is implemented with Java. Java provides a general component framework, called JavaBeans, for building reusable software components designed for Java language rather a document-centric framework. The initial release of JavaBeans [8] did not contain a hierarchical or logical structure for JavaBean objects, but its latest release [3] allows JavaBean objects to be organized hierarchically. However, the JavaBeans framework provides no higher-level document-related functions. Moreover, it has not inherently been designed for mobility. Therefore, it is difficult for a group of JavaBean objects in the containment hierarchy to migrate to another computer.

Recently, several projects have started constructing component frameworks for compound documents. Of these, the Bonobo framework for software components and compound documents is being developed by the GNOME project [6]. 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. XML-based technologies can also offer rich documents, but they inherit the problems of compound documents, because, when a computer receives XML-based documents, the computer may not have viewer/editor programs for them. This framework uses mobile agent technology. However, the technology assumes each mobile agent to be an isolated entity that migrates independently and it does not support any document-centric approaches. To customize distributed computing, particularly network processing between computers, several researchers have explored active networking technologies [20]. However, these existing technologies have focused on configurations for low-level network processing, e.g., routing and QoS protocols and they are not suitable for end-users.

We presented a compound document framework for providing software components designed for compound documents [17, 18]. Although the previous framework was an early prototype of the framework presented here and it enabled components to carry and forward other components over a distributed system, the previous work was designed for distributed documents under the documents' control, whereas the framework presented in this paper supports various networking for documents.

3 Design Principles

This section briefly outlines the framework presented here.

3.1 Requirements

This framework must satisfy the following requirements.
Composition: Like OpenDoc and JavaBeans, our framework must compose 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 and can determine which components or computers it will go to according to its program code and content, and then migrate to that destination.
Reconfiguration: The network processing of documents, components, or plain data can be easily and naturally de-

fined 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. Our notion makes documents both secure and portable. The framework permits only the code of a component to access its content and it automatically invokes the code before or after the component is viewed, modified distributed, duplicated, and saved. When a user receives a document, he/she can view or edit the document by using its code instead of any applications deployed at its current computer. This results in increasing in the size of the document but this will not be excessively large in comparison with documents created from modern office applications as will be discussed in Section 6.

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 freely move into any other component except itself or its descendant components, as long as the destination component accepts it. The destination may be at a different computer. 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 in other components or on secondary disks, whereas each contained component cannot control its container component.

Component interaction

Component technologies to develop standalone or distributed systems have been needed to support various mechanisms to enable interactions between components. Our compound documents themselves, on the other hand, are visible and mobile. Early experience with this framework suggests that components embedded in a compound document only require simple interactions between one another. Therefore, 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 upon which they had relied.

3.3 Network processing

This framework provides two approaches to enable 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 four 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. Network processing components, e.g., carrier, and forwarder components can also carry or forward other network processing components over a network like visual components. These components can be assembled and operated through GUI manipulations and embedded into a document as visual components.

4 Design

This framework consists of two parts: runtime systems and components. The framework itself is independent of any programming language, but its prototype implementation is constructed on Java. It can exchange components between runtime systems, even when their underlying systems, e.g., operating systems and hardware, are different, because Java VM conceals differences between the underlying systems.

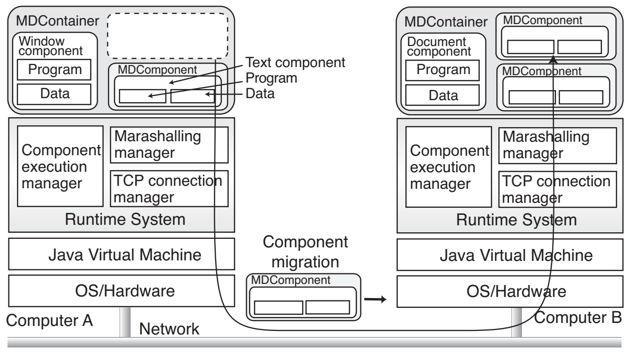


Figure 1. Component migration between two computers.

4.1 Component runtime system

Each runtime system is a middleware system for managing and executing components. Figure 1 outlines the basic structure of a runtime system. Each runtime system governs all the components inside 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.

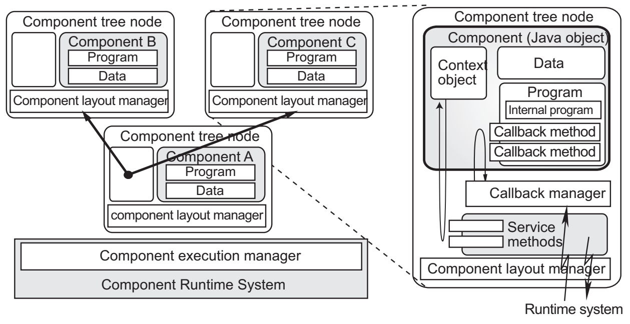


Figure 2. Component hierarchy and structure of components.

Component migration

When each runtime system saves or migrates a component over a network, it marshals the component, the component's inner components, and information about their containment relationships and visual layouts, called component nodes, into a bit-stream and then later unmarshals the components and information from the bit-stream. The runtime system then transmits the marshaled component to its destination through an extension of the HTTP protocol.[1] Since the runtime system transmits both the code and state of the component to the destination, the component can continue processing, even when the destination is 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.

The current implementation uses the Java object serialization package for marshaling and duplicating the states of components. 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 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.

Component migration

The runtime system marshals the components into a bit-stream to duplicate components 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.[2] To reduce the size of the bit-stream, 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.[3]

Component hierarchy management

The container technology developed by Enterprise Java Beans provides interfaces for components and enables these to transparently adapt to runtime services, e.g., transaction management. 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 inside it and provides interfaces between its component and the runtime system (Figure 2). When a component is created in a runtime system, the system creates a component tree node for the newly created component and runs the node inside it. When a component migrates to another location or duplicates itself, the runtime

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

[2]Since the framework treats a component and its clones as independent, it does not support any consistency between them.

[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.

system migrates its node with the component and makes a replica of the node.

As we can see from Figure 3, a hierarchy of components is maintained in the form of a tree structure, which has the component tree nodes of components. Each node is defined as a subclass of `MDContainer` or `MDComponent`, where the first supports components, which can contain more than one component inside it and the second supports components, which cannot contain any components.[4] For example, when a component has two other components inside it, the nodes that contains the two inner components are attached to the node that wraps the first 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.

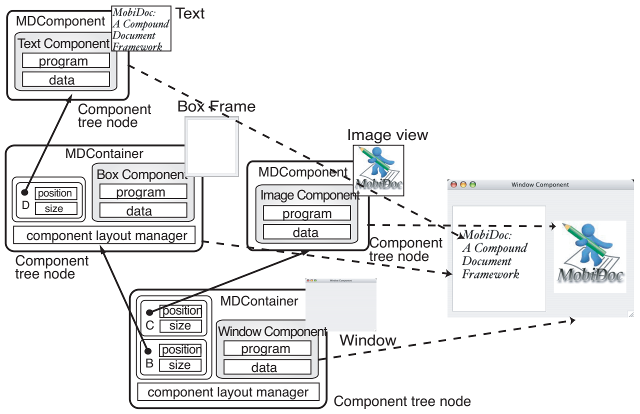


Figure 3. Component Hierarchy

4.2 Component manipulation

Each component tree node defines the protocols that let components embedded in it to communicate with one another. 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.

Visual layout management

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, preferable size, and maximum size of the visible estate of its component, but the estate is controlled by the node of its container component. If a component contains more than one component inside it, its node is responsible for assigning its inner components their rectangular estates within its estate according to the node's layout manager, and for controlling the sizes, positions, offsets, and order of their estates. When a visual component moves to another visual component, the moving component's estate may be allocated at different positions or be different in size. We can customize visual layout management by defining it as a subclass of the `MDContainer` or `MDComponent` class. We can also define its new layout manager as an instance of a subclass of the `java.awt.LayoutManager` class.

Visible operations

Each component tree node can dispatch certain events to its components to notify them when certain actions happen within their surroundings. 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 (Figure 4 (a)). The user can resize the selected component, move it to another component, save it, and terminate it by dragging its handles (Figure 4 (b)). Each node can define its own document-wide operations, such as mouse clicks and keystrokes as well as the built-in operations of the framework.

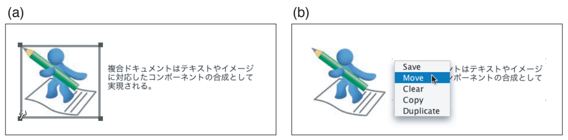

(a)
複合ドキュメントはテキストやイメージに対応したコンポーネントの合成として実現される。
(b)
Save
Move
Clear
Copy
Duplicate
ントはテキストやイメージ
ンポーネントの合成として

Figure 4. Editing layout of component and popup-menu for controlling component

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. When a component wants a service, it migrates into one of the components that can provide that service.

5.1 Carrier component

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

[4]The runtime system basically provides a node derived from the `MDContainer` class for components except for the visual components that has been designed not to have any inner components, e.g., text-viewer and sound-player components.

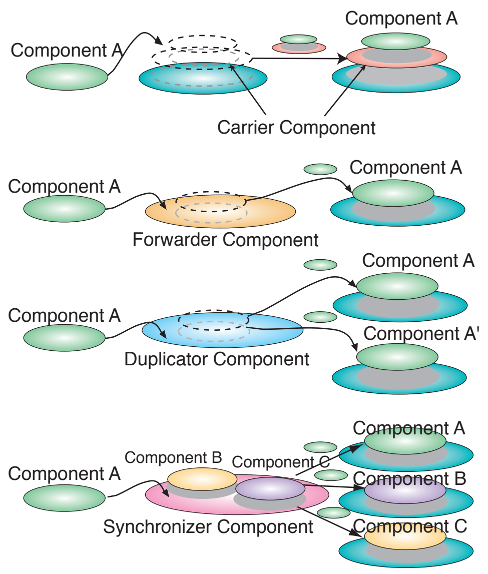


Figure 5. Carrier component and basic forwarder components for network processing.

Component distribution control

We developed a language enabling mobile agents to specify their own itineraries from multiple destinations [16]. The carrier components presented in this paper can define their itineraries based on the language and migrate to other computers along their itineraries. A carrier component may be deployed at another location by forwarder components. When the movement of a component deviates from its registered itinerary, the runtime system issues an exception to the component.

Component access control

Moreover, carrier components can encapsulate or restrict their inner components, because they can control their inner components, while carrying them. This is useful in protecting content in visual components against illegal access or modification. For example, we provided a special component, called a safe component, which is like an armored car. It is a container of other components and has a secret-key-based cryptographic procedure inside it. When a component visits the safe component, the safe component automatically serializes and encrypts the visiting component under a secret key. It next migrates itself to the destination as a whole with all its inner components and its cryptographic procedure except for its secret keys. After arriving at the destination, the safe component keeps its inner components secure inside it. Safe components can be implemented independently of cryptographic algorithms because the algorithms must be selected according to the requirements of the application. A non-standardized cryptographic algorithm can be embedded into a safe component without losing any interoperability because the component can convey the procedure for the algorithm and perform it at both the source and destination computers.

5.2 Forwarder component

Forwarder components are statically or dynamically deployed at components. The forwarder component is a key component in the framework. A variety of processes for components, e.g., duplication and synchronization, can be defined in derivations from the forwarder component. When a forwarder component receives a component, it processes the visiting component and then forwards it to its target component.

Basic forwarder components for network processing

The current implementation provides basic operations for component migration (Figure 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 places. 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 component.

Synchronizer component can strand its inner components until it can determine 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 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. Figure 6 shows application-specific document-delivery (for workflow management systems) executed by combining basic components.

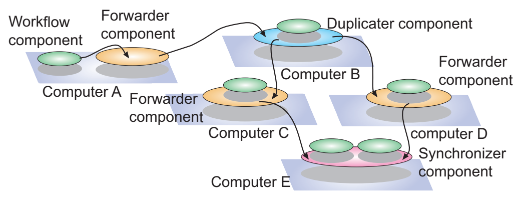
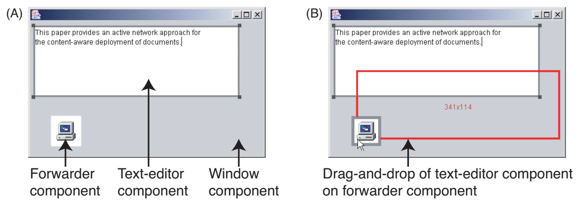


Figure 6. Combination of basic components for network processing



This paper provides an active network approach for the content-aware deployment of documents.

This paper provides an active network approach for the content-aware deployment of documents.

Figure 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.

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 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 Figure 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.

Figure 8 shows 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.

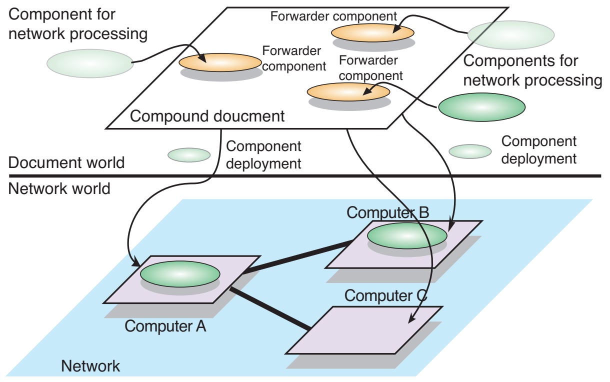


Figure 8. Compound document for deploying components at computers.

6 Current Status

We implemented the framework using Java language (JDK1.4 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.

Figure 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 copy-and-paste. End-users can also define

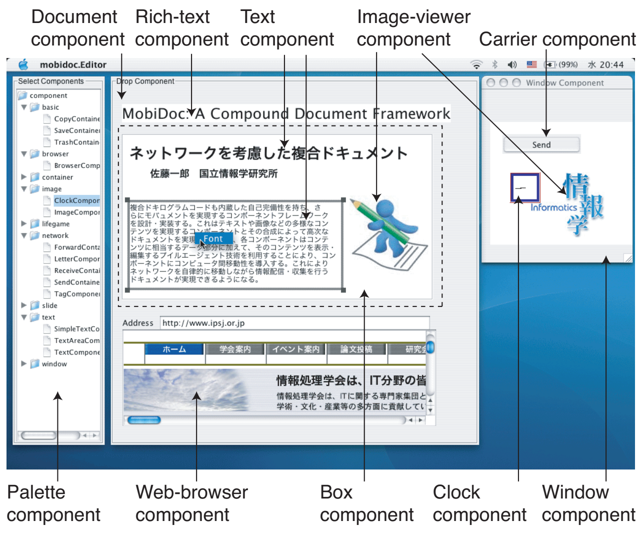


Figure 9. Example compound documents

and customize their application-specific network processing by combing forwarder components through GUI manipulations in the same way as if they were editing visual components in documents.

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 JDK 1.4.2). The time for component migration measured from one container to another container in the same hierarchy was 10 ms, 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 ms. The cost was the sum of marshaling, compression, opening a TCP connection, transmission, acknowledgment, decompression, security and consistency verification, unmarshaling, visual space layout, and drawing of 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 tends 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.[5] 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 Kbytes (0), 6.3 Kbytes (1,000), 9.9 Kbytes (10,000), and 19.5 Kbytes (100,000), whereas the size of the document created with MS-Word was 19.5 Kbytes (0)21.0 Kbytes (1,000) 47.1 Kbytes (10,000), and 306.2KB (100,000), where the numbers in parentheses represent the length of the text where the text-content was a part of this paper.
- The second is image content within the dotted box in Figure 9. The content is composed of three components: box, text, and image-viewer components, where the first contains the second and third components. The content was 68 Kbytes, where the document corresponding to the content created with MS-Word was 24 Kbytes.

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 means 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 size of the document tended to be large, 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 contain a few types of components. The size of our components was reasonable because network bandwidth and the capacity for storage have recently increased.

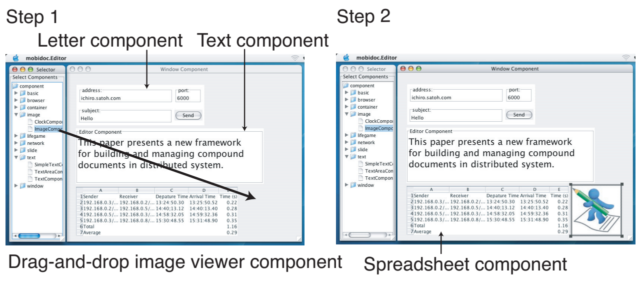
7 Experiences

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

7.1 Compound document letter

Although documents are often sent to their destinations through electronic mail systems, it is difficult for such systems to send the documents to multiple destinations along specified itineraries. To solve this problem, this framework

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



address: ichiro.satoh.com
port: 6000
subject: Hello

Editor Component
This paper presents a new framework for building and managing compound documents in distributed system.

	A	B	C	D	E
1	Sender	Receiver	Depature Time	Arrival Time	Time (s)
2	192.168.0.3/...	192.168.0.2/...	13:24:50.30	13:25:50.52	0.22
3	192.168.0.2/...	192.168.0.5/...	14:40;13.12	14:40:13.40	0.28
4	192.168.0.5/...	192.168.0.3/...	14:58:32.05	14:59:32.36	0.31
5	192.168.0.3/...	192.168.0.8/...	15:30:48.55	15:31:48.90	0.35
6	Total				1.16
7	Average				0.29

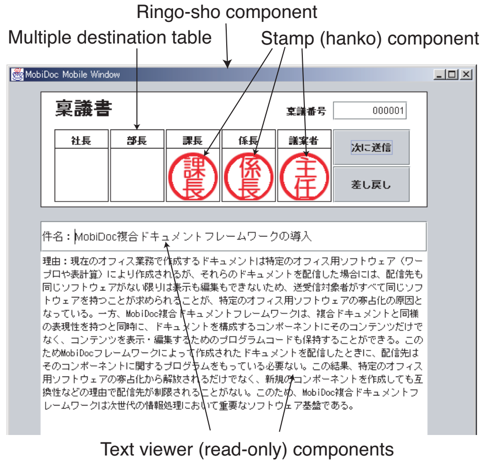
Figure 10. Letter document

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 graphical user interface corresponding to the header part of 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 over to all sections involved with the proposed issue. When the managers of the sections deem the proposal to be worthy, they give it their hanko, or right computer (Windows PC) through its carrier component and has then resumed its animation at the right of the computer.

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. We constructed a 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



稟議書

稟議番号 000001

社長	部長	課長	係長	議案者
		課長	係長	主任

件名：MobiDoc複合ドキュメントフレームワークの導入

理由：現在のオフィス業務で作成するドキュメントは特定のオフィス用ソフトウェア（ワープロや表計算）により作成されるが、それらのドキュメントを配信した場合には、配信先も同じソフトウェアがない限りは表示も編集もできないため、送受信対象者がすべて同じソフトウェアを持つことが求められることが、特定のオフィス用ソフトウェアの寡占化の原因となっている。一方、MobiDoc複合ドキュメントフレームワークは、複合ドキュメントと同様の表現性を持つと同時に、ドキュメントを構成するコンポーネントにそのコンテンツだけでなく、コンテンツを表示・編集するためのプログラムコードも保持することができる。このためMobiDocフレームワークによって作成されたドキュメントを配信したときに、配信先はそのコンポーネントに関するプログラムをもっている必要ない。この結果、特定のオフィス用ソフトウェアの寡占化から解放されるだけでなく、新規のコンポーネントを作成しても互換性などの理由で配信先が制限されることがない。このため、MobiDoc複合ドキュメントフレームワークは次世代の情報処理において重要なソフトウェア基盤である。

Figure 11. Ringi-sho compound document.

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 12 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.

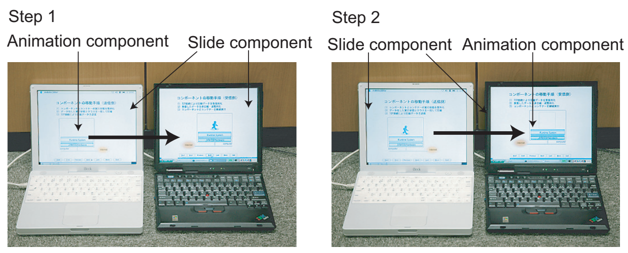


Figure 12. Migration of components between slide components running on different computers

7.3 Application-specific document distribution

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 13. 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 as many replicas of the component as 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. Also, the writer can add his/her visual components to the page component. It 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 be easily deployed and coordinated according to the requirements of applications.

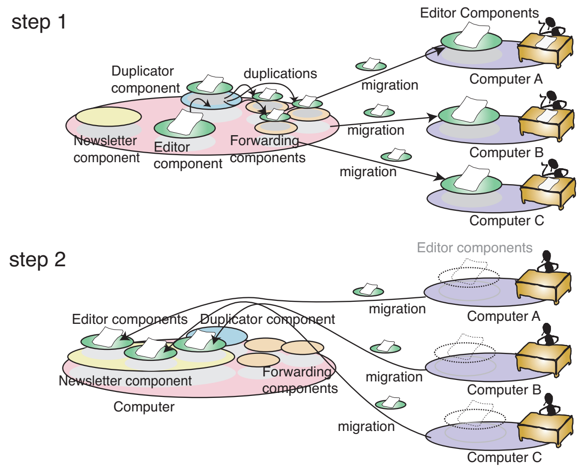


Figure 13. Newsletter editing system.

8 Conclusion

We presented a compound document framework for document-centric network processing. The framework made three contributions. The first introduced the notion of component hierarchy and mobile components. This notion enabled an enriched document to be composed of various components and migrate between 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 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 digital content because it prevented the content of components from being accessing from the external systems. The third 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., forwarding, duplication, and synchronization. Since the operations were implemented as 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. 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. 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, but the current implementation relies on Java's layout manager. We need a sophisticated and flexible mechanism to enable adaptation.

References

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

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

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

[4] P. Dourish et al, A Programming Model for Active Documents, Proceedings of 13th Symposium on User Interface Software and Technology (UIST'2000), pp.41 - 50, ACM Press, 2000.

[5] D. P. Friedman, M. Wand, and C. T. Haynes, Essentials of Programming Languages, MIT Press, 1992.

[6] The GNOME Project, Bonobo, http://developer.gnome.org/ arch/component/bonobo.html, 2002.

[7] 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.

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

[9] 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.

[10] J. Morin, HyperNews, a Hypermedia Electronic-Newspaper Environment based on Agents, Proceedings of HICSS-31, pp.58-67, 1998.

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

[12] D. Rogerson, Inside COM, Microsoft Press, 1997.

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

[14] I. Satoh, 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.

[15] I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, vol. 7, no.1, pp.73-83, Kluwer, January 2004.

[16] 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.

[17] I. Satoh, Network Processing of Documents, for Documents, by Documents Proceedings of ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'2005), Lecture Notes in Computer Science (LNCS), December 2005.

[18] I. Satoh, A Document-centric Component Framework for Document Distributions, to appear in Proceedings of 8th International Symposium on Distributed Objects and Applications (DOA'2006), Lecture Notes in Computer Science (LNCS), October 2006.

[19] Sun Microsystems, Inc., Enterprise JavaBeans Technology (EJB) http://java.sun.com/products/ejb, 2002.

[20] D. L. Tennenhouse et al., A Survey of Active Network Research, IEEE Communication Magazine, vol. 35, no. 1, 1997.