Technical Note #10074
Version InDesign CS

Feb 2004

ADOBE SYSTEMS INCORPORATED
Corporate Headquarters
345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000
<table>
<thead>
<tr>
<th>Rev #</th>
<th>Date</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>19 Oct 2002</td>
<td>Ian Paterson</td>
<td>Created initial draft including the use-cases and scenarios</td>
</tr>
<tr>
<td>0.2</td>
<td>24 Oct 2002</td>
<td>Ian Paterson</td>
<td>Rolled in detail about the scenarios catered for in ref implementation and more detail on the model</td>
</tr>
<tr>
<td>0.3</td>
<td>1 Nov 2002</td>
<td>Ian Paterson</td>
<td>Tidied up, added UML diagrams and added detail on ref impl.</td>
</tr>
<tr>
<td>0.4</td>
<td>4 Nov 2002</td>
<td>Ian Paterson</td>
<td>Round of edits to finalise document.</td>
</tr>
<tr>
<td>0.5</td>
<td>18 Nov 2002</td>
<td>Ian Paterson</td>
<td>Rolled in Seoras’ comments, regenerated all UML diagrams and described the simplified architecture.</td>
</tr>
<tr>
<td>0.6</td>
<td>18 Dec 2002</td>
<td>Ian Paterson</td>
<td>Integrated Paul’s review comments.</td>
</tr>
<tr>
<td>0.7</td>
<td>16 Dec 2003</td>
<td>Jane Bond</td>
<td>Update to InDesign CS</td>
</tr>
<tr>
<td>0.8</td>
<td>27 Feb 2004</td>
<td>Ian Paterson</td>
<td>Review updates and re-organise slightly.</td>
</tr>
</tbody>
</table>
## Contents

Abstract .................................................................................................................. 7
Terminology and definitions .................................................................................. 7
Introduction ........................................................................................................... 8
  Industry background ............................................................................................ 8
  InDesign CS support for locking ....................................................................... 9
  Functional requirements .................................................................................... 9
Use cases ............................................................................................................... 10
  Creating access rights in a multi-user document ........................................... 12
  Configuring the client workspace/ logging-in ............................................... 12
  Checking out document-layers ........................................................................ 13
  Checking-in a document layer ........................................................................ 13
Design and architecture for a MUDA system ..................................................... 13
  Design overview ............................................................................................... 14
  InDesign CS document-object model and document layers ....................... 14
  Re-using API implementations of persistent interfaces to store data ........ 14
  Document-layer metadata and observer ....................................................... 15
  Document metadata and document-observers ............................................. 16
  Command facade ............................................................................................. 16
  Factoring into model and user-interface plug-ins ........................................ 17
  Session state ..................................................................................................... 17
  Checking out and creating a replica ............................................................. 17
  Responding to document signals during the opening of a replica .............. 18
  Checking in changes to a document layer ..................................................... 19
Quick-start guide to the minimal implementation ............................................. 19
  Administration functions ................................................................................ 19
  Setting client workspace and authenticating ............................................ 20
  Checking out a document-layer from a shared-master ................................ 20
  Editing the shared-master ............................................................................. 22
  Checking in changes to a document-layer ................................................... 23
Limitations of the minimal implementation and extensions .............................. 23
  Lack of scalability of the document layers approach .................................. 23
  Being notified when others change the shared layout .................................. 23
  Client-server design ....................................................................................... 23
  Overriding the normal open action ............................................................ 24
  "Set active layer" command followed by tool selection ............................. 24
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other approaches to multi-user access</td>
<td>25</td>
</tr>
<tr>
<td>Other locking schemes</td>
<td>25</td>
</tr>
<tr>
<td>Round-tripping of layout in a custom format</td>
<td>25</td>
</tr>
<tr>
<td>Package for GoLive and XML features</td>
<td>26</td>
</tr>
<tr>
<td>Appendix</td>
<td>28</td>
</tr>
<tr>
<td>Persistent implementations for re-use</td>
<td>28</td>
</tr>
<tr>
<td>Protection of resources and access</td>
<td>28</td>
</tr>
<tr>
<td>Web Distributed authoring and versioning (WebDAV)</td>
<td>29</td>
</tr>
<tr>
<td>References</td>
<td>29</td>
</tr>
</tbody>
</table>
Abstract

This document describes a minimal solution to the multi-user document access (MUDA) problem within InDesign CS; allowing designers to work on different parts of the same InDesign CS document at the same time. It is not intended as a scalable solution that could deal with many end-users; rather, it shows how to capitalise on existing architecture in the InDesign CS document object model to build a minimal solution that may be of value in some situations. The minimal solution is based on document layers within InDesign CS documents.

This document illustrates how to build a system where multiple users can appear to be opening the same shared-master document, checking out for edit parts of the document that have been assigned to them, then checking-in their pieces of work into the shared-master document. The system works by enforcing mutually exclusive access on document-layers in the shared-master document.

This document considers briefly the limitations of locking within the InDesign CS API. There are locking-related interfaces in the InDesign CS public API (ILockPosition, ITextLockData) as well as commands to lock document layers, which prevent page items and guides on those layers being selected by an end-user. The strengths and weaknesses of approaches that take into account existing mechanisms within the InDesign CS API are discussed very briefly. Other approaches based on feature-sets new to InDesign CS are also discussed.

Terminology and definitions

- access right; the ability to execute an operation on an object. For the minimal multi-user document access system described here, access rights will be specified for operations on document layers. See “Protection of resources and access” on page 28 for a slightly more formal definition of the concept.

- client workspace; a folder on a client machine where local copies or replicas of shared-master documents will be kept.

- domain; a user, process or procedure for which access rights might be specified. For this investigation domains will consist of end-users or possibly groups of end-users.

- merging; integrating changes to content in a portion of a replica document into the shared-master document.

- page independence assumption; that a document does not contain more than one page to minimise cross-page dependencies. A newspaper workflow that included InDesign CS
Introduction

This section introduces the problem of multi-user document access in an industry context. It identifies the need for mechanisms to support synchronisation of the activities of designers working within InDesign CS to build a multi-user document access system. Synchronisation of activities of several designers working on the same InDesign CS document is the main problem; how to co-ordinate the activities of a team so that they do not corrupt each other’s work.

A further objective is to define a minimal access rights model for a newspaper publishing environment; this requires us to identify what the resources are that we need to protect, what kinds of protection are required for these resources, and how fine-grained the access domains for these resources should be.

An additional objective is to define the use cases that a minimal multi-user document access system should satisfy. These will include configuring a shared-master InDesign CS document, checking out a part of the shared-master document for edit as a local replica, and integrating the changes made on a local replica into the shared-master document.

Industry background

Newspapers do not typically work with spreads containing more than one page and pages very often have more than one article on them. In addition, when using InDesign CS, each page would be represented as a separate document. This concept of page independence allows for efficiencies in the workflow by reducing cross-page bottlenecks. It is conventional in the newspaper industry to factor content into articles which can span multiple pages.
If a newspaper is using InCopy CS to edit the stories and style the text, then several editors can work on the stories on one page. The designer working on the layout can see that the stories have been updated because the Links panel notifies when a linked story has been updated and they can refresh the InDesign CS document with the new copy. This works well with a single designer; however, several designers may wish to work simultaneously on the layout of stories contained within the same InDesign CS document. This would not be possible with InDesign CS at the time of writing without additional effort; an InDesign CS document can only be opened by one end-user for read and write access; this is enforced at the operating system level.

One solution, borrowed from the domain of WebDAV and source code control systems, is to open a replica of a shared file into a client workspace. A designer could make changes to the local replica without necessarily locking out others from working on other portions of the shared-master document, as long as it was clear what part of the shared-master document they were working on. When they were done, they would merge their changes back into the shared-master document.

**InDesign CS support for locking**

This section describes how InDesign CS provides locking-related features to end-users. It also touches briefly on some of the API related to locking, which are discussed later in this report in more detail. Locking within InDesign CS consists of the following:

1. **The page items and/or guides on document layers can be locked through the user-interface of the Layers panel, or by processing a command of type kLockLayerCmdBoss (content) or kLockGuideLayerCmdBoss (guides).** This has the effect of preventing selection of objects on the locked document-layer.

2. **Page item position can be locked through the user-interface, or by processing a command of type kSetLockPositionCmdBoss to change the data stored on the ILockPosition interface of a page item.**

3. **If the InCopy CS plug-in suite for InDesign CS is present, then the text content of stories can be locked through a command of type kSetStoryLockStateCmdBoss.** This command changes the state of the ITextLockData interface aggregated on kTextStoryBoss. Locking is applied to the story content (represented by kTextStoryBoss) rather than the layout (represented by multi-column items, kMultiColumnItemBoss in kSplineItemBoss containers). The ITextLockData interface on kTextStoryBoss is available in both InDesign CS and InCopy CS.

This gives two initial choices that could be used to build a multi-user document access system where the prime requirement is support multiple designers working on layout within a single InDesign CS document: document layers (class kDocumentLayerBoss), or individual page items (ILockPosition interface).

**Functional requirements**

Systems integrators were polled for what they considered to be the main functional requirements of a multi-user document access system. They came up with a list which included the following:
1. Ability for a designer to design/define a page that contains placeholder or live content frames and assign read/write access to those frames for another designer to work on while the original designer also works on the document at the same time.

2. Allowing object locking including all sub-trees.

3. Ability to send notification updates to other document users when work has been saved by a specific user of the document.

4. Include a user-interface to allow the assigning/releasing of page items.

5. Provide a dual view of the document that allows the document user to see only their assigned content frames and design versus the entire page.

6. Include Read/Write icons for content frames and page items.

Of these requirements, only the first and second are considered in the present case study. The third is an important requirement as it makes explicit the co-operative nature of a multi-user document access system. The other requirements either require writing user-interface code or are discussed in “Limitations of the minimal implementation and extensions” on page 23.

---

Use cases

Use cases express how end-users derive benefit from a system. They represent the interactions between agents or actors and the system. Representing use cases provides a powerful means of determining the functional requirements of a system, rather than trying to conjure up a list of functional requirements in an ad-hoc way.

The system in the current investigation is a “multi-user document access system” (MUDA) of unspecified design; however, an assumption is made that documents have been stratified into document layers, each of which perhaps represents an article or piece of work for a single designer. The actors considered are as follows:

1. Administrator: e.g. a senior editor, responsible for setting up the shared documents which would be worked on by a team.

2. A designer working on a layout of one or more layers. Each designer may be responsible for laying out one or more articles.

Note that we do not consider sub-editors who might be responsible for the copy associated with articles in the publication. The rationale for this is that the current investigation focuses more on enabling multiple designers to edit layout of a single InDesign CS document. InCopy CS does an admirable job of letting multiple contributors edit the content of and style text of articles displayed within a single InDesign CS document.
The diagram shows some use cases and roles in a multi-user document access system. The administrator sets up the document for multiple users to access and assigns protection rights. The designer represents a typical end-user of the system, whose main objective is to be able to edit a layout that may be opened by another end-user. The
Unfilled use cases are outside the scope of the minimal implementation that we describe in this document.

A diagram containing the use cases that the system should support is shown in Figure 1. Note that it is assumed that the budgeting of who does what would be determined by a section or senior editor, who would operate as an administrator to set up access-lists in the shared-master document. Some key use cases are considered in more detail below.

Creating access rights in a multi-user document

**Context and preconditions**

This would be performed by a senior editor or section editor; it would have to be done before designers can work on a shared layout. Assume that there is already a standard template with some initial budgeting done for the layout. The portions of the layout that would be edited by different individuals would have to be held on separate document-layers within this model.

**Flow of events**

1. The administrator would assign access rights to the end-users over the document layers. For each document-layer that could be edited, there would be a corresponding access list specifying end-users who could check-out that document-layer for editing.

2. The newly created shared-master document would be saved.

**Caveat.** This use-case was not directly supported by the present case-study. Although an example file is provided that has access-rights associated with document-layers, these access-rights were created by test code and no user interface was delivered to support this use case in our samples. This example file can be found in the folder examplefiles within the folder `<sdk>/source/sdksamples/multiuser`.

Configuring the client workspace/ logging-in

**Context and preconditions**

An end-user would have to specify where they wanted to create replicas of the shared-master document, and in each editing session might be required to identify themselves to the system.

**Flow of events**

1. The end-user would select a folder on their client workstation where replicas of the shared-master documents could be created. The system could choose to persist this information across sessions.

2. The end-user would confirm their identity to the system by making some gesture such as a selection in a drop-down list or authenticating with a user-name and password combination. The system could choose to persist this information across sessions to require an individual to identify themselves only once, or could require authentication on each editing session (i.e. each time InDesign CS is re-started).
Caveat. This use-case was not directly supported by the present case-study. Although example-files are provided which have access rights associated with document layers, these access rights were created by test code and no user interface was delivered to support this use case.

**Checking out document-layers**

**Context and preconditions**
This would be performed by an end-user who was working on the layout of one specific document portion.

**Flow of events**
1. The end-user would choose a shared-master document.
2. The system would display a list of the document-layers in this shared-master document along with information about whether a document-layer was exclusively locked by other end-users.
3. The end-user would choose one of the document-layers that was not exclusively locked by another end-user and check it out for editing.
4. The system would note that this document-layer had been checked-out exclusively by this particular end-user and would disallow further attempts to check this document-layer out for editing until it had been checked-back in or the edit abandoned.

**Checking-in a document layer**

**Context and preconditions**
This would happen when a designer had finished their edits on the layout of a document-layer that they had checked out for editing.

**Flow of events**
1. The end-user would make a check-in gesture to inform the system that their changes should be integrated into the shared-master document.
2. The system would merge the changed document-layer in their client-replica into the shared-master document.
3. The document-layer would be unlocked in the shared-master document and become available for other end-users to edit.

**Design and architecture for a MUDA system**

This section provides some architecture that can be used to implement a multi-user document access system within InDesign CS. For the minimal implementation, multiple users were able to edit different document layers (kDocumentLayerBoss); document-layers are one of the few components of the document object model that have some lock-related capability. Metadata
was added to kDocumentLayerBoss to provide an exclusive-lock mechanism that allowed the
MUDA system to mediate access to document-layers in a shared-master document.

Design overview

This section provides a high-level overview of the overall system design and attempts to discuss
how the design caters for the use case discussed previously.

The minimal implementation of a MUDA system described in this document is based on
document layers. If there are several end-users who wish to edit a single document, they cannot
simultaneously open the same InDesign CS document, which we termed as the shared-master
document. What they can do, with some coding effort on our part, is to open a replica of the
shared-master document into their own client workspace.

As in the “Checking out document-layers” use case, the end-user would choose a document
layer in the shared-master document, obtain an exclusive lock on this layer, and then start to
edit the corresponding layer in their local replica of the shared-master. The exclusive lock on a
particular document layer in the shared-master would be specified by metadata that we add to
the shared-master document, rather than being some intrinsic feature of the InDesign CS API.

Once the exclusive lock was obtained on a particular document-layer in the shared-master, no
other end-user would be able to edit this layer until the end-user who had it locked had
checked-in, thus revoking the lock.

The following sections describes solutions to the implicit problems in the above high-level
description, such as:

● how to work with document layers to support the task of multi-user access,
● how one might specify metadata to support the locking protocol,
● how to create replica documents using the InDesign CS API,
● how to open the shared-master document in a headless mode,
● what session state needs to be maintained,
● how to merge back in edited content in the shared-master document.

InDesign CS document-object model and document layers

The document object model (DOM) for an InDesign CS document is relatively complex. The
portion of the InDesign CS DOM that is required for the present investigation relates to
document-layers and spread-layers. This is considered in detail in the Programming Guide
chapter in "Document structure".

Re-using API implementations of persistent interfaces to store data

Frequently it is necessary to persist data of various types within some document object, such as
the document (kDocBoss), document workspace (kDocWorkspaceBoss) or session workspace
(kWorkspaceBoss) or on page items. Whilst it is possible to create a persistent interface and
implement the ReadWrite method yourself, as in the SDK sample BasicPersistentList, there are
some persistent implementations already available in the API which can be re-used with
cautions.
If you are re-using persistent implementations (see Table 1) then you should always use an ImplementationAlias resource statement to be on the safe side, as persistent implementation IDs have to be unique within a boss class. An ImplementationAlias with an ID from your own unique plug-in ID space should enforce this constraint. The example below shows an ImplementationAlias that associates the persistent API implementation used to store a SysFile (kSysFileDataImpl) with an implementation ID that is unique to the MultiUser plug-in. This means that the implementation ID kMultiUserRemotePathImpl can be added to boss classes even if these already aggregate the implementation with ID kSysFileDataImpl.

```plaintext
resource ImplementationAlias(1) {
    {kMultiUserRemotePathImpl, kSysFileDataImpl,}
};
```

**Document-layer metadata and observer**

This section defines the metadata that is added to document layers (kDocumentLayerBoss) to specify access rights, as well as provide the capability of exclusive locking by a single end-user. The key concept is to use the state in a shared-master document to synchronise the activities of the client workflow plug-ins. This eliminates the need to have a separate server component to synchronise the activities of the client workflow plug-ins.

To enable mutually-exclusive access to the document-layers, one can associated an access list with each document layer. It would be desirable if the MUDA system could create an exclusive lock on a document layer, perhaps through a combination of some private data and logic.

In addition it is convenient to represent the identity of the individual who has the exclusive-lock on a document part, to enable other users within the system to determine who might have locked a part that is critical in their workflow, and which they need unlocked. This would let any other end-users determine who has a particular part checked out for edit.

To represent the access rights on a part, one can represent a list of names. This access list can be the names of individual end-users. It might consist of groups to which the users are assigned in a more elaborate workflow.

For multiple users to be able to share a layout in a single shared-master document, an administrator user needs to be able to set up the access rights on the document-layers of the document that they want people to be able to checkout exclusively. This would entail them segregating content into document-layers and then creating access lists for the document-layers. No user-interface was delivered in the minimal implementation for this capability, although the API is there in the MultiUser plug-in to create access lists for document-layers.

In the absence of a server component, the client plug-ins that implement the MUDA system can still co-operate when the shared-master document represents state that the individual client plug-ins should care about. For instance, an individual designer should not be able to open for editing a part of the shared-master document if this part is already being edited by another user. Metadata is added to the part of interest (kDocumentLayerBoss) to enable specialisation of the normal behaviour; for instance, to open a replica of a document with all the parts locked except for those for which the end-user has the appropriate access rights.
Additionally an end-user is prevented from inadvertently unlocking any parts (layers) for which they do not have the correct access rights.

Note that although the access-list represents individual end-users, it would be appropriate in a more robust implementation to assign these users to groups to represent less redundant data at the level of each document part.

There is a small set of commands delivered by the MultiUser plug-in to change these data interfaces on each document part, as well as utility methods in the IMultiUserCommands interface to make queries on a particular document, for instance the exclusive-lock state or the access-list. Note that the exclusive lock state is not the same as the layer-lock state when a file is opened; it is an exclusive lock that can only be obtained by one end-user at a time and it is represented by metadata added by the MultiUser plug-in to document layers (kDocumentLayerBoss) in the shared-master document.

Document metadata and document-observers

Metadata was added to the document workspace (kDocWorkspaceBoss) and populated for replica documents opened as part of a check-out. This metadata was essential to the operation of the MultiUser plug-in, since it specified what shared-master document the replica document had been cloned from and what document-layer in the shared-master document had been checked out for editing.

A document-observer was added to the document boss class (kDocBoss) to receive notifications about document-level events; for instance, document-layers being added or deleted, and notification about when a document is fully opened.

The path to the shared-master document is stored in the document workspace (kDocWorkspaceBoss) of any replica document that gets opened. This enables the client plug-in to find the original shared-master document when the changes to the replica are to be checked-in.

The name of the document-layer that has been checked out from the master document is maintained in the document workspace as well. This enables the correct document-layer in the shared-master document to be updated when the check-in is performed.

An observer interface is added to the class that represents the document (kDocBoss) to allow notification about events such as the document becoming fully opened, or a new layer is added, or existing layers are deleted.

Command facade

A useful pattern to shield client code from complexity is to use a facade. A facade is a pattern from [Gamma et al, 1995]; it “provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use”. In the case of the current implementation, there are several interfaces and helper classes that maintain state which the client plug-in needs to change and/or be aware of. Rather than having the client code (in the MultiUserUI plug-in) manipulate the helpers (e.g. MultiUserHelper) and interfaces (e.g. IMultiUserSessionState) directly, the interface IMultiUserCommands provides a straightforward to use API that hides the detail of the abstractions responsible for managing the state or performing the operations requested.
Factoring into model and user-interface plug-ins

The multi-user document access (MUDA) minimal implementation is factored into two plug-ins; one provides the user interface (MultiUserUI) and is explicitly dependent on the plug-in MultiUser, which provides the 'model'.

The MultiUserUI plug-in exercises the capabilities of the model plug-in through the IMultiUserCommands facade interface.

This factoring into model and UI plug-ins is common in the application itself, with the user interface plug-in dependent on the model plug-in (but not vice versa). For instance, the XML subsystem delivers its main capability in XML.pln, with the XMediaUI plug-in providing a user-interface to this subsystem.

Session state

There is some state that is maintained in the session boss object (kSessionBoss, referred to by gSession) which is used by the MultiUser plug-in to co-ordinate its activities. An interface of type IMultiUserSessionState is added to the session boss class (kSessionBoss) to maintain this state. For instance, the client workspace for the end-user in the session is stored in this interface, along with the identity of the end-user.

Checking out and creating a replica

In a source code control system or a WebDAV-based system, the end-user or client opens a replica of a document by a check-out gesture and makes some local edits on the replica. A replica of the master document is created in the client workspace. Once the local edits are complete, the end-user makes a check-in gesture.

The model for storing the shared-master documents in the minimal implementation is to have the shared documents on a file-server. This avoided the need to have to write a server component to mediate access to these documents. However a real-world system would invariably involve a server component that acted upon messages sent from a client (e.g. over a TCP/IP socket) and might for example enable the client to open a stream over the socket to copy data from the server machine over the network to create a local replica.

The end-user would select the shared-master document to open; we can use the open manager service (IOpenManager) to create a file-selection dialog. Once the end-user has selected a shared-master document, it can be copied into the client workspace and then opened.

Since the model involved copying files from one path in the local area network to another, it was sufficient to use the methods provided by ICoreFilename to perform the necessary copy. To use this interface it is only necessary to create an instance of kCoreFilenameBoss and initialise it with the path that a file is to be copied from/to.

Once the file is copied into the client workspace, then you can process a command of type kOpenFileWithWindowCmdBoss to open the document with a view.
Responding to document signals during the opening of a replica

The command to open a document sends a set of signals at determinate points whilst a document is opened. Client code can listen for these signals by adding responder boss classes in its plug-in.

The document should not be changed during an open operation; to add metadata to objects in a document, one has to wait until the document is fully opened. A document observer can be attached when the signal is received that the document is opening, and this observer can listen for the post-notification about the command to open the document (kOpenDocCmdBoss).

The class MultiUserDocResponder listens for the following signals and acts upon them as follows:

- **kDuringOpenDocSignalResponderService**: an observer is attached to the document, i.e. a document observer.
- **kBeforeCloseDocSignalResponderService**: the document observer is detached.
- **kAfterCloseDocSignalResponderService**: the multi-user feature is turned ‘off’ until a further custom-open is performed. Turning the feature off means that unless a checkout operation is performed, the newly opening document is not decorated with MultiUser specific metadata, which may not be present in a valid state in the session.

Recall that an observer interface was added in to the class that represents a document (kDocBoss) to allow notification about events such as the document becoming fully opened. When we attach an observer to the newly-opened replica document, we register to be notified along IID_IDOCUMENTLIST. This will result in an IObserver:Update message when the document has been fully opened, as the command to open a document (kOpenDocCmdBoss) will notify along this protocol with its own ClassID once the command to open the document has completed. The observer adds metadata to the replica document (path to the shared-master document and the document-layer checked out for editing). A plug-in at this point can also change the document file handler on the replica document, if it is desired to customise the normal Save action to perform the equivalent of a check-in operation. However, this was not performed as part of the minimal implementation.

We also want to attach an observer to each of the document layers (kDocumentLayerBoss) in an opened document. It will observe changes to the document layer lock state. We can do this once we have received the message with ClassID of kLockLayerCmdBoss along with protocol IID_IDOCUMENTLAYER. When lock state changes are made by something other than the MUDA system, to something other than the part we have check out for editing, we revert the lock state.

The code in the MultiUser plug-in has an observer that is added into the document class (kDocBoss). This observer is attached by the responder on the signal kDuringOpenDocSignalResponderService, and it attaches itself to the subject (ISubject) interface of the document (kDocBoss) on which it is aggregated. The implementation MultiUserDocObserver then creates some metadata in the document workspace (kDocWorkspaceBoss) using a command of type kMultiUserSetDocMetaCmdBoss. It acquires this metadata from the state stored on the session in the interface IMultiUserSessionState.

The command to close a document sends out notification messages, in additional to the responder signals, which provide another opportunity for client code to clean-up before a
document is closing or immediately after it is fully closed. The command type of kCloseDocCmdBoss notifies document observers before the document is closed. The same command post-notifies observers of the document-list observers that the document has gone away. This observer would have to have the document-list as the subject, rather than the document, which would of course have disappeared as a valid object by the time the notification about it being fully closed has come through. The command kCloseDocCmdBoss notifies observers of the document-list after the document is closed with ClassID kDocumentClosedMsg, along protocol IID_IDOCUMENTLIST.

Checking in changes to a document layer

When a designer has changed the portion of the layout represented in the document layer that they checked out for editing and wish to check it in, the main task is to update the document layer (kDocumentLayerBoss) in the shared-master document with the changed content. Below is the event sequence we implemented in MultiUser sample:

1. the MultiUser plug-in opens the shared master document as a headless document,
2. it deletes the document-layer in the shared-master document
3. it creates a new document-layer based on the edited layer in the replica
4. it copies the old metadata from the deleted layer to the new layer
5. it makes sure that the document-layer is no longer marked as exclusively locked out for editing.
6. it closes the shared master document.

When the replica document itself is closed, the remaining clean-up is to make sure that the observers on the document layers are detached correctly and that the document observer itself stops listening for document-level notifications. This is achieved by listening for a signal sent before the replica document is closed (kBeforeCloseDocSignalResponderService) with a document-responder, which then sends an IObserver::AutoDetach message to the document observer. This document observer in turn sends IObserver::AutoDetach messages to tell the document-layer observers to stop listening for document-layer-level changes.

Quick-start guide to the minimal implementation

The MultiUser and MultiUserUI plug-ins provide a minimal implementation of a multi-user document access system. This system does not provide administrator capability through the user-interface, although the API delivered by the MultiUser plug-in does, and this API was used to set up a sample shared document through test code.

Administration functions

There are no administrator-level functions accessible through the GUI; the example files have access rights for document-layers created through the test code (MultiUserUnitTest), which
has some hard-wired paths (so the tests will fail unless the paths are changed to make sense for your system).

**Setting client workspace and authenticating**

You have to set the client workspace before you can either log-in or open a shared-master document; this is some folder on your local machine, so that the MUDA system knows where you want it to save the replica document. The concept is identical to that of a WebDAV server or a source code control system, where you have to specify a client workspace folder for local copies of server files. The client workspace can be any folder on your local machine.

*FIGURE 2 Specifying the folder for the client workspace*

This screenshot shows the MultiUser user-interface before the client workspace has been chosen. Note that no other menu items are enabled, because the system does not know who the user is or where local replicas of the shared files should be kept.

A more realistic implementation would persist this setting (e.g. see the SDK samples named XDocBookWorkflow, CustomPrefs, BasicPersistInterface); the minimal implementation requires the end-user to set this from session to session.

Once you have specified the client workspace, you should find that you are able to log-in. You should provide one of the names indicated on the dialog; these already have some form of access rights to the sample document (shared.indd) provided in the examplefiles folder in `<sdk>/source/sdksamples/multiuser`.

*FIGURE 3 User interface for identifying user*

This screenshot shows the user interface to identify the user of the system. The user name selected should be one of the choices indicated at the bottom of the screen. In a less-minimal implementation, the widget to enter the name would be something like a drop-down list.

**Checking out a document-layer from a shared-master**

Once you have authenticated (“logged-in”), the menu item to allow a document-layer to be checked out for edit should become enabled.
This screenshot shows the user interface before checking out a part of the shared document. Since nothing is checked out, the Check-in menu item is disabled.

At this point you have provided enough information for the MUDA system to open a document, as it knows your identity and where you want to copy the replica of the shared-master document to. Choose the file named “shared.indd” in the examplefiles folder in <sdk>/source/sdksamples/multiuser.

This screenshot shows the file selection dialog that appears initially when you execute the menu item to check out a part of a shared document; you are initially prompted to locate the shared file whose part you are interested in, located on a file server on the local network.

The first thing that you would see on choosing the document named “shared.indd” is a dialog with the different document layers, with some information about their state, and access-lists. At this point you should choose the document layer that you want to edit. If you choose a layer
that your user-profile doesn’t have access to, then in debug mode you’ll see an assert and in both debug/release a warning that you don’t have access rights to do what you want to do.

**FIGURE 6** User interface showing parts available and locked

![User Interface Screenshot]

This screenshot shows...

If you have chosen a document layer that you have access right, then you should see a document (replica, in your client workspace) opening and an alert to indicate your success.

**FIGURE 7** Notification on successful check-out

![Notification Screenshot]

This screenshot appears on successfully checking out a part (document layer) within the shared master document for edit. There is a counterpart notification that appears when trying to check out a part for which the user does not have appropriate access rights.

**Editing the shared-master**

Inspect the layer-panel. Note how you can only unlock the layer that you checked out for editing. The rule is that the layer you checked out for editing will be the only one changed in the shared-master when you save the replica document.
Checking in changes to a document-layer

A menu item enables the edited layer in the replica to be checked in. When you are done with editing the layer that you can edit in the replica, you can check in the edited document-layer. When you execute this menu item, you should be notified of your success.

FIGURE 8 Notification on successful check-in

This screenshot shows the notification that appears when the part (document layer) checked out for edit has been successfully checked back in.

Limitations of the minimal implementation and extensions

Lack of scalability of the document layers approach

A mechanism that is based around document layers has some utility when the numbers of layers can be kept small. However for more than a handful of collaborating designers, the document-layer approach would rapidly become unwieldy.

Being notified when others change the shared layout

A desirable feature of a multi-user document access system is notification about changes to other parts of the layout. The minimal implementation described in this document did not attempt to perform this kind of dynamic update. An idle task that polled perhaps once a minute for changes to the state of the shared-master document that one was editing could be used to determine if updating of parts of the replica were required (see the SDK sample named PanelTreeView for an idle task example). To integrate the changed content in the replica to bring its state into line with the shared master, code similar to that to integrate the changed content into the shared master would be required (see MultiUserHelper::CheckInPart). Additional metadata such as a timestamp indicating the last time a document-layer had been edited would be useful to allow the state of the document-layers in the replica to be synchronised with the shared-master document state.

Client-server design

Performance in the minimal implementation is relatively poor, since to change a fairly small amount of persistent data in the shared-master document, the document is opened, changed and saved, which has a fair amount of overhead. It would be preferable to have a server component that persists the state required to support the client workflow plug-ins outside of
an InDesign CS document. For instance, it is straightforward to write a server application in Java that listens on a socket for connections, and maintains state in some XML document or database, say.

**Overriding the normal open action**

Although this was *not* performed as part of the minimal implementation in the interests of clarity, a common question is how to override the normal menu item behaviour. For instance, files are opened with the gesture to execute the menu item “File >> Open” or a keyboard shortcut (e.g. “Ctrl + O”).

The normal gesture to Open a file could be overridden for multi-user document access; this would prevent a shared-master document being opened normally without going through a check-out protocol. One way to do this is to use an action filter (IActionFilter) to override the normal action associated with a particular ActionID, which is associated with a menu-item. However for the purposes of the minimal implementation, we did not override the normal “Open” menu item. Instead, we added a check out menu item in the MultiUserUI implementation. In a pure multi-user environment, it might be desirable to turn off the normal “Open” item with an action filter. Your implementation of IActionComponent would handle the incoming request for an ActionID (e.g. kOpenActionID) to be processed. It can also selectively disable some menu items (and the associated shortcuts that map to a given ActionID) by specifying that these ActionID’s should be handled by your action component, and then setting the type to custom enabling (kCustomEnabling), disabling in IActionComponent::UpdateActionStates.

**“Set active layer” command followed by tool selection**

The end-user can create objects in the layers that are apparently locked through the user interface of the Layers panel if they select a tool from the Toolbox panel. This is academic in a sense because only the content in the document layer that was checked out for edit should be saved back to the shared-master document.

However, it might be that you would wish to prevent this from happening. This means that you would have to reverse the effect of the command that sets the active layer. If you inspect the documentation for the command of type kSetActiveLayerCmdBoss, note that it notifies along IID_ILAYOUTCONTROLDATA with a subject (ISubject) that is the layout widget; an object of type kLayoutWidgetBoss, aggregating the interface ILayoutControlData. The trickiest part of this process is AutoAttach’ing your own observer; you could observe changes in the active context (ISession::GetActiveContext), and attach to the current layout widget boss object, detaching from the old layout widget if you were still attached.

The command itself lets you determine the last active layer and the new active layer; you could therefore register to be notified when this command had been processed, and set the active layer back to be the only layer that the end-user was supposed to be editing. The old layer is stored in the IUIDData interface on the command of type kSetActiveLayerCmdBoss.

**EXAMPLE 9 Finding last active layer and newly active layer**

```cpp
// What was the old active layer?
InterfacePtr<IUIDData> oldLayerUID((IPMUnknown *)changedBy, IID_IUIDDATA);
```
Other approaches to multi-user access

This document has described a minimal solution to enable multi-user document access that is based around document layers (kDocumentLayerBoss). Benefits and limitations of this approach and two accompanying sample plug-ins are discussed. Note that this minimal implementation was developed against InDesign 2. With InDesign CS, there are some other possible approaches that could be tried. This section is more speculative but identifies avenues that are promising for developers who have a good understanding of XML.

Other locking schemes

If the notion of a part is refined to be at the level of a page item, then once again we consider adding metadata to the document parts, such as the access rights associated with them and the current locking state and identity of the client who has locked the part. Having an access list per page item that represents all the data would not be ideal; therefore you might only wish to store a UIDRef per page item, and persist the access lists in the document workspace.

Locking a layer means that nothing in the layer can be selected; there is an equivalent notion of locking on page items to mean that they cannot be selected in a layout view. The interface ILockPosition is exposed on guides (kGuideItemBoss) and page items that are selectable (descendants of kDrawablePageItemBoss). This interface represents a flag (ILockPosition::kLock or ILockPosition::kUnlock) to say whether a page item can be selected. Its state is changed by a command of type kSetLockPositionCmdBoss, taking an item list of page items to change the lock-state on. This command notifies document observers with ClassID of kSetLockPositionCmdBoss, along protocol IID_ILOCKPOSITION. Since there is no pre-notification, this means that there is no opportunity to cancel the command before it is processed, and you would have to reverse the state of the locking to reflect the policy given the access rights assigned to the page item.

On saving the document the MUDA system could update the shared master-document in headless mode by deleting the page items that were marked as open for edit by the client and replacing these page items by copying the page items from the replica, exactly as happens with the current minimal implementation based on layers.

Round-tripping of layout in a custom format

The SDK sample named CHMLFilter, which can be found in the folder <sdk>/source/sdksamples/chmlfilter, shows one approach to round-tripping a layout. This SDK sample has a custom import service, which imports text and graphics, and a custom export service, which exports the geometry of text frames and preserves geometry of graphic frames and the datalinks associated with them.
CHMLFilter would need to be extended to allow it to export and import information about the linking between text frames and datalinks for placed stories. With styled text held as InCopy stories (rather than in the chml files as at present), it would be a relatively small piece of work to extend this to support exporting a layout to a format that could be round-tripped and preserve the geometry, text linking and datalinks to stories and images.

XML-tagging could be used to identify the articles within the publication and this information preserved on export and restored on import. With the ability to identify different articles by the XML tags applied to the content items, it would be possible to create a merged CHML file from different designers. That is, to compute the merge, you would take only the layout information from tagged objects that were associated with the article that each designer was editing the layout for.

Package for GoLive and XML features

One requirement expressed by systems integrators is to be able to have a high-fidelity representation of an InDesign layout in a format such as XML, which could be relatively easily manipulated outside the application.

The application is already able to produce a high-quality XML representation of a layout via the Package for GoLive feature, which is described at length from a programmer’s viewpoint in the tech-note #10084. The file layout.xml that is produced by the Package for GoLive feature has a representation that could be used as the basis a multi-user document access system. There are also some other features in the XML subsystem that can be used to assist in an implementation; see the tech-note #10044 on XML productivity features in InDesign CS. You can use

If designers are trying to edit a shared layout consisting of graphic frames and text frames with images and stories held in external assets, then you could imagine the situation where you want to add some private data to graphic frames and text frames to let you control access rights to change their geometry and position. Rather than adding your own private data to document objects, a convenient way to do this is to tag the content items using the XML API, or through the user-interface as an end-user. For instance, if designer A is going to edit the layout for Article A (based on an InCopy story, say) then you might want to tag the story flowing through the text frames that they were going to be editing the layout of with the name of the article.
The diagram shows two stories which are tagged to identify them. A tagged content item is associated with an element in the logical tree-structure. The assets that belong to one article are tagged with the same tag.

Figure 10 shows a page from a document containing two stories, tagged to identify them. When the document is exported through the Package for GoLive feature, the geometry of each frame and the linking is represented in the layout.xml file. In addition, the identity of the element in the logical tree-structure associated with each text frame is preserved. The attribute aid:AssociatedXMLElement in the layout.xml file represents the element in the XML logical tree-structure associated with the content item. A fragment of the layout.xml is shown below.

<aid:TextFrame ... aid:AssociatedXMLElement="ro_dili2i5" ...>

You would need to merge the layout.xml files exported by multiple designers. You could do this outside InDesign CS, using any convenient XML toolkit, say from your own Java application.

The missing part of the jig-saw is that InDesign CS does not support round tripping of this information, and you would have to write some parsing code that allowed an end-user to open a layout based on the information in an XML file. This is not as difficult as it might sound, as you would be able to implement a custom SAX content handler to parse the XML using XML parser service (kXMLParserServiceBoss) built into InDesign CS. A description of the building block named SAX content handler can be found in the tech-note #10044 on XML productivity features in InDesign CS. You register to handle elements in the XML stream being imported, and your custom SAX content handler would have to handle the entire tree of elements in the layout.xml file and create a layout to the specification in the input XML. You might want to wrap this in a custom import service, as in the SDK sample at <sdk>/source/sdksamples/xdocbookworkflow. However, depending on how much of the information in the XML files you wanted to round-trip, there would still be a fair amount of work.

Once the layout had been opened, text frames and graphics frames created and content placed, then it would be possible to lock object positions; all object positions other than for objects in the article checked out for edit would be locked.
Appendix

This section provides additional background to support the investigation.

**Persistent implementations for re-use**

Note that the implementations listed in Table 1 come from required plug-ins or the application core and should be available to re-use without having to specify a plug-in dependency in your own ODFRez code.

**Table 1 Persistent implementations that can be re-used**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Interface type</th>
<th>Implementation ID</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool16</td>
<td>IBoolData</td>
<td>kPersistBoolDataImpl</td>
<td>defaults to kTrue</td>
</tr>
<tr>
<td>int32</td>
<td>IIntData</td>
<td>kPersistIntDataImpl</td>
<td>value of (-1)</td>
</tr>
<tr>
<td>PMString</td>
<td>IStringData</td>
<td>kPersistStringDataImpl</td>
<td>empty string</td>
</tr>
<tr>
<td>K2Vector&lt;PMString&gt;</td>
<td>IStringListData</td>
<td>kStringListDataImpl</td>
<td>empty vector</td>
</tr>
<tr>
<td>PMReal</td>
<td>IGraphicAttrRealNumber</td>
<td>kGraphicAttrRealNumberImpl</td>
<td>0.0</td>
</tr>
<tr>
<td>SysFile</td>
<td>ISysFileData</td>
<td>kSysFileDataImpl</td>
<td>N/A</td>
</tr>
<tr>
<td>UID</td>
<td>IPersistUIDRefData</td>
<td>kCPersistUIDRefDataImpl</td>
<td>kInvalidUID</td>
</tr>
<tr>
<td>UIDList</td>
<td>IPersistUIDListData</td>
<td>kPersistUIDListDataImpl</td>
<td>nil list</td>
</tr>
<tr>
<td>list of int32</td>
<td>IIntList</td>
<td>kPersistIntListImpl</td>
<td>empty vector</td>
</tr>
</tbody>
</table>

**Protection of resources and access**

Protection of resources can help prevent end-users corrupting each other’s work. The notion of protection of resources is well-understood in operating systems, where preventing misuse of resources is a critical feature. Informally, protection is expressed by the “need to know” principle. This may be formalised in terms of a protection domain, defining a set of objects and types of operations that may be invoked on these objects. Ability to execute an operation on an object is termed an access right. A domain is a collection of access rights <object-name, rights-set>. A domain within an operating system might be a user, process or procedure. In publishing, we may not be so concerned with process or procedure abstractions, but the concept of access rights for users (and by extension groups of users) becomes central.

The protection model for a system can be expressed in an access matrix, which specifies the access rights associated with all the objects in the system for all the valid domains. A domain could be an individual user or group of users. For instance, the group “designer” might have read-only access to the copy of stories. An individual designer might have write access to the
layout associated with a particular story; that is, they could move and resize text frames, create and link new frames, and so on.

It is necessary to establish and control access rights within a system, and this is a level of access in itself. For instance, in a newspaper context, a senior editor might assign access rights to portions of a document to different designers and sub-editors. It is assumed that an individual who can assign access privileges is a member of an administrator group. A multi-user document access system would be expected to expose different capabilities to normal end-users and administrator users.

**Web Distributed authoring and versioning (WebDAV)**

The WebDAV protocol from W3C (see http://www.webdav.org), an extension to HTTP, provides a framework for distributed authoring. It attempts to provide file-system features, such as asset-level locking, through Web protocols. Despite the name, the existing WebDAV standard does not support versioning per se.

There are planned extensions to WebDAV that should support versioning; for instance the DeltaV protocol adds versioning within the WebDAV model by extending the base WebDAV protocol.

**References**
