THE QUEEN'S AWARDS FOR ENTERPRISE 2001

BBC

R&D White Paper

WHP 110

April 2005

CICO: a web-deployable application for remote file access

S.J.E. Jolly

Research & Development
BRITISH BROADCASTING CORPORATION

BBC Research & Development
White Paper WHP 110

CICO: A Web-Deployable Application for Remote File Access

Stephen Jolly

Abstract

We set out the rationale for a web-deployable application for checking media files in and out of a repository in the context of an ongoing investigation into the use of web technologies in production environments. The use cases for such an application are listed, and a specification fulfilling those use cases is presented. "CICO" (Check-In/Check-Out), an application written to that specification is then documented, along with justification for the design decisions made as part of the development process. Possible future changes to and uses of the application are then discussed.

Additional key words: CICO, MDP, Media Dispatch, SOAP

©BBC 2005. All rights reserved.

White Papers are distributed freely on request.

Authorisation of the Chief Scientist is required for publication.

©BBC 2005. Except as provided below, no part of this document may be reproduced in any material form (including photocoping or storing it in any medium by electronic means) without the prior written permission of BBC Research & Development except in accordance with the provisions of the (UK) Copyright, Designs and Patents Act 1998.

The BBC grants permission to individuals and organisations to make copies of the entire document (including this copyright notice) for their own internal use. No copies of this document may be published, distributed or made available to third parties whether by paper, electronic or other means without the BBC's prior written permission. Where necessary, third parties should be directed to the relevant page on BBC's website at http://www.bbc.co.uk/rd/pubs/whp for a copy of this document.

BBC Research & Development
White Paper WHP 110

CICO: A Web-Deployable Application for Remote File Access

Stephen Jolly

Contents

©BBC 2005. All rights reserved.

BBC Research & Development
White Paper WHP 110

CICO: A Web-Deployable Application for Remote File Access

Stephen Jolly

1 Introduction

Among the projects currently being undertaken by the Desktop Production team at BBC Research & Development is that of "Hello World", a vision for an integrated collection of web-based tools and applications that will assist production teams with all aspects of the creation of a programme, from planning and brainstorming all the way through to delivery of the final media package. Hello World will take the team's existing work on file formats, digital capture, ingest, multi-channel audio, commodity servers and media exchange, and incorporate it into a web-based framework with a friendly user interface.

Enabled by the increasing proliferation of PC and internetworking technology, an increasing trend within the BBC is that of working from home. In the context of programme creation, this leads to a requirement that members of production teams should be capable of accessing the files associated with their project (including any media files) from remote computers and locations. In addition, exchange of media with third parties is a common requirement during the production process, be they organisations partnering with the BBC, or contracted parties such as post-production facilities.

2 Scenarios and Use Cases

There are a number of other example scenarios in which transfer of files in or out of a production workgroup has been envisaged:

- Transfer of a finished programme to regional sites for playout.
- Transfer of material from an archive for potential inclusion in a programme.
- Transfer of material with possible alternate uses to an archive.
- Transfer of material to a technical service (eg colour-correction, subtitling) for processing.

Assuming that files associated with a project are stored on a dedicated server, or "content repository", these scenarios can be mapped onto four independent use cases:

- Check-In: A file and any associated metadata (collectively referred to as a "content object") are transferred from a client application to the repository for storage.
- Check-Out: A content object is retrieved from the repository by a client application. Where appropriate, the client application may "lock" the content object, preventing access to it by other clients for as long as the lock persists.
- Notify: The user is notified that a particular transfer has taken place.

- Remote Transfer: One repository checks a file in or out of another repository at the request of a user.

More details of the scenarios (including some more sophisticated examples) and the use cases they correspond to can be found in Appendix A.

In the context of Hello World, these scenarios and use cases can be addressed by the use of a web-based or web-deployable file transfer application. A high priority was placed on user-friendliness, and on minimising any requirement for pre-installed software on client machines. The application should not be tied to any particular file transfer protocol, and ideally, it would permit transfers to be negotiated using the Media Dispatch Protocol (MDP) [1] currently being developed by the Media Dispatch Group (MDG) of the Pro-MPEG Forum [2].

No existing file transfer application met these requirements, and it was decided to develop one independently, a Check-In/Check-Out (CICO) tool that would demonstrate the advantages of a web-deployable file transfer application in the scenarios described above, but would not necessarily implement all the features of a production-ready software utility.

3 Specification

The application was designed to fulfil the requirements of the following specification:

- To feature a user-friendly rich GUI interface.
- To support, at least, the Check-In and Check-Out use cases.
- To support file transfers via the MDP, and at least one other protocol (eg FTP).
- To suport content discovery by presenting the contents of the server to the user as a "tree" of folders containing files.
- To store appropriate descriptive metadata with each file.

A number of other desirable features were identified. Support for multiple users was one. This leads to a second: a common requirement when files are being shared between multiple users is that of multiple access. This leads either to file locking, in which only one user at a time is given permission to modify a file, or to version control, in which multiple versions of the same file may be saved on the server, with simultaneous changes resolved either manually or (where possible) by the software. In actuality, neither approach was fully implemented; new versions of an existing file are simply added to the server, and do not overwrite old versions.

4 Implementation

CICO is a client-server application written in Java. It uses the Apache Axis servlet and client library to implement an RPC interface via SOAP over HTTP. Internally, the code is divided into a number of packages:

- `bbc.cico.client` contains the classes required by a CICO client appliction.
- `bbc.cico.server` contains the classes required by the web-service interface to a CICO repository, or "server".

- `bbc.cico.core` contains classes that implement functionality common to both clients and servers.
- `bbc.cico.exception` contains exceptions that represent fault conditions within CICO clients and servers.
- `bbc.cico.clientgui` contains a GUI for the `client` classes, implemented in Java Swing.
- `bbc.cico.clientcli` contains a simple (check-in only) command-line interface (CLI) to the `client` classes.
- `bbc.cico.test` contains redundant code and test routines that do not form part of a working implementation.

The assistance of A. Sheikh and P. de Nier in the preparation of this object model is gratefully acknowledged.

5 Core Classes

The `bbc.cico.core` classes form the heart of the CICO object model. Typically an abstract `core` class is defined to represent a concept with project-wide scope, such as a "server" or a "folder". Classes in the `client` and `server` packages will extend these base classes appropriately.

5.1 Content Repositories

The most general possible representation of a content repository within the CICO architecture is the `bbc.cico.core.Server` abstract class. It defines two properties common to all classes that represent content repositories: a root folder in the repository's folder tree, and a "friendly name" by which the repository will be presented to the user. The `core` package also contains an abstract extension of this class: `RemoteServer`, which defines methods by which a CICO client can access a CICO server.

Not all RPC methods are contained within the `RemoteServer` class. It was decided to put relevant parts of the RPC interface into individual classes representing the different kinds of object stored on the server (ie content objects and folders). The `RemoteServer` class therefore only contains methods pertaining to the server itself, such as `getFolderTree()` (which returns a representation of the repository's folder tree) and `getFriendlyName()` (which returns the friendly name for the repository set by the user, or a client-dependent default if no user-set name is available – this will typically be the URI or domain name of the repository).

5.1.1 Folders

The actual return type of the `getFolderTree()` method is a `bbc.cico.core.Folder` object that represents the root folder of the tree. The `Folder` class is also abstract, and defines all the methods necessary to set and retrieve the contents and metadata of folders in the filesystem heirarchy of the repository[1]. Such methods include two similar functions: `getSubFolders()` (returning a `List` of `Folder` objects) and `getFolderContents()` (returning a `List` of `ContentObject`s); two methods

[1]Note that there is no requirement for the conceptual folders represented by `Folder` objects to represent actual directories in the local filesystem of the network server running the CICO servlet. The servlet may use any method to store filesystem metadata, such as a relational database or an object structure in memory.

with similar functionality are provided to avoid having to make the two return types inherit from a new superclass.

An important feature of the `Folder` object is that it implements all the methods defined by the `javax.swing.tree.MutableTreeNode` interface. This means that a `Folder` can be treated as a node of a `javax.swing.JTree` GUI tree component and display all the expected functionality, with subfolders displayed as child nodes, etc.

5.1.2 Content Objects

Files within a content repository and a set of metadata describing them are represented by instances of the `bbc.cico.core.ContentObject` class. The CICO object model represents a file as a `bbc.cico.core.ContentFile`, and specifies a 1:1 mapping between `ContentFile` objects and `ContentObject` instances.

Similarly to the `Folder` class, the `ContentObject` class implements `MutableTreeNode`. Content objects are not displayed in the folder tree in the current GUI client, however. The `ContentObject` class contains getter and setter methods for most of the metadata applicable to content objects:

- Name: the name by which the content object will be presented to the user.
- Description: a short textual description of the content object.
- Technical Description: a short textual description identifying any technical information about a content object that a user is likely to want to know (eg video/audio codec, aspect ratio, colour space).
- Copyright Status: a string that can take the values "Unknown", "Red", "Orange" and "Green"; the latter three meaning "no rights", "check rights" and "all rights" respectively.
- Related Object*, RelationshipDetails*: these metadata allow one content object to be identified as an exact copy (perhaps in a different medium) or an altered version of another content object.
- Lock Status*: whether or not a content object is "locked", and hence unavailable for access by other users.

Support for the types denoted by an asterisk is not fully implemented in the current version of the software. Read-only support is also provided for the "Check-In Time", "Creation Time" and "Modified Time" metadata types. Setting these timestamps is handled by the server. This metadata set was chosen for content objects because it is simple and yet general enough to be applicable to practically any kind of content that might be stored in a repository, and not merely to media files.

The types of metadata are enumerated by one of several classes in the object model based on the "Typesafe Enum" design pattern [3]: `bbc.cico.core.ContentObjectMetadataTypes`. The "Copyright Status" type is similarly enumerated by the `bbc.cico.core.CopyrightStatus` class.

The `bbc.cico.core.ContentFile` abstract class defines methods that act on a file stored in the content repository. These include self-explanatory functions such as `getFileSize()` and `getFilePath()`, and also a `getFileHash()` method that returns an MD5 hash of the file contents. Although subclasses of `ContentFile` must implement this last method, no use of it has been made in the current software. Developers should be aware that not only is it a processor-intensive method, but that implementations of it in different subclasses can vary considerably in terms of speed.

5.1.3 Local Repository Classes

The core classes contain one concrete implementation of each of the abstract classes described above, to represent directories in the local filesystem, and Content Objects for which the file is stored in the local filesystem and the metadata inside the application. These classes are called `LocalFolder`, `LocalContentObject` and `LocalContentFile`.

The hash function in LocalContentFile makes use of the "Fast MD5 Hash" library [4], which incorporates "fast" native-code implementations for Microsoft Windows and Linux running on x86 hardware; on other platforms a slower Java-only implementation is used.

5.2 Transferring files

A separate set of core classes describes the functionality required to perform file transfers. The first of these is concrete: `bbc.cico.core.TransferDescription`, which describes the files to be transferred, their source and destination and the identity of the user responsible for initiating the transfer. The class also contains details of the "transfer options" selected by the user: such things as scheduling information, and flags to indicate whether or not a given transfer must take place via an encrypted connection, etc.

A limitation of the current class model is that the `TransferOptionEnum` class that represents transfer options directly enumerates a subset of the options defined by the Media Dispatch Protocol. A more flexible implementation would allow different transfer types to implement their own subclasses of an abstract `TransferOption` class, defining options appropriate to the type of transfer in question.

The `TransferDescription` class stores information about the files to be transferred as a `List` of `FileTransfer` objects, each referencing two `ContentFile`s representing the transfer endpoints and an instance of the `bbc.cico.core.TransferDirection` enumeration class to identify whether the `FileTransfer` represents an upload or a download, from the point of view of the client.

As each `FileTransfer` is added to a `TransferDescription`, it is checked to ensure that the file transfer that the former represents is between the local machine and the remote repository identified by the latter. This behaviour must be modified if the CICO application is updated to implement the "Remote Transfer" use case (see Appendix A).

5.2.1 The Transfer Class

An instance of the `TransferDescription` class does not represent an actual transfer of files; that function is performed by subclasses of the `Transfer` abstract class. However, a `Transfer` subclass contains (and is always initialised by) a `TransferDescription` object. The `Transfer` class contains methods for performing tasks such as getting the transfer's status, along with functions that perform actions on the transfer (eg initiating it; aborting it), or act on the instance's embedded `TransferDescription`.

5.2.2 The TransferFactory

`Transfer`s are created by the `bbc.cico.core.TransferFactory` singleton, which selects the appropriate subclass of `Transfer` to create (eg `FTPTransfer`, `MDGTransfer`) based on the transfer method preferred by the remote repository identified by the `TransferDescription` with which the new `Transfer` is to be initialised. The `TransferFactory` singleton must be updated to support every new subclass of `Transfer` that is added to the software.

5.2.3 Transfer Status Monitoring

The status of transfers is given by classes implementing the `bbc.cico.core.TransferStatusMonitor` interface. Such classes can return progress information, both for the files within the transfer and for the transfer as a whole, and an instance of the `TransferStateEnum` enumerated type. This latter class defines the following possible states in which a transfer may be found:

- "Assembly": the transfer is being set up by the user or software, and has not yet started.
- "Queued": the transfer has been set up and is ready/waiting to start.
- "Running": the transfer is in progress.
- "Complete": the transfer has stopped.
- "Unknown": the state of the transfer is unknown.

When the transfer is running, an appropriate instance of a class implementing `bbc.cico.core.FileTransferStatusMonitor` may be obtained by calling the `getFileTransferStatusMonitor()` method of the `Transfer`'s associated `TransferStatusMonitor` instance with a `FileTransfer` as its parameter; this can be used to retrieve information about the transfer progress of individual files within the transfer as a whole.

5.2.4 The Transfer Procedure

To transfer a file using the CICO client, the following procedure must be followed:

- A new `TransferDescription` is created, identifying the user[2] requesting the transfer and the `Server` to/from which the file will be transferred.
- The `TransferDescription.addFileTransferPair()` method is called for each file the user wishes to transfer, identifying source and destination.
- The `createNewTransfer()` method of `TransferFactory` is called to create a new `Transfer` object representing the transfer.
- If no errors have occurred, `initiateTranfer()` is called on the `Transfer` to start the process of transferring the file.
- A `TransferStatusMonitor` instance is obtained by calling the `Transfer`'s `getTransferStatus()` method. This is used to provide feedback on the progress of the transfer to the user.

[2]User support in the current code is minimal. A `bbc.cico.core.User` class has been created and is referenced in certain function calls, but no use is made of the information. Wherever an implementation of `User` is required, a default value is supplied, obtained by calling the `User.getDefaultUser()` static method.

5.2.5 Implementations of the Transfer Classes

Two implementations of `Transfer` and its associated abstract classes have been provided. One, comprising the `MDGRPC`, `MDGRPCService`, `MDGRPCServiceLocator`, `MDGFileTransferStatusMonitor`, `MDGTransfer`, `MDGTransferAgent` and `MDGTransferStatusMonitor` classes is designed to carry out transfers using the Media Dispatch Protocol, although at the time of writing the implementation had not been completed, due to no implementation of an MDP Transfer Agent being sufficiently complete to allow testing.

The other, working, implementation transfers files via FTP, and comprises the `FTPTransfer`, `FTPTransferStatusMonitor` and `FTPFileTransferStatusMonitor` classes. FTP support is provided via the JvFTP [5] library, to which the aforementioned classes act largely as wrappers. A very brief `NullFTPConsole` class that implements the JvFTP console interface is provided to stop the FTP library from writing status information to standard output.

5.2.6 Transfer Agents

Transfer methods can be divided into two kinds: those for which the CICO client connects directly to the repository to transfer files, and those in which file transfer is handled by an external "transfer agent". In the latter case, the client may use different agents for different kinds of transfer, and so agents are represented by implementations of the `TransferAgent` interface.

At present the interface defines only a single method: `getTransfersInProgress()`, which returns to the client a list of the transfers currently underway.

6 Server Classes

Classes exclusive to CICO servers are stored in the `bbc.cico.server` package.

Client-server communication takes place via a custom SOAP API, defined by the bbc.cico.server.CicoRPCInterface class, which can be found in Appendix D. The API defines methods for creating and deleting content objects, locking them to prevent concurrent access, setting and retrieving content object metadata, accessing folder information and server metadata and getting authentication tokens for checking objects in and out.

6.1 The Server Singleton

An individual server is represented by the `bbc.cico.server.Server` singleton. This extends the minimal `bbc.cico.core.Server` class.

Creating a `bbc.cico.server.Server` instance involves a slight complication. When instantiated, the `Server` class initialises its back-end, which may be dynamic, eg an SQL database. Since the `Server` class cannot guarantee the availability of such services, it may fail to complete its instantiation successfully, leading to the throwing of a `bbc.cico.exception.BackendInitialisationException`. To avoid having to catch this condition (and the similar, self-explanatory `bbc.cico.exception.InvalidConfigurationFileContentsException`) every time the singleton's `getInstance()` static method is called, the latter requires the programmer to issue it with a guarantee: it must receive `Boolean.TRUE` as a parameter if (and only if) the programmer has previously called `Server.instantiate()`, otherwise an `Error` will be thrown.

The `Server` singleton performs two tasks. It provides configuration information (backed by a "CICOServer.ini" file in the classpath), and provides an instance of `bbc.cico.server.BackEnd` to

server-side implementations of the `bbc.cico.server.CicoRPCInterface` API. They can use this to retrieve content object metadata. Access to the configuration file is provided by the IniEditor library [6]. For an example configuration file, see Appendix C.

6.2 The BackEnd Interface

The `bbc.cico.server.BackEnd` interface defines the methods that will be used within the server to access content object and folder metadata. It provides a layer of abstraction that permits servers to store these entities in multiple ways. In the current software only a relational-database-backed back-end has been provided, but there are a number of other obvious candidates for implementation, such as a back-end that wraps part of the server's local filesystem.

6.2.1 MySQLBackEnd

The one `BackEnd` implementation in the current software is `bbc.cico.server.MySQLBackEnd`. This uses the MySQL Connector/J library [7] to connect to a MySQL database that stores the metadata. The database schema is given in Appendix B. Files are stored on the server in a dedicated folder.

A simple helper class is associated with the `MySQLBackEnd`, `MySQLBackEndMetadataHelper`. This provides a single static method, `inDatabase()`, which returns `true` if the type of metadata passed to it can be found in the MySQL database. This is required, since some kinds of metadata (eg file size) must be obtained by filesystem calls.

6.3 RPC Data Formatting

Implementing the client-server API in SOAP places limitations on the data formats that can be transferred. Specifically, all objects transfered between client and server must be expressible as XML-SCHEMA entities.

The Apache Axis [8] servlet and libraries have been used at both the client and the server end in the CICO application. For full details of the constraints on Java objects that are permitted to be transferred via SOAP, see the JAX-RPC specification [9]. Briefly though, a number of common Java objects are suitable for transferral unmodified (eg `java.lang.String`, `java.util.Date`).

Only one of the CICO API calls required the creation of a special class to transfer its data: `getFolderTree()`, which returns an array of `bbc.cico.core.Folder` objects. The special class in question is `RPCFolderRepresentation`, which represents the state of the `Folder` as three `Strings`, identifying the FID[3] of the folder, the FID of its parent folder (or null in the case of the root folder), and the folder's name.

The resultant data structure can be transferred via SOAP, as can arrays of `RPCFolderRepresentation` objects. If every folder in the repository is represented as an object in such an array, the result is an "Adjacency List" representation of the repository's entire folder tree [10]. Any standards-compliant SOAP implementation will be able to handle this data; this is important for the sake of interoperability.

[3] A string uniquely identifying a folder within the scope of the repository.

7 Client Classes

The `bbc.cico.client` package contains all the classes common to CICO clients, except for a set of RPC client stub classes that are automatically generated from the WSDL[4] description of the repository's SOAP interface. These latter classes have been packaged in the file `CicoWSDL.jar`, which is included in the source tree.

The RPC stubs are called by a set of classes that represent the server and its contents as subclasses of the `Server`, `Folder`, `ContentObject` and `ContentFile` classes in the `bbc.cico.core` package, which are described above. These subclasses are named `CicoServer`, `CicoFolder`, `CicoContentObject` and `CicoContentFile` respectively. In the current implementation, for the sake of simplicity, only the server's folder tree is cached at the client end. Under certain circumstances, this can lead to a large number of RPC calls per user operation (eg viewing the contents of a folder in a graphical client that displays metadata along with every content object). This is not an issue for demonstration use over a local network, but the potential limitations on the scalability and deployability of the application that arise as a consequence of this choice of approach should be recognised.

A helper class called `PigsEarToCicoFolderTree` exists to assist the `CicoServer` class by turning an array of `RPCFolderRepresentation`s into a doubly-linked tree structure in which nodes contain a reference to their parent (if any) and a list of their children. The latter is the preferred form for representing the tree in memory, and is the representation on which the `Folder` class has been based.

7.1 RemoteServer

The `CicoServer` class is not a direct subclass of `bbc.cico.core.Server`; a `bbc.cico.client.RemoteServer` abstract class has been defined and placed between them in the heirarchy. As the name might suggest, subclasses of `RemoteServer` (such as `CicoServer`) represent entities that the client should regard as content repositories. Subclasses of `RemoteServer` must implement the `javax.swing.tree.MutableTreeNode` interface, and may thus be placed inside a `javax.swing.JTree` GUI component. The single branch of a `RemoteServer` node will be the root folder of the repository.

A quirk of the `RemoteServer` class is that its instances[5] are required to separate initialisation from instantiation in a similar manner to the `bbc.cico.server.Server` singleton. This separation is due to the way in which client representations of repositories are created. The repositories that are to be presented to the user are determined from configuration file settings (in a manner described in detail below) and then instances of the appropriate `RemoteServer` subclasses are created using Java's `Class.forName().newInstance()` method. Only the default constructor can be called using this method, and hence initialisation of the class must be handled separately. `RemoteServer` subclasses must implement a `init()` method that takes as arguments `String`s representing the name of the repository and its URI[6].

It is important that a CICO client application knows what kinds of repository it can connect to. Another way of expressing this requirement is that a client must know all the subclasses of `RemoteServer` in its classpath. Attempts were made to obtain this information dynamically,

[4]Web Services Description Language [11] - a document in WSDL format describing the functions available to RPC clients is generated automatically by Apache Axis in this case.

[5]Since the `RemoteServer` class is abstract, references to its "instances" refer of course to instances of its concrete subclasses.

[6]Uniform Resource Identifier, eg "http://cicoserver.broadcaster.co.uk:8080/axis/services/CICO/".

but they were not successful. For now, whenever a new subclass of `RemoteServer` is made available, details of it must be added manually to the `getSubClassesOfRemoteServer()` method in `bbc.cico.client.Client`.

7.2 LocalServer

A slightly unusual subclass of `RemoteServer` is also included in the `bbc.cico.client` package. This is `LocalServer`, which represents part of the client's local filesystem as a remote content repository. It is the logical client-side companion to the `LocalFolder`, `LocalContentObject` and `LocalContentFile` classes included in the `bbc.cico.core` package. Although it has been useful for testing purposes and is an elegant demonstration of the flexibility of the CICO class model, no current client implementations make use of it.

7.3 The Client Singleton

The core of the `bbc.cico.client` classes is the `Client` singleton. This performs a similar role to the `bbc.cico.server.Server` singleton, providing access to configuration information and storing the identity of the current user.

7.4 Configuration

The configuration system for CICO clients is sophisticated. Java provides two cross-platform configuration methods that are used by the client, and as a last resort, configuration information can be stored in a file, as is the case for CICO servers (described above). The application logic behind this process is contained in the `fillServers()` method of the `Client` singleton, which is called from the latter's constructor.

The primary configuration method for CICO clients is the Java Preferences API. This acts as an abstraction layer for all the various system-dependent application-configuration systems. On Microsoft Windows platforms, the Windows Registry will be used; on Unix-based platforms the information will be stored in a flat file somewhere, and so on. From the programmer's point of view, key/value pairs can be written to and read from the Preferences system and will persist between program executions. In this manner, the `Client` singleton will attempt at startup to read the name, type and URI of all repositories to be presented to the user.

If no server information can be obtained from the Preferences system, the `Client` will look at the System Properties. These are effectively environment variables, containing information about a running application and its environment. They are not persistent, but as is explained below, they can be set by the deployer of a CICO system if the client application is delivered over the web by Java Web Start technology. A first-time user of a CICO system can thereby be presented with a fully-configured application simply by clicking on a website link. As was the case for Preferences, System Properties entries are key/value pairs, and hence the same information is stored in the same format as before.

If no server information can be found in either the Preferences or System Properties, the `Client` will look for the file "CICOClient.ini" in the user's home directory (or equivalent on non-Unix systems). If the file exists, the server information will be read from it using the IniEditor library. For an example configuration file, see Appendix C.

The `Client` class takes no action if it fails to obtain any server information whatsoever. An empty list of known servers will be stored, and the decision regarding what to do in this eventuality is left to the user interface implementation (which may itself delegate the decision to the user).

The obvious place for a client application to *store* (potentially updated) configuration information is in the Preferences system, but again, if and when to do this is up to the user interface.

8 The GUI Client

A Graphical User Interface (GUI) has been written that extends the `bbc.cico.client` code. All classes that implement the GUI are included in the `bbc.cico.clientgui` package. The Java Swing toolkit has been used to implement the GUI, making the application as portable as Java is.

8.1 The Repository Browser

The application's main class is called simply `CICO`. The `main()` method of this class initialises the `bbc.cico.client.Client` singleton and then creates the main GUI window by instantiating the `RepositoryBrowser` class. If information about the previous size and position of the main `RepositoryBrowser` window is stored in the Preferences system then these settings are recovered and reapplied at this point.

It is the constructor of the `RepositoryBrowser` class and the methods called by it that handle the business of setting up the contents of the window. The layout of the `RepositoryBrowser` window is shown in Fig. 1.

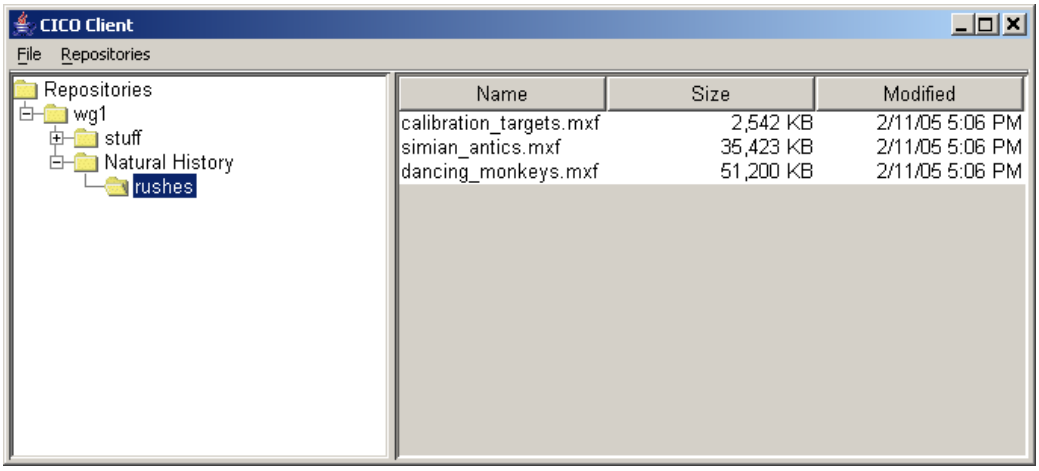

Figure 1: The main "`RepositoryBrowser`" window of the GUI CICO client (shown here running on the Microsoft Windows platform).

The `RepositoryBrowser` is a subclass of `javax.swing.JFrame`. The frame's content pane is divided into two by a `javax.swing.SplitPane` component. On the left is a `bbc.cico.clientgui.RepositoryTree` displaying the repositories known to the client and their folder structures; on the right is a `javax.swing.JTable` containing information about the content objects in the selected folder (if any). An application menu at the top of the frame contains options for checking in and checking out, and for adding, removing and (locally) renaming content repositories. These options are also available via appropriate context menus triggered on objects selected by the user. The `bbc.cico.clientgui.PopupListener` class is responsible for displaying these appropriately.

When the repository browser is closed, information regarding the position and size of its frame is written to the local Preferences system.

8.2 The Repository Tree

The `RepositoryTree` class is a subclass of `javax.swing.JTree`. The few differences between the two are related to a partial implementation of drag-and-drop check-in/check-out. The `ContentFileListTransferable`, `RepositoryTreeDropTargetListener` and `RepositoryTreeTransferHandler` classes are also related to this functionality.

A number of problems were encountered during the implementation of drag-and-drop. Firstly, Java does not appear to receive the relevant drag/drop notification messages from the window manager when running under KDE on SuSE Linux 9.1. Although this problem can be avoided by running the CICO application on Windows, it raises serious questions regarding whether or not drag-and-drop can be implemented in a fully cross-platform manner.

Secondly, under Windows (and presumably on other operating systems that support the drag-and-drop gesture) the data associated with a drag-and-drop file transfer is simply a list of paths to the files that are being dragged. It is up to the drop target application to perform whatever operations the user intended. This means that the files must be available in the local filesystem before the drag-and-drop operation can be completed. This is not a problem for checking in, but poses an obvious problem for checking out to which no easy solution is apparent.

Information regarding the servers in the repository tree is also written to the Preferences system when the repository browser is closed, to be restored when the client GUI is next started.

8.3 The Content Object Table

Although the content object table component is an unmodified `JTable`, it is backed by an application-specific implementation of the `javax.swing.table.TableModel` interface: `bbc.cico.clientgui.ContentObjectTableModel`. This fixes the number of columns to three, determines their contents and provides the `JTable` with filename, size and modification timestamp `Strings` for each content object (if any) in the `List` of `ContentObjects` in the currently-selected `Folder`.

8.4 Actions

All user-triggerable actions have been written as implementations of the JFC[7] `Action` interface. This helps eliminate unnecessary repetition of code by facilitating the triggering of a single event in many ways. For example, a check-in could be initiated via the main application menus, the folder-specific context menu or (theoretically) as a result of a drag-and-drop gesture. All of these would trigger the `actionPerformed()` method of the `CheckInAction` class. In addition, the short-cut keys and menu mnemonics associated with actions are automatically recovered from the respective `Action` classes by the relevant other Java Swing components. Enabling or disabling the `Action` automatically enables and disables the relevant menu options.

The following `Actions` are provided:

- `CheckInAction`: Enabled when a repository folder is selected in the `RepositoryTree`; checks a new file into the selected folder.

- `CheckOutAction`: Enabled when a content object is selected in the content object table; checks out the selected content object to a location of the user's choice.

[7] Java Foundation Classes - a collection of classes included with Java runtime environments, providing core functionality common to many applications

- `DeleteContentObjectAction`: Enabled when a content object is selected in the content object table; deletes the selected content object from the repository and refreshes the client's display.

- `AddServerAction`: Always enabled; adds a new repository to the `RepositoryTree`. Fails if a repository with that URI is already referenced from the tree.

- `RemoveServerAction`: Enabled when a repository is selected in the `RepositoryTree`; removes that repository from the tree.

- `RenameServerAction`: Enabled when a repository is selected in the `RepositoryTree`; changes the *local* name for the repository. This name will be persistent between CICO client sessions on the local machine, but will not affect other installations or the respository itself.

- `WipePreferencesAction`: Disabled under all circumstances by default; this `Action` wipes all the information that the client has stored in the local Preferences system. Used for debugging and development purposes only.

The final class in the GUI is `bbc.cico.clientgui.GUIExceptionHandler`. This catches otherwise-unhandled `Exception`s and formats them as message dialog boxes that are then presented to the user, making use of the `javax.swing.JOptionPane.showMessageDialog()` method.

8.5 Threading

The use of Java Swing necessarily involves the use of threads, although a conscious effort has been made to minimise multi-threading in the GUI client. In the current GUI, the only code that is multi-threaded is that associated with transferring files and updating the transfer progress dialog. These processes cannot run synchronously as they would cause the user interface to freeze for as long as the transfer was in operation: potentially indefinitely.

Transfer progress is displayed by `javax.swing.ProgressMonitor` instances. These create a dialog on screen containing a progress bar. To cause the progress bar to update regularly and to destroy the dialog when the transfer is complete, the instantiated `ProgressMonitor` and a `bbc.cico.core.TransferStatusMonitor` are used to initialise a `bbc.cico.clientgui.TransferProgressRunnable` thread class, which retrieves the transfer state and progress from the `TransferStatusMonitor` at regular intervals (twice per second) and updates the `ProgressMonitor`. If the transfer has finished, the `TransferProgressRunnable` destroys the progress dialog.

During transfer methods for which the CICO client connects directly to the repository to transfer files, it is desirable not to freeze the main `RepositoryBrowser` window, as this will prevent it from repainting itself as other windows are moved across it, etc. It is therefore necessary for such transfers to run inside a thread separate from both the main GUI thread and the `TransferProgressRunnable`. This is implemented in the `initiateTransfer()` method of the relevant `bbc.cico.core.Transfer` subclass (eg `bbc.cico.core.FTPTransfer`): the operations relating to the actual transfer of a set of files are placed inside an inline `java.lang.Thread` and started immediately. Execution returns at once, and hence the main GUI thread is not blocked. The transfer `Thread` is automatically destroyed as soon as the transfer is complete.

9 The CLI Client

As a proof of principle, and to solve a specific problem (that of automatically adding files to a repository as they are copied to or created on the server by other means), a simple check-in-only Command Line Interface (CLI) client was also written. At present this is the only way of integrating the CICO repository with the local filesystem of the machine on which it runs while maintaining the integrity of the database.

The client consists of a single class: `bbc.cico.clientcli.CicoCheckIn`. Three command-line arguments are required: the URI of the server, the path to the destination folder in the repository's folder tree and the path of the file to check in. If an incorrect number of parameters are provided, the client prints a description of what is required to standard output.

Much of the code is taken from `bbc.cico.clientgui.CheckInAction`. The only significant addition is a short routine to walk the repository's folder tree and obtain a `bbc.cico.core.Folder` object representing the destination folder identified on the command line.

10 Exceptions

A large number of subclasses of `java.lang.Exception` have been defined. These may be found in the `bbc.cico.exception` package. Since discussion of exception-handling has been largely omitted from this document for the sake of simplicity, and the difference between different classes in the package is rarely more than their names (which are largely self-explanatory), they will not be discussed in detail here.

One `Exception` that does need explaining is the `MissingJavaFeatureException`. This has been used as a poor substitute for generics (also known as "parameterised classes" or "templates"). Essentially, there is no language feature (in versions of Java ≤ 1.4) that can be used to prevent a programmer from making a `LocalContentObject` the child of a `CicoFolder`, for example. The only constraint placed on the children of `CicoFolder`s is that they are subclasses of `bbc.cico.core.ContentObject`. To prevent this kind of programmer error, a "sanity check" measure has been implemented: whenever a method of one subclass takes an instance of a different parent class as an argument (as would be the case for the `CicoFolder`'s `addContentObject()` method, for example), it is checked at the start of the method and a `MissingJavaFeatureException` is thrown if its type is inappropriate.

11 Packaging and Deployment

Three different combinations of the various CICO packages were produced for deployment.

- CICOServer.jar contains the `core`, `exception` and `server` packages and their dependencies. It is placed on the repository server in Axis' directory for packaged web services (the `WEB-INF/lib/` subdirectory of the Axis installation root), and then deployed as detailed in Appendix E.
- CICOClientCli.jar contains the `core`, `exception`, `client` and `clientcli` packages. The `bbc.cico.clientcli.CICOCheckIn` class is flagged as containing the main class of the JAR file, so a command-line check-in can be initiated by running `java -jar CICOClientCli.jar` <*parameters*>.
- CICOClient.jar contains the `core`, `exception`, `client` and `clientgui` packages. The `bbc.cico.clientgui.CICO` class is flagged as containing the main class of the JAR file.

The JAR file has been designed to be deployed via Java Web Start [12] (JWS), a technology that permits Java applications to be downloaded and launched simply by clicking on a link on a website. CICOClient.jar and its dependencies are placed in a folder on a webserver with a JNLP[8] file such as that given in Appendix F. This fulfils the server-side requirements for an application to be launched via JWS. The client-side requirements are that a compatible Java Runtime Environment (JRE) be installed, such as that available free of charge from Sun Microsystems.

CICOClient.jar and all its dependencies are cryptographically signed to permit them to be executed outside Java's applet sandbox when deployed via JWS, and hence have unrestricted access to the client machine's filesystem and network connectivity. For demonstration purposes, the signing certificate used was itself self-signed, and hence users must click through a warning message before being able to start the application for the first time.

12 Applications and Further Development

The software as it stands will address the majority of the scenarios listed in Appendix A. Having said that, there are a number of obvious ways of extending the functionality it currently provides.

- The architecture does not currently implement the Remote Transfer use case – content objects cannot be transferred directly from repository to repository.
- The client cannot currently manage asynchronous transfers; access to the repository browser is blocked during file transfers and a progress dialog appears.
- Neither versioning nor locking of files to prevent multiple access issues have been fully implemented. At present, checking out a file is equivalent to making a local copy of it, and checking in a file with an identical name to one already present on the repository will neither overwrite nor update the original - they will appear in the GUI client with the same name, but with different timestamps.
- Drag and drop is an important method of initiating check-in and check-out actions, but had not been implemented in the current GUI client. Doing so would result in a significant improvement in the usability of the software.
- There is currently no convenient way of creating, deleting, moving or renaming folders: these actions must be done manually via an SQL client.

During the development of the CICO software, a number of other potential applications were identified, including the following:

- Check-out to portable handheld devices, such as PDAs and Portable Media Players. Many of these devices can be made to appear as an external disk drive when connected to a PC, and hence the existing software can be used to check content out to them. Particularly in the case of PDAs (many of which support Java applications and full independent network connectivity), the ability to run a CICO client on the device itself would also be desirable, however.

[8] Java Network Launching Protocol, an XML format used to give details of a web-deployed application: details of the files and libraries required to launch it, and descriptive information regarding it that can be presented to a user.

- A user-friendly interface to the Media Dispatch Protocol. At the time of writing, an MDP Transfer Agent capable of performing simple file transfers mediated by an MDP manifest was available [13], but no user-friendly software was available to make use of it. Further work would be needed on both the Transfer Agent (to complete the implementation of its API) and the CICO Transfer classes written to interface to it. To reduce the complexity of a CICO installation, the creation of a new `bbc.cico.server.BackEnd` implementation that wrapped (part of) the repository's filesystem would also be desirable[9].

13 Conclusions

A client-server application for checking files in and out of a content repository has been written around a flexible framework that allows the software to be easily extended and modified to support different transfer protocols, file and metadata storage systems and client user interfaces.

A graphical user interface, an FTP transfer system and a hybrid filesystem/relational-database server back-end have been implemented and tested. Together they form a complete demonstration system of check-in/check-out principles. A command-line check-in-only interface has also been implemented to facilitate automated checking-in of content.

The shortcomings of the existing software have been evaluated, and a number of potential applications for the technology identified, which require additional work to varying degrees.

References

[1] P Brightwell. Broadcast media exchange for B2B applications. BBC R&D White Paper 097, Sep 2004.

[2] The Pro-MPEG Forum Media Dispatch Group. `http://www.pro-mpeg.org/publicdocs/mdg.html`.

[3] E Gamma et al. *"Design Patterns : Elements of Reusable Object-Oriented Software"*. Addison Wesley, Feb 1996.

[4] T Macinta & S Paavolainen. Fast MD5 Implementation in Java. `http://www.twmacinta.com/myjava/fast_md5.php`.

[5] B Petrovicova. JvFTP: an FTP client library for Java 1.2 and higher. `http://jvftp.sourceforge.net`.

[6] N Haldimann. IniEditor: a small Java library to read and edit INI-style configuration files. `http://www.ubique.ch/code/inieditor/`.

[7] MySQL AB. MySQL Connector/J: an implementation of Sun's JDBC 3.0 API for the MySQL relational database server. `http://dev.mysql.com/doc/connector/j/en/index.html`.

[8] Apache Software Foundation. Apache Axis: an implementation of SOAP for Java clients and servlet containers. `http://ws.apache.org/axis/`.

[9] R Chinnici et al. Java(TM) API for XML-Based RPC. JCP Spec JAX-RPC 1.1, Oct 2003.

[9] This would be conceptually similar to the `bbc.cico.client.LocalServer` class, but since the `bbc.cico.client.RemoteServer` and `bbc.cico.server.BackEnd` interfaces are very different, the acutal implementations would be quite different.

[10] J Celko. *Trees and Hierarchies in SQL*. Morgan Kaufman, San Francisco, Jun 2004.

[11] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`.

[12] Sun Microsystems. Java web start technology. `http://java.sun.com/products/javawebstart/`.

[13] N Harris et al. MediaDispatch: a Transfer Agent for the Media Dispatch Protocol. `http://sourceforge.net/projects/mediadispatch`.

[14] D Rayers. private communication. Jun 2004.

[15] Apache Software Foundation. Axis Installation Instructions. `http://ws.apache.org/axis/java/install.html`.

Appendix A CICO Scenarios and Use Cases

The following scenarios have been identified [14] as being situations that programme-makers will find themselves in during the production of a programme using the digital tools of the future. For each scenario, the actors are identified, the CICO use cases that correspond to the scenario are given, and the flow of events that will allow a CICO application to be used to address the requirements of the scenario is set out. The use cases referred to here are defined in Section 1.

"I've shot some stuff. It is on my portable PC and I want to send it back to the production centre. I have a fast link."

This scenario maps onto the "Check In" use case.

Actors:

- The user is the "I" of the scenario.
- The "production workgroup" CICO server manages the production workgroup's content repository.

Flow of Events: The user opens the CICO graphical client application, and connects to a "production centre" CICO server using a transfer agent running on the portable PC. The "Check In" flow of events then takes place for each file the user wishes to send to the production centre.

"I'm doing a shoot and I'm on location. I want to look at yesterday's rushes, mark them up and rearrange them and then send them back to base."

This scenario maps onto the "Check In" use case.

Actors:

- The user is the "I" of the scenario.
- The "base" CICO server manages the content repository of the user's "base".

Flow of Events: The user reviews, rearranges and marks up yesterday's rushes on his or her PC using appropriate software. The user then opens the CICO graphical client application and connects to the "base" CICO server using a transfer agent running on the user's PC. The "Check In" flow of events then takes place for each file the user wishes to send to the production centre.

"I've made my programme and I want to submit it to a broadcaster (for review, playout and archiving)."

This scenario maps onto the "Check In" use case.

Actors:

- The user is the "I" of the scenario.
- The "BBC" CICO server manages the BBC's content repository.

Flow of Events: The user opens the CICO graphical client application, and connects to a "BBC" CICO server using a transfer agent running locally. The "Check In" flow of events then takes place for the file comprising the programme.

"I'm part way through making my programme and I want to send off some stuff to a facilities house for special production effects. I then want to automatically get the output from the house when it is available."

This scenario maps onto the "Check In", "Check Out" and "Notify" use cases.

Actors:

- The user is the "I" of the scenario.
- The "BBC" CICO server manages the BBC's content repository.
- The facilities house is the entity responsible for taking the user's content and adding special production effects to it.

Flow of Events: The user opens the CICO graphical client application and connects to a "BBC" CICO server using a transfer agent running locally. The "Check In" flow of events then takes place for the file comprising the original material. The client then registers a notification, asking that the server notify the user when the facilities house checks in new material into the same folder. The user then contacts the facilities house by external means (eg email) to inform them how to access the original material and where to put the modified material. The facilities house then "Checks Out" the original material, modifies it, and "Checks In" the modified material. The "BBC" server then notifies the user of the availability of new material by unspecified means (eg email). Finally, the user "Checks Out" the modified material.

"We have a programme for playout and we want the same content transferred to a regional site so that it can be played from their server as well."

This scenario maps onto the "Check In" and "Remote Transfer" use cases.

Actors:

- The user is the "we" of the scenario.
- The "BBC" CICO server manages the BBC's (central) content repository.
- The remote server manages the remote site's content repository.

Flow of Events: The user opens the CICO graphical client application and connects to a "BBC" CICO server using a transfer agent running locally. The "Check In" flow of events then takes place for the file comprising the programme (if necessary). The "Remote Transfer" flow of events then requests the "BBC" server's Transfer Agent to execute the "Check In" flow of events to transfer the content to the remote server.

"I want to retrieve a particle programme from an archive because I may want to include some of it in my current production."

This scenario maps onto the "Check Out" use case.

Actors:

- The user is the "I" of the scenario.
- The "archive" CICO server manages the archive content repository.

Flow of Events: The user opens the CICO graphical application and connects to an "archive" CICO server using a transfer agent running locally. The "Check Out" flow of events then takes place for the file comprising the programme. The I&A server does not permit the user to lock the file and prevent others from accessing it.

"I have some rushes that could be useful to others and I want to submit them to an archive"

This scenario maps onto the "Check In" use case.

Actors:

- The user is the "I" of the scenario.
- The "archive" CICO server manages the archive content repository.

Flow of Events: The user opens the CICO graphical application and connects to an "archive" CICO server using a transfer agent running locally. The "Check In" flow of events then takes place for the files comprising the programme.

"I've heard that there is a [technical quality check / ARC / Colour Correct / Subtitling / Image Analysis] service available and I have some media that needs checking. I want to submit it to the service and retrieve the results"

This scenario maps onto the "Check In", "Check Out" and "Notify"use cases.

Actors:

- The user is the "I" of the scenario.
- The "service" CICO server manages the content repository of the technical service (which may well be limited to short-term storage of the content being processed).

Flow of Events: The details depend on whether or not the results of the service are textual or comprise a new media file that is an updated version of the original.

In the former case, the user opens the CICO graphical client application and connects to a "service" CICO server using a transfer agent running locally. The "Check In" flow of events then takes place for the file comprising the original material. The service, if automated, is triggered by a "notification" on the folder to which the original material was uploaded. Otherwise, the service is notified of the presence of the material by unspecified means (eg email). The service then "Checks Out" the original material, analyses it and returns the textual results to the user by unspecified means (eg email).

In the latter case, the user opens the CICO graphical client application and connects to a "service" CICO server using a transfer agent running locally. The"Check In" flow of events then takes place for the file comprising the original material. The client then registers a notification, asking that the server notify the user when the service checks in the updated file. The service, if automated, is triggered by a similar notification on the original material. Otherwise, the service is notified of the presence of the material by unspecified means (eg email). The service then "Checks Out" the original material, modifies it, and "Checks In" the modified material. The "service" server then notifies the user of the availability of new material by unspecified means (eg email). Finally, the user "Checks Out" the modified material.

If the material to be modified resides on a CICO server rather than on the hard drive of the user, the transfers can take place using the "Remote Transfer" use case.

Appendix B The CICO Data Model

This appendix describes the database schema used by the CICO application. The schema was kept as simple as possible, consistent with the minimum functionality of the application. Allowing the functionality of the application to be extended at a later date without requiring a major rewrite was also a consideration.

The table structures are presented as MySQL `CREATE` commands. Within each `CREATE` command, the columns are defined first, followed by the table's key (if applicable) and any "unique" columns.

A few bits of jargon:

- An `INT` column contains integers; a `TEXT` column text. A `TIMESTAMP` column contains a date and time.
- A `NOT NULL` column cannot be left empty. `TIMESTAMP` columns are intrinsically `NOT NULL`, since if you add a null entry, it is replaced with the current date and time.
- If an integer column is defined with the `AUTO_INCREMENT` flag, adding a row with a null entry for that column results in the database filling it with a number equal to the highest value for that column in other rows, plus one. This feature is not common to all SQL implementations.
- In a `UNIQUE` column, no two rows can have the same value. For variable-length fields such as `TEXT`, the number of characters on which to force uniqueness must be defined.
- Defining a column as `PRIMARY KEY` enforces uniqueness in the same way as `UNIQUE`. Defining multiple columns as `PRIMARY KEY` enforces unique combinations—the table cannot contain two rows with the same combination of values in those columns.
- InnoDB is a database engine provided with MySQL that supports certain advanced features such as foreign keys.

Users

```
CREATE TABLE ‘users‘ (
‘UID‘ INT NOT NULL AUTO_INCREMENT ,
‘name‘ TEXT NOT NULL ,
‘email‘ TEXT NOT NULL ,
‘password‘ TEXT NOT NULL ,
PRIMARY KEY ( ‘UID‘ ) ,
UNIQUE ( ‘email‘(255) )
) TYPE=InnoDB;
```

Each user is assigned a unique user ID (`UID`). Users are identified by their `email` address, and there can be only one user account per address. The `password` field is of type `TEXT`, but it is intended that hashes of the passwords be stored in it (possibly making use of MySQL's `SHA1()` function.)

Groups

```
CREATE TABLE ‘groups‘ (
‘GID‘ INT NOT NULL AUTO_INCREMENT ,
‘name‘ TEXT NOT NULL ,
‘description‘ TEXT ,
PRIMARY KEY ( ‘GID‘ ) ,
UNIQUE ( ‘name‘(255) )
) TYPE=InnoDB;
```

Each group is identified by a unique group ID (`GID`), and by a unique `name`.

Group Membership

```
CREATE TABLE ‘group_membership‘ (
‘GID‘ INT NOT NULL ,
‘UID‘ INT NOT NULL ,
‘group_admin‘ BOOL DEFAULT ’0’ NOT NULL ,
INDEX (‘GID‘),
FOREIGN KEY (‘GID‘)
REFERENCES ‘groups‘(‘GID‘)
ON DELETE CASCADE,
INDEX (‘UID‘),
FOREIGN KEY (‘UID‘)
REFERENCES ‘users‘(‘UID‘)
ON DELETE CASCADE,

   PRIMARY KEY ( ‘GID‘ , ‘UID‘ )
   ) TYPE=InnoDB;
```

The existence of a `GID`/`UID` pair in this table implies that the user identified by the `UID` is a member of the group identified by `GID`. If `group_admin` is true, the user has admin rights for that group, and can make changes to its membership. The foreign key constraints ensure that if a user or group is deleted, all memberships relating to it are also deleted.

Permissions

```
CREATE TABLE ‘permissions‘ (
‘COID‘ INT NOT NULL ,
‘GID‘ INT NOT NULL ,
‘can_check_out‘ BOOL DEFAULT ’0’ NOT NULL ,
‘can_update‘ BOOL DEFAULT ’0’ NOT NULL ,
‘can_lock‘ BOOL DEFAULT ’0’ NOT NULL ,
‘can_change_perms‘ BOOL DEFAULT ’0’ NOT NULL ,
```

can_check_out	Members of the group can access the resource
can_update	Members of the group can overwrite the resource with an updated version and alter its metadata
can_lock	Members of the group can lock the resource to prevent other users from updating it
can_change_perms	Members of the group can change the permissions of the resource

Table 1: The meanings of the permission types in the `permissions` table

```
PRIMARY KEY ( ‘COID‘ , ‘GID‘ ) ,
INDEX (‘GID‘),
FOREIGN KEY (‘GID‘)
REFERENCES ‘groups‘(‘GID‘)
ON DELETE CASCADE,
INDEX (‘COID‘),
FOREIGN KEY (‘COID‘)
REFERENCES ‘content_objects‘(‘COID‘)
ON DELETE CASCADE
) TYPE=InnoDB;
```

The permissions table defines what members of a group can do with a content object and its instances. The meaning of the permissions is as detailed in Table 1.

Folders

```
CREATE TABLE ‘folders‘ (
‘FID‘ INT NOT NULL AUTO_INCREMENT ,
‘name‘ TEXT NOT NULL ,
‘description‘ TEXT ,
‘creation_ts‘ TIMESTAMP ,
‘parent_folder‘ INT ,
PRIMARY KEY ( ‘FID‘ )
) TYPE=InnoDB;
```

The folders table is a “pig’s ear” representation of a tree of folders. Each folder has a name and an optional description (that could be displayed in a tooltip, for example). The creation timestamp `creation_ts` is filled automatically by MySQL, and may be ignored by clients. The `parent_folder` column contains the `FID` of a folder’s parent, or NULL if the folder is a root folder.

COID	A unique identifier for the content object, for use as a key within the database
URI	A globally unique identifier for the content object (possibly related to the UMID if the primary content object instance is an MXF file, for example; possibly generated by the database)
name	A human-friendly name for the content object
description	A textual description of the content object
tech_description	Any relevant technical information associated with the content object
copyright	Copyright information associated with the content object
filename	The filename of the instance as it will be provided to the user (eg the original filename of the instance)
folder	The FID of the folder in which this content object is located
check_in_ts	The date and time when the content object was first checked in
modified_ts	The date and time when the content object was last modified
creation_ts	The date and time when the content object was created

Table 2: The purposes of the columns in the content_objects table

Content Objects

```
CREATE TABLE ‘content_objects‘ (
‘COID‘ INT NOT NULL AUTO_INCREMENT ,
‘URI‘ TEXT ,
‘name‘ TEXT NOT NULL ,
‘description‘ TEXT ,
‘tech_description‘ TEXT ,
‘copyright‘ ENUM(‘‘Unknown’’, ‘‘Red’’, ‘‘Orange’’, ‘‘Green’’) ,
‘filename‘ TEXT NOT NULL ,
‘folder‘ INT NOT NULL ,
‘check_in_ts‘ TIMESTAMP ,
‘modified_ts‘ TIMESTAMP ,
‘creation_ts‘ TIMESTAMP ,
PRIMARY KEY ( ‘COID‘ ) ,
INDEX ( ‘folder‘ ) ,
FOREIGN KEY ( ‘folder‘ )
REFERENCES ‘folders‘(‘FID‘)
ON DELETE CASCADE,
INDEX ‘URI‘ ( ‘URI‘ ( 32 ) )
) TYPE=InnoDB;
```

The content objects table contains all the information that the CICO application possesses about an individual content object. The purposes of each of the columns are given in Table 2:

Checkouts

```
CREATE TABLE ‘checkouts‘ (
‘COID‘ INT NOT NULL,
‘UID‘ INT NOT NULL ,
‘locked‘ BOOL DEFAULT ’0’ NOT NULL ,
‘reason‘ TEXT ,
‘check_out_ts‘ TIMESTAMP ,
PRIMARY KEY ( ‘COID‘, ‘UID‘ ) ,
INDEX ( ‘COID‘ ) ,
FOREIGN KEY ( ‘COID‘ )
REFERENCES ‘content_objects‘(‘COID‘)
ON DELETE CASCADE ,
INDEX ( ‘UID‘ ) ,
FOREIGN KEY ( ‘UID‘ )
REFERENCES ‘users‘(‘UID‘)
ON DELETE RESTRICT
) TYPE=InnoDB;
```

An entry in this table implies that the user identified by `UID` has checked out the content object identified by `COID` at the date/time stored in `check_out_ts` for the (optional) given reason. If `locked` is true, then the file may not be checked out by other users, nor checked in again, until such a time as the user who locked the file (or an admin?) unlocks it again. A user cannot be deleted while he/she has instances checked out.

Log

```
CREATE TABLE ‘log‘ (
‘timestamp‘ TIMESTAMP NOT NULL ,
‘UID‘ INT NOT NULL ,
‘entry‘ TEXT NOT NULL
);
```

The log table provides an audit trail: every action taken by a user is logged.

COIDseq

```
CREATE TABLE ‘COIDseq‘ ( ‘id‘ INT NOT NULL);
```

The COIDseq table is used to emulate the sequence functionality provided by more fully-featured databases.

IIDseq

```
CREATE TABLE ‘IIDseq‘ ( ‘id‘ INT NOT NULL);
```

The IIDseq table likewise.

Appendix C Example CICO Configuration Files

C.1 CICOClient.ini

```
[Servers]
numServers = 2

[Server0]
serverType = bbc.cico.client.CicoServer
friendlyName = wg1
serverURI = http://cicoserver.broadcaster.com:8080/axis/services/CICO

[Server1]
serverType = bbc.cico.client.CicoServer
friendlyName = test
serverURI = http://testserver.company.co.uk:8080/axis/services/CICO
```

C.2 CICOServer.ini

```
[CICOServer]
#The ContentArchivePrefix is prepended to filenames to form the path at which
#files that are checked in will be stored.  Give a trailing / if you only want
#to specify a directory!
ContentArchivePrefix = /tmp/cico/
Organisation = BBC R&D
TransferMethod = ftp
FQDN = cicoserver.broadcaster.com
TransferEndpointURI = ftp://cicoserver.broadcaster.com/cico/
```

Appendix D The CICO Client-Server SOAP API

The Client-Server API is defined here as a Java Interface that details the methods that comprise the API.

```
/*
 * CicoRPCInterface.java
 *
 * Created on 20 October 2004, 16:03
 */

package bbc.cico.server;

/**
 * The interface offered to clients via SOAP
 * @author  steve
 */
import java.rmi.Remote;
import java.rmi.RemoteException;
import bbc.cico.core.*;

public interface CicoRPCInterface extends java.rmi.Remote \{

    /** Returns a string representing the organisation, for display purposes
     *  (eg "BBC R&D")
     * @throws RemoteException if the RPC fails.
     */
    public String getOrganisation() throws RemoteException;

    /** Returns a representation of the Server's folder tree as an array of
     * bbc.cico.core.RPCTreeNodeRepresentation "classes"
     * @throws RemoteException if the RPC fails.
     */
    public RPCFolderRepresentation[] getFolderTree() throws RemoteException;

    /** Returns an array of COIDs that identify the contents of the
     * specified folder.  "contents" in this context DOES NOT include
     * subfolders.
     * @param the FID identifying the folder in question
     * @returns an array of Strings, each of which is a COID
     * @throws RemoteException if the specified folder does not exist.
     */
    public String[] getFolderContents(String FID) throws RemoteException;

    /** Returns the URI that transfer agents should use to transfer files
     *  to the server
     * eg ftp://user:password@ftp.broadcaster.com/path/to/transferarea/
     * @throws RemoteException if the RPC fails.
     */
    public String getTransferEndpointURI() throws RemoteException;

    /** Returns a string that identifies how the server wishes to exchange
     *  files with a client (eg HTTP, MDP).  The strings are defined in
     * bbc.cico.core.TransferMethodEnum.
     * @throws RemoteException if something untoward occurs.
     */
    public String getTransferMethod() throws RemoteException;
```

```
/** Locks a ContentObject, preventing access to it.
 * @param the COID of the ContentObject to lock.
 * @returns Boolean.TRUE if the ContentObject was successfully locked.
 * @throws RemoteException if the RPC fails.
 */
public Boolean lockContentObject(String COID) throws RemoteException;

/** Deletes a ContentObject, removing its file and metadata.
 * @param COID is the COID of the ContentObject to be deleted.
 * @returns Boolean.TRUE if the ContentObject was successfully deleted.
 * @throws RemoteException if the RPC fails.
 */
public Boolean deleteContentObject(String COID) throws RemoteException;

/** Creates a new ContentObject in the specified folder, fills it with
 * default metadata and points it at a non-existent local essence file
 * @param the FID of the new ContentObject's parent folder
 * @returns the COID of the new ContentObject
 * @throws RemoteException if the RPC fails.
 */
public String createContentObject(String FID) throws RemoteException;

/** Returns a string token that authorises a client to check out a
 * particular ContentObject. The token will be presented to the local
 * TransferAgent by the remote TransferAgent as proof that the transfer
 * is permitted.
 * @throws RemoteException if the RPC fails.
 */
public String getCheckoutAuthenticationToken(String COID)
    throws RemoteException;

/** Returns a string token that authorises a client to check in a
 * particular ContentObject. The token will be presented to the local
 * TransferAgent by the remote TransferAgent as proof that the transfer
 * is permitted.
 * @throws RemoteException if the RPC fails.
 */
public String getCheckInAuthenticationToken(String FID)
    throws RemoteException;

/** Returns the specified metadata for the given Content Object in
 * String form. Note that when metadataEnumString indicates that the
 * file path is desired, this should be the file path in the local
 * Transfer Agent context, not the local filesystem context.
 * @param COID is the COID in String form
 * @param metadataEnumString is a String representation of the
 * bbc.cico.core.ContentObjectMetadataTypes instance that identifies the
 * required metadata.
 * @throws RemoteException if the RPC fails.
 */
public String getContentObjectMetadata(String COID,
                                       String metadataEnumString)
    throws RemoteException;

/** Alters the specified metadata for the given Content Object
 * @param the COID in String form, a String representation of the
 * bbc.cico.core.ContentObjectMetadataTypes instance that identifies the
 * metadata to be changed, along with a String containing the new value.
```

```
     * @throws RemoteException if the RPC fails.
     */
    public void setContentObjectMetadata(String COID,
                                         String metadataEnumString,
                                         String newValue)
        throws RemoteException;

    /** Returns whether or not the specified Content Object is locked
     * @param the Content Object's COID in String form.
     * @returns Boolean.TRUE if the content object is locked;
     * Boolean.FALSE otherwise.
     */
    public Boolean getLockStatus(String COID) throws RemoteException;
\}
```

Appendix E Axis Deployment and Undeployment Descriptors

The following descriptors should be passed to the Axis AdminClient using the following command:
On Unix:

```
java -cp $AXISCLASSPATH org.apache.axis.client.AdminClient
     -lhttp://localhost:8080/axis/services/AdminService deploy.wsdd
```

On Windows:

```
java -cp %AXISCLASSPATH% org.apache.axis.client.AdminClient
     -lhttp://localhost:8080/axis/services/AdminService deploy.wsdd
```

`AXISCLASSPATH` should be set to include all the Axis libraries - see step 6 of the Axis installation guide [15] for details. Alter the URI of your Axis AdminService appropriately.

E.1 deployment.wsdd

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CICO" provider="java:RPC">
    <parameter name="className" value="bbc.cico.server.CicoRPC"/>
    <parameter name="allowedMethods" value="*"/>
    <beanMapping qname="CICO:RPCFolderRepresentation"
   xmlns:CICO="urn:BeanService"
       languageSpecificType="java:bbc.cico.server.RPCFolderRepresentation"/>
  </service>
</deployment>
```

E.2 undeployment.wsdd

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="CICO"/>
</undeployment>
```

Appendix F JNLP Manifest for CICO GUI Client

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://webserver.broadcaster.com/JARPATH/" href="CICO.jnlp">
  <information>
    <title>CICO</title>
    <vendor>BBC Research and Development</vendor>
    <description>An application for remote check-in/check-out</description>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4"/>
    <jar href="CICOClient.jar"/>
    <jar href="CicoWSDL.jar"/>
    <jar href="axis.jar"/>
    <jar href="commons-discovery.jar"/>
    <jar href="commons-logging.jar"/>
    <jar href="inieditor.jar"/>
    <jar href="jaxrpc.jar"/>
    <jar href="jvftplib.jar"/>
    <jar href="saaj.jar"/>
    <property name="numServers" value="1"/>
    <property name="Server0Type" value="bbc.cico.client.CicoServer"/>
    <property name="Server0URI" value="http://cicoserver.broadcaster.com:8080/axis/services/CICO"/>
    <property name="Server0Name" value="broadcaster"/>
  </resources>
  <application-desc main-class="bbc.cico.clientgui.CICO"/>
</jnlp>
```

Note that the URIs in this description file will need to be changed to suit each individual installation: the first should be the URI of the web-accessible folder in which CICOClient.jar and its dependencies may be found; the second occurs in the (optional) `System.Properties` section, which will be used by clients the first time they are launched on a given computer to configure themselves; the `"Server0URI"` value should be changed to the URI of the SOAP RPC interface of a CICO server, and the `"Server0Name"` value to a user-friendly name for the repository.