DataObjectProxy Users Guide

Abstract

This document explains how to use the **DataObjectProxy** and related classes in D2K modules.

Revision History

Latest entry ***at the top*** please			
Version	Author/Contributor	History	Date(yyyy-mm-dd)
004	Robert E. McGrath, Fang Guo, Barry Sanders	Edits, add info. about logging.	2007-1-23
003	Robert E. McGrath	Info. about software dependencies	2006-11-27
002	Robert E. McGrath	Revised, with info about cache, multifiles	2006-11-20
001	Robert E. McGrath	Initial version	2006-10-23

Acknowledgements

This work was funded by the National Center for Advanced Secure Systems Research (NCASSR) at the University of Illinois at Urbana-Champaign (UIUC), a multi-institutional cybersecurity research team. NCASSR focuses basic and applied research necessary to develop next generation information security technologies that address national cybersecurity needs. NCASSR is led by the National Center for Supercomputing Applications (NCSA) and supported by funding from the Office of Naval Research (ONR).

Contents

1. Introduction

D2K ([2]) data analysis projects use and create many data objects, including input data, intermediate results, documentation, and output data. Storing data in local or network files or databases is not adequate: it can be difficult to locate the data when needed, either because the location is not known, or because the contents of the files or tables are not known, or both.

At NCSA, the Tupelo project is designing a high performance, large scale repository to meet these requirements ([6]). The overall software developments needed to integrate D2K with Tupelo and other similar repositories are discussed in [4].

In general, the goal is to provide a facility to manage data objects associated with a D2K project, i.e., multiple runs of one or more D2K itineraries, building on the best practices and standards as defined by the NCSA Design Principles for Cyberenvironments [7]. This will enable D2K applications to access and exploit with the general Cyberinfrastructure, and reuse other work as much as possible.

D2K support has a variety of development targets (see also [4]):

1. Standard D2K modules (i.e., user visible modules) to interact with repositories, e.g., modified versions of file I/O, and new modules, e.g., for sophisticated data browsing of remote data.
2. Options within the D2K infrastructure, e.g., to automatically record provenance or other metadata "behind the scenes" (i.e., not directly visible to user modules)
3. Modifications of the D2K Toolkit and other environments, e.g., to have the modules, itineraries, and other data be accessed via shared repositories rather than local disk
4. Features to support distributed execution, e.g., modifications to the D2K Web Service to use repositories instead of local files

This note describes the DataObjectProxy class, which is designed to meet address the first goal.

2. Overview of the Design

The current D2K environment includes modules that access databases and modules that access local files. Files may also be accessed via HTTP. The former case is managed by the "Connection" class. Access to local or remote files is written into the consuming classes. In these cases, the user provides a path or URL (as a string), and the module manages the set up and data accesses.

An example is the classes that parse files. This process is organized with a factory class that accepts a path name, creates an instance of the required class of parser, which is passed to the consumer. The parser opens and reads the file, and the consumer draws data from the parser. Figure 1 shows the D2K modules that a user itinerary can use to parse a delimited file. The **CreateDelimitedFileParser** module (Figure 1b) reads the path as a string (from the **InputFileName** module, Figure 1a). This path is passed to a instance of **DelimitedFileParser** (Figure 1c), which is responsible for accessing the file. The parser object is passed to the **ParseFileToTable** module (Figure 1d), which calls the parser to obtain data.

In this example, consider what would be required to read the file from a remote source, such as a Web page. First, the **InputFileName** will need to be extended to understand URLs. Second, the parser class will need to implement different access methods depending on whether the input is from local disk or remote file. These changes will need to be replicated throughout many modules and supporting classes.

The **DataObjectProxy** is a wrapper that manages the data access methods. The initial implementation supports local files and remote files accessible through the WebDAV protocol [3]. Other access methods may be added in the future.

Fundamentally, every module or class that does file I/O should be modified to use a **DataObjectProxy** instead. Figure 2 illustrates this for the example above. The **InputFileName** should be changed to obtain a URL, including server name, user, and password, if needed (Figure 2a). The new class should create an instance of **DataObjectProxy** (Figure 2b), which is passed to the **CreateDelimitedFileParser** module (Figure 2c). The **DelimitedFileParser** will need a new method to access data via the **DataObjectProxy** rather than opening the file directly from a path name. The consumer **ParseFileToTable** class (Figure 2d) will receive the **DelimitedFileParser**, which is used exactly as before.

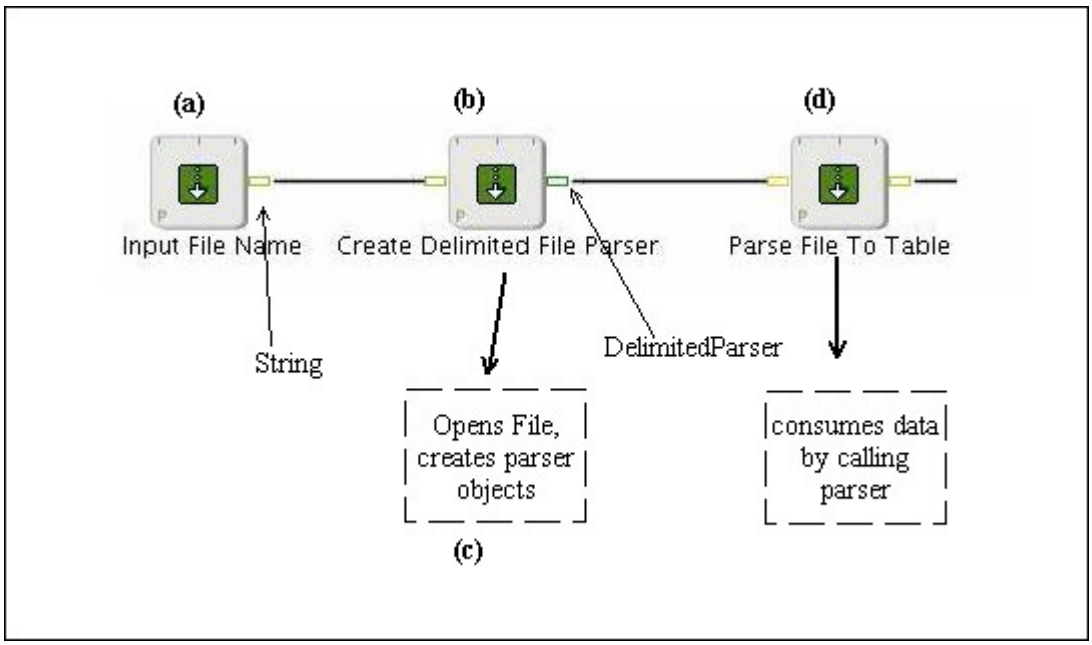

Figure 1. Example Itinerary using the old modules.

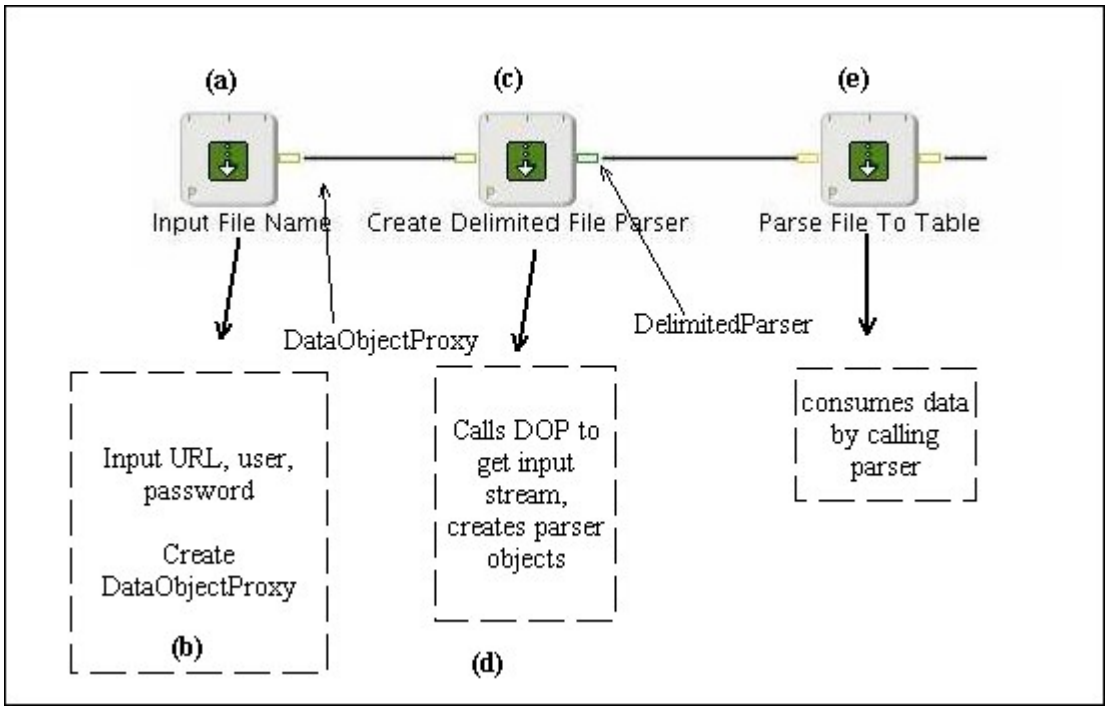

Figure 2. Example using the new modules (compare to Figure 1).

3. Usage

This section gives some examples of how to use the **DataObjectProxy.** There are three steps:

1. create a **DataObjectProxy** for a given URL or file.
2. read or write data via the proxy

3. close the proxy to clean up.

The **DataObjectProxy** provides mechanisms to manage a local temporary file, if needed. The **DataObjectProxy** may throw exceptions, which may be raised by dependent software.

3.1 Creating a DOP.

A **DataObjectProxy** object is created using the **DataObjectProxyFactory**. The object is created from a URL which may be a local file or a URL.[1] The basic usage is:

```
DataObjectProxy dataobj = DataObjectProxyFactory.getDataObjectProxy( url, username, password);
```

It is important to note that the **DataObjectFactory** will create an instance of an appropriate sub class of **DataObjectProxy**, depending on the URL. When the URL is for 'file:', the class will implement access to local files, when the URL is 'http:', the class will implement WEBDAV access. In either case, the consumer will use the object in the same way.

```
/* Four D2K properties are defined:
 *     FileName
 *     HostURL
 *     Username
 *     Password
 */

public void doit() throws Exception {
    String fn = getFileName();
    String hosturl = getHostURL();
    URL url;

    /* set up a URL for the object */
    if (hosturl == null || hosturl.trim().length() == 0) {
      /* no host is given, so this is a local file */
      File file = new File(fn);
      url = file.toURI().toURL();
    } else {
      /* construct a URL for the remote object */
      url = new URL(createURL(hosturl, fn));
    }

  /* Use the properties to create a DataObjectProxy for the specified URL or file. */
    DataObjectProxy dataobj = DataObjectProxyFactory.getDataObjectProxy(url,
                                        getUsername(), getPassword());

   /*  Note: the DataObjectProxy returned above will be an instance of either LocalDataObjectProxy
     * or WebdavDataObjectProxy, depending on the URL.
     */

    /* pass the proxy to the next module */
    pushOutput(dataobj, 0);
}
```

Figure 3. Example 'doit' method from a D2K module.

[1] In the initial version, the URL must use HTTP or SHTTP. Other protocols (such as FTP) have not been tested and may or may not work.

Figure 3 shows example code that might be used in a D2K module. This code collects a URL or path (along with username and password, if needed), and creates a **DataObjectProxy** for the indicated object. The proxy is pushed to the next module.

An alternative way to create a **DataObjectProxy** is to reset the URL for an existing proxy.

```
DataObjectProxy dataobj2 = dataobj1.resetURL( newurl, newusername, newpassword);
```

This method for proxy reuse has the same effect, but will reuse the username and password if they are not specified. This may be convenient when accessing several objects on the same server.

3.2 Reading

Data can be read from **DataObjectProxy** via a Java **InputStream**. The consuming program obtains a reader, which then is used just as any other data stream would be used. For example:

```
BufferedReader reader = new BufferedReader( new InputStreamReader( dataobj.getInputStream()));
```

Figure 4 shows example code for reading from a **DataObjectProxy**. Note that the code after the reader is set up is identical to the way any data source would be read in a Java class.

When the data object is actually on a remote server, it may be accessed two ways, as an **InputStream** from the server (actually from HTTP), or by copying the data to a local file and accessing the local copy. The latter is used when the consuming code needs to randomly access the data or otherwise expects a local file, since the HPPT protocol does not provide a mechanism for random access within remote files.[2]

The **DataObjectProxy** can be forced to download to a local copy by calling the **getLocalFile** method. (If the object is already a local file, this method has no effect.) Figure 5 shows a segment of code to illustrate. One reason to force the proxy to cache the file would be to be able to randomly access the object. Figure 5 shows a **reset()** on the file. This would not be possible for a stream from a remote server.

```
private void doit() throws Exception {
    /* local variables … */

    /* connect to the DataobjectProxy, read from a stream  */
    DataObjectProxy dop = (DataObjectProxy) pullInput(0);
    InputStream is = dop.getInputStream();
    BufferedReader reader = new BufferedReader(new InputStreamReader(is)));

    // read the file in one row at a time
    int currentRow = 0;

    while ( ((line = reader.readLine()) != null) &&  (currentRow < NUM_ROWS_TO_COUNT)) {
      lines.add(line);
   /* … */
```

Figure 4. Segment of code to illustrate reading from a DataObjectProxy.

[2] Some servers support access to byte ranges, which could, theoretically, be used to implement random access.

```
public void doit() throws Exception {
     /* local variables */

    /* connect to the DataobjectProxy, read from a local file  */
     DataObjectProxy dop = (DataObjectProxy) pullInput(0);
    /* Note:  if the target is remote, the object will be downloaded to this local file */
     InputStream is = dop.getLocalInputStream();
     BufferedReader reader = new BufferedReader(new InputStreamReader(is));

     // read the file in one row at a time
     int currentRow = 0;

     while ( ((line = reader.readLine()) != null) && (currentRow < NUM_ROWS_TO_COUNT)) {
        lines.add(line);

        /* … */
      }

         /* note: since this is a local file, the program can randomly access: */

       reader.reset()

       /* read again from the beginning … */
```

Figure 5. Reading from a DataObjectProxy from a file (compare to Figure 4).

3.3 Writing

Data can be written to **DataObjectProxy** from a Java **File** or **InputStream**. The producing program obtains the local file or stream, which then is used just as any other data stream would be used. For example:

```
       File f = … /* open file… */
       dataobj.putFromFile(f);
or
       InputStream is = ... /* open stream… */
       dataobj.putFromIS(is);
```

```
/*  This example needs  one D2K property:  Filename  */
public void doit() throws Exception {
     Object object = pullInput(0);  /* the object to be written */
     DataObjectProxy dop = (DataObjectProxy) pullInput(1);

     File localFile = dop.initLocalFile();  // a local temporary file

    /* open the local file, write the serialized object */
     try {
        out = new ObjectOutputStream(new FileOutputStream(localFile);
        out.writeObject(object);
     } catch (Exception e) {
        throw new IOException("Unable to serialize object"+ e);
     }
     out.flush();
     out.close();
     /* write the object to the store.  Remote access will transfer the file to the server here */
     dop.putFromFile(localFile);
     dop.close();  /* flush the transmission if needed. */
  }
```

Figure 6. Writing from a local file to a DataObjectProxy object.

In some cases, it will be necessary to create a temporary local file to write the data to, which will be written to the **DataObjectProxy** when completed. In Figure 6, the temporary file was obtained from an input property. The **DataObjectProxy getLocalFile** method can be used to generate a temporary file.

```
File localFile = dataobj.getLocalFile();
```

3.4 Close, cleanup

The **DataObjectProxy** object should be "closed" when the access is completed. The **close()** method cleans up and finishes any data transfers, and closes any connections. In the case where a local file has been allocated, the **close** method will clean up the temporary file.

```
DataObjectProxy dop = (DataObjectProxy) pullInput(0);
FileWriter fw = new
      FileWriter( dop.initLocalFile());

  fw.write(...); fw.flush(); fw.close();

  dataobj.putFromFile(dataobj.getLocalFile());

  /* Very important to call close.  Cleans up the local temporary file,
     closes the remote connection to flush all output.  */
  dop.close();
```

Figure 7. Closing the DataObjectProxy.

3.5 Errors

The **DataObjectProxy** interface returns exceptions wrapped in an instance of **DataObejctProxyException**. The exceptions may well reflect problems in the I/O, which depend on the means used. In the case of local files, the errors may be an **IOException** or a security failure, for instance. In the case of a remote URL, the failures usually will be HTTP status codes (e.g., 401 (Unauthorized)) or **MalformedURLException**, for example.

Note that the **DataObjectProxy** implementations are built on top of other software. Unfortunately, this means that the stack trace for an exception may extend deep into other packages, e.g., for the HTTP client protocol.

Modules that use a **DataObjectProxy** usually should handle the **DataObjectProxyException**, to either fix the problem or raise a clearly understandable exception for other modules or the user.

The **DataObjectProxy** can only access objects if the storage system supports the required access. In particular, access to remote objects requires the target HTTP server supports WEBDAV. In practice, a server may support WEBDAV for some objects and not others on the same server, it is up to the server. This cannot be determined from the URL, the only way to determine if the object can be accessed is to ask the server.

When a **DataObjectProxy** is created for a URL, it checks whether access is supported. If not, it will raise a **DataObjectProxyNotSupportedException**.

3.6 Example: Copy Object From One Server to Another

The **DataObjectProxy** can be used without creating a local file. For example, Figure 8 shows example code to copy from one remote URL to a second URL using two instances of **DataObjectProxy**. In this example, the data is passed through memory of the calling program, as the 'put' call pulls data from the remote location through a Java stream.

```
public void copy(String srcURL, String destURL) {

    try {
        /*  set up proxy to read from source */
        URL url1 = new URL(srcURL);
        DataObjectProxy dop1 =DataObjectProxyFactory.getDataObjectProxy(url1,uname1,pword1);
        InputStream is = dop1.getInputStream();

        /* set up proxy to write to dest */
        URL url2= new URL(destURL);
        DataObjectProxy dop2 =DataObjectProxyFactory.getDataObjectProxy(url2,uname2,pword2);

        /* execute the copy:  put from the input stream */
        dop2.putFromStream(is);

        dop1.close();
        dop2.close();
    }
    catch (Exception e) {
        ..
    }
}
```

Figure 8. Example code to copy through memory.

4. Multiple File Operations

The operations described above operate on a single file or URL. The **DataObjectProxy** also provides options to read or write the contents of a directory, creating a similar hierarchy at the destination. These operations recursively descend the hierarchy, copying directories and files as they are discovered.

There are three methods of interest. Each starts at the URL specified and traverses the hierarchy to a specified depth. In the current implementation, the depth is DEPTH_0, DEPTH_1, or DEPTH_INFINITY, meaning the directory itself, the immediate children, or all the descendents.

The **downloadDir** method retrieves the directories and files from the source, and copies to the destination. As the name suggests, this is used to copy whole directories from a URL. Figure 9 shows an example of usage.

```
public void doit() throws Exception {

    DataObjectProxy srcdop = (DataObjectProxy) pullInput(0);
    DataObjectProxy desdop = (DataObjectProxy) pullInput(1);

    Int depth = DEPTH_INFINITY;

    System.out.println("Start downloading "+ srcdop.getURL()+ " to "+desdop.getURL());
    srcdop.downloadDir(desdop,depth);
    System.out.println("End downloading "+ srcdop.getURL()+ " to "+desdop.getURL());
    desdop.close();
    srcdop.close();
}
```

Figure 9. Example code for downloadDir, multiple files.

A similar method, **uploadDir**, moves files from the source to a destination. Again, this can be thought of as copying files up to a server. Figure 10 shows example code.

```
public void doit() throws Exception {

      DataObjectProxy srcdop = (DataObjectProxy) pullInput(0);
      DataObjectProxy desdop = (DataObjectProxy) pullInput(1);
      int depth = DEPTH_INFINITY

      System.out.println("Start uploading "+ srcdop.getURL()+ " to "+desdop.getURL());
      desdop.uploadDir(srcdop,depth);
      System.out.println("End uploading "+ srcdop.getURL()+ " to "+desdop.getURL());
  }
```

Figure 10. Example code for uploadDir, multiple files.

The two methods (**uploadDir** and **downloadDir**) do the same thing, and the source and destination can be local or remote for either. That is, both **uploadDir** and **downloadDir** can copy local-local, local-remote, remote-local, or remote-remote. The only difference is the invocation.

A related method discovers all the descendents and returns a list of their URLs in a vector. Note that the order of this list may vary from different sources. Figure 11 gives an example of this function. Figure 12 shows example output for the case where the **DataObjectProxy** is pointing to a remote server, and Figure 13 shows the output for a local directory.

A variant of this method returns only the non-directories from the list. Example code is shown in Figure 11, and example output is shown in Figure 12 and Figure 13.

```
      System.out.println("List the contents:");
       Vector list = srcdop.getChildrenURLs(depth);
       Enumeration en = list.elements();
       while (en.hasMoreElements()) {
           Object s =en.nextElement();
           System.out.println("  "+s);  // should be a URL
       }

      System.out.println("Just the directories:");
       Vector list = srcdop.getChildrenURLs(depth, true);
       Enumeration en = list.elements();
       while (en.hasMoreElements()) {
           Object s =en.nextElement();
           System.out.println("  "+s);  // should be a URL
       }
```

Figure 11. Example code for getChildrenURLs, multiple files.

```
List the contents:
 http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/Bob/DataObjectProxy.java
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/4/DataObjectCacheRecord.java
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/Merge/4/DataObjectProxyException.java
Just the directories:
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/Bob/DataObjectProxy.java
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/4/DataObjectCacheRecord.java
  http://vesta.ncsa.uiuc.edu:8088/slide/files/Bob/Merge/Merge/4/DataObjectProxyException.java
```

Figure 12. Example output from a server

```
List the contents:
  file:/tmp/XX/
  file:/tmp/XX/getdirectory.itn
  file:/tmp/XX/putdirectory.itn
  file:/tmp/XX/rmdirectory.itn
Just the files:
  file:/tmp/XX/getdirectory.itn
  file:/tmp/XX/putdirectory.itn
  file:/tmp/XX/rmdirectory.itn
```

Figure 13. Example output from a local file system

Important Restriction

As noted above, the multiple file options work for local files or for remote servers that implement the WEBDAV protocol. A standard Web Server does *not* provide the capability to retrieve directory listings, so it is not possible to list, get, or put multiple files from a regular web server, only from a WebDAV enabled server.

5. Read Cache

The **WebdavDataObjecProxyImpl** class implements a simple cache for downloading temporary copies of files. This cache is transparent to the calling program.

The cache is configured to manage one or more directories, usually in the working area of the D2K environment, e.g., under the 'data' directory. The cache manager allocates temporary files, maintains a list of the cache contents, and detects changed or old copies that must be deleted and reloaded from the source. When a current copy of the file is found in the cache, the download is not repeated, and the cached copy returned to the caller.

The cache is only used when downloading to a local file, and only when the calling program does not specify a destination file. In the examples above, Figure 6 would use the cache, while Figure 4 or Figure 7 would not use the cache because they use an input stream and are uploading a file respectively.

The cache cleaning policy is implemented by an implementation of the **DataObjectCachePolicy** interface. The default implementation (**DataObjectCacheDefaultPolicy**) removes any file older than 24 hours.

The implementation of the cache is transparent to the user, and is designed to be easy to replace with alternative implementations. The initial version saves its persistent data in a local XML file, managed using JAXB ([8]) to both read and write the XML. This module could be replaced with a module that uses a database or other mechanism to save the state.

6. Notes on Software Configuration

The **DataObjectProxy** and related classes are built on top of external Java packages which must be included in the class path of any itinerary that uses the **DataObjectProxy**. In addition, the **DataObjectProxyCache** (See section 5, above) requires configuration files and local disk.

6.1 Software Dependencies

The **WebdavDataObjectProxyImpl** requires nine Jar files, which implement HTTP and WebDAV protocols. Table 1 lists these Jar files. These jars come from the Apache Jakarta project, an open source HTTP client and from the Scientific Middleware (SAM) project from the Collaboratory for Multi-scale Chemical Science (CMCS) project [5].

The **DataObjectProxyCache** implementation uses JAXB to manage its stored state. The Jar files for JAXB are listed in Table 2. The JAXB package uses Java activation, so these Jar file must be in the class path for the *itinerary*—usually, they should be in the *modules* directory rather than the *lib* directory.

Table 1. Jars required for the DataObjectProxy (WebDAV support). Note: the table suggests the order in the class path, e.g., in Toolkit.lax.

Jar File	Source
lib/commons-httpclient.jar	Apache Jakarta [3]
lib/commons-httpclient-contrib.jar	Apache Jakarta [3]
lib/dsmgmt-1.0dev.jar	CMCS [5][3]
lib/sam-webdavlib.jar	CMCS [5]
lib/jakarta-slide-webdavlib-2.1.jar	Apache Jakarta [3]
lib/security-1.0dev.jar	CMCS [5]
lib/util-1.0dev.jar	CMCS [5]
lib/commons-logging.jar	Apache Jakarta [3]
lib/jdom-1.0.jar	Jdom.org, available from Apache Jakarta [3]

Table 2. Jars required for the DataObjectProxyCache implementation (uses JAXB). Note that these Jars usually need to be in the *modules* directory (rather than the *lib* directory) because JAXB uses reflection/activation.

Jar	Source
activation.jar	Java, included in JAXB Reference Implementation [8]
classes12.jar	Java, included in JAXB Reference Implementation [8]
jaxb-api.jar	JAXB Reference Implementation [8]
jaxb-impl.jar	JAXB Reference Implementation [8]
jaxbv1-impl.jar	JAXB Reference Implementation [8]
jaxb-xjc.jar	JAXB Reference Implementation [8] (Needed in compile but not in runtime)
jaxp-api.jar	JAXB Reference Implementation [8]
jsr173_1.0_api.jar	JAXB Reference Implementation [8]
dom.jar	Apache XML [1, 9]
sax.jar	Apache XML [1, 9]
xalan.jar	Apache XML [1, 9]
xercesImpl-2.7.1.jar	Apache XML [1, 9]. Note: the JAXB reference implementation depends on this specific version of Xerces.

6.2 Configuration of the Cache

The local read cache has three functional parts: one or more directories for cached copies, an XML file with the state of the cache, and the XML schema for the cache state.

6.2.1 Directories

The read cache is implemented by the **DataObjectProxyCache** and related classes. These classes manage a pool of recently downloaded files, keeping track of the original URL, the date downloaded, and other information about the cached copies. When a download is requested by calling **DataObjectProxy.getLocalInputStream()**, when the target is on a remote server, the cache checks to see if an up to date copy is available on local disk. If so, it is used and the download is skipped. Otherwise, the target is read into a temporary file in the cache, and a record added to the cache state.

Thus, the cache consists of one or more directories of local files and an XML file with the current state of the cache. The layout of the cached files is up to the implementation. The default implementation creates a default directory in the top level of the D2K toolkit run time.

The cache state is stored in an XML file which is read and updated by the cache management code. The current implementation uses the schema in **resources/cacheManager.xsd**, and stores the file in

[3] The Jar files were obtained within NCSA. It is not certain where the sources are maintained.

resources/testCacheManager.xml. These should be present in the top level directory of the D2K Toolkit or equivalent working directory.

6.2.2 Schema for the Cache Records

The cache state consists of a series of records, each describing one local copy from a remote server. Figure 14 shows the current version of the schema. Table 3lists the fields for one record.

```
<xs:schema version="1.0">

<xs:element name="dataObjectCacheInfo" type="DataObjectCacheRecords"/>

<xs:complexType name="DataObjectCacheRecords">
    <xs:sequence>
      <xs:element name="cacheList" type="DataObjectCacheRecord" minOccurs="0"
maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="DataObjectCacheRecord">
    <xs:sequence>
      <xs:element name="calendar" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="ID" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="localFile" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="URL" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="ETag" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="username" type="xs:string" minOccurs="1" maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="rid" type="xs:string"/>
</xs:complexType>

</xs:schema>
```

Figure 14. The XML Schema for the cache state.

Table 3. Description of the fields in a cache record.

Field	Description
calendar	The date of the initial download
ID	The name of the local file, used as a key.
localFile	The name of the local file
URL	The source URL
ETag	The HTTP ‘etag’, a unique hash for each version of the URL.
username	The user name used to retrieve the file.
attribute rid	The ‘record id’ used as a key.

6.2.3 Use of JAXB

The configuration file is an XML file. The schema for this file is in resources/cacheManager.xsd, as discussed above. The schema is used to automatically generate Java classes to parse XML into java data structures, and to write XML from Java. The classes are generated with the JAXB utility *xjc*.

```
xjc.sh -p ncsa.d2k.modules.core.io.proxy -d /path/to/source/resources cacheManager.xsd
```

Xjc needs to execute *when (and only when) the schema changes*.

Xjc generates classes that represent the data structures described in the schema. These are listed in Table 4. (A script to invoke xjc is included in the 'bin' directory in CVS.)

Table 4. The generated classes (in package ncsa.d2k.modules.core.io.proxy).

Generated Class	Description
ObjectFactory.java	Utilities for invoking JAXB
DataObjectProxyCacheRecord.java	The Java object for a single record.
DataObjectProxyCacheRecords.java	The sequence of records.

When the cache manager runs it calls the JAXB libraries in lib/jaxb-xjc.jar, lib/jaxb-api.jar, lib/jaxb-impl.jar. These libraries check and parse the input XML file, and create arrays of objects using the generated classes. The cache manager extracts the configuration data from these classes. When the state changes, the cache manager updates the Java classes and then calls JAXB to write out XML.

Again, if the schema for the configuration file remains unchanged, there is no need to regenerate the JAXB classes.

6.3 Logging

As discussed above, the WEBDAV implementation uses DSI and Apache HTTP classes. These classes produce log messages which are not under the control of the D2K logging facilities. Parts of theses packages uses log4j and other parts use java.util.logging. The log4j can be controlled by a log4j.properties file. Figure 15 shows suggested settings to suppress log messages.

To control the java.util,logging messages, the DataObjectProxy implementation has an initialization that maps the D2K logging levels to equivalent settings for Java logging, and forces the logging to that level. This code is illustrated in Figure 16, Figure 17, and Figure 18. The effect is for the D2K logging settings to be applied to the java logging.

```
log4j.rootLogger=FATAL

log4j.logger.org.apache.commons.httpclient=FATAL
log4j.logger.httpclient.wire.header=SEVERE
log4j.logger.httpclient.wire.content=SEVERE

log4j.logger.org.scidac.cmcs=FATAL
log4j.logger.org.scidac.cmcs.dsmgmt=FATAL
log4j.logger.org.scidac.cmcs.dsmgmt.dsi=FATAL
```

Figure 15. Suggested settings to suppress log messages.

```
      org.apache.log4j.Level ll = D2KLogger.logger.getEffectiveLevel();
      java.util.logging.Level l = convertLevel(ll);
      setJavaUtilLogger(l);
```

Figure 16. Example code to set log levels.

```
private void setJavaUtilLogger(java.util.logging.Level l) {
    LogManager lm = LogManager.getLogManager();

    java.util.Enumeration e = lm.getLoggerNames();

      while (e.hasMoreElements()) {
        Logger jlogger = lm.getLogger((String) e.nextElement());
         jlogger.setLevel(l);
      }
}
```

Figure 17. Example code to set Java logging.

```
private java.util.logging.Level convertLevel(org.apache.log4j.Level lin) {

    if (lin == null) {
         // Fatal
        return null;
      }

      if (lin == org.apache.log4j.Level.DEBUG) {
         return java.util.logging.Level.FINEST;
      } else if (lin == org.apache.log4j.Level.WARN) {
         return java.util.logging.Level.WARNING;
      } else if (lin == org.apache.log4j.Level.INFO) {
         return java.util.logging.Level.INFO;
      } else if (lin == org.apache.log4j.Level.ERROR) {
         return java.util.logging.Level.SEVERE;
      } else if (lin == org.apache.log4j.Level.FATAL) {
         return java.util.logging.Level.SEVERE;
      } else if (lin == org.apache.log4j.Level.OFF) {
         return java.util.logging.Level.OFF;
      } else if (lin == org.apache.log4j.Level.ALL) {
         return java.util.logging.Level.ALL;
      } else {
         return java.util.logging.Level.INFO;
      }
   }
```

Figure 18. Example code to map D2K log levels to Java log levels.

7. Conclusion

This document has presented the basic usage of the **DataObjectProxy** and the **DataObjectProxyCache**.

8. References

1. Apache XML Project. *Xerces-2 Java*. http://xml.apache.org/xerces2-j/index.html.
2. Automated Learning Group. *D2K - Data to Knowledge*. 2005, http://alg.ncsa.uiuc.edu/do/tools/d2k.
3. Jakarta Project. *Slide*. 2006, http://jakarta.apache.org.
4. McGrath, Robert E., *D2K Interface to Standard Content Repositories: Integrating D2K Data Analysis with Tupelo Storage*. NCSA Request For Comments, 2006. https://ncsapoint.ncsa.uiuc.edu/alg/Shared%20Documents/Forms/AllItems.aspx?RootFolder=%2falg%2fShared%20Documents%2fRFC%2fRepository&View=%7b22EC86B9%2d7D34%2d40CD%2d80D2%2dD083BF873467%7d
5. Myers, James D., Thomas C. Allison, Sandra Bittner, Brett Didier, Michael Frenklach, William H. Green Jr., Yen-Ling Ho, John Hewson, Wendy Koegler, Carina Lansing, David Leahy, Michael Lee, Renata McCoy, Michael Minkoff, Sandeep Nijsure, Gregor von Laszewski, David Montoya, Carmen Pancerella, Reinhardt Pinzon, William Pitz, Larry A. Rahn, Branko Ruscic, Karen Schuchardt, Eric Stephan, Al Wagner, Theresa Windus, and Christine Yang. *A Collaborative Informatics Infrastructure for Multi-scale Science*. In *Challenges of Large Applications in Distributed Environments (CLADE) Workshop*, 2004, 24-33.
6. NCSA. *Cybertechnologies*. 2006, http://www.ncsa.uiuc.edu/Projects/cybertechnologies.html.
7. NCSA, *Design Principles for Cyberenvironments (in preparation)*. NCSA, 2006.
8. Sun Microsystems, Inc. *Java Architecture for XML Binding (JAXB)*. 2006, http://java.sun.com/webservices/jaxb/.
9. The Apache XML Project. *Xalan-java Version 2.7.0*. 1006, http://xml.apache.org/xalan-j/.