
D2K Support for Standard Content Repositories: Design Notes

Request For Comments

Abstract

This note discusses software developments to integrate D2K [1] with Tupelo [6] and other similar repositories. D2K itineraries will access the repositories through standard modules or options to modules. The D2K infrastructure, toolkit, and distributed computing facilities may also be extended. This note considers ideas for new capabilities for D2K.

Revision History

Latest entry <i>at the top</i> please			
Version	Author/Contributor	History	Date(yyyy-mm-dd)
003	Robert E. McGrath		2006-07-17
002	Robert E. McGrath		2006-07-14
001	Robert E. McGrath	Initial version	2006-06-20

Contents

1.	Introduction	3
2.	Background: Repository, Repository Access, Tools	3
3.	D2K Support: D2K Modules.....	4
3.1	Extensions of Existing Modules: "File Access"	4
3.1.1	"Input File"	5
3.1.2	"Output Files"	6
3.1.3	Summary.....	7
3.2	Metadata Operations: "New" Features.....	7
3.2.1	Writing Metadata.....	8
3.2.2	Read and Search	8
4.	D2K Infrastructure Features	8
5.	D2K Toolkit/Environment Features.....	9
6.	Other Operations	9
7.	Web, Grid Service Features	9
8.	Sketch of Implementation	9
9.	Summary.....	10
10.	Acknowledgements.....	10
11.	References.....	10

1. Introduction

D2K ([1]) 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 [5].

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.

This work assumes the existence of general purpose infrastructure. Specifically, we assume that there are one or more repositories with standard access methods to store, annotate, find, and retrieve objects. In addition, we assume that there are tools to manage and query the repository.¹ (With the introduction of the proposed technology, D2K itineraries may become important tools for managing and using repositories.

D2K support has a variety of ramifications, which can be viewed as possible phases of development. Each target has its own value and challenges.

This note discusses the following development targets:

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

In the following sections, these features are discussed. Analysis of the current set of core IO modules, we can classify the capabilities into three categories:

1. extensions of current modules, especially, file access methods
2. metadata operations, which may have no direct analogs in current modules
3. other new capabilities, such as versioning and bulk operations

2. Background: Repository, Repository Access, Tools

We assume that there will be one or more standard repository access protocols, to access a variety of distributed stores. At NCSA, the Tupelo project is designing a high performance, large scale repository to meet these requirements ([6]). Other projects may wish to use WEBDAV [2] or even plain read-only Web services. And we expect some D2K users to work with local files, i.e., using the local file system as a repository. (Many D2K applications use databases as stores as well—we do not propose to replace this functionality, merely extend the range of non-database capabilities.)

We initially focus on the WEBDAV protocol [2], as discussed in [5]. The essence of the reasoning is that WEBDAV is robust and widely available today, and will be supported as an access method for Tupelo [6].

¹ With the introduction of the technology discussed here, D2K itineraries may become important tools for managing and using repositories. However, the goal is to interoperate with other general purpose tools.

However, we do not want to “lock in” limitations of WEBDAV, so care should be taken to have a general model that can expand to future services, such as the Java Content Repository [3].

We want to interoperate with many repositories, and want to be able to use generic tools. E.g, for WEBDAV, we can use standard tools to access and manage data outside of a D2K application itinerary.

3. D2K Support: D2K Modules

One area of development is user-visible D2K modules. In the current releases of the D2K Toolkit there are a number of standard modules that enable the user to input data from files and save objects to files. Table 1 lists some commonly used modules. We would like to extend the current behavior of D2K to easily input from repositories and output to repositories. Ideally, this should be done *without requiring major changes to existing modules and itineraries*.

Table 1. Examples of Existing D2K Modules

Example modules:	Description
...core.io.file.getInputFileName	Get input file, output as a string
...core.io.file.input.Create*Parser	Input a file name (string), create parser module, parser opens file as buffered input stream.
...core.io.file.input.GetTextFile	Input a file name (string), module opens file for random access read
...core.io.file.input.XMLtoGraph	Input a file name (string), module opens file for read
...core.io.file.output.file.WriteTableToFile	Input a file name (string), module opens file for write, incrementally appends lines of text

In general, can be done by creating new versions of the standard modules that input from files and output to files. The older modules would be deprecated, to encourage migration to the new capabilities.

This work raises a number of technical problems that must be addressed. These include:

- I/O issues such as incremental read and write, random access versus streaming access, cache management
- Synchronization, locking, versioning
- General mechanisms to handle many flavors of metadata
- Details of reverse engineering existing modules, infrastructure, and toolkit (and minimizing the cost of adoption.)

The following sections sketch the three capabilities, along with design questions.

3.1 Extensions of Existing Modules: “File Access”

WEBDAV and similar protocols support ‘get’ and ‘put’ operations that are analogous to “read from file” or “write to file”. Ideally, we would like to replace/extend everything that uses “InputFileName” with a more generic version that will (as seamlessly as possible) work with either a local path or a URL to a stored object.

The first design issue is to note that moving from private local files to potentially shared distributed objects will require more information to enable access. Access an object in a repository requires:

- A URL
- Access control information
- Exception handling associated with the remote access or local access

Thus, every module that currently uses a file name (a string) will need to be modified to use a URL, and the access code will need to manage the authentication and exceptions. Table 2 suggests two design alternatives.

Table 2. Ideas on object references

How to transparently handle both local file names and URL references

One idea: Make the filename relative to some repository plus path (possibly plus authentication). The default is null, which gives the current behavior (I think). In this approach an open question is, how to pass this to consumers.

Another idea would be to modify all the modules that use path names to recognize URLs, and use WEBDAV (e.g.) to get the data. The URL could be input using the same "InputFileName", though we would want to document the new feature. In this approach, the open question is what the URL's should look like, and how can the receiver know what access methods to use?

A second issue is that, in the current standard modules, the consumer of the data is responsible for opening the file in a way that matches its I/O pattern. Specifically, modules that do multiple passes on input data, or append on output, will open files using appropriate buffering methods, e.g., Java buffered file input or output.

However, remote access to objects does not necessarily support the required access pattern. For example, HTTP usually supports file copy, but has little or no partial I/O. Fetching data via an HTTP connection is not random access, and putting data via HTTP does not support append or random access. Therefore, it will be necessary to assure that the I/O requirements of the consumer are met by the remote data mechanism. For example, if the data must be randomly accessed, it may be necessary to transparently cache the whole file in a local store, which is used by the consumer.

This challenge argues for a specialized I/O module, through which consumers fetch data. This would replace code in existing modules, substituting a generic input or output stream. For example, in a module that current receives a file name as an input, and opens a file to read, would be changed to accept an input stream as a parameter. A new module would be inserted to manage the file or remote open, which would output an InputStream. Note, though that there will be hidden dependencies on the location of the input module and the consumer: it would not be safe to distribute them to different proximities.

A third issue is that the current modules assume the data is private to the user. Using data from a potentially shared repository will require synchronization and access control. Some of this may be handled transparently by an I/O module, but some aspects will inevitably require input from the modules or user. Also, synchronization and access control must implement policies, e.g., for a community, or for a specific repository. There are many issues that need to be handled here.

3.1.1 "Input File"

The current design of the standard D2K modules that read data from files is to have two or three modules. For parsing data, there are typically 4 modules:

1. Input File Name – outputs a string (file or path)
2. Create*Parser – opens the file, creates a parser, outputs parser
3. parser (hidden) – opens file, outputs parsed data
4. Parse module – inputs parser, calls parser to get data

Figure 1 shows a fragment of an itinerary that parses text data. The parser is created by the middle module and passed to the right hand module in Figure 1.

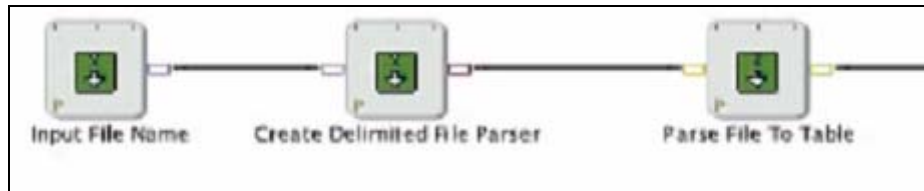


Figure 1

There are variations on this approach, such as ReadFile, ParseXMLtoGraph. The ReadFileNames from T2K core and a similar module in M2K inputs multiple files, e.g., from a directory.

In all the cases, though, the consumer receives a filename and opens an input stream.

We propose to change this design, to separate the I/O from the user of the input stream. This will enable a specialized module to open either a local file or a remote object, isolating the consumer from the details of the I/O mechanism.

{ illustrate }

This will require redesign of a number of widely used modules, which will be deprecated in favor of new, more general modules. *This is a major redesign, but need not interfere with older code.*

Problems to be handled:

1. need a general "getURL" class that will work for any repository (for now, assume WEBDAV).
2. Probably need the equivalent of a "connection" object for an open repository, from which to pull the data when needed. Default would be to output and input file stream (??)
3. need a new, more general "get input files" module that can handle URLs and files. This should also be a "multi file" input, and should handle authentication if needed. Ideally, the module should default to the simple behavior of the current Input File Name when accessing local files.
4. Need to revise existing modules to use the input stream rather than file names.

Possible issues:

1. there are many kinds of Java input streams, we may need variations, or else modules that convert from input stream to, say, object input stream.
2. Some consumers require random access, others can handle one-pass through a stream. Therefore, for a remote object we need 2 options:
 - a. Stream
 - b. Copy to local file, stream from FIS
3. Note: can input stream be passed outside of current VM? (E.g., in distributed itinerary?)
4. Other??

3.1.2 "Output Files"

There are several modules that write objects to files, including WriteTableToFile, OutputSerializedObject. These modules take two inputs, a file name (string) and the object to write to the file.

Again, we would like to generalize this so that the output could be a local file or a URL for a repository location. As with the input operations, this will require redesign to have a module to get the URL or file name, another to open and return an output stream, and revising the consumer modules to use the output instead of the file name.

Issues:

1. Many services support file upload but don't support partial I/O. Some data producers expect to be able to append and sometimes to random access the output. WebDAV upload don't really support stream output, though JCR and others may do so. Therefore, will need options for
 - a. Write to local file, upload when finished
 - b. Write to stream (receiver will manage)
2. Many issues of synchronization, access control, locking, etc. are raised by writing to remote repo. Instead of private local files. E.g., when creating an object, need to have an identity on server. How to do this in a non-intrusive way?
3. Passing around output streams
4. Others?

3.1.3 Summary

Extending the "file I/O" to work with remote file access will require a few simple changes, and some challenging additions.

First, we need to isolate the actual access from the consumer. I.e., instead of passing a file name to a parser, we should have a generic 'InputData' (or 'outputData') module that manages the file or remote data access, passing an Input (or Output) stream to the consumer.

Second, the I/O module must cache a local copy of a remote file, to allow random access and appending.

Third, the I/O module must manage authentication and network errors.

Several design issues must be considered:

1. How should the URL+path+auth information be collected and passed? How should local and remote accesses be specified.
2. How should the consumer specify local cache or not?
3. How should synchronization be handled?
4. What are the restrictions when passing IOStreams, e.g., modules that cannot be distributed

3.2 Metadata Operations: "New" Features

Part of the value added for using the repository and protocols such as WEBDAV is the additional metadata available. Since few existing D2K modules annotate objects, and searching by attributes is rarely done, we are breaking new ground in these user visible modules.

Within the infrastructure, however, there is a lot of metadata but no uniform treatment of it. Itineraries and modules are managed via local configuration files and Java objects. Runtime parameters and statistics are managed through Java objects. Persistence is achieved through data bases or local (private) files. Many of these features could (and probably should) use a generic repository, though implementation is likely to require significant reengineering of many parts of the code.

Reading metadata is similar to querying. For example, there should be a "find file(s) where <x>" feature in D2K, similar to a standard Java file finder. There should be a generic "get properties associated with this URL". The requirements for this need to be investigated, most likely, this can be built from generic code.

The NCSA Cyberenvironments roadmap calls for development of standard tools that automatically collect process metadata. In the case of D2K, this would take the form of automatically publishing notations of data movement, processing statistics (iteration number), and so on. In addition, there might be user visible features, such as, add annotation, and "save object to repository" (write the contents plus some

metadata in a standard way). These changes involve significant reverse engineering of existing code, yet to be determined. This is discussed in Section 4 below.

3.2.1 Writing Metadata

We should be able to create new modules that can write metadata to a URL. In the case of WEBDAV, this means writing properties to an object. In the more general case, the object may be just an address (e.g., in JCR, a “node” can have metadata but no “content”.)

The D2K module needs to collect the URL and the metadata to create or write. This can be done in several ways:

1. As D2K properties specified in the module
2. Through a GUI that lets the user set values
3. Through input from other modules, e.g., a list of properties to store

Important Challenges:

1. will need to have a model of how this metadata is represented in D2K. This should be considered carefully. WEBDAV has a simple, but limited model. Ideally, we should have a more flexible model that will work with other repositories. We need to consult with, e.g., Tupelo.
2. need to be able to discover existing metadata,
3. will need to implement access controls, create, overwrite or not, etc.

3.2.2 Read and Search

Here we can envision an extension of the Input Files discussed above, in which we provide an enhanced “Browse” button that can browse repositories, displaying metadata, filtering and searching by attributes, and selecting.

In addition to the need for a model of metadata, we will need a significant amount code to support the browser. Note that DSI has implemented something like this, so we should consider reusing it. However, the DSL model for metadata may or may not be what we want. We need to consult with others about this.

4. D2K Infrastructure Features

In addition to the user visible features, we may envision new features inside the D2K infrastructure. The envisioned NCSA Cyberenvironment calls for the automatic collection of “process metadata” from workflows and applications. In the case of D2K, this would take the form of instrumentation of the data flows and module statistics, to optionally emit descriptions of each step of computation, a la the recent extensions to Kepler [4]. Technically, this will involve changes and extensions to many parts of the (invisible) D2K infrastructure.

Another goal is to support posting and polling for output and input, to enable looser coupling of applications. The user visible modules discussed above could be extended further to use generic services to discover, poll, and post data in standard ways.

5. D2K Toolkit/Environment Features

The D2K Toolkit itself manages a computing environment that has an implicit repository of modules, itineraries, local data files, and output models and visualizations, as well as “proximities” and settings for run time environment. This environment should be redesigned to be able to use a distributed (and shared) environment instead of local files. Technically, this involves revising the relevant parts of the toolkit and infrastructure to be able to use repositories as well as files.

These features raise complex technical issues of caching and synchronization. For example, if modules and itineraries are shared, there will need to be a protocol for safely managing updates, as well as access control, and other features.

The scope of these changes is yet to be determined.

6. Other Operations

Taking WEBDAV [2] as an example, we see there are a number of features that may be of interest.

- access control, locking, and versioning, at least on servers that support it.
- bulk operations,
- select by attribute (rather than specific file name)
- Coordination via global messaging, e.g., an Enterprise Service Bus such as Mule [8].

We need to consider how to use these features.

7. Web, Grid Service Features

The D2K Server and D2K Web Server provide capabilities for spreading computation across multiple systems. Within these multi-tiered configurations using local file systems as a repository is hazardous and not recommended. However, distributed repositories offer a means for creating both a distributed computing environment and for managing data objects.

8. Sketch of Implementation

Need “repository client” interface, with implementations for different protocols

Analogous to ‘connection’ objects, though has to be heavier weight due to ‘dumbness’ of services.

Note: extends file I/O, but:

- Not local
- Not private

Need coherent model of I/O...

Features

- ‘client’ for remote repositories
- Manage connections, I/O. E.g., web dav (HTTP), file transfers
- Manage local cache of files, as needed
- Manage synchronization?

Design issue:

One object per connection?

One object per session? Synchronize across multiple modules

Another tier? E.g., a local cache manager, subscribe to it?

9. Summary

This note has sketched a set of changes to D2K to enable support for distributed repositories.

10. Acknowledgements

Thanks for discussion and advice from: Joe Futrelle, Jim Myers, Xavier Llorà, Bernie Achs, Loretta Auvil, Fang Guo

11. References

1. Automated Learning Group. *D2K - Data to Knowledge*. 2005, <http://alg.ncsa.uiuc.edu/do/tools/d2k>.
2. Goland, Y., E. Whitehead, A. Faizi, S. Carter, and D. Jensen, *HTTP Extensions for Distributed Authoring -- WEBDAV*. IETF RFC 2518, 1999.
3. Java, *Content Repository API for Java™ Technology Specification*. Java Specification Request 170 version 0.16.2, 2005.
4. Kepler-project.org. *Kepler Provenance Framework*. 2006, <http://kepler-project.org/Wiki.jsp?page=KeplerProvenanceFramework>.
5. 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>
6. NCSA. *Cybertechnologies*. 2006, <http://www.ncsa.uiuc.edu/Projects/cybertechnologies.html>.
7. NCSA, *Design Principles for Cyberenvironments (in preparation)*. NCSA, 2006.
8. SymphonySoft. *MULE Universal Message Objects*. 2006, <http://mule.codehaus.org/>.