10.10

AN ARCHITECTURE FOR THE LEAD DATA REPOSITORY

Doug Lindholm*, Anne Wilson, Tom Baltzer
Unidata Program Center / UCAR

1. INTRODUCTION

The Linked Environments for Atmospheric discovery (LEAD) is a multi-institutional Information Technology Research effort funded by the National Science Foundation (NSF). The goal of LEAD is to create a Grid and Web Service based framework to support mesoscale meteorology research and education.

LEAD presents unique challenges integrating large data volumes from real-time observational systems as well as those that are dynamically created during the execution of adaptive workflows. The LEAD Data Repository (LDR), which manages these data, must be able to autonomously handle storage and retrieval requests generated by the LEAD orchestration in addition to directly satisfying user requests.

During the design process, the responsibilities of the LDR were broken down into separable concerns and constructed using loosely coupled implementations of abstract interfaces. The result is a flexible framework that supports the needs of various scientific data communities.

2. DATA REPOSITORY COMPONENTS

A data repository, much like a file system, has a number of responsibilities to fulfill. It must enable a user or an application to store and retrieve data responsively and reliably. There must be a data discovery mechanism such as browsing or query support. The data repository must also manage metadata for the data products and coordinate a number of other technical capabilities to support these requirements.

Another key feature that must not be overlooked is ease of use. LEAD users and others in the Unidata community have a wide variety of needs and expectations. The data repository framework must be flexible to support a variety of use cases from students in a classroom to developers of sophisticated scientific workflows.

It is helpful to factor out an abstraction of the data repository responsibilities to support such flexibility. By defining modular functional components, various loosely coupled implementations can be plugged in to support the needs of the target community.

2.1 Storage Locator

Some part of the data repository subsystem must be responsible for managing the physical storage resources for the data. The role of the Storage Locator is to provide the data repository with a physical URL to a location for storing the data. This could simply be a preconfigured scratch directory or a more complex capability that considers user authorization, optimizations, and desired access protocols.

2.2 Data Mover

Once a location has been identified, the Data Mover has the sole responsibility of moving the data from its source location to its destination. This could be a simple file copy or it could involve a complex subsetting or aggregation of the data. It is useful to think beyond a simple file based abstraction.

2.3 Metadata Generator

To support data discovery and use, there must be metadata associated with the data. If metadata is not available by other means, an automated capability can be used to harvest metadata from the data products. The Metadata Generator is responsible for gathering the minimum set of information about the data to support the discovery and use requirements of the data repository.

2.4 Metadata Crosswalk

To promote a loose coupling of components, the data repository design should not require a consistent metadata language across all modules. A Metadata Crosswalk can be provided to translate from one metadata schema to another to ease the barriers of interoperability.

2.5 Cataloger

The Cataloger is responsible for maintaining the metadata for the data repository's data holdings. The Cataloger should provide a user interface for browsing or querying the contents of the repository and in turn provide access information to the user or application.

2.6 Unique ID Generator

**Corresponding Author Address:*
Doug Lindholm,Unidata,PO Box 3000, Boulder, CO 80307
lind@unidata.ucar.edu

It is useful to name data products with a unique identifier as opposed to a physical URL. This supports the notion of replicas and data access transparency. The Unique ID Generator is responsible for creating an identifier that is unique within the scope of the data repository.

2.7 Name Resolver

The Name Resolver is responsible for providing a mapping between the Unique ID and a physical URL.

3. THREDDS DATA REPOSITORY FRAMEWORK

To support the varied needs of the LEAD and Unidata communities, we have designed the THREDDS Data Repository (TDR) framework that encapsulates the modular components of a data repository behind a simple interface or façade. The TDR defines an interface for each component described above and provides simple high level interfaces (such as putData and getData) which in turn delegate to the component interfaces to perform the desired task. As a result of the separation of concerns inherent in the design, various implementations of these components can be plugged into a TDR instantiation.

The goal is to provide an easily usable interface to a data repository subsystem. As a result, some compromises have been made to keep the higher level TDR interface simple. The interfaces are designed to support roughly 80% of the use cases – the most common ones. However, the TDR does expose the underlying native implementations of the functional components so there is a hook to access the other 20% by more sophisticated users.

We are currently developing functional component implementations and TDR configurations to support three unique use cases. Although the requirements of each system vary significantly, this framework promotes a great deal of code reuse.

3.1 Local THREDDS Data Repository (TDR-lite)

TDR-lite packages implementations of the data repository functional components that support storing data in a local space such that the data can later be accessed on that system with no networking requirements. One such scenario would be an instructor who wants to package up a learning module, including data, on a laptop to be presented in a classroom setting without network connectivity.

TDR-lite provides the user with a lightweight desktop client application that enables data discovery (e.g. from an existing THREDDS Data Server) and provides the option to store the data to a personal TDR workspace. The client then invokes a local Java implementation of the TDS that orchestrates the other configured components. In this configuration, the Storage Locator simply returns a URL to a pre-configured directory and the Data Mover does a simple http download to this location. The Metadata Generator will ensure that metadata is available to be cataloged by the Cataloger. For this scenario it is sufficient to store the physical URL in the metadata so a unique ID and Name Resolver is not needed. Because this implementation is local to a single user, concerns of security and scalability are greatly reduced.

The user can then access the data via a hierarchy of logically named data products (much like a file system) in the TDR THREDDS catalog using THREDDS enabled clients such as the Integrated Data Viewer.

3.2 THREDDS Data Repository Service (TDR-classic)

The Classic implementation of the TDR is as a web service that supports remote data ingest and access for multiple users. One such scenario would involve a researcher putting together a case study to share with the community. In this scenario, security and scalability become a greater concern and the functional component implementations must be suitable to meet those needs.

The TDR service can be invoked by a user via a web interface or programmatically by another application. The required functionality is much like that of the TDR-lite but the implementations must be able to deal with authentication, authorization, and handling multiple users. The data can in turn be accessed by others via a THREDDS Data Server.

3.3 LEAD Data Repository (LDR)

The LEAD system provides some much bigger challenges of scalability. Not only will the TDR be offered as a service, but the individual functional components can also be implemented as distributed services. The Storage Locator will make use of sophisticated resource monitoring. The Data Mover will be implemented using NCSA's trebuchet software which supports gridftp and other data transport mechanisms. Metadata can be harvested from data products via the NetCDF Common Data Model. A Crosswalk will translated the generated metadata from the THREDDS schema to the LEAD

schema. The Cataloger will then send the metadata to the myLEAD catalog developed at Indiana University. LEAD needs to be able to support replicas so unique IDs and name resolution are a must. The Replica Locator Service (RLS) is being considered as an implementation of the TDR Name Resolver interface. LEAD's grid security measures will be applied throughout this implementation.

CONCLUSION

We have described here an architecture for the THREDDS Data Repository (TDR) -- a data repository framework suitable to meet the needs of a wide variety of communities. The most ambitious needs of the LEAD project are driving the development. The TDR is designed as a collection of highly configurable, loosely coupled modules that enable plugging in alternate implementations to support the needs of other communities. Through the use of open source software, code reuse, and an extensible framework we hope the TDR can grow to support additional communities.

REFERENCES

THREDDS Data Repository (TDR)
http://www.unidata.ucar.edu/projects/LEAD/ThreddsDataRepository.html

THREDDS
http://www.unidata.ucar.edu/projects/THREDDS/

Common Data Model (CDM)
http://www.unidata.ucar.edu/software/netcdf/CDM/

Integrated Data Viewer (IDV)
http://www.unidata.ucar.edu/software/idv

LEAD
http://lead.ou.edu/

Unidata
http://www.unidata.ucar.edu