

The Language Data Repository: Project Abstract

Neal Audenaert • Texas A&M University • neal_audenaert@acm.org

Data Types: Lexicon, Common

Functions: Store, Create, Query

Acknowledgments

I would like to thank Lisa Ann Lane who has served as my advisor and mentor on this project for the past two years. Her frequent advice and numerous suggestions have been invaluable. None of this would have been possible without her help. I am also greatly indebted to a team of five students at Texas A&M whose hard work during this past fall has helped to begin putting together all the pieces of the LDR system: Ryan Saunders, who help lead this team and who has been the individual responsible for the design and development of the client module and served as the technical lead for the project, Matt Benton, Travis Reed, Che Wilcox and Blaine Young. Thanks to all for your tremendous efforts.

Abstract

The Language Data Repository (LDR) project is working to develop a software system capable of storing transcripts and recordings of spoken language data and capable of hosting software tools to aid in the analysis of that data. The proposed software architecture will enable multiple researchers to store linguistic data from multiple languages on either local machines or non-local machines that can be accessed via a network by multiple users simultaneously. Such a software system will offer two main improvements over current methods of recording transcripts of linguistic data. First, by utilizing machine-readable storage, it will enable linguists to use computational tools to aid in linguistic analysis, thereby reducing some of the “grunt work” traditionally associated with this aspect of research. In turn, there will be an increase in the ability to quickly and accurately test and evaluate linguistic hypotheses. Second, the LDR system, by providing a common format for the representation of data and by reducing the need to obtain physical access to the data, will enhance linguists' ability to document their research and share their results with a greater number of colleagues than previously possible. This paper will present the concept behind the development of the Language Data Repository, describe the ways in which the LDR system can be extended to meet user-specific needs and outline some of the critical factors in the development of the LDR.

Design Considerations

The LDR is intended to be a general-purpose computational system for the archiving and analyzing of linguistic data, but what exactly does this mean? In examining the needs of the linguistic community, a short list of core requirements for such a system has been identified to answer that question. The six items listed below have served as the primary catalyst behind the development of the concept for the LDR.

- The system must provide for the reliable archiving of primary and secondary data collected from any spoken human language.

- The system must support multiple users, working together and independently, located both centrally and disparately while accurately attributing the work of the individual researchers who collected and catalogued the data.
- The system must allow access to data stored in multiple physical locations.
- The system must allow for analysis from many different theoretical perspectives.
- The system must be platform independent.

Two issues in the above list are worthy of special note. First, the second and third items cannot be satisfied under the traditional paradigm of desktop computing in which an application is run on a stand-alone computer. To meet these requirements, the LDR will follow a network paradigm of computing. Under this paradigm, the network is the computer, and the machine on a user's desk is simply his or her access to the network. Data is then stored somewhere on the network (or rather in many places on the network) and loaded into main memory somewhere else on the network, yet accessed by each user from the machine on his or her desktop, while (ideally) the network remains transparent. Second, the fourth item cannot be fully satisfied by a single piece of software for the simple reason that many of the theories that will be needed to analyze linguistic data have not yet been conceived. A single program might try to provide a theory neutral approach, but then it may lose some of the power to be gained by the application of a specific theory to a specific problem. Conversely, a theory specific program may offer the user a uniquely powerful tool for a particular analysis, but disenfranchise those who wish to employ some other theory. In order to meet the evolving analytical needs of the linguistic community, the LDR system will not be a single program but rather an architecture to glue together many separate analysis tools and incorporate them into a single unified system. These tools then can be developed by any software development group and incorporated into the system by the user as s/he needs them.

Design Overview

This section will describe, at a high level, the major components of the LDR system, how these components are involved in the creation, storage and retrieval of data, and how these components allow for the extension of the system. The LDR architecture is composed of client, data server and repository modules. Each module, when instantiated as an executable program, forms a system component that can be distributed to a node on a network and accessed by other components through a well defined application programming interface (API). For the current prototype development, these components communicate via Java's proprietary remote method invocation (RMI). Plans for future development of the system call for the use of CORBA instead of RMI due to the fact

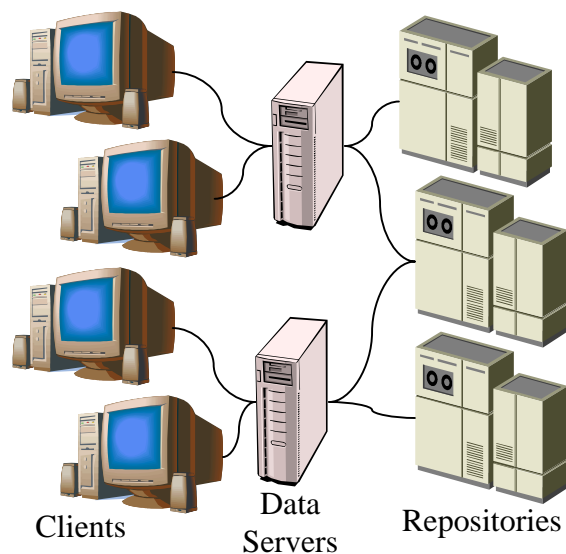


Figure 1: System Modules

that CORBA is much more widely used and that it allows much more flexibility in choosing a programming language to implement a distributed component.

The client is the module that the user interacts with directly. This is the program that the user will execute from her or his desktop computer in order to start working with the LDR system. It will provide the mechanisms for hosting tools, contacting the data server and repositories, and for setting user preferences. The data server module provides a run time instantiation of the linguistic data and acts as a cache between the client and the repository. The data server also provides a layer of abstraction for the repositories, allowing the client to interact with a number of different repositories as though they were a single entity. The repository module provides long-term data storage, and handles the execution of data queries. The API for this module will define the behavior of repository components, while the internal implementation is left unspecified. This allows changes to the underlying storage mechanism to be made with no effect to the rest of the system. The abstraction of data storage implementation provides one of the two primary extension points of the LDR system (the other being the analysis tools) in that it allows an existing corpus to be integrated into the LDR system by building a repository module that is aware of the underlying data storage format for that corpus. The general framework for instantiation of linguistic data as objects at runtime and for the storage and retrieval of that data is described below in the data persistence section.

Representation of Data

The interaction of analysis tools, both with the LDR system and with other tools, as well as the development of repository modules for existing corpora depends on a well-defined, stable description of what exactly comprises linguistic data. For the LDR system this description comes in the form of the LDR data model. This data model will be comprised of three different components:

- A conceptual description providing both textual and graphical description of the data and the relationships between the data.
- An XML based description providing either DTDs or schema. XML representations of data following this set of DTDs or schemas will be used for the exchange of data between components.
- An API describing the classes used to instantiate data objects and the ways in which those objects can be manipulated.

This model will define the atomic and composite elements of linguistic data, the relationships between data, and the constraints that govern that data. A complete, well designed, well documented data model will be critical to the success of the system. Any data that is needed by the linguistic community and that cannot be represented by the system's data model constitutes a serious flaw in the design of the system. The development of such a data model is a tremendous undertaking and will not be feasible without significant input from the linguistic community. It is hoped that work being conducted by those presenting at this workshop and elsewhere will result in the standardization of a general model of linguistic data that can then be implemented by the LDR system. This will save a tremendous amount of time and effort in the development

of the LDR system and help to ensure that it fully meets the needs of the linguistic community.

The Client Module

The client module is the portion of the system that users will most directly interact with. It is responsible for hosting analytical tools and supporting system configurability. The main requirements in developing this part of the system are that it provide easy, powerful access to all aspects of the system to individuals with widely varying technical capabilities and that it allow extensive customization to best meet the user's individual preferences. The user community has been grouped into three categories: general users, power users, and programmers. To address the needs of each category of user, the client module employs a three-layered approach to accessing the functionality of the system. Each layer is intended to meet the need of one category of user and provides complete access to the functionality of the client (to the maximum extent possible for that type of interaction). The first layer, the GUI layer, provides access to client functionality through a graphical user interface. This is intended to allow general users who have little or no technical background to interact with the system. While relatively easy to learn and use, GUI systems can make repetitive tasks tedious. The second layer reduces this problem by allowing users with some technical background to access the system through the use of a scripting language. These power users could then write macro scripts that would automate repetitive, multi-step and frequent tasks. Rather than implement a new proprietary scripting language, the client will utilize an existing scripting language in order to allow it to capitalize on the knowledge of existing developers. The JavaScript language is currently being used due to its large developer community and the availability of tools to integrate it into other programming languages. This scripting language will form one of the core components of the client module and will be used internally in many of the start-up/shut-down routines, to help manage user-configured properties of the system and to integrate tools. The third layer is the application programming interface (API). This layer will allow programmers to develop software to access the system, and is intended to support the development of analytical tools that can be easily plugged into the system while it is running.

The client module is actually composed of a set of managers, each of which is responsible for a distinct area of the client's functionality. Each manager encapsulates the details of implementing the functionality associated with its area of responsibility and provides the corresponding services to the rest of the system. The following managers are currently being developed:

- **LDRClient** – This is the top-level manager, and the executable program that the user will run. It is responsible for creating, configuring, and destroying the other managers. It encapsulates the module's interaction with the operating system.
- **ScriptManager** – This manager integrates the scripting functionality of the client and manages declared JavaScript macros.
- **ToolManager** – This manager handles all the details of hosting plug-in tools.
- **LayoutManager** – This manager handles the usage of screen real estate in order to give an appearance of total integration between tools and visible system components. It serves as the visual platform in which tools display themselves.

- **ConfigurationManager** – This manager handles a user’s preferences and other important information.
- **ConnectionManager** – This manager encapsulates communication with the data server and other clients. It services requests for data objects and connections to other machines.
- **SecurityManager** – This manager restricts access to sensitive functionality in order to reduce the risk that 3rd party tools might introduce hostile code to the system.

The Persistence Framework

The persistence framework is the aspect of the system that handles creating, retrieving, and updating data objects that maintain their state over time. This framework has been adapted from a design pattern described by Reese (1997). It views persistent objects as objects that can be saved to a data store, and restored, either individually or as a group of objects (e.g. a data query), from that data store to the state they were in when they were last used. It also provides a mechanism for creating new persistent objects and for transaction management (i.e. managing groups of objects in such a way that modifications to one object can be saved, if and only if all modifications to the group can be saved).

The framework being developed for the LDR system allows those developing data classes (i.e. classes to represent linguistic data) to focus their development efforts on representing linguistic data and the operations that can be performed on that data rather than concerning themselves with the mechanisms of the persistence operations. Since the information about the storage technology to be used for data persistence is not fundamentally part of the data being stored, this information should not be included in the data classes. The persistence framework, therefore, should also separate both the data and the operations specific to the underlying storage technology (relational database, object-relational database, flat files or other information retrieval system) from the data classes. This provides two levels of abstraction, isolating the data classes from the persistence operations and isolating the persistent objects from the data storage format used by the data store. Although the actual framework will be distributed across a network, the implementation of the distributed features is omitted here since it is not part of the core persistence operations.

Figure 2 shows a static UML diagram of the persistence framework. The general principle behind this is that each linguistic data class has a corresponding peer class. The data class is responsible for representing linguistic data, the peer class is responsible for saving that data in the format specified by the repository module. In the LDR persistence framework, the class, *Data*, is the abstract super-class for all data classes. It handles the persistence operations for the data classes (except for actually assigning the attributes of a data class based on restored data). The sub-classes of *Data* will provide the implementation of the LDR data model. The *DataSet* class provides the operations necessary for the management of groups of data, including transaction management. The *LDRRepositoryPeer* is an abstract super-class that defines the general behavior of the peer classes (including behavior that is specific to the LDR repository) and implements

the DataPeer interface. This interface defines operations to restore a data object from the data store, save a data object to the data store and commit or abort a save in progress.

The DataServerManager and the RepositoryManager serve as the chief manager classes for their respective modules (analogous to the LDRClient for the client module). They are responsible for managing execution of data queries, access to loaded data, creation of new data, and other related services. Their primary function in the persistence framework is to serve as factories for data and peer classes. (See Reese 1997 for more information about the factory pattern.) The

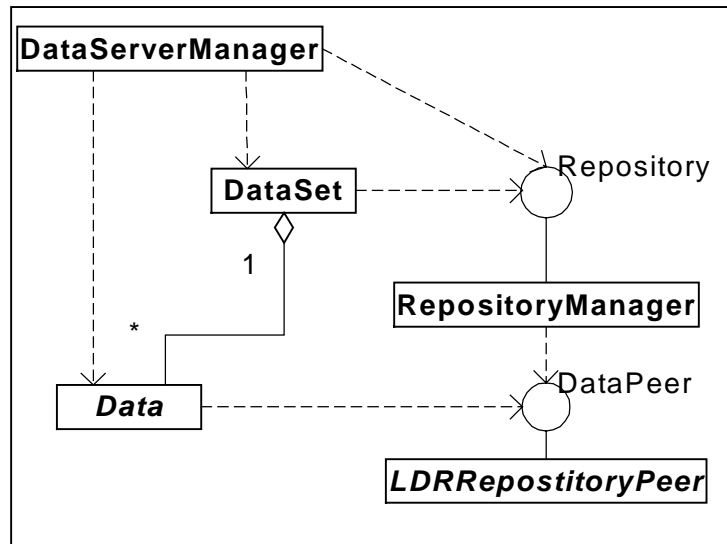


Figure 2: UML Diagram of the Persistence Framework

DataServerManager serves as a factory to create new data objects, restore existing objects (based on the data ID) and create new data set objects from collections of data objects. It also serves as a form of cache, querying the repository only if the data to be restored has not already been loaded from the repository as a result of some other request for data. The RepositoryManager provides similar services for peer classes, creating and returning to the data server new peer classes for each type of data defined by the data model.

Extending the LDR system by adding a different repository implementation can be accomplished by developing a repository manager that implements the Repository interface and a set of peer classes that implement the LDR data model. Each of these peer classes must implement the DataPeer interface (note that these peer classes do not need to descend from a super-class as they do in the LDR repository implementation). A repository implementation may choose not to support particular types of data defined for the LDR data module. In this case calls to the factory or query operations of the repository will send an appropriate error message back if a data server tries to access the unsupported data. It will also be necessary for the repository to provide a mechanism by which the repository may be queried to determine what data is supported. Once the repository and appropriate peer classes have been implemented, the new repository can simply be plugged into the system and accessed by clients through the data server in the same way any other repository would be accessed.

Conclusions

The development of the LDR system is in its second year of design and initial development. The system concept has been established and carefully reviewed and current development efforts are focused on implementing a prototype of some of the basic elements of the system architecture including the persistence framework and the client. These prototypes are being implemented entirely in Java and use RMI to access

distributed components of the system. The prototype repository module will use a PostgreSQL database for data storage. So far, no major technical problems have been encountered and current projections indicate that this initial prototype development will be completed late spring or summer of 2001. While many important architectural issues will be left for a future iteration of the development cycle, the prototypes in development will provide a solid basis for future development of the LDR system and will greatly enhance the ability of users to interact with the development team in order to refine the system. Once these prototypes have been completed, the next stage of research will begin in earnest on designing and implementing the data model. The development and implementation of the data model will provide the first opportunity to thoroughly examine whether or not the system will truly be able to meet its requirements. Although the LDR system is still in its infancy, the outlook is very optimistic. The development is proceeding well, if slowly at times, and the feedback from the linguistic community has been very positive.

References

G. Reese. *Database Programming with JDBC and Java*. O'Reilly & Associates, Inc, 1997.