A Flexible, Evolvable Data Broker

Sidney C. Bailin[1] and Theodora Bakker[2]
[1]*Knowledge Evolution, Inc., 1748 Seaton Street NW, Washington DC 20009, U.S.A.*
[2]*North Shore-LIJ Health System, 175 Fulton Ave # 200, Hempstead, NY 11550, U.S.A.*

Keywords: Data Broker, Information Sharing, Data Integration, Linked Data, Semantic Web, ORCID.

Abstract: We describe a Data Broker application developed for New York University Langone Medical Center. The Broker was designed to accommodate an evolving set of data sources and destinations with little or no additional coding. A schema-less RDF database, currently implemented in OpenRDF but amenable to other implementations, is a key to this flexibility. The independence from a database schema allows the Broker to operate equally well in other institutions and in other applications. Thus, although it was built for one specific purpose, it can be reused as a general-purpose tool.

1 INTRODUCTION

This paper describes a Data Broker application developed for New York University Langone Medical Center (NYULMC). The Broker integrates several sources of data describing different aspects of NYUMC researchers' work. Data sources are integrated with each other and with the Open Researcher and Contributor ID system (ORCID, 2015).

The end-user of the Broker is the organization that wishes to federate, combine, or warehouse information from a variety of its databases. In operates in the background, with a web-based administrator interface for stopping, restarting, and viewing the contents of the database.

The Broker was designed to accommodate an evolving set of data sources and data destinations. In particular, the design attempts to make the addition of a data source or destination as simple as possible, with little or no coding. For this reason, the Broker will work in other institutions, and in other applications besides the exchange of researcher information.

2 REPOSITORY

The Broker stores its information in a repository. The current implementation of this repository uses the Sesame OpenRDF system (Sesame, 2015). We decided to use an RDF triple store as the initial implementation because it allows us to store information without imposing a database schema (Günes et al., 2015). Since we expect to learn increasingly about the required data structures as more data sources are identified and integrated, we did not want to commit prematurely to a database schema. This schema-less design also supports the deployment of the Broker in very different environments.

OpenRDF was chosen as the platform for the RDF triple store because it is open and lightweight. Currently, the Broker uses a native Sesame triple store, meaning the database is provided by the OpenRDF software itself. However, the Broker accesses Sesame through the OpenRDF Storage and Inference Layer (SAIL) application program interface (API). Since there are many large-scale and higher performance triple stores that implement the SAIL API, should the scale of the Broker at some point exceed what the native Sesame triple store can support, one of these larger-scale systems can be easily substituted for it, without any change to the Broker code.

In implementing the repository, however, we also recognized that some other form of database management system (DBMS) might eventually be desired. This might be a relational DBMS, or one of the many no-SQL alternatives that are gaining currency in the database world (NOSQL, 2015). For this reason, the Broker's repository is defined as an abstract interface, of which the Sesame repository is just one possible implementation. This design will

Bailin, S. and Bakker, T..
A Flexible, Evolvable Data Broker.
In *Proceedings of the 6th International Workshop on Software Knowledge (SKY 2015)*, pages 41-45
ISBN: 978-989-758-162-5

facilitate the replacement of OpenRDF with another, possibly non-RDF DBMS if that is desired at some point. Of course, there is then a trade-off between the flexibility provided by RDF and the possibly more rigid commitments of an alternative DBMS.

3 CONFIGURING SOURCES AND DESTINATIONS

The Broker converts data received from sources into its own data structures, and converts its own data structures into those expected by the destinations. In this sense, the Broker plays a role analogous to the well-known software *mediator* pattern (Gamma et al., 1994). Unlike the mediator pattern, however, the Broker is an application in its own right, mediating between applications rather than objects within a single software system.

There are three categories of data structures managed by the Broker: 1) The Broker's own structures; 2) Data source structures; 3) Data destination structures. Each source and destination is defined to the Broker in terms of the data structures the source or destination uses.

The Broker's own structures are required in order to provide a persistent store independent of the sources and destinations. Since there is potentially a lot of overlap between source and destination information, simply aggregating them into a persistent store would be highly redundant.

The Broker repository code does not, however, reference the Broker structures explicitly. Instead, it makes extensive use of Java reflection to reference classes, methods, and fields anonymously, so that the repository code need not change even if the structures change (Oracle, 2015a). This combination of Java reflection and schema-less RDF provides the core flexibility of the tool. It is complemented by the use of the Java-XML Binding (JAXB) to generate Java classes automatically from XML Schema documents specifying the source and destination data structures (Oracle, 2015b). JAXB, reflection, and RDF allow the Broker to evolve without additional coding.

The Broker requires some basic information about each source or destination:

- How information will be moved between the Broker and the source or destination;
- How the source or destination data are structured;
- How to map between the source and destination data and the Broker's own structures.

One provides this information through a set of start-up parameters.

The question of how information will be moved breaks down, in turn, to the following questions:

- *What vehicle is used to communicate with the source or destination?* For example: web-based communication using HTTP, or some form of remote program call, or direct file or database access;
- *Which party is the active party in the exchange?* For a data source, the Broker can poll the source periodically to check for updates, or it can let the source contact the Broker with updates as they occur. For a destination, the Broker can contact the destination when there are updates, or it can wait for the destination to request them.

These questions are answered by choosing a software pattern for implementing the source or destination. The patterns are implementations of the Broker's abstract interfaces IDataSource and IDataDest, respectively. These interfaces are analogous in purpose to those of the same name found in other frameworks, such as .Net, but they are otherwise unrelated. Here, for example, is IDataSource:

```
public interface IDatasource<T_Struct,
T_IO> {
  public String getSourceName();
  public void setSourceName(String s);
  public XMLGregorianCalendar
    getLastUpdatedDate();
  public void
    setLastUpdatedDate(
       XMLGregorianCalendar date);
  public String dateForQuery();
  public Lists acceptData();
  public Lists acceptData(String d);
  public T_IO readData();
  public T_IO readData(String d);
  public T_Struct unmarshall(
    T_IO marshalledData);
  public Lists transform(
    T_Struct data);
  public Lists transform(
    T_Struct data,
    String researcherId);
}
```

The Broker provides several default implementations of these interfaces, sufficient for most purposes.

Besides the source or destination's data structures, each source or destination requires that a transformation be defined between its structures and the Broker's. The transformation enables the Broker to transform a source's structures into its own core structures, and to transform its own core structures into a destination's structures.

The transformation can be written directly in Java, but for the majority of cases in which fields are simply mapped to other fields, with minimal change, the Broker supports a simple XML-based specification of the mapping.

3.1 Generating Source and Destination Structures from XSDs

The Broker's classes representing a source or destination can – and if possible, should – be generated by the JAXB xjc tool from an XML Schema Definition (XSD). This allows one to create and modify sources and destinations without directly writing Java code. Figure 1 illustrates the role of the XSD in generating the Broker's source and destination structures, and the role of the XML mapping file in mapping between those structures and the Broker's own structures.

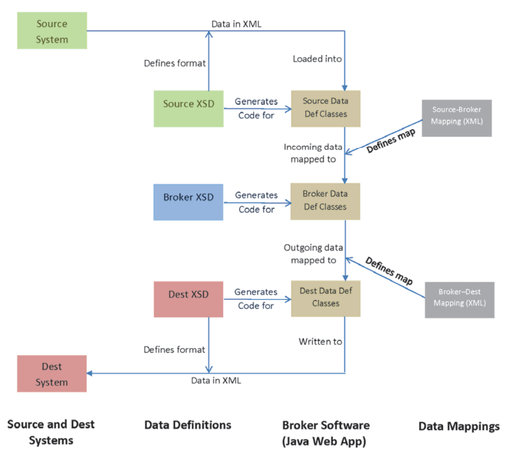

Figure 1: XSDs can be used to generate the Broker's own structures as well as those the Broker uses to represent sources and destinations.

4 DATA EXCHANGE DYNAMICS

The Broker can interact with a source or destination either actively or passively. The active approach involves the Broker periodically polling the data source for any updates, and for a destination, periodically sending any updates. The passive approach involves the Broker waiting to be called by a data source with updated data, and for a destination, waiting for a request for updated data.

Polling by the Broker is accomplished by two threads that run in the background, one for sources and one for destinations. Each poller is configured with a "sleep interval," which is the number of milliseconds between poll attempts. The sleep intervals are set by start-up properties.

If the source or destination plays an active role in sending information to or requesting information from the Broker, it addresses these requests to a servlet interface provided by the Broker.

5 DATA MAPPING

The Broker supports a simple method of defining a mapping from source data structures to the Broker's core structures, or from the Broker's core structures to a destination's data structures. The mapping is defined in an XML file whose path is given by a start-up property.

The root element of the XML file must be called *<assignments>,* and within this, the mapping is defined by a series of *<assignment>* elements.

Each assignment element must contain a *<from>* element and a *<to>* element. The *<from>* element is a fully qualified path of a field in the data source's structures, starting from the top-level class. The top-level class is the class with which the data source is instantiated from one of the pattern classes.

The *<to>* element is, typically, a relative path, whose context is taken from the mapping of the next highest assignment.

The "to" path may contain a sequence of dot-separated fields, so that the source field is mapped to a Broker field at a lower level in the Broker structure. Usually, the intermediate components along the path should be single valued. There are cases in which an intermediate component must be a collection, but this should only be done with great caution. When the Broker is mapping a "from" value to a "to" value, it handles such cases by creating a new instance for the intermediate component, and adding it to the collection represented by that intermediate field. The Broker has no way of determining whether there is an existing element of the collection that should be used instead. When the Broker encounters a collection component inside a "to" path, it issues a warning to this effect.

5.1 Data Mapping Mismatches

5.1.1 Single vs. Multi-valued Fields

If the "from" field is multi-valued (that is, it is an instance of a Java Collection), then the "to" field must also be multi-valued. It is, however, permitted to map a single-valued field to a multi-valued field. In this case, the Broker will simply create an element within the Collection to represent the mapped value.

5.1.2 Structural Mismatch

There may be cases in which a field in the source is not mapped to a field in the immediate mapped context. That is, the mapping may have to move certain fields around because of a mismatch in the source and Broker data structures. To support such cases, the Broker supports two additional notations.

A ".." tells the Broker to go up one level from the mapped context. Any number of "../" components may be used, followed by field specifications starting down from the resulting structure.

The second alternative notation allows one to specify an absolute path. This can be useful if one wants to avoid mapping upward through many levels using "../" to reach the root level, but simply want to start the path at the root level, proceeding down to some other field in the target structure.

5.2 External Keys

A data source might not provide all of the relevant information about a referenced structure, but only some of it. To accommodate this fact, the mapping notation allows one to identify a data source field to use as a key to retrieve an entity from the Broker's repository.

For example, in the NYULMC-ORCID application, the source structure may include a description of a researcher's grants. Each grant description may include the name of the grant's funding agency, but it will not include all of the funding agency's Broker information. When we update the researcher's Broker information, we want the Broker to identify the funding agency by name and then use the actual Broker entity representing the funding agency. To do this, we include a <*key*> element inside the <assignment> element.

The <*key*> element can also be used in a destination mapping file. In this case, it specifies a field within the "to" destination structure that should be set to the value of the broker field.

For mapping to a destination, there is another optional element analogous to the <*key*> element for source mapping. In this case, we want to map a Broker structure to a scalar value within the destination's structures.

For example, if a grant's funding agency should be represented in the destination by just the name of the agency, omitting all other information describing the agency, one specifies this through a <*uses*> element:

```
<assignment>
  <from>Researcher.grants.agency</from>
  <to>fundingAgency</to>
  <uses>name</uses>
</assignment>
```

This tells the Broker that when mapping a grant's funding agency to the destination structure, it should just use the funding agency's name.

5.3 Special Cases

5.3.1 Explicit Conversion

If the default conversion provided by the Broker does not suffice for a particular field, one can provide a Java method for converting that field.

5.3.2 Annotation with a Fixed Value

Sometimes we need to specify that a sibling field of the "to" field be assigned a fixed value, as a form of annotation of the "to" field itself. For example, in mapping a Grant to the ORCID Funding structure, the grant number is included as an external identifier. A sibling element, the external identifier type, is used to specify that this identifier is, in fact, a grant number. In the Broker mapping file, we specify this by using a <*set*> element:

```
<assignment>
  <from>Researcher.grants.number</from>
  <to>externalIdentifierValue</to>
  <set>
     identifierType="grant_number"
  </set>
</assignment>
```

6 RELATED WORK

The Data Broker roughly falls into the category of an Extract, Transform, Load (ETL) system, and is therefore closely related to the field of data warehousing. It is distinguished from most ETL systems in its use of schema-less RDF. Although there have been efforts to provide ETL for RDF data (Knap et al., 2014), that is not what the Broker does; rather, the Broker uses RDF as a vehicle to mediate between other data stores without committing to a schema.

7 CONCLUSION

The Data Broker design and implementation show how declarative knowledge can be removed from program code and into configuration files, resulting in a highly reusable and evolvable software system. As flexible as the current design is, however, we believe we can go further. For example, the representation of source and destination structures as Java classes requires that the Broker be stopped by the system administrator in order to add a source or destination. The new or revised XML files are then compiled into Java by *xjc*, and the system is restarted. However, with appropriate dynamic class compilation and loading, the requirement to stop the Broker could be eliminated.

We would also like to consider eliminating the native Broker structures entirely, allowing sources and destinations alone to define the available knowledge. This would closely reflect the schema-less RDF representation, and move the system closer to the notion of a semantic data bus (Rilee et al., 2012).

REFERENCES

ORCID, 2015. *http://orcid.org.*

Sesame, 2015. *http://rdf4j.org.*

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

Güneş, A., Özsu, T., Daudjee, K., and Hartig, O. Executing Queries over Schemaless RDF Databases. *Proceedings of 2015 IEEE 31st International Conference on Data Engineering (ICDE).*

Knap, T., Kukhar, M., Machac, B., Skoda, P., Tomes, J., Vojt, J. UnifiedViews: An ETL Framework for Sustainable RDF Data Processing. *11th Extended Semantic Web Conference,* ESWC 2014.

NOSQL, 2015. *http://nosql.org.*

Oracle, 2015a. The Java Reflection API. *https://docs.oracle.com/javase/tutorial/reflect.*

Oracle, 2015b. Java Architecture for XML Binding (JAXB). *http://www.oracle.com/technetwork/articles/javase/index-140168.html.*

Rilee, M., Curtis, S., Clark, P., and Bailin, S. Frontier, a decision engine for designing stable adaptable complex systems: Adaptive framework. *Proceedings of the 2012 IEEE Aerospace Conference,* 3-10 March 2012.