A Web Services-Based Framework for Building Componentized Digital Libraries

Pablo A. Roberto, Rodrygo L. T. Santos, Marcos André Gonçalves, Alberto H. F. Laender

Department of Computer Science – Federal University of Minas Gerais
31270-901 Belo Horizonte, MG, Brazil

Abstract

We present a new Web services-based framework for building componentized digital libraries (DLs). We particularly demonstrate how traditional RDBMS technology can be easily deployed to support several common digital library services. Configuration and customization of the framework to build specialized systems is supported by a wizard-like tool which is based on a generic metamodel for DLs. Such a tool implements a workflow process that segments the DL designer tasks into well-defined steps and drives the designer along these steps. Both the framework and the configuration tool are evaluated in terms of several performance and usability criteria. Our experimental evaluation demonstrates the feasibility and superior performance of our framework, as well as the effectiveness of the wizard tool for setting up DLs.

Key words: Digital libraries, Web services, component-based software development, software evaluation

1 Introduction

Digital libraries (DLs) are among the most advanced and complex types of information system, going far beyond search engines, since they offer many other valued services. They are normally designed for specific user communities, which must be involved with many aspects, from specification to utilization, in order to guarantee their success (Fox and Marchionini, 1998).
Many of the existing DLs are based on monolithic architectures and their development projects are characterized by intensive cycles of design, implementation and tests (Suleman, 2002). Several have been built from scratch, aiming to meet the requirements of a particular community or organization (for instance, see (Laender et al., 2004)).

One way to deal with these issues is through the creation of specific software component toolkits, in which each component is responsible for a small part of the functionality of a DL and integrates with other components in order to build a complete system (Suleman, 2002). Such toolkits offer a generic, extensible and reusable framework for building DLs, allowing, for example, to reduce the necessary effort to develop them.

In this article, we present WS-ODL, a new Web services-based framework to build componentized digital libraries. The framework components operate on top of the Fedora architecture (Lagoze et al., 2006), which provides the framework repository and some basic infrastructure services. All communication among the components and between them and Fedora is done via Web services, using SOAP (Simple Object Access Protocol)\(^1\), which provides advantages like enhanced interoperability and validation of input parameters. Besides Fedora, we make use of standard relational database technology to support the functionality of some components, mainly those providing more advanced services, e.g., searching using structured and unstructured data jointly and multidimensional browsing. Since the framework configuration is not a trivial task, we have also developed a wizard-like tool that guides the user throughout the installation process.

Our experimental evaluation demonstrates that this framework is feasible and that its components present superior performance when compared to those provided by the ODL framework (Suleman, 2002), which inspired the development of ours. The experiments also show that the wizard-based approach is very effective, allowing the users to install a DL system much more easily and rapidly than by doing it via command-line. We also discuss some drawbacks of the framework in its current version, which should be considered when choosing it as a solution to build digital libraries.

In summary, the main contributions of this article are:

- The design and implementation of a software framework for building componentized digital libraries;
- The development of a wizard tool for easily setting up running digital libraries based on the framework;
- An extensive experimental evaluation of both the framework and the wizard tool comprising performance, scalability, and usability issues.

\(^1\)http://www.w3.org/TR/soap/
The remainder of this article is organized as follows. In Section 2, we describe related work. In Section 3, we overview the architecture of all components that make up the WS-ODL framework. The wizard-based installation tool is described in Section 4. In Section 5, we discuss the experimental evaluation of our framework. Finally, in Section 6, we present our conclusions and perspectives for future work.

2 Related Work

The Open Digital Libraries (ODL) (Suleman, 2002) project was one of the first efforts to advocate a componentized approach for the development of digital libraries. ODL proposed an extension to the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) (Lagoze and Van de Sompel, 2001) to support interactions among digital library components. This framework had some advantages mainly due to the use of the OAI and the software componentry paradigms, like simplicity, openness and reuse. Nevertheless, it presents several problems with respect to performance, scalability and interoperability. Performance issues are due to specific implementation choices which led to scalability problems, mainly regarding the components that implement information retrieval functions. Interoperability issues are related to the use of the XOAI protocol, a not widely adopted OAI-PMH extension developed specifically for that framework.

To overcome some of the deficiencies mentioned above, mainly regarding interoperability, we chose to use the Web services technology in our framework. A Web service is a software system designed to support an interoperable interaction between applications in a network. It communicates with other systems via an interface described in WSDL (Web Service Description Language) and SOAP messages, that are transported using HTTP and XML. SOAP is a protocol for information exchange in a decentralized and distributed environment. Through the use of the XML technology, SOAP defines an extensible framework for message transfer on top of other low level protocols. The framework is designed to be independent of programming models and specific implementation semantics. More details on these technologies can be found on the W3C site².

Fedora (Lagoze et al., 2006) is an extensible repository for storage, management and dissemination of complex objects and their relationships. These objects can have local or distributed content, and can be attached to different “disseminations”, which makes it possible to have various dynamically created representations of a given object. The relationships among the objects are ex-

² http://www.w3.org/
pressed in RDF (Resource Description Framework) and stored in the Kowari triplestore\(^3\). The architecture is implemented as a Web service and provides the basis for the construction of other applications, like a digital library, for example. Our framework supports the construction of a DL on top of Fedora while improving part of its functionality, as will be shown. It should be emphasized that other repository infrastructures could also be used, provided that they could explore Web services intra-communication facilities.

Regarding implementation choices, we chose to explore open source standards and languages such as Java and traditional DBMS technology, e.g., MySQL. In particular, we wanted to investigate how much support traditional DBMS technology could give to the implementation of the functionality of our components. In this context, Grossman et al. (1997) showed that it is possible to use the standard relational model to implement an information retrieval system and that such a system, in specific scenarios, presents good performance and scales. Moreover, it makes it possible to integrate structured and unstructured data, which provides a more powerful search mechanism, comparatively to traditional search engines. This mechanism is very interesting to a digital library, where users have needs more specific than those of a common Web user. We adopt, adapt, and extend these ideas by showing how several other DL common services can be implemented using the same technology.

With respect to the task of setting up components in order to build digital libraries, some tools have also been described in the literature. BLOX (Suleman et al., 2005) is a tool that hides most of the complexity involved in the task of configuring distributed componentized digital libraries. However, during the configuration task, users interact with this tool in an unguided manner: its interface comprises a set of windows, each one representing the configuration of a software component. In another approach, the Greenstone suite (Buchanan et al., 2005) incorporates a wizard that allows non-specialist users to create and organize digital collections from local or remote documents. This tool, on the other hand, does not deal with the configuration of service provider components.

Our wizard tool (Santos et al., 2006) implements the setup task as a workflow, regarding the configuration of each of the major aspects within the domain of a digital library (e.g., repositories, services) as a step in this workflow, and describes each of the involved concepts through a contextual help system in the form of tooltips. According to Ekenstierna and Ekenstierna (2002), a wizard is a special form of user assistance that steps users through the completion of tasks that have clear structures. Wizards are not suitable for tutorials – they should operate on real data, and not be considered for instructional user assistance. A great drawback with wizards is their linearity. But, since many

\(^3\) [http://www.kowari.org/](http://www.kowari.org/)
tasks are of a linear kind, wizards can be very useful – above all when tasks are performed infrequently, which is the case when constructing digital libraries. The average user thinks that the wizard is a very convenient technique, mostly thanks to its interactive properties – it is unlikely for a user to make errors. On the other hand, average users are uncertain if they are able to solve similar tasks without using the wizard. As for the contextual help approach, Ekenstierna and Ekenstierna claim that this help feature gives the user immediate assistance about a specific object and its context, without leaving the current working area.

3 The WS-ODL Framework

In this section, we describe the WS-ODL framework developed to construct digital libraries from a pool of Web services-based components. This framework comprises three major parts (see Figure 1):

- A data repository (dark-gray boxes in Figure 1), based on the Fedora architecture, that supports basic infrastructure services, such as an OAI data provider and object versioning;
- A set of Web services (middle-gray box in Figure 1) to provide specific digital library services, such as searching and browsing;
- A client layer (light-gray boxes in Figure 1), responsible for generating user interfaces, for sending data to the Web services, and for treating the responses returned by these services. This layer is composed of a set of Java servlets, which execute XSL transformations (XSLT) on the data returned by the Web services.

Following, we first give an overview of the WS-ODL working scheme and then we describe each of the components that comprise our framework.
3.1 WS-ODL Working Scheme

According to Figure 1, the operation of a typical component of our framework (e.g., the recent works exhibition component) is as follows: (1) the client, via a Web browser, makes an HTTP request to the servlet; (2) the servlet requests to the Web service the relevant identifiers (pids), e.g., pids for recent works; (3) the Web service communicates with the database or the Fedora research service (Fedora Resource Index Query Service) to get the requested pids; (4) the database or the Fedora service returns the pids to the Web service; (5) the Web service passes the pids to the servlet; (6) the servlet sends the pids to the Fedora repository; (7) the Fedora repository returns the metadata corresponding to the pids; (8) the servlet executes an XSL transformation on the metadata; (9) the data transformed is shown in the client Web browser.

The WS-ODL framework can be used in a distributed fashion. For example, Fedora can be hosted in one server, the Web services in another one, and the client layer in a third one. This loosely-coupled architecture along with the use of Web services allows for an enhanced flexibility and interoperability of the systems built using our framework.

3.2 Recent Works Exhibition Component

The Recent Works Exhibition component has the objective of stimulating the users’ curiosity by exhibiting the works most recently added to the digital library repository, in decreasing order of inclusion date. Users may additionally specify, as a parameter, the number of works they want to see. Otherwise, a default number of ten works is shown.

The pids for recent works come from a table \( (\text{doFields}) \) in the Fedora’s relational database that stores (meta-)information about digital objects. The Recent Works Exhibition Web service accesses this table and retrieves the fields \( \text{pid} \) and \( \text{mDate} \), ordered by the latter, which contains the last modification date of each of the stored digital objects. The retrieved data is passed to the servlet, which in turn uses it to get the actual digital objects from the Fedora repository. Although based on the Fedora architecture, this component can work with other repositories as well with little effort, thus not depending on this particular one.

The option of making a direct access to the database and then to access the repository to get the actual data may seem a waste of resources, but this can be justified by the fact that Fedora does not have ordering capabilities in its search API. Therefore, the alternative option, i.e., to get all data and order them locally in the client, would be clearly much more inefficient than the
adopted strategy.

3.3 Browsing Components

Our framework incorporates two browsing components with different features. The first one is composed of a set of Web services and servlets designed to provide a navigational structure to the metadata stored in the Fedora repository. This structure integrates with the other components, allowing the user to have a rich experience in utilizing a digital library built from these components.

The Web services associated with this component are able to obtain: pids referring to a given author name, pids referring to the works of a collection, the name of a collection given its pid, and collection information given a work pid. In the context of this component, a collection is a related group of objects. For example, papers belonging to a conference proceedings form a collection. With this, we can have a collection for the proceedings of JCDL 2007, VLDB 2007, SIGMOD 2007, and so on. This is similar to the OAI set concept (Lagoze and Van de Sompel, 2001).

The servlets make use of the information obtained from the Web services to produce author’s works pages and collection’s works pages. The functionality of this component is similar to that described in Section 3.2. However, instead of accessing the database to obtain the pids, it uses the Fedora ri:search service, which permits querying relationships (expressed as RDF triples) between the digital objects stored in the repository. Accordingly, to create an author’s works page, one of the Browsing Web services issues to ri:search the iTQL query in Figure 2.

```
select $object $date
from <#ri>
where $object <dc:creator> 'Somebody' and $object <dc:date> $date
order by $date desc
```

Fig. 2. Retrieving an author’s works page

This query searches, in the Fedora’s Kowari triplestore, for all RDF triples (in the form subject-predicate-object) that satisfy the predicate $dc:creator$ is ‘Somebody’ and returns the matching subjects (binded by the variable $object$). Since we want the result ordered by date, we have to bind the variable $date$ as well and associate it to the subjects of interest, which is done in the second part of the where clause through the predicate $dc:date$ is $date$. As a side effect, the object’s date will also be returned. The Web service obtains the response from the query and returns only the pids, which are then used by

---

4 $dc:creator$ and $dc:date$ are two of the fifteen fields defined by the Dublin Core metadata standard (http://dublincore.org/)
a servlet to retrieve the complete digital objects from the Fedora repository. 
The collection’s works page can be constructed in a similar fashion.

The Multidimensional Browsing component (Raghavan et al., 2005), our sec-
second component for browsing, provides a hierarchical navigation structure 
based on a metamodel expressed in XML (Figure 3) and on a relational 
database. It allows for other navigation paradigms (e.g., by date) to be deliv-
ered to the user, by just altering its metamodel configuration and updating the 
database. It can also work as a substitute for author’s works and collection’s 
works pages. The metamodel basically contains dimension names and XPath expressions used to obtain the desired data from the digital object, which is 
then stored in a database. This component has been created by the Virginia 
Tech DL Research Group but we have modified its structure in order to 
accept multivalued attributes for digital objects (e.g., more than one author).
An interesting feature of this component is that it is independent of the repos-
itory being used by the framework. This way, if Fedora is to be replaced by 
any other architecture, this component will demand no modification.

```xml
<browse>
  <dimension name="coverage">
    <level>
      <levelname>Coverage</levelname>
      <path>//dc:coverage</path>
    </level>
  </dimension>
  <dimension name="date">
    <level>
      <levelname>Date</levelname>
      <path>//dc:date</path>
    </level>
  </dimension>
</browse>
```

Fig. 3. Example fragment from the Multidimensional Browsing metamodel

### 3.4 Searching Component

A searching service, structured or keyword-based, is an important part of 
any digital library, since it provides to users a practical way of finding the 
information they desire. The Fedora architecture has searching capabilities 
in its API and provides a search interface to the end user. However, this 
search service has some limitations. For example, it does not support the 
retrieval of query results ranked by relevance. In this sense, our Searching 
component can work as a substitute for the Fedora’s native search API, adding 
the capability for processing relevance-based queries and the possibility of 
integrating structured and unstructured data in the same query.

---

5 http://www.w3.org/TR/xpath
6 http://www.dlib.vt.edu/
Our Searching component takes three input parameters – the query keywords, the type of search and the number of documents to be returned – and produces an HTML page with the results satisfying the query. The search type can be either Boolean (AND / OR) or by relevance. In the latter case, documents are ordered according to their similarity to the query, which is calculated based on the standard vector space model (Salton et al., 1975). The number of documents limits the size of the response, since the user is normally only interested in the top documents.

This component is also independent of the repository and works like the Recent Works Exhibition component, making use of a relational database to return the pids for the objects satisfying the user query. This database includes relations that simulate an inverted file, and an SQL query can be issued to obtain the desired pids. So, there is a table representing a document (with its doc_id and weight), a table representing the terms (with the term identifier and idf – inverse document frequency), a table for the association document-term (with the term frequency – tf – within the document), and a table used by structured queries, indicating the document field in which a given term occurs. Figure 4 illustrates the database relations. This component is not just a simple SQL-based search service, since it is capable of ranking the retrieved documents by their similarity to the query. Moreover, matching between query and document can be partial, which in general does not occur with regular SQL queries.

```
SELECT D.fedora_pid, SUM(DT.tf * I.idf * I.idf) / D.weight AS rank
FROM doc D, doc_term DT, idf I, doc_term_struct DTS
WHERE I.term IN ('digital', 'libraries') AND D.doc_id = DT.doc_id AND
  DT.term = I.term AND D.doc_id = DTS.doc_id AND
  DTS.field = 'dc:creator' AND DTS.content LIKE '%smith%'
GROUP BY DT.doc_id
ORDER BY rank DESC LIMIT 30
```

Fig. 5. Combination of keyword-based and structured searches
Besides the aforementioned components, which constitute the core of our framework, in this Section, we present two additional components that provide basic functions to support other components, as well as a set of components that perform maintenance tasks required by the framework.

**User Registration.** This is a simple component by means of which users can register themselves for future activities in the digital library, like annotate an object. When registering, users inform an e-mail address and a password. This information is validated and stored in the Fedora repository as an inactive object that is not seen by other components.

**Annotation.** This component allows users to make comments about items in the digital library. They do it by choosing the object to be annotated and by informing their e-mail address and password, as well as the text to be annotated. Information is validated and, if is correct, stored in the Fedora repository as a new inactive digital object.

**Administration Components.** These components are used to insert data into the digital library, as illustrated in Figure 6. The Data Inclusion component uses as input an XML file, in a format similar to that of a response to a ListRecords OAI request (Lagoze and Van de Sompel, 2001), containing all records to be inserted in the repository. From this file, it generates a set of FOXML (Fedora Object XML) files (object0, object1, etc.), which are then inserted into the repository in the order they were created (based on their suffixes) with the aid of the Fedora API. The Source Database component is used to create a database that works as a middleware between Fedora and the browse and search databases. If it did not exist, the Browse and Searching components would have to communicate directly with Fedora to obtain the data they need. Since this communication (HTTP-based) is slower than a database communication, we chose to store the data in an intermediary database, to improve performance when creating the other databases. This way, the Source Database component also improves the generality of the framework, since the Fedora OAI data provider could be replaced by any other component with the same functionality without affecting the creation of the databases. It basically issues OAI requests to the Fedora OAI data provider and stores the results in the database. These requests are incremental, meaning that only new content is retrieved from the repository. If the object has suffered any modification and it is versioned in Fedora, only the last version is retrieved. From the source database and a metamodel, the Browse Database component builds the browse database, used by the Multidimensional Browsing component. The creation of this database is also done in an incremental fashion. The last component (Search Database) creates the database used by the Searching
component. It is possible to inform the delimiters between the words, the stop words and which fields are to be indexed to create the database. The final result will be a database similar to that depicted in Figure 4.

![Figure 6. Administration components](image)

### 4 The Wizard Tool

In order to facilitate the setup of DLs based on our framework, a wizard-like tool is provided to guide the designer during the installation task. Its architecture basically follows the MVC framework (Burbeck, 1987), which comprises three layers: the model layer, the view layer, and the controller layer. In order to allow for the produced configurations to outlive the execution of the wizard tool, we have extended the MVC framework with the addition of a persistence layer.

The model layer was primarily designed based on configuration requirements gathered from the ODL framework (Suleman, 2002). Later, it was extended in order to support the configuration workflow of different component pools. Such extension was conceived inspired on the definition of a digital library taken from the 5S (Streams, Structures, Spaces, Scenarios, Societies) model (Gonçalves et al., 2004). Accordingly to 5S, a typical digital library is informally defined as a set of mathematical components (e.g., collections, services), each component being precisely defined as functional compositions or set-based combinations of formal constructs from the model. Our configuration model was devised regarding the components that make up a 5S-like digital library as configurable instances of software components provided by a component pool. By “configurable instances” we mean software components whose behaviors are defined as sets of user-configurable parameters. The class diagram in Figure 7 shows a simplified view of the devised model. As shown in the diagram, a Digital Library is implemented as a set of configurable instances of the Provider class—a subtype of Component, the class that models the
software components supplied by the Pool being used. A Provider instance may be typed either Repository or Service, according to its role within the digital library. For orthogonality purposes, the Digital Library itself is also implemented as a configurable instance of Component. Additionally, components may be declared mandatory, as well as dependent on other components, if their execution depends on services provided by these other components. The configuration of each component is implemented as a set of Parameters, semantically organized into parameter groups.

![Class diagram for the model layer](image)

Fig. 7. Class diagram for the model layer

The persistence layer is responsible for loading and saving components configurations. Besides that, it is up to this layer the tasks of setting environment variables and preparing databases that support the execution of some components. Its working scheme is based on two XML documents: a configuration log and a pool descriptor. The first and simpler one, the configuration log, acts as a cache for the persistence layer, comprising basic information about the currently configured digital libraries running in the server. In Figure 8, for example, the configuration log lists four component instances in a given system configuration, named ‘MyLIB’.

```xml
<system id="mylib">
  <baseDir>/test/www/wsodl</baseDir>
  <baseUrl>http://localhost:8080/test/wsodl</baseUrl>
  <name>MyLIB</name>
  <description>This is a test configuration based on the WS-ODL framework.</description>
  <instances>
    <instance id="lib1" component="library"/>
    <instance id="rep1" component="fedora"/>
    <instance id="sea1" component="search"/>
    <instance id="brw1" component="browse"/>
  </instances>
</system>
```

Fig. 8. Fragment from the wizard configuration log

The second document, the pool descriptor, details every component in the pool, including all configuration parameters associated to them, as exemplified in Figure 9 for the Searching service. The definition of this component includes its type – which indicates where in the installation workflow its instances will be configured – and a list of its component dependencies (in this case, search instances depend on admin instances). It also includes a label and a description to be displayed by the view layer. The parameters associated to this
component are grouped together according to their semantics. For instance, the **Indexation** group lists a set of parameters responsible for managing the inverted files used by the Searching service. The description of each configuration parameter contains path entries of the form `document:xpath_expression` that uniquely locate the parameter in each of its source documents. Since some path entries may be dependent on auto-detected or user-entered information, both only known at runtime (e.g., the base directory of the wizard tool and the current digital library identifier), the pool descriptor document also comprises a list of definitions to be used in path entries declaration. For example, in Figure 9, the path entry for the `fieldstoindex` parameter is declared relatively to the definition `fedoraHome`, which is derived from a user-entered value representing the base directory of the digital library being configured.

```
<component id="search" type="model.pool.library.Service" dependencies="admin">
  <label>Searching</label>
  <description>A mechanism for both structured and keyword-based searches.</description>
  <group>
    <label>Indexation</label>
    <parameter id="fieldstoindex" type="enum:eval('./md_fields -f #mdprefix')" min="1">
      <path>#fedoraHome/wsodl/config.xml:/config/search/fieldstoindex</path>
      <default>dc:title,dc:subject,dc:description</default>
      <label>Fields to Index:</label>
      <description>A list of fields to be indexed by the Searching service.</description>
    </parameter>
  </group>
</component>
```

Fig. 9. Fragment from the pool descriptor document

For validation purposes, the type of each parameter is defined by regular expressions. Besides that, parameter types may be complex (i.e., defined as a composition of simple types) and may include cardinality constraints, as pointed by the `min` attribute in Figure 9. Pointers to the values of previously-configured parameters in the workflow sequence may be used as terminals in type expressions, as well as values evaluated from external script calls. This way, the domain of a given parameter type may be constrained based on values set in other parameters or even on values determined from outside the framework being configured (e.g., operating system properties). In Figure 9, for instance, the type declaration for `fieldstoindex` is determined by the values returned by the `md_fields` script, which takes the previously configured `mdprefix` parameter (associated to the repository infrastructure) as input and outputs the fields corresponding to this format (e.g., Dublin Core fields).

Both the configuration log and the pool descriptor XML documents are handled via DOM – Document Object Model\(^7\). Configuration loading and saving are performed through XPath expressions. Based on the specification of each component (from the pool descriptor document), previously-configured instances of them are loaded into memory; besides, an empty instance of each component is created to serve as a template for new instances to be added later. Loading is performed in a lazy fashion, i.e., objects are created only

\(^7\) [http://www.w3.org/DOM/](http://www.w3.org/DOM/)
when needed. On the other hand, saving is only performed at the end of the whole configuration task, as well as some additional tasks, such as environment variables and database setup, performed via system calls. Specializing the wizard tool to assist the configuration of digital libraries based on different component pools can be done just by providing a description document for each pool, as well as eventual accessory scripts for performing deployment tasks. In fact, during the wizard development, we have produced a specific version to work with the components provided by the ODL framework.

The remaining parts of the wizard architecture, the \textit{view} and \textit{controller} layers are integrated – a common simplification of the original MVC framework. They are responsible for handling user interactions, for performing the corresponding modifications to the configured instances of the defined model, and for displaying these updated instances back to the user. Once user interactions are directly transmitted to component instances, users modify a clone rather than the original configuration of each instance. This allows users to cancel all the modifications performed to a given component instance at any time.

Since the configuration workflow is organized into steps, a wizard-like interface was a natural choice (Figure 10). In this interface, each step comprises the configuration of a major aspect of a digital library: the library itself, its collections or metadata catalogs, and the services it may provide. In each of these steps, the parameters associated to the component instances they list are presented in dynamically created, tab-organized forms. Each tab corresponds to a parameter group. Form elements are designed according to the type of the parameter they represent: repeatable parameters are shown as lists, parameters representing file descriptors present a file chooser dialog, parameters with values restricted to an enumerable domain are displayed as a combo box, strings and integers are simply shown as text fields. The semantics of every parameter is displayed as a tooltip near the parameter label. Type-checking is performed against every value entered by the user; in case of an erroneous value, a corresponding exception is raised and the user is properly notified.

![Fig. 10. Wizard GUI: configuring the Searching component](image-url)
In order to verify the viability of our framework, we performed several tests, including an installation test, a data loading test and a performance test. These are discussed below.

### 5.1 Installation Test

In the installation test, we wanted to measure how difficult it is to configure our framework, mainly by non-specialist users. It was divided into two comparable experiments: the first one, a manual, command-line driven installation of the framework, and the second one, comprising two sessions of installations guided by our wizard tool. These two experiments are detailed in the following subsections.

#### 5.1.1 Manual Installation

In this experiment, users were given an installation guide with instructions on how to configure all parts of our framework. We had eight users, acting as digital library administrators, being four from Computer Science (CS) and four from Library and Information Science (LIS). The time taken by this task was measured at five checkpoints (A - install and configure Fedora; B - install Apache Axis; C - install the Web services on Axis; D - install the servlets; and E - insert data in the constructed digital library), comprising macro-phases of the installation process. The number of errors made in each phase was also measured. Tables 1 and 2 show the results. Times are displayed in the form $hh:mm$.

<table>
<thead>
<tr>
<th>Task</th>
<th>CS</th>
<th>LIS</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0:35</td>
<td>0:43</td>
<td>0:39</td>
</tr>
<tr>
<td>B</td>
<td>0:12</td>
<td>0:14</td>
<td>0:13</td>
</tr>
<tr>
<td>C</td>
<td>0:26</td>
<td>0:30</td>
<td>0:28</td>
</tr>
<tr>
<td>D</td>
<td>0:10</td>
<td>0:08</td>
<td>0:07</td>
</tr>
<tr>
<td>E</td>
<td>0:20</td>
<td>0:08</td>
<td>0:09</td>
</tr>
<tr>
<td>Total</td>
<td>1:43</td>
<td>1:53</td>
<td>1:48</td>
</tr>
</tbody>
</table>

#### As we can see, the installation process is not trivial, with users taking almost two hours to complete the task. The main reason behind this is the fact that the process is all command-line driven. The most complex task is the installation of Fedora, followed by the installation of the Web services on Axis. This occurs because these two tasks are the most intensive in terms of using the
command-line. Besides that, the installation of Fedora is naturally complex, due to the activities involved (e.g., configuring a database).

As expected, CS users tend to be more skilled in this kind of task, since they are used to dealing with computers. Despite this, they completed the task just a little faster than LIS users (about 10%, on average), although making much less errors (about 53%, on average). This probably happened because CS users payed more attention to the installation guide. This way, they made less errors, but took almost the same time as LIS users, since they carefully read the guide. Nevertheless, the experiment demonstrates that our framework can be used by non-specialist users, and that they spend just a little more time than CS users, although making more errors.

5.1.2 Assisted Installation

In order to evaluate the effectiveness of the wizard-guided approach, we have conducted two experiment sessions with the tool. The first one also involved four users from Computer Science (CS) and four from Library and Information Science (LIS) and consisted of performing two configuration tasks and filling in an evaluation questionnaire. Both tasks highly explore all interface elements of the wizard tool, such as lists and file choosers. The first and simpler task, aimed at helping users to get familiar with the tool, consisted of modifying a few parameters of a pre-configured digital library. The second and more complex one consisted of configuring a whole library from scratch. This task was designed to be comparable to the one performed in the command-line driven installation test. Though data insertion is considered out of the scope of the wizard tool but is performed in the command-line installation experiments, the comparison was still possible just by discarding the time spent at checkpoint “E” while comparing the overall times. Table 3 shows the completion time and correctness from the two experiments conducted with the wizard prototype (namely, tasks #1 and #2), as well as those for the users who also performed the command-line driven configuration experiment (task #2'). For comparison purposes, the performance of an expert user – the developers of the wizard tool and of the WS-ODL framework – is also shown at the end of the table. Time is displayed in the form \(hh:mm:ss\) and correctness stands for the number of correctly executed items in the configuration task divided by the total number of items in that task.

Comparison between the wizard-guided and the command-line driven processes for task #2 shows that configuring WS-ODL components with the aid of the wizard tool is much faster (about 500%, on average) than manually (hypothesis accepted by statistical analysis: \(t\) test with \(\alpha = 0.05\)). Configuration correctness is also substantially increased (about 34%, on average) with the aid of the tool (hypothesis accepted by statistical analysis: \(t\) test with \(\alpha = 0.05\)).
This is mainly due to its type-checking and component dependency checker systems. Fastness and correctness attest the effectiveness of the wizard-based against the command-line driven process. Effectiveness was also subjectively rated by users who participated in both tasks and measured based on a 5-point bipolar scale, ranging from 1 (worst rating) to 5 (best rating). On average, the effectiveness of the wizard-guided process, in terms of easing the configuration task, was rated 4.5.

From the questionnaire filled in by the users who performed the wizard-guided configuration tasks, we devised other two metrics: didactic applicability and satisfaction, both measured based on 5-point bipolar scales, ranging from 1 (worst rating) to 5 (best rating). On average, in terms of understanding of the concepts being configured (i.e., concepts pertaining to the domain of the component pool on top of which the wizard is running), the didactical applicability of the wizard was subjectively rated 3.75. This was an unexpected yet not unwelcome high value, since the design of wizards is not intended for didactical purposes. Satisfaction was measured in terms of comfort and ease of use. On average, users subjectively rated them 4.25 and 4, respectively.

In the second wizard-guided experiment session, we wanted to better assess the users’ understanding of the tasks being accomplished. For such, we devised a higher level configuration task, in which users were given a description of the library to be constructed rather than a step-by-step procedure on how to construct it, as was done in the previous tasks. For this task, we had eight new volunteers, being six from Computer Science (CS) and two from Library and Information Science (LIS), that fit our desired profile, i.e., users with experience in dealing with visual applications and with knowledge about digital libraries at a system level. Though similar to the previous tasks, this one was considered quite more difficult, demanding a broader comprehension from the users. Completion time and correctness measures (see task #3, in Table 3), however, were not significantly affected, what further confirms the effectiveness of the wizard tool. In this experiment, satisfaction and ease of use were subjectively rated 3.5 and 3.83, respectively. These lower values, comparatively to those obtained in the first wizard-guided experiment, appear to be due to the much less assistive nature of the task designed for this session.

### Table 3

Completion time and correctness per task

<table>
<thead>
<tr>
<th>User</th>
<th>Completion Time</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Task #1</td>
<td>Task #2</td>
</tr>
<tr>
<td>CS Mean</td>
<td>00:07:01</td>
<td>00:11:26</td>
</tr>
<tr>
<td>CS StDev</td>
<td>00:01:11</td>
<td>00:04:15</td>
</tr>
<tr>
<td>LIS Mean</td>
<td>00:11:05</td>
<td>00:18:33</td>
</tr>
<tr>
<td>LIS StDev</td>
<td>00:03:33</td>
<td>00:02:08</td>
</tr>
<tr>
<td>Global Mean</td>
<td>00:09:33</td>
<td>00:15:00</td>
</tr>
<tr>
<td>Global StDev</td>
<td>00:03:17</td>
<td>00:04:55</td>
</tr>
<tr>
<td>Expert</td>
<td>00:01:53</td>
<td>00:04:33</td>
</tr>
</tbody>
</table>
By regarding the task to be completed as a set of items, we could measure two other important points. First of all, we measured the performance of users as a function of the times they spent in each item of the task relatively to the time an expert user (theoretically, a lower-bound time) spent at the corresponding item. This way, we could observe the evolution of users’ performance along the task, as shown in the graph in Figure 11.

From the graph in Figure 11, we can observe that most users soon reach a performance level close to that of the expert user (horizontal line at the relative performance value of 1), what we see as an indication that the wizard tool is easy to use. We can also observe that there are performance peaks in some users’ curves, denoting performances up to 4 times higher than the expert’s in the execution of a given item. However, this abrupt gains are counterbalanced across several other items, ending up with an overall performance lower than the expert’s, as expected.

As a second point, we measured the temporal evolution of users along the execution of the configuration task, as depicted in Figure 12. The graph shows, for each of the 18 task items, the corresponding cumulated time spent by each user. From the graph, we can see, for instance, the points in which users’ curves most diverge from the expert’s curve, what corresponds to the task items with greatest complexities. In case of this task, the creation of directories demanded by some components (items 2 and 10 in the graph), followed by the instantiation of the repository component (item 5), were the most time-consuming items in the whole task.
5.2 Data Loading Test

The second experiment was a data-loading test, in which we wanted to measure the time taken to make an entire collection available using our framework. We have measured the time for six activities in this process: (1) transform the input file containing the records in a unique FOXML file; (2) split this file, in order to create one file per digital object; (3) insert these objects in the Fedora repository; (4) create the source database; (5) create the browsing database, and (6) create the search database. We repeated the experiment 10 times for each activity, and took the average execution time. Activities 1 to 3 are related to Fedora and can be different in case another repository is chosen. Activities 4 to 6 are related to our framework, no matter which repository is used.

We used three data collections: BDBComp (Laender et al., 2004) (4,142 objects), LDB (29,480 objects) and Citeseer \(^8\) (574,901 objects). The LDB collection comprises objects from three data sources (Lattes \(^9\), DBLP \(^10\), and BDB-Comp \(^11\)), with replicas discarded and scope constrained only to the Brazilian computer science community, since Lattes is a curriculum information system used by the whole Brazilian scientific community. The machine used for this experiment was a 32-bit Pentium 4 HT 3.2GHz, with 2GB of RAM and a HD of 400GB. The measured times are shown in Table 4.

It can be observed that the most time consuming activity is the insertion of objects in Fedora. This happens because, for each object inserted, Fedora has to generate and store many RDF triples in the Kowari triplestore used to maintain object to object relationships. The larger and more related, the more

---

\(^8\) [http://citeseer.ist.psu.edu/](http://citeseer.ist.psu.edu/)

\(^9\) [http://lattes.cnpq.br/](http://lattes.cnpq.br/)

\(^10\) [http://dblp.uni-trier.de/](http://dblp.uni-trier.de/)

\(^11\) [http://www.lbd.dcc.ufmg.br/bdbcomp](http://www.lbd.dcc.ufmg.br/bdbcomp)
triples are generated. In our case, about 30 triples are generated per object in average, and this has a major impact in the performance of this activity. Besides that, the Kowari version used in Fedora 2.0 (the version we used) has a memory leak problem, which also causes performance problems, especially with large volumes of data.

For the Citeseer collection, the whole activity could not be performed. One of the reasons is the memory leak mentioned above. The other is related to the collection size. Since this collection has more than 570,000 objects, the RDF triples generation process exhausted the 32-bit memory address space of the machine used, causing the Java Virtual Machine used to run the framework to crash with an out-of-memory exception. Since this activity is a pre-requisite for the others, they were not performed either. Note that these are problems specific to the chosen repository infrastructure, not the framework itself. Solutions to these problems include upgrading to Fedora 2.1, which comes with a bug fixed Kowari version, and using a 64-bit machine. Alternatively, we could replace the whole repository infrastructure.

An interesting point to note is the growth rate of data loading time based on the size of the collections used in the experiments. Table 5 shows the time increase from BDBComp loading time to LDB loading time for each of the data loading activities considered, as well as for the process as a whole. From the table, we can see that for all but the last two activities the increase in data loading time was lower than the increase in collection size, namely, 7.11. Besides that, it was not a linear increase. However, the last two activities, which involved creating databases for the Browsing (A5) and Searching (A6) services, showed a time increase quite above the increase in collection size. This fact may be related to how the components responsible for creating these databases work, since they use DOM, which is likely to be best suited for applications where documents must be accessed repeatedly or randomly. Once our application is strictly sequential and one-pass, an alternative could be to use SAX – Simple API for XML, which is likely to run faster and use less memory in this case. Despite the worse performance of the activities A5 and A6, the overall time increase was lower than that of the collection sizes.

For comparison purposes, we also measured the data loading time taken by

<table>
<thead>
<tr>
<th>Activity</th>
<th>BDBComp</th>
<th>LDB</th>
<th>Citeseer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>4.90</td>
<td>23.24</td>
<td>9,648.00</td>
</tr>
<tr>
<td>A2</td>
<td>125.15</td>
<td>248.54</td>
<td>6,768.00</td>
</tr>
<tr>
<td>A3</td>
<td>1,362.60</td>
<td>4,238.05</td>
<td>–</td>
</tr>
<tr>
<td>A4</td>
<td>49.30</td>
<td>294.82</td>
<td>–</td>
</tr>
<tr>
<td>A5</td>
<td>117.00</td>
<td>3,333.32</td>
<td>–</td>
</tr>
<tr>
<td>A6</td>
<td>42.34</td>
<td>668.60</td>
<td>–</td>
</tr>
</tbody>
</table>
Table 5
LDB loading time relative to BDBComp loading time

<table>
<thead>
<tr>
<th>Activity</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>4.74</td>
</tr>
<tr>
<td>A2</td>
<td>1.98</td>
</tr>
<tr>
<td>A3</td>
<td>3.11</td>
</tr>
<tr>
<td>A4</td>
<td>5.98</td>
</tr>
<tr>
<td>A5</td>
<td>28.49</td>
</tr>
<tr>
<td>A6</td>
<td>15.79</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5.17</strong></td>
</tr>
</tbody>
</table>

the ODL framework (Suleman, 2002) for the BDBComp and LDB collections. We measured the time ODL took to generate its recent works, browsing and searching databases. To complete these tasks, ODL took 1,740 seconds for the BDBComp collection and 11,520 seconds for LDB. Regarding BDBComp, the two frameworks are practically identical, with a slight advantage of ours (1,701 seconds); but, concerning LDB, our framework data loading is 1.31 times faster than ODL, finishing the process in 8,807 seconds. One of the reasons behind this is the fact that ODL harvests the collection via OAI-PMH to each one of its databases, while our framework harvests only once (to the source database).

5.3 Performance Test

This experiment consists of a time comparison between the components developed and those from the ODL framework. To perform this test and to exercise the components utilization, we constructed a digital library prototype with functionality similar to that presented by BDBComp (Laender et al., 2004).

Using Apache jMeter\(^{13}\), we measured response times of HTTP requests issued to the components. We took two kinds of measure: the time taken for each component to return the relevant pids and the time taken for each component to show the response to the user. Table 6 presents the results for the LDB collection (mean time, standard deviation and gain over ODL). In the table, there are three kinds of entry: ODL entries, representing ODL components; WS-ODL entries, representing the components of our framework; and WS-ODL 2 entries, which represent a variation of our components, as explained below.

It can be noticed that our framework is much faster than ODL, in terms of retrieving pids. Regarding the time to produce the Web pages, our framework is slower than ODL in all but one case. However, this happens because our components perform considerably more work than the ODL components. For example, if a retrieved object belongs to a collection, we call an additional Web service to obtain the name of this collection, in order to create a link to this collection’s works page. We also insert links in each author name,

\(^{13}\) [http://jakarta.apache.org/jmeter/](http://jakarta.apache.org/jmeter/)
Table 6
Comparison of components response time

<table>
<thead>
<tr>
<th>Component</th>
<th>PID Time (ms)</th>
<th>Page Time (ms)</th>
<th>Gain Over ODL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StDev</td>
<td>Mean</td>
</tr>
<tr>
<td>ODL Recent Works</td>
<td>232.55</td>
<td>5.96</td>
<td>1,748.18</td>
</tr>
<tr>
<td>WS-ODL Recent Works</td>
<td>31.59</td>
<td>3.58</td>
<td>3,312.18</td>
</tr>
<tr>
<td>WS-ODL Recent Works 2</td>
<td>31.59</td>
<td>3.58</td>
<td>465.63</td>
</tr>
<tr>
<td>ODL Search</td>
<td>229.81</td>
<td>17.29</td>
<td>1,068.64</td>
</tr>
<tr>
<td>WS-ODL Search</td>
<td>54.44</td>
<td>4.38</td>
<td>3,590.52</td>
</tr>
<tr>
<td>WS-ODL Search 2</td>
<td>54.44</td>
<td>4.38</td>
<td>447.68</td>
</tr>
<tr>
<td>ODL Browse Author</td>
<td>180.35</td>
<td>18.93</td>
<td>662.55</td>
</tr>
<tr>
<td>WS-ODL Browse Author</td>
<td>38.02</td>
<td>5.73</td>
<td>1,598.11</td>
</tr>
<tr>
<td>WS-ODL Browse Author 2</td>
<td>38.02</td>
<td>5.73</td>
<td>249.58</td>
</tr>
<tr>
<td>ODL Browse Collection</td>
<td>169.57</td>
<td>16.20</td>
<td>1,466.13</td>
</tr>
<tr>
<td>WS-ODL Browse Collection</td>
<td>60.11</td>
<td>3.87</td>
<td>694.97</td>
</tr>
<tr>
<td>WS-ODL Browse Collection 2</td>
<td>60.11</td>
<td>3.87</td>
<td>658.78</td>
</tr>
</tbody>
</table>

leading to the author’s works page. The Web service call, the additional data retrieved and the links penalizes our performance. If we do not perform all this additional work, our framework is again faster than ODL, as we can see from WS-ODL 2 entries. Our browsing collection page is created faster than ODL’s in both cases, because there is not much extra work to do in this case. For this reason, the difference between our two versions is not significant.

Three main reasons explain the better performance of our framework: (1) the use of Java servlets and Web services technologies, which provides superior performance than that offered by CGI, the basis of ODL; (2) an effective use of the repository infrastructure, which offers means to achieve better performance (e.g., objects caching); (3) a more effective use of the relational database technology, especially by the Searching component.

Besides the comparison with the ODL framework, in order to contrast our implementation with a special purpose system, we compared the performance of our components against the corresponding services implemented in BD-BComp. This comparison comprised the Recent Works Exhibition and the Browsing (by author and collection) components of both BD-BComp and our framework. Table 7 presents the results.

Table 7
Performance comparison against a monolithic implementation

<table>
<thead>
<tr>
<th>Component</th>
<th>Page Time (ms)</th>
<th>Size</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StDev</td>
<td>(bytes)</td>
</tr>
<tr>
<td>BDBComp Browse Author</td>
<td>49.35</td>
<td>7.05</td>
<td>12902</td>
</tr>
<tr>
<td>WS-ODL Browse Author</td>
<td>1598.11</td>
<td>33.67</td>
<td>27896</td>
</tr>
<tr>
<td>BDBComp Browse Collection</td>
<td>676.97</td>
<td>13.70</td>
<td>47786</td>
</tr>
<tr>
<td>WS-ODL Browse Collection</td>
<td>112.61</td>
<td>6.17</td>
<td>49779</td>
</tr>
<tr>
<td>BDBComp Recent Works</td>
<td>44.56</td>
<td>3.01</td>
<td>6412</td>
</tr>
<tr>
<td>WS-ODL Recent Works</td>
<td>691.11</td>
<td>3.87</td>
<td>658.78</td>
</tr>
</tbody>
</table>

In Table 7, the column Size represents the size (in bytes) of the pages built by each component – in the case of our framework – or service – in the case of BD-BComp. Column Rate shows the page building rate for our components and the BD-BComp services. This measure was introduced in order to make their
response times comparable, since they produce pages with uneven sizes. From the table, we can observe that BDBComp presented a quite superior performance relatively to our framework. A main reason explain this fact: BDBComp is a digital library built from scratch, with an integrated architecture and a database specifically designed for it. It was built to be as efficient as possible. Our framework, in turn, was developed upon several software layers with generality as its main goal. The overhead introduced by this loosely-coupled, multilayer architecture naturally reduces our performance when compared to such a targeted system as BDBComp.

6 Conclusions and Future Work

This work presents a framework for the construction of digital libraries according to a componentized approach. The framework is divided in three layers: the data repository (in this case based on the Fedora architecture), the Web services, and the client. This architecture allows the distribution of the digital library components, which, by itself, leads to a greater flexibility of the system. Moreover, the use of Web services facilitates interoperability and standardized access to the system data, by human or machine users. The framework is accompanied by a wizard-like tool that implements a workflow process, driving the designer along the configuration of the framework.

Fedora utilization readily provides to the framework a repository to store digital objects, which can be simple, like the ones used in our experiments, or complex, like multimedia objects. Also, Fedora provides some basic infrastructure services, like an OAI data provider. The use of the relational database technology provides support to create DL components with advanced functions, such as search combining structured and textual data and multidimensional browsing. It also improves the performance of the framework, since the above components get data from a database, instead of accessing the Fedora repository, which is slower. The generality of the framework is also improved, since the databases provide a separation between Fedora and the components. This way, Fedora can be replaced by another repository without affecting these components.

In the client layer, the DL administrator has the flexibility to create the layout that is most suited to the users, through the use of XML style sheets (XSL), which are informed to the components via XML configuration files. Although XSL creation is not a trivial task, the framework provides basic XSL models for each component, and these can be improved by the users. Alternatively they can use graphical third-party tools to create the styles they want.

Through an experimental evaluation involving Computer Science and Library
and Information Science users, we showed that the WS-ODL framework is feasible and presents performance superior to the ODL framework, which inspired the development of ours. We also showed that providing a wizard-like tool to guide the framework installation is key to facilitate its use by non-specialist users when deploying a DL.

Nevertheless, the framework presents some problems. A major problem comes from the Kowari triplestore used by Fedora. Its memory leaks and RDF triples generation slow down the data insertion process, or even makes it impracticable, as seen in Section 5.2. From the discussion above, it is important to analyze what will be the volume of data treated by the digital library being built, and whether large volumes are to be inserted at once. In these cases, Fedora, and therefore our framework, may not be a good solution in their current version. The use of another repository is a possible way to solve this.

In the future, we plan to extend the wizard tool in order to support the installation of systems other than digital libraries, and to enhance some of its interface aspects based on users’ suggestions as well as on observations we made during the experiment sessions. We also plan to address the problems that come from the duplication of repository data in the databases and to investigate the use of other repositories. Currently, we are upgrading to Fedora 2.1 and intend to test the framework in a 64-bit machine. Besides that, we plan to use the framework to improve the services or even substitute the current version of BDBComp.

7 Acknowledgments

This work was partially supported by projects I3DL (MCT/CNPq/Protem grant number 680154/01-9) and 5S-VQ (MCT/CNPq/CT-INFO grant number 551013/2005-2). Authors would like to thank Elisa Tuler de Albergaria e Raquel Oliveira Prates for their valuable help in preparing and conducting our usability experiments as well as all the volunteers that participated in these experiments.

References


