Greenstone3 : A modular digital library.

Katherine Don

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Greenstone Digital Library Version 3 is a complete redesign and reimplementation of the Greenstone digital library software. The current version (Greenstone2) enjoys considerable success and is being widely used. Greenstone3 will capitalize on this success, and in addition it will

- improve flexibility, modularity, and extensibility
- lower the bar for "getting into" the Greenstone code with a view to understanding and extending it
- use XML where possible internally to improve the amount of self-documentation
- make full use of existing XML-related standards and software
- provide improved internationalization, particularly in terms of sort order, information browsing, etc.
- include new features that facilitate additional "content management" operations
- operate on a scale ranging from personal desktop to corporate library
- easily permit the incorporation of text mining operations
- use Java, to encourage multilinguality, X-compatibility, and to permit easier inclusion of existing Java code (such as for text mining).

Parts of Greenstone will remain in other languages (e.g. MG, MGPP); JNI (Java Native Interface) will be used to communicate with these.

A description of the general design and architecture of Greenstone3 is covered by the document *The design of Greenstone3: An agent based dynamic digital library* (design-2002.ps, in the docs/manual directory).

This documentation consists of several parts. Section 1 is for administrators, and covers Greenstone3 installation, how to access the library, and some administration issues. Section 2 is for users of the software, and looks at using the sample collections, creating new collections, and how to make small customizations to the interface. The remaining sections are aimed towards the Greenstone developer. Section 3 describes the run-time system, including the structure of the software, and the message format. Section 4 describes how to add new features to Greenstone, such as how to add new services, new page types, new plugins for different document formats. Section 5 describes how to make Greenstone run in a distributed

fashion, using SOAP as an example communications protocol. Finally, there are several appendices, including how to install Greenstone from CVS, some notes on Tomcat and SOAP, and a comparison of Greenstone2 and Greenstone3 format statements.

Contents

1 Greenstone installation and administration **5**
- 1.1 Get and install Greenstone 5
- 1.2 How the library works 5
  - 1.2.1 Restarting the library 6
- 1.3 Directory structure 6
- 1.4 Sites and interfaces 6
- 1.5 Configuring Tomcat 8
- 1.6 Configuring a Greenstone library 8
  - 1.6.1 Site configuration file 9
  - 1.6.2 Interface configuration file 11
- 1.7 Run-time re-initialization 11

2 Using Greenstone3 **14**
- 2.1 Using a collection 14
- 2.2 Building a collection 15
  - 2.2.1 Using the Librarian Interface 15
  - 2.2.2 Importing from Greenstone2 16
  - 2.2.3 Using command line building 16
- 2.3 Collection configuration files 18
  - 2.3.1 collectionInit.xml 18
  - 2.3.2 collectionConfig.xml 20
  - 2.3.3 buildConfig.xml 22
- 2.4 Formatting the collection 22
  - 2.4.1 Changing the service text strings 27
- 2.5 Customizing the interface 29
  - 2.5.1 Modifying an existing interface 29
  - 2.5.2 Defining a new interface 30
  - 2.5.3 Changing the interface language 30

3 Developing Greenstone3: Run-time system **32**
- 3.1 Overview of modules?? 32
- 3.2 Start up configuration 33
- 3.3 Message passing 35
- 3.4 'describe'-type messages 35
- 3.5 'system'-type messages 41
- 3.6 'format'-type messages 42
- 3.7 'status'-type messages 42
- 3.8 'process'-type messages 44
  - 3.8.1 'query'-type services 45
  - 3.8.2 'browse'-type services 46
  - 3.8.3 'retrieve'-type services 47
  - 3.8.4 'process'-type services 49

3.8.5 'applet'-type services . . . 50
3.8.6 'enrich'-type services . . . 51
3.9 Page generation . . . 52
3.9.1 'page'-type requests and their arguments . . . 52
3.9.2 page format . . . 53
3.9.3 Receptionists . . . 55
3.9.4 Collection specific formatting . . . 55
3.9.5 CGI arguments . . . 55
3.9.6 Page action . . . 55
3.9.7 Query action . . . 56
3.9.8 Applet action . . . 56
3.9.9 Document action . . . 57
3.9.10 XML Document action . . . 57
3.9.11 GS2Browse action . . . 58
3.9.12 System action . . . 58
3.10 Other code information . . . 58

4 Developing Greenstone3 : Adding new features **59**
4.1 Creating and using new services . . . 59
4.1.1 Creating the service . . . 60
4.1.2 Loading the service . . . 60
4.1.3 Using the service . . . 60
4.2 creating new actions/pages . . . 61
4.3 new interfaces . . . 61
4.4 New types of collections . . . 61
4.5 The gs2 Interface . . . 64

5 Distributed Greenstone **65**
5.1 Serving a site using soap . . . 65
5.2 Connecting to a site web service . . . 66

A Using Greenstone3 from CVS **67**

B Tomcat **68**
B.1 Proxying Tomcat with apache . . . 69
B.2 Running Tomcat behind a proxy . . . 69

C SOAP **70**
C.1 Debugging SOAP . . . 70

D Tidying up the formatting for imported Greenstone2 collections **71**
D.1 Format statements: Greenstone2 vs Greenstone3 . . . 71
D.2 Cleaning up macros . . . 71

1 Greenstone installation and administration

This section covers where to get Greenstone3 from, how to install it and how to run it. The standard method of running Greenstone3 is as a Java servlet. We provide the Tomcat servlet container to run the servlet. Standard web servers may be able to be configured to provide servlet support, and thereby remove the need to use Tomcat. Please see your web server documentation for this. This documentation assumes that you are using Tomcat. To access Greenstone3, Tomcat must be started up, and then it can be accessed via a web browser.

Ant (Java's XML based build tool) is used for compilation, installation and running Greenstone. The `build.xml` file is the configuration file for the Greenstone project, and `build.properties` contains parameters that can be altered by the user.

1.1 Get and install Greenstone

Greenstone3 is available for download from Sourceforge:
`https://sourceforge.net/projects/greenstone3`. There are Windows, Linux, and source releases. The binary releases are self-installing executables: download and run the file to install. A series of prompts will guide you through the installation process. The source release is a gzip'd tar file. Unzip and untar this, check build.properties, then run `'ant install'` to configure and compile the code.

The Greenstone3 library can be launched by running the server program. This is accessible from the Start menu on Windows, or by running the `gs3-server.sh/bat` script in the top level `greenstone3` directory. This program will start up the Tomcat web server and launch a browser.

Alternatively, you can start it up using Ant: run `'ant start'`, which starts up Tomcat, then in a browser go to `http://localhost:8080/greenstone3`
(or `http://your-computer-name:your-chosen-port/greenstone3`).
This gets you to a welcome page containing links to four servlets: the `test` servlet (this allows you to check that Tomcat is running properly); the standard `library` servlet which serves `localsite` site with the `gs2` interface; the `gs3library` servlet which serves `localsite` using the `default` Greenstone3-style interface; and the `gateway` servlet, which serves `gateway` site with the `default` interface. The `gateway` site uses a SOAP connection to communicate with `localsite`, and demonstrates the library working in a distributed fashion. The SOAP connection is not enabled by default - to enable it, run `'ant deploy-localsite'`.

Greenstone3 is also available through CVS (Concurrent Versioning System). This provides the latest development version, and is not guaranteed to be stable. Appendix A describes how to download and install Greenstone3 from CVS.

1.2 How the library works

The standard library program is a Java servlet. We use the Tomcat servlet container to present the servlets over the web. Tomcat takes CGI-style URLs and passes the

arguments to the servlet, which processes these and returns a page of HTML. As far as an end-user is concerned, a servlet is a Java version of a CGI program. The interaction is similar: access is via a web browser, using arguments in a URL.

Other types of interfaces can be used, such as Java GUI programs. See Section 4.3 for details about how to make these.

1.2.1 Restarting the library

You can restart Tomcat by clicking 'Restart Server' on the little server program. You should restart the server any time you make changes in the following for those changes to take effect:

- `$GSDL3HOME/WEB-INF/web.xml`
- `$GSDL3SRCHOME/packages/tomcat/conf/server.xml`
- any classes or jar files used by the servlets

1.3 Directory structure

Table 1 shows the file hierarchy for Greenstone3. The first part shows the common stuff which can be shared between Greenstone users—the source, libraries etc. The second part shows the file hierarchy for the web directory, which comprises the greenstone3 context for Tomcat, and is accessible via Tomcat. The main directories are for sites and interfaces: there can be several sites and interfaces per installation, and they are described in the following section.

Two environment variables used by Greenstone3 are often mentioned in this manual: `$GSDL3SRCHOME` and `$GSDL3HOME`. `$GSDL3SRCHOME` refers to the top-level `greenstone3` directory, while `$GSDL3HOME` refers to the `web` directory. The web directory contains everything needed to serve the Greenstone3 library using Tomcat, and doesn't necessarily need to live with the rest of the Greenstone3 source.

1.4 Sites and interfaces

Sites and interfaces contain the content and presentation information, respectively, for the digital library. A site is comprised of a set of collections and possibly some site-wide services. An interface (in this web-based servlet context) is a set of images along with a set of XSLT files used for translating xml output from the library into an appropriate form—HTML in general.

One Greenstone3 installation can have many sites and interfaces, and these can be paired in different combinations. One instantiation of a servlet uses one site and one interface, so every specified pairing results in a new servlet instance. For example, a single site might be served with two different interfaces. This provides different modes of access to the same content. e.g. HTML vs WML, or perhaps

Table 1: The Greenstone directory structure

directory	description
greenstone3	The main installation directory—$GSDL3SRCHOME is set to this directory
greenstone3/src	Source code lives here
greenstone3/src/java/	main Greenstone3 java source code
greenstone3/src/packages	Imported source packages from other systems e.g. indexing packages may go here
greenstone3/lib	Shared library files
greenstone3/lib/java	Java jar files not needed in the Greenstone3 runtime
greenstone3/lib/jni	Jar files and shared library files (.so, .jnilib, .dll) needed for JNI components
greenstone3/resources	any resources that may be needed
greenstone3/resources/soap	soap service description files
greenstone3/bin	executable stuff lives here
greenstone3/bin/script	some Perl and/or shell scripts
greenstone3/packages	External packages that may be installed as part of greenstone, e.g. Tomcat
greenstone3/docs	Documentation
greenstone3/gli	Greenstone Librarian Interface code
greenstone3/gs2build	collection building code
greenstone3/web	This is where the web site is defined. Any static HTML files can go here. This directory is the root directory used by Tomcat when serving Greenstone3. $GSDL3HOME is set to this directory.
greenstone3/web/WEB-INF	The web.xml file lives here (servlet configuration information for Tomcat)
greenstone3/web/WEB-INF/classes	Individual class files needed by the servlet go in here, also properties files for java resource bundles - used to handle all the language specific text. This directory is on the servlet classpath
greenstone3/web/WEB-INF/lib	jar files needed by the servlets go here
greenstone3/web/sites	Contains directories for different sites—a site is a set of collections and services served by a single MessageRouter (MR). The MR may have connections (e.g. soap) to other sites
greenstone3/web/sites/localsite	An example site - the site configuration file lives here
greenstone3/web/sites/localsite/collect	The collections directory
greenstone3/web/sites/localsite/images	Site specific images
greenstone3/web/sites/localsite/transforms	Site specific transforms
greenstone3/web/interfaces	Contains directories for different interfaces - an interface is defined by its images and XSLT files
greenstone3/web/interfaces/default	The default interface
greenstone3/web/interfaces/default/images	The images for the default interface
greenstone3/web/interfaces/default/js	The javascript libraries for the default interface
greenstone3/web/interfaces/default/style	The CSS stylesheets for the default interface
greenstone3/web/interfaces/default/transforms	The XSLT files for the default interface
greenstone3/web/applet	jar files needed by applets can go here

providing a completely different look and feel for different audiences. Alternatively, a standard interface may be used with many different sites—providing a consistent mode of access to a lot of different content.

Collections live in the `collect` directory of a site. Any collections that are found in this directory when the servlet is initialized will be loaded up. Public collections will appear on the library home page, while private collections will be hidden. These can still be accessed by typing in cgi arguments. Collections require valid configuration files, but apart from this, nothing needs to be done to the site to use new collections. Collections added while Tomcat is running will not be noticed automatically. Either the server needs to be restarted, or a configuration request may be sent to the library, triggering a (re)load of the collection (this is described in Section 1.7).

There are two sites that come with the distribution: `localsite`, and `gateway`. `localsite` has several demo collections, while `gateway` has none. `gateway` specifies that a SOAP connection should be made to `localsite`. Getting this to work involves setting up a soap server for localsite: see Section 5 for details. There are also two interfaces provided in the distribution: `default` and `gs2`. The default interface is a generic Greenstone3 interface, while the `gs2` interface aims to look like the old Greenstone2 interface.

Each site and interface has a configuration file which specifies parameters for the site or interface—these are described in Section 1.6.

1.5 Configuring Tomcat

The file `$GSDL3HOME/WEB-INF/web.xml` contains the configuration information for Tomcat. It tells Tomcat what servlets to load, what initial parameters to pass them, and what web names map to the servlets. There are four servlets specified in web.xml (these correspond to the four servlet links in the welcome page for Greenstone3): one is a test servlet that just prints "hello greenstone" to a web page. This is useful if you are having trouble getting Tomcat set up. The other three are the Greenstone library servlets described in Section 1.1, `library`, `gs3library` and `gateway`. Each servlet must specify which site and which interface to use. Having multiple servlets provides a way of serving different sites, or the same site with a different style of presentation. `site_name` and `interface_name` are just two examples of initialization parameters used by the library servlets. The full list is shown in Table 2.

For more details about Tomcat see Appendix B.

1.6 Configuring a Greenstone library

Initial Greenstone3 system configuration is determined by a set of XML configuration files. Each site has a configuration file that binds parameters for the site, `siteConfig.xml`. Each interface has a configuration file, `interfaceConfig.xml`, that specifies parameters for the interface. Collections also have several config-

Table 2: Greenstone servlet initialization parameters

name	sample value	description
library_name	library	the web name of the servlet
interface_name	default	the name of the interface to use
site_name	localsite	the name of the local site to use (use either site_name or the three remote_site parameters)
remote_site_name	org.greenstone.site1	the name of a remote site (can be anything??)
remote_site_type	soap	the type of server running on the site
remote_site_address	http://www.greenstone.org/greenstone3/services/localsite	The address of the server
default_lang	en	the default language for the interface
receptionist_class	MyReceptionist	(optional) specifies an alternative Receptionist to use (default is DefaultReceptionist)
messagerouter_class	NewMessageRouter	(optional) specifies an alternative MessageRouter to use (default is MessageRouter)
params_class	GS2Params	(optional) specifies an alternative GSParams class to use

uration files; these are discussed in Section 2.3. The configuration files are read in when the system is initialized, and their contents are cached in memory. This means that changes made to these files once the system is running will not take immediate effect. Tomcat needs to be restarted for changes to the interface configuration file to take effect. However, changes to the site configuration file can be incorporated sending a system command to the library. There are a series of system commands that can be sent to the library to induce reconfiguration of different modules, including reloading the whole site. This removes the need to restart the system to reflect these changes. These commands are described in Section 1.7.

1.6.1 Site configuration file

The file `siteConfig.xml` specifies the URI for the site (`localSiteName`), the HTTP address for site resources (`httpAddress`), any `ServiceClusters` that the site provides (for example, collection building), any `ServiceRacks` that do not belong to a cluster or collection, and a list of known external sites to connect to. Collections are not specified in the site configuration file, but are determined by the contents of the site's collect directory.

The HTTP address is used for retrieving resources from a site outside the XML protocol. Because a site is HTTP accessible through Tomcat, any files (e.g. images) belonging to that site or to its collections can be specified in the HTML of a page by a URL. This avoids having to retrieve these files from a remote site via the XML protocol[1].

[1]Currently, sites live inside the Tomcat greenstone3 root context, and therefore all their content is accessible over HTTP via the Tomcat address. We need to see if parts can be restricted. Also, if we use a different protocol, then resources from remote sites may need to come through the XML. Also, if we are running locally without using Tomcat, we may want to get them via file:// rather than

```
<siteConfig>
  <localSiteName value="org.greenstone.localsite"/>
  <httpAddress value="http://localhost:8080/greenstone3/sites/localsite"/>
  <serviceClusterList/>
  <serviceRackList/>
  <siteList/>
</siteConfig>

<siteConfig>
  <localSiteName value="org.greenstone.gsdl1"/>
  <httpAddress value="http://localhost:8080/greenstone3/sites/gsdl1"/>
  <serviceClusterList>
    <serviceCluster name="build">
      <metadataList>
        <metadata name="Title">Collection builder</metadata>
        <metadata name="Description">Builds collections in a
           gsdl2-style manner</metadata>
      </metadataList>
      <serviceRackList>
        <serviceRack name="GS2Construct"/>
      </serviceRackList>
    </serviceCluster>
  </serviceClusterList>
  <siteList>
    <site name="org.greenstone.localsite"
      address="http://localhost:8080/greenstone3/services/localsite"
      type="soap"/>
  </siteList>
</siteConfig>
```

Figure 1: Two sample site configuration files

Figure 1 shows two example site configuration files. The first example is for a rudimentary site with no site-wide services, which does not connect to any external sites. The second example is for a site with one site-wide service cluster - a collection building cluster. It also connects to the first site using SOAP. These two sites happen to be running on the same machine, which is why they can use `localhost` in the address. For site `gsdl1` to talk to site `localsite`, a SOAP server must be run for `localsite`. The address of the SOAP server, in this case, is `http://localhost:8080/greenstone3/services/localsite`.

Another element that can appear in a site configuration file is `replaceList`. This must have an `id` attribute, and may contain one or more `replace` elements. Replace elements are discussed in Section 2.3. The list found in a `siteConfig.xml` file can be applied to any collection by adding a `replaceListRef` element (with the appropriate `id` attribute) to its `collectionConfig.xml` file.

---

http://.

1.6.2 Interface configuration file

The interface configuration file `interfaceConfig.xml` lists all the actions that the interface knows about at the start (other ones can be loaded dynamically). Actions create the web pages for the library: there is generally one Action per type of page. For example, a query action produces the pages for searching, while a document action displays the documents. The configuration file specifies what short name each action maps to (this is used in library URLs for the a (action) parameter) e.g. QueryAction should use `a=q`. If the interface uses XSLT, it specifies what XSLT file should be used for each action and possibly each subaction. This makes it easy for developers to implement and use different actions and/or XSLT files without recompilation. The server must be restarted, however.

It also lists all the languages that the interface text files have been translated into. These have a `name` attribute, which is the ISO code for the language, and a `displayElement` which gives the language name in that language (note that this file should be encoded in UTF-8). This language list is used on the Preferences page to allow the user to change the interface language. Details on how to add a new language to a Greenstone3 library are shown in Section 2.5.3.

An `optionList` element can be used to disable or enable some optional functionality for the interface. Currently there are three options that can be enabled:

highlightQueryTerms	Whether search term highlighting is available or not
berryBaskets	Whether berry basket functionality is available or not
displayAnnotationService	Whether any annotation services (specified in the site config file) should be displayed with a document or not.

An interface may be based on an existing one, for example, the gs2 interface is based on the default interface. This means that it will use any images or templates from the base one unless overridden in the current one. The `baseInterface` attribute of the `<interfaceConfig>` element is used to specify the base interface.

1.7 Run-time re-initialization

When Tomcat is started up, the site and interface configuration files are read in, and actions/services/collections loaded as necessary. The configuration is then static unless Tomcat is restarted, or re-configuration commands issued.

There are several commands that can be issued to Tomcat to avoid having to restart the server. These can reload the entire site, or just individual collections. Unfortunately at present there are no commands to reconfigure the interface, so if the interface configuration file has changed, Tomcat must be restarted for those changes to take effect. Similarly, if the Java classes are modified, Tomcat must be restarted then too.

Currently, the runtime configuration commands can only be accessed by typing

```
<interfaceConfig>
  <actionList>
    <action name='p' class='PageAction'>
      <subaction name='home' xslt='home.xsl'/>
      <subaction name='about' xslt='about.xsl'/>
      <subaction name='help' xslt='help.xsl'/>
      <subaction name='pref' xslt='pref.xsl'/>
      <subaction name='nav' xslt='nav.xsl'/><!-- used for the
            collection header frame -->
      <subaction name="html" xslt="html.xsl"/> <!-- used to put an
            external page into a frame with a collection header-->
    </action>
    <action name='q' class='QueryAction' xslt='basicquery.xsl'/>
    <action name='b' class='GS2BrowseAction' xslt='classifier.xsl'/>
    <action name='a' class='AppletAction' xslt='applet.xsl'/>
    <action name='d' class='DocumentAction' xslt='document.xsl'/>
    <action name='xd' class='XMLDocumentAction'>
      <subaction name='toc' xslt='document-toc.xsl'/>
      <subaction name='text' xslt='document-content.xsl'/>
    </action>
    <action name='pr' class='ProcessAction' xslt='process.xsl'/>
    <action name='s' class='SystemAction' xslt='system.xsl'/>
    <action name='g' class='GeneralAction'>
      <subaction name="berry" xslt='berry.xsl'/>
    </action>
  </actionList>
  <languageList>
    <language name="en">
      <displayItem name='name'>English</displayItem>
    </language>
    <language name="fr">
      <displayItem name='name'>Franais</displayItem>
    </language>
    <language name='es'>
      <displayItem name='name'>Espaol</displayItem>
    </language>
  </languageList>
  <optionList>
    <option name="highlightQueryTerms" value="true"/>
    <option name="berryBaskets" value="true"/>
  </optionList>
</interfaceConfig>
```

Figure 2: Default interface configuration file

Table 3: Example run-time configuration arguments.

`a=s&sa=c`	reconfigures the whole site. Reads in siteConfig.xml, reloads all the collections. Just part of this can be specified with another argument `ss` (system subset). The valid values are `collectionList`, `siteList`, `serviceList`, `clusterList`.
`a=s&sa=c&sc=XXX`	reconfigures the XXX collection or cluster. `ss` can also be used here, valid values are `metadataList` and `serviceList`.
`a=s&sa=a`	(re)activate a specific module. Modules are specified using two arguments, `st` (system module type) and `sn` (system module name). Valid types are `collection`, `cluster` `site`.
`a=s&sa=d`	deactivate a module. `st` and `sn` can be used here too. Valid types are `collection`, `cluster`, `site`, `service`. Modules are removed from the current configuration, but will reappear if Tomcat is restarted.
`a=s&sa=d&sc=XXX`	deactivate a module belonging to the XXX collection or cluster. `st` and `sn` can be used here too. Valid types are `service`.

arguments into the URL; there is no nice web form yet to do this.

The arguments are entered after the `library?` part of the URL. There are three types of commands: configure, activate, deactivate. These are specified by `a=s&sa=c`, `a=s&sa=a`, and `a=s&sa=d`, respectively (`a` is action, `sa` is subaction). By default, the requests are sent to the MessageRouter, but they can be sent to a collection/cluster by the addition of `sc=xxx`, where `xxx` is the name of the collection or cluster. Table 3 describes the commands and arguments in a bit more detail.

2 Using Greenstone3

Once Greenstone3 is installed, the sample collections can be accessed. The installation comes with several example collections, and Section 2.1 describes these collections and how to use them. Section 2.2 describes how to build new collections.

2.1 Using a collection

A collection typically consists of a set of documents, which could be text, HTML, word, PDF, images, bibliographic records etc, along with some access methods, or "services". Typical access methods include searching or browsing for document identifiers, and retrieval of content or metadata for those identifiers. Searching involves entering words or phrases and getting back lists of documents that contain those words. The search terms may be restricted to particular fields of the document.

Browsing involves navigating pre-defined hierarchies of documents, following links of interest to find documents. The hierarchies may be constructed on different metadata fields, for example, alphabetical lists of Titles, or a hierarchy of Subject classifications. Clicking on a bookshelf icon takes you to a lower level in the hierarchy, while clicking on a book or page icon takes you to a document.

In the standard interface that comes with Greenstone3 [2], collections in a digital library are presented in the following manner. The 'home' page of the library shows a list of all the public collections in that library. Clicking on a collection link takes you to the home page for the collection, which we call the collection's 'about' page. The standard page banner for a collection looks something like that shown in Figure 3.

Figure 3: A sample collection page banner

The image at the top left is a link to the collection's home page. The top right has buttons to link to the library home page, help and preferences pages. All the available services are arrayed along a navigation bar, along the bottom of the banner. Clicking on a name accesses that service.

Search type services generally provide a form to fill in, with parameters including what field or granularity to search, and the query itself. Clicking the search button carries out the search, and a list of matching documents will be displayed. Clicking on the icons in the result list takes you to the document itself.

[2] of course, this is all customizable

Once you are looking at a document, clicking the open book icon at the top of the document, underneath the navigation bar, will take you back to the service page that you accessed the document from.

2.2 Building a collection

There are three ways to get a new collection into Greenstone3. The most common way is to use the Greenstone Librarian Interface to create a collection. If you have existing collections in a Greenstone2 installation, these can be imported into Greenstone3. Thirdly, you can use the Perl command line building scripts directly.

Collections live in the `collect` directory of a site. As described in Section 1.4, there can be several sites per Greenstone3 installation. The collect directory is at `$GSDL3HOME/sites/site-name/collect`, where site-name is the name of the site you want your new collection to belong to.

The following three sections briefly describe how to create a collection using GLI, how to import a collection from Greenstone2, and how to use command line building. Once a collection has been built (and is located in the collect directory), the library server needs to be notified that there is a new collection. This can be accomplished in two ways[3]. If you are the library administrator, you can restart Tomcat. The library servlet will then be created afresh, and will discover the new collection when it scans the collect directory for the collection list. Alternatively, an activate collection command can be issued to the servlet, using the arguments `a=s&sa=a&st=collection&sn=collname`, where `collname` should be replaced with the collection name—this tells the library program to (re)load the `collname` collection.

2.2.1 Using the Librarian Interface

The Greenstone Librarian Interface (GLI) can be used to create collections. The procedure is the same as for Greenstone2, but it works in a Greenstone3 context. It can be started under Windows by selecting Greenstone Librarian Interface from the Greenstone 3 Digital Library menu in the Program Files section of the Start menu. On Linux, run `ant gli` from the `greenstone3` directory, or run `./gli4gs3.sh` from the `$GSDL3SRCHOME/gli` directory.

Currently, the GLI works almost exactly the same as for Greenstone2[4]. Collection configuration is done in a Greenstone2 manner. The main difference is that Greenstone3 has different sites and interfaces and servlets, whereas Greenstone2 has a single collect directory, and a single runtime cgi program.

The GLI for Greenstone3 has a couple of new configuration parameters: site and servlet. It operates within a single site—you can edit, delete, and create new collections within this site. A servlet is also specified for that site—this is used when previewing a collection. While you are working in one site, you cannot

[3]and eventually there will also probably be automatic polling for new collections

[4]Eventually the GLI will be modified to use Greenstone3 XML configuration files.

edit collections from another site. However, you can base a collection on one from another site. To change the working site and/or servlet, go to Preferences->Connection in the File menu. By default, the GLI will use site `localsite`, and servlet `library`.

Collection building using the GLI will use the Greenstone2 Perl scripts and plugins. At the conclusion of the Greenstone2 build process, a conversion script will be run to create the Greenstone3 configuration files. This means that format statements are no longer 'live'—changing these will require changes to the Greenstone3 configuration files. Clicking the Preview Collection button will re-run the configuration file conversion script. If you change anything on the Format panel, you will need to click Preview Collection. Just reloading the collection via a browser will not be enough.

Detailed instructions about using the GLI can be found in Sections 3.1 and 3.2 of the Greenstone2 User's Guide (`GS2-User-en.pdf`). This can be found in your Greenstone2 installation, or in the `$GSDL3SRCHOME/docs/manual` directory if you have installed Greenstone3 from a distribution.

2.2.2 Importing from Greenstone2

Pre-built Greenstone2 collections can also be used in Greenstone3. The collection folder should be copied to the collect directory of the site it is to appear in (or a symbolic link may be used if possible). The Greenstone3 run time system requires different configuration files for a collection, so you need to run a conversion script. All this does is create the new `collectionConfig.xml` and `buildConfig.xml` from the old `collect.cfg` and `build.cfg` files. It does not change the collection in any way, so it can still be used by Greenstone2 software.

The conversion script is `convert_coll_from_gs2.pl`. To run it, make sure you have run `source setup.bash` (or `setup` in Windows) in the `$GSDL3SRCHOME/gs2build` directory (as well as running the standard `gs3-setup` command). Then you need to specify the path to the collect directory and the collection name as parameters to the conversion script. For example,

```
convert_coll_from_gs2.pl -collectdir
   $GSDL3HOME/sites/localsite/collect gs2mgdemo
```

The script attempts to create Greenstone3 format statements from the old Greenstone2 ones. The conversion may not always work properly, so if the collection looks a bit strange under Greenstone3, you should check the format statements. Format statements are described in Section 2.4.

Once again, to have the collection recognized by the library servlet, you can either restart Tomcat, or load it dynamically.

2.2.3 Using command line building

This is the same procedure as for Greenstone2 command line building, with the addition of a final step to create the Greenstone3 configuration files. The basic

steps are (for a new collection called testcol):

Linux:

```
cd greenstone3
source gs3-setup.sh
cd gs2build
source setup.bash
cd ../
mkcol.pl -collectdir $GSDL3HOME/sites/localsite/collect testcol
put source documents and metadata into
            $GSDL3HOME/sites/localsite/collect/testcol/import
edit $GSDL3HOME/sites/localsite/collect/testcol/etc/collect.cfg as
            appropriate
import.pl -collectdir $GSDL3HOME/sites/localsite/collect testcol
buildcol.pl -collectdir $GSDL3HOME/sites/localsite/collect testcol
rename the $GSDL3HOME/sites/localsite/collect/testcol/building
            directory to index
convert_coll_from_gs2.pl -collectdir $GSDL3HOME/sites/localsite/collect
            testcol
%$
```

Windows:

```
cd greenstone3
gs3-setup
cd gs2build
setup
cd ..
perl -S mkcol.pl -collectdir %GSDL3HOME%\sites\localsite\collect testcol
put source documents and metadata into
            %GSDL3HOME%\sites\localsite\collect\testcol\import
edit %GSDL3HOME%\sites\localsite\collect\testcol\etc\collect.cfg as
            appropriate
perl -S import.pl -collectdir %GSDL3HOME%\sites\localsite\collect testcol
perl -S buildcol.pl -collectdir %GSDL3HOME%\sites\localsite\collect testcol
rename the %GSDL3HOME%\sites\localsite\collect\testcol\building directory
            to index
perl -S convert_coll_from_gs2.pl -collectdir
            %GSDL3HOME%\sites\localsite\collect testcol
```

Once the build process is complete, Tomcat should be prompted to reload the collection—either by restarting the server, or by sending an activate collection command to the library servlet.

Metadata for documents can be added using `metadata.xml` files. A `metadata.xml` file has a root element of `<DirectoryMetadata>`. This encloses a series of `<FileSet>` items. Neither of these tags has any attributes. Each `<FileSet>` item includes two parts: firstly, one or more `<FileName>` tags, each of which encloses a regular expression to identify the files which are to be assigned the metadata. Only files in the same directory as the `metadata.xml` file, or in one of its child directories, will be selected. The filename tag encloses the regular expression as text, e.g.:

```
<FileName>example</FileName>
```

This would match any file containing the text 'example' in its name. The second part of the `<FileSet>` item is a `<Description>` item. The `<Description>` tag has no attributes, but encloses one or more `<Metadata>` tags. Each `<Metadata>` tag contains one metadata item, i.e. a label to describe the metadata and a corresponding value. The `<Metadata>` tag has one compulsory attribute: `'name'`. This attribute gives the metadata label to add to the document. Each `<Metadata>` tag also has an optional attribute: `'mode'`. If this attribute is set to `'accumulate'` then the value is added to the document, and any existing values for that metadata item are retained. If the attribute is set to `'set'` or is omitted, then any existing value of the metadata item will be deleted.

Figure 4 shows an example metadata.xml file. Here, only one file pattern is found in each file set. However, the `Description` tag contains a number of separate metadata items. Note that the `Title` metadata does not have the `mode=accumulate` attribute. This means that when this title is assigned to a document, any existing `Title` information will be lost.

2.3 Collection configuration files

Each collection has two, or possibly three, Greenstone3 configuration files, `collectionConfig.xml`, `buildConfig.xml`, and optionally `collectionInit.xml`, that give metadata, display and other information for the collection. Currently, `collectionConfig.xml` and `buildConfig.xml` are generated from `collect.cfg` and `build.cfg`. At some stage, the collection building process and the Librarian Interface will be modified to use these files directly. `collect.cfg` and/or `collectionConfig.xml` includes user-defined presentation metadata for the collection, such as its name and the *About this collection* text; gives formatting information for the collection display; and also gives instructions on how the collection is to be built. `build.cfg` and/or `buildConfig.xml` are produced by the build-time process and include any metadata that can be determined automatically. It also includes configuration information for any ServiceRacks needed by the collection.

All the configuration files should be encoded using UTF-8.

The format of `collect.cfg` and `build.cfg` are not discussed here. Please see the Greenstone2 manuals for more information regarding these files.

2.3.1 collectionInit.xml

This optional file is only used for non-standard, customized collections. It specifies the class name of the non-standard collection class. The only syntax so far is the class name:

```
<collectionInit class="XMLCollection"/>
```

Section 4.4 describes an example collection where this file is used. Depending on the type of collection that this is used for, one or both of the other configuration files may not be needed.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DirectoryMetadata SYSTEM "http://greenstone.org/dtd/DirectoryMetadata
         /1.0/DirectoryMetadata.dtd">
<DirectoryMetadata>
  <FileSet>
    <FileName>ec160e</FileName>
    <Description>
      <Metadata name="Title">The Courier - No.160 - Nov - Dec 1996 -
         Dossier Habitat - Country reports: Fiji , Tonga (ec160e)</Metadata>
      <Metadata mode="accumulate" name="Language">English</Metadata>
      <Metadata mode="accumulate" name="Subject">Settlements and housing:
        general works incl. low- cost housing, planning techniques, surveying,
        etc.</Metadata>
      <Metadata mode="accumulate" name="Subject">The Courier ACP 1990 - 1996
        Africa-Caribbean-Pacific - European Union</Metadata>
      <Metadata mode="accumulate" name="Organization">EC Courier</Metadata>
      <Metadata mode="accumulate" name="AZList">T.1</Metadata>
    </Description>
  </FileSet>
  <FileSet>
    <FileName>b22bue</FileName>
    <Description>
      <Metadata name="Title">Butterfly Farming in Papua New Guinea
        (b22bue)</Metadata>
      <Metadata mode="accumulate" name="Language">English</Metadata>
      <Metadata mode="accumulate" name="Subject">Other animals (micro-
        livestock, little known animals, silkworms, reptiles, frogs,
        snails, game, etc.)</Metadata>
      <Metadata mode="accumulate" name="Organization">BOSTID</Metadata>
      <Metadata mode="accumulate" name="AZList">T.1</Metadata>
      <Metadata mode="accumulate" name="Keyword">start a butterfly farm
        </Metadata>
    </Description>
  </FileSet>
</DirectoryMetadata>
```

Figure 4: Sample metadata.xml file

2.3.2 collectionConfig.xml

The collection configuration file is where the collection designer (e.g. a librarian) decides what form the collection should take. So far this file only includes the presentation aspects needed by the run-time system. Instructions for collection building have yet to be defined. Presentation aspects include collection metadata such as title and description, display text for indexes, and format statements for search results, classifiers etc. The format of `collectionConfig.xml` is still under consideration. However, Figure 5 shows the parts of it that have been defined so far.

Display elements for a collection can be entered in any language—use `lang='en'` attributes to specify which language they are in.

The `<metadataList>` element specifies some collection metadata, such as creator. The `<displayItemList>` specifies some language dependent information that is used for collection display, such as collection name and short description. These `displayItem` elements can be specified in different languages.

The `<search>` element provides some display and formatting information for the search indexes, while the `<browse>` element concerns classifiers, and the `<display>` element looks at document display.

Inside the `<search>` and `<browse>` elements, `<displayItem>` elements are used to provide titles for the indexes or classifiers, while `<format>` elements provide formatting instructions, typically for a document or classifier node in a list of results. Placing the `<format>` instructions at the top level in the `search` or `browse` element will apply the format to all the indexes or classifiers, while placing it inside an individual `index` or `classifier` element will restrict that formatting instruction to that item.

The `<display>` element contains optional formatting information for the display of documents. Templates that can be specified here include `documentHeading` and `DocumentContent`. Other formatting options may also be specified here, such as whether to display a table of contents and/or cover image for the documents.

Format elements are described in Section 2.4.

An optional `<replaceList>` element can be included at the top level. This contains a list of strings and their replacements. This is particularly useful for Greenstone2 collections that use macros.

The format is like the following:

```
<replaceList>
<replace scope='text' macro="xxx" text="yyy"/>
<replace scope='metadata' macro="xxx" bundle="yyy" key="zzz"/>
<replace scope='all' macro='xxx' metadata='yyy'/>
</replaceList>
```

Scope determines on what text the replacements are carried out: `text`, `metadata`, and `all` (both text and metadata). An empty scope attribute is equivalent to scope=all. Each replace type can be used with all scope values. Replacing uses Java's 'String.replaceAll' functionality, so macro and replacement text are actually regular expressions. The

```
<collectionConfig xmlns:gsf="http://www.greenstone.org/greenstone3/
  schema/ConfigFormat" xmlns:xslt="http://www.w3.org/1999/XSL/Transform">
  <metadataList>
    <metadata name="creator">greenstone@cs.waikato.ac.nz</metadata>
    <metadata name="public">true</metadata>
  </metadataList>
  <displayItemList>
    <displayItem name=’name’ lang=’en’>Greenstone3 MG demo collection</displayItem>
    <displayItem name=’description’ lang=’en’>This is a demonstration
      collection for the Greenstone3 digital library software.</displayItem>
    <displayItem name=’icon’ lang=’en’>gs3mgdemo.gif</displayItem>
    <displayItem name=’smallicon’ lang=’en’>gs3mgdemo_sm.gif</displayItem>
  </displayItemList>
  <search>
    <index name="ste">
      <displayItem name=’name’ lang="en">chapters</displayItem>
      <displayItem name=’name’ lang="fr">chapitres</displayItem>
      <displayItem name=’name’ lang="es">captulos</displayItem>
    </index>
    [ ... more indexes ...]
    <format>
      <gsf:template match="documentNode"><td valign=’top’>
       <gsf:link><gsf:icon/></gsf:link></td><td><gsf:metadata name=’Title’/>
      </td></gsf:template>
    </format>
  </search>
  <browse>
    <classifier name="CL1" horizontalAtTop=’true’>
      <displayItem name=’name’ lang=’en’>Titles</displayItem>
    </classifier>
    [... more classifiers ...]
    <classifier name="CL4">
      <displayItem name=’name’ lang=’en’>HowTo</displayItem>
      <format>
        <gsf:template match="documentNode">
          <br /><gsf:link><gsf:metadata name=’Keyword’ />
            </gsf:link></gsf:template>
      </format>
    </classifier>
  </browse>
  <display>
    <format>
      <gsf:option name="coverImages" value="false"/>
      <gsf:option name="documentTOC" value="false"/>
    </format>
  </display>
</collectionConfig>
```

Figure 5: Sample collectionConfig.xml file

first example is a straight textual replacement. The second example uses dictionary lookups. xxx will be replaced with the (language-dependent) value for key zzz in resource bundle yyy. The third example uses metadata: xxx will be replaced by the value of the yyy metadata for that document.

Appendix D.2 gives some examples that have been used for Greenstone2 collections.

2.3.3 buildConfig.xml

The file `buildConfig.xml` is produced by the collection building process. Generally it is not necessary to look at this file, but it can be useful in determining what went wrong if the collection doesn't appear quite the way it was planned.

It contains metadata and other information about the collection that can be determined automatically, such as the number of documents in the collection. It also includes a list of `ServiceRack` classes that are required to provide the services that have been built into the collection. The serviceRack names are Java classes that are loaded dynamically at runtime. Any information inside the serviceRack element is specific to that service—there is no set format. Figure 6 shows an example. This configuration file specifies that the collection should load up 3 ServiceRacks: `GS2Browse`, `GS2MGPPRetrieve` and `GS2MGPPSearch`. The contents of each `<serviceRack>` element are passed to the appropriate ServiceRack objects for configuration. The `collectionConfig.xml` file content is also passed to the ServiceRack objects at configure time—the `format` and `displayItem` information is used directly from the `collectionConfig.xml` file rather than added into `buildConfig.xml` during building. This enables formatting and metadata changes in `collectionConfig.xml` to take effect in the collection without rebuilding being necessary. However, as these files are cached, the collection needs to be reloaded for the changes to appear in the library.

2.4 Formatting the collection

Part of collection design involves deciding how the collection should look. Greenstone3 has a default 'look' for a collection, so this is optional. However, the default may not suit the purposes of some collections, so many parts to the look of a collection can be determined by the collection designer.

In standard Greenstone3, the library is served to a web browser by a servlet, and the HTML is generated using XSLT. XSLT templates are used to format all the parts of the pages. These templates can be overridden by including them in the `collectionConfig.xml` file. Some commonly overridden templates are those for formatting lists: search results list, classifier browsing hierarchies, and for parts of the document display.

Real XSLT templates for formatting search results or classifier lists are quite complicated, and not at all easy for a new user to write. For example, the following

```
<buildConfig>
  <metadataList>
    <metadata name="numDocs">11</metadata>
    <metadata name="buildType">mgpp</metadata>
  </metadataList>
  <serviceRackList>
    <serviceRack name="GS2Browse">
      <indexStem name="gs2mgppdemo"/>
      <classifierList>
        <classifier name="CL1" content="Title"/>
        <classifier name="CL2" content="Subject" />
        <classifier name="CL3" content="Organization" />
        <classifier name="CL4" content="Howto" />
      </classifierList>
    </serviceRack>
    <serviceRack name="GS2MGPPRetrieve">
      <indexStem name="gs2mgppdemo"/>
      <defaultLevel name="Sec" />
    </serviceRack>
    <serviceRack name="GS2MGPPSearch">
      <indexStem name="gs2mgppdemo"/>
      <defaultLevel name="Sec" />
      <levelList>
        <level name="Sec" />
        <level name="Doc" />
      </levelList>
      <fieldList>
        <field shortname="ZZ" name="allfields" />
        <field shortname="TX" name="text" />
        <field shortname="DL" name="dls.Title" />
        <field shortname="DS" name="dls.Subject" />
        <field shortname="DO" name="dls.Organization" />
      </fieldList>
      <searchTypeList>
        <searchType name="form" />
        <searchType name="plain" />
      </searchTypeList>
      <indexOptionList>
        <indexOption name="stemIndexes" value="3"/>
<indexOption name="maxnumeric" value="4"/>
      </indexOptionList>
      <defaultIndex name="idx" />
      <indexList>
        <index name="idx" />
      </indexList>
    </serviceRack>
  </serviceRackList>
</buildConfig>
```

Figure 6: Sample buildConfig.xml file (gs2mgppdemo collection)

is a sample template for formatting a classifier list, to show Keyword metadata as a link to the document.

```
<xsl:template match="documentNode" priority="2"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="collName"/>
    <td><a href="{$library_name}?a=d&amp;c={$collName}&amp;
           d={@nodeID}&amp;dt={@docType}"><xsl:value-of
           select="metadataList/metadata[@name='Keyword']"/></a>
    </td>
</xsl:template>
```

To write this, the user would need to know that:

- the variable `$library_name` exists,
- the collection name is passed in as a parameter called `collName`
- metadata for a document is found in a `<metadataList>` and that its form is `<metadata name="Keyword">the value</metadata>`
- the arguments needed for the link to the document are `a, sa, c, d, a, dt`.

We can use XSLT to transform XML into XSLT. Greenstone3 provides a simplified set of formatting commands, written in XML, which will be transformed into proper XSLT. The user specifies a `<gsf:template>` for what they want to format—these typically match `documentNode` or `classifierNode` (for a node in a classification hierarchy).

The template above can be represented as:

```
<gsf:template match='documentNode'>
  <td><gsf:link><gsf:metadata name='Keyword'/></gsf:link></td>
</gsf:template>
```

Table 4 shows the set of `'gsf'` (Greenstone Format) elements. If you have come from a Greenstone2 background, Appendix D.1 shows Greenstone2 format elements and their equivalents in Greenstone3 .

The `<gsf:metadata>` elements are used to output metadata values. The simplest case is `<gsf:metadata name='Title'/>`—this outputs the Title metadata for the current document or section. Namespaces are important here: if the Title metadata is in the Dublin Core (dc) namespace, then the element should look like `<gsf:metadata name='dc.Title'/>`. There are three other attributes for this element. The attribute `multiple` is used when there may be more than one value for the selected metadata. For instance, one document may fall into several classification categories, and therefore may have multiple Subject metadata values. Adding `multiple='true'` to the `<gsf:metadata>` element will retrieve all values, not just the first one. Multiple values are separated by commas by default. The `separator` attribute is used to change the separating string. For example, adding `separator=': '` to the element will separate all values by a colon and a space.

Table 4: Format elements for GSF format language

Element	Description
<gsf:text/>	The document’s text
<gsf:link>...</gsf:link>	The HTML link to the document itself
<gsf:link type=’document’>... </gsf:link>	Same as above
<gsf:link type=’classifier’>... </gsf:link>	A link to a classification node (use in classifierNode templates)
<gsf:link type=’source’>... </gsf:link>	The HTML link to the original file—set for documents that have been converted from e.g. Word, PDF, PS
<gsf:icon/>	An appropriate icon
<gsf:icon type=’document’/>	same as above
<gsf:icon type=’classifier’/>	bookshelf icon for classification nodes
<gsf:icon type=’source’/>	An appropriate icon for the original file e.g. Word, PDF icon
<gsf:metadata name=’Title’/>	The value of a metadata element for the current document or section, in this case, Title
<gsf:metadata name=’Title’ select=’select-type’ [separator=’y’ multiple=’true’]/>	A more extended selection of metadata values. The select field can be one of those shown in Table 5. There are two optional attributes: separator gives a String that will be used to separate the fields, default is “, “, and if multiple is set to true, looks for multiple values at each section.
<gsf:metadata name=’Date’ format=’formatDate’/>	The value of a metadata element for the current document, formatted in some way. Current formatting options available are formatDate: turns ’20040201’ into ’01 February 2004’, and formatLanguage: turns ’en’ into ’English’, both in a language dependent manner.
<gsf:choose-metadata> <gsf:metadata name=’metaA’/> <gsf:metadata name=’metaB’/> <gsf:metadata name=’metaC’/> </gsf:choose-metadata>	A choice of metadata. Will select the first existing one. the metadata elements can have the select, separator and multiple attributes like normal.
<gsf:switch preprocess= ’preprocess-type’> <gsf:metadata name=’Title’/> <gsf:when test=’test-type’ test-value=’xxx’>...</gsf:when> <gsf:when test=’test-type’ test-value=’yyy’>...</gsf:when> <gsf:otherwise>...</gsf:otherwise> </gsf:switch>	switch on the value of a particular metadata - the metadata is specified in gsf:metadata, has the same attributes as normal.

Table 5: Select types for metadata format elements

Select Type	Description
current	The current section
parent	The immediate parent section
ancestors	All the parents back to the root (topmost) section
root	The root or topmost section
siblings	All the sibling sections
children	The immediate child sections of the current section
descendants	All the descendent sections

Sometimes you may want to display metadata values for sections other than the current one. For example, in the mgppdemo collection, in a search list we display the Titles of all the enclosing sections, followed by the Title of the current section, all separated by semi-colons. The display ends up looking something like: *Farming snails 2; Starting out; Selecting your snails* where *Selecting your snails* is the Title of the section in the results list, and *Farming snails 2* and *Starting out* are the Titles of the enclosing sections. The `select` attribute is used to display metadata for sections other than the current one. Table 5 shows the options available for this attribute. The `separator` attribute is used here also, to specify the separating text.

To get the previous metadata, the format statement would have the following in it:

```
<gsf:metadata name='Title' select='ancestors' separator='; '/>;
    <gsf:metadata name='Title'/>
```

The `<gsf:choose-metadata>` element selects the first available metadata value from the list of options.

```
<gsf:choose-metadata>
  <gsf:metadata name='dc.Title'/>
  <gsf:metadata name='dls.Title'/>
  <gsf:metadata name='Title'/>
</gsf:choose-metadata>
```

This will display dls.Title if available, otherwise it will use dc.Title if available, otherwise it will use the Title metadata. If there are no values for any of these metadata elements, then nothing will be displayed.

The `<gsf:switch>` element allows different formatting depending on the value of a specified metadata element. For example, the following switch statement could be used to display a different icon for each document in a list depending on which organization it came from.

```
<gsf:switch  preprocess='toLower;stripSpace'>
  <gsf:metadata name='Organization'/>
  <gsf:when test='equals' test-value='bostid'>
     <!-- output BOSTID image --></gsf:when>
  <gsf:when test='equals' test-value='worldbank'>
     <!-- output world bank image --></gsf:when>
  <gsf:otherwise><!-- output default image--></gsf:otherwise>
</gsf:switch>
```

Table 6: Formatting options

option name	values	description
coverImages	true, false	whether or not to display cover images for documents
documentTOC	true, false	whether or not to display the table of contents for the document

Preprocessing of the metadata value is optional. The preprocess types are `toLower` (make the value lowercase), `toUpper` (make the value uppercase), `stripSpace` (removes any whitespace from the value). These operations are carried out on the value of the selected metadata before the test is carried out. Multiple processing types can be specified, separated by ; and they will be applied in the order specified (from left to right).

Each option specifies a test and a test value. Test values are just text. Tests include `startsWith`, `contains`, `exists`, `equals`, `endsWith`. Exists doesn't need a test value. Having an otherwise option ensures that something will be displayed even when none of the tests match.

If none of the gsf elements meets your needs for formatting, XSLT can be entered directly into the format element, giving the collection designer full flexibility over how the collection appears.

The collection specific templates are added into the configuration file `collectionConfig.xml`. Any templates found in the XSLT files can be overridden. The important part to adding templates into the configuration file is determining where to put them. Formatting templates cannot go just anywhere—there are standard places for them. Figure 7 shows the positions that templates can occur.

There are also formatting instructions that are not templates but are options. These are described in Table 6. They are entered into the configuration file like `<gsf:option name='coverImages' value='false'/>`

Note, format templates are added into the XSLT files before transforming, while the options are added into the page source, and used in tests in the XSLT.

2.4.1 Changing the service text strings

Each collection has a set of services which are the access points for the information in the collection. Each service has a set of text strings which are used to display it. These include name, description, the text on the submit button, and names and descriptions of all the parameters to the service.

These text strings are found in `.properties` files, in `$GSDL3HOME/WEB-INF/classes`. The names of the files are based on class names. Subclasses can define their own properties, or can use their parent class ones. For example, `AbstractSearch` defines strings for the `TextQuery` service, in `AbstractSearch.properties`. `GS2MGSearch` just uses these default ones, so doesn't need its own properties file.

A particular collection can override the properties for any service. For example, if a collection uses the `GS2MGSearch` service rack (look in the `buildConfig.xml`

```
<collectionConfig>
  <metadataList/>
  <displayItemList/>
  <search>
    <format> <!--Put here templates related to searching and
        the query page. The common one is the documentNode
        template -->
      <gsf:template match='documentNode'>...</gsf:template>
    </format>
  </search>
  <browse>
    <classifier name='xx'>
      <format><!-- put here templates related to formating a
        particular classifier page. Common ones are documentNode
        and classifierNode templates-->
        <gsf:template match='documentNode'>...</gsf:template>
        <gsf:template match='classifierNode'>...</gsf:template>
        <gsf:template match='classifierNode' mode='horizontal'>...
          </gsf:template>
      </format>
    </classifier>
    <classifier>...</classifier>
    <format><!-- formatting for all the classifiers. these will
      be overridden by any classifier specific formatting
      instructions --></format>
  </browse>
  <display>
    <format><!-- here goes any formatting relating to the display
        of the documents. These are generally named templates,
        and format options -->
      <gsf:template name='documentContent'>...</gsf:template>
      <gsf:option name='TOC' value='true'/>
    </format>
  </display>
</collectionConfig>
```

Figure 7: Places for format statements

file for a list of service racks used), and the collection builder wants to change the text associated with this service, they can put a `GS2MGSearch.properties` file in the resources directory of the collection. After a reconfigure of the collection, this will be used in preference to the one in the default resources directory.

2.5 Customizing the interface

Format statements in the collection configuration files provide a way to change small parts of the collection display. For large scale customizations to a collection, or ones that apply to a site as a whole, a second mechanism is available. The interface is defined by a set of XSLT files that transform the page data into HTML. Any of these files can be overridden to provide specialized display, on a site or collection basis.

The first section looks at customizing the existing interface, while the second section looks at defining a whole new interface. The last section describes how to add a new language translation of an interface.

2.5.1 Modifying an existing interface

Most of an interface is defined by XSLT files, which are stored in `$GSDL3HOME/-interfaces/interface-name/transform`. These can be changed and the changes will take effect straight away. If changes only apply to certain collections or sites, not everything that uses the interface, you can override some of the files by putting new ones in a different place. XSLT files are looked for in the following order: collection, site, interface, default interface. (This currently only apples to sites, and therefore collections, that reside in the same Greenstone installation as the interface.)

Sites and collections can have a transform directory, which is where customized XSLT files should go. Any XSLT files in here will be used in preference to the interface files when using this collection. For example, if you want to have a completely different layout for the about page of a collection, you can put a new `about.xsl` file into the collection's `transform` directory, and this will be used instead. This is what we do for the Gutenberg sample collection.

This also applies to files that are included from other XSLT files. For example the `query.xsl` for the query pages includes a file called `querytools.xsl`. To have a particular site show a different query interface either of these files may need to be modified. Creating a new version of either of these and putting it in the site `transform` directory will work. Either the new `query.xsl` will include the default `querytools.xsl`, or the default `query.xsl` will include the new `querytools.xsl`. The `xsl:include` directives are preprocessed by the Java code and full paths added based on availability of the files, so that the correct one is used.

Note that you cannot include a file with the same name as the including file. For example `query.xsl` cannot include `query.xsl` (it is tempting to want to do

this if you just want to change one template for a particular file, and then include the default. but you cant).

You can add the argument `o=xml` to any URL and you wil be returned the XML before transformation by a stylesheet. This shows you the XML page source. It can be useful when you are trying to write some new XSLT statements.

2.5.2 Defining a new interface

A new interface may be needed if different instantiations of the library require different interfaces, or different developers want their own look and feel. Creating a new interface will allow modifications to be made while leaving the original one intact.

A new interface needs a directory in `$GSDL3HOME/interfaces`, the name of this directory becomes the interface name. Inside, it needs `images` and `transform` directories, and an `interfaceConfig.xml` file. The `interfaceConfig.xml` file may specify a base interface, in which case the new interface only needs to define XSLT for the parts that are different. Otherwise, it will need a full set of XSLT files.

To use a new interface, the `$GSDL3HOME/WEB-INF/web.xml` file must be edited: either change the interface that a current servlet instance is using, or add another servlet instantiation to the file (see Section 1.4 or Appendix B). The Tomcat server must be restarted for this to take effect.

2.5.3 Changing the interface language

The interface language can be changed by going to the preferences page, and choosing a language from the list, which includes all languages into which the interface has been translated.

It is easy to add a new interface language to Greenstone . Language specific text strings are separated out from the rest of the system to allow for easy incorporation of new languages. These text strings are contained in Java resource bundle properties files. These are plain text files consisting of key-value pairs, located in `$GSDL3HOME/WEB-INF/classes`. Each interface has one named `interface_name.properties` (where `'name'` is the interface name, for example, `interface_default.properties`, or `interface_gs2.properties`). Each service class has one with the same name as the class (e.g. `GS2Search.properties`). To add another language all of the base `.properties` files must be translated. The translated files keep the same names, but with a language extension added. For example, a French version of `interface_default.properties` would be named `interface_default_fr.properties`.

Keys will be looked up in the properties file closest to the specified language. For example, if language `fr_CA` was specified (French language, country Canada), and the default locale was `en_GB`, Java would look at properties files in the following order, until it found the key: `XXX_fr_CA.properties`, `XXX_fr.properties`, `XXX_en_GB.properties`, then `XXX_en.properties`, and finally the default `XXX.properties`.

These new files are available straight away—to use the new language, add e.g. `l=fr` to the arguments in the URL. To get Greenstone to add it in to the list of languages on the preferences page, an entry needs to be added into the languages list in the `interfaceConfig.xml` file (see Section 1.6.2). Modification of this file requires a restart of the Tomcat server for the changes to be recognized.

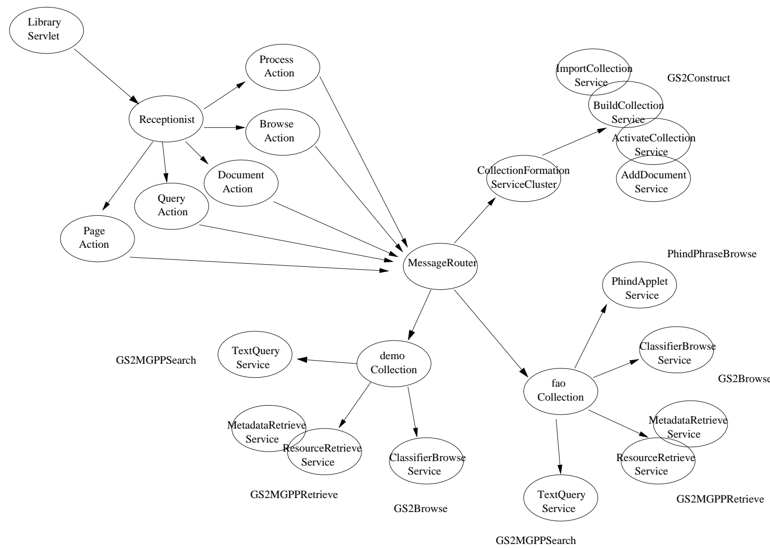

Figure 8: A simple stand-alone site.

3 Developing Greenstone3: Run-time system

[TODO: rewrite this section
runtime object structure diagram. describe the modules.
class hierarchy,
directory structure and where everything lives
message format.
overall description of message passing sequence.
configuration process - start up and runtime

page generation
]

3.1 Overview of modules??

A Greenstone3 'library' system consists of many components: MessageRouter, Receptionist, Actions, Collections, ServiceRacks etc. Figure 8 shows how they fit together in a stand-alone system. The top left part is concerned with displaying the data, while the bottom right part is the collection data serving part. The two sides communicate through the MessageRouter. There is a one-to-one correspondence between modules and Java classes, with the exception of services: for coding and/or run-time efficiency reasons, several Service modules may be grouped together into one ServiceRack class.

MessageRouter: this is the central module for a site. It controls the site, loading up all the collections, clusters, communicators needed. All messages pass through the MessageRouter. Communication between remote sites is always done between

MessageRouters, one for each site.

Collection and ServiceCluster: these are very similar, and group a set of services into a conceptual group.. They both provide some metadata about the collection/cluster, and a list of services. The services are provided by ServiceRack objects that the collection/cluster loads up. A Collection is a specific type of ServiceCluster. A ServiceCluster groups services that are related conceptually, e.g. all the building services may be part of a cluster. What is part of a cluster is specified by the site configuration file. A Collection's services are grouped by the fact that they all operate on some common data—the documents in the collection. Functionally Collection and ServiceCluster are very similar, but conceptually, and to the user, they are quite different.

Service: these provide the core functionality of the system e.g. searching, retrieving documents, building collections etc. One or more may be grouped into a single Java class (ServiceRack) for code reuse, or to avoid instantiating the same objects several times. For example, MGPP searching services all need to have the index loaded into memory.

Communicator/Server: these facilitate communication between remote modules. For example, if you want MR1 to talk to MR2, you need a Communicator-Server pair. The Server sits on top of MR2, and MR1 talks to the Communicator. Each communication type needs a new pair. So far we have only been using SOAP, so we have a SOAPCommunicator and a SOAPServer.

Receptionist: this is the point of contact for the 'front end'. Its core functionality involves routing requests to the Actions, but it may do more than that. For example, a Receptionist may: modify the request in some way before sending it to the appropriate Action; add some data to the page responses that is common to all pages; transform the response into another form using XSLT. There is a hierarchy of different Receptionist types, which is described in Section 3.9.3.

Actions: these do the job of creating the 'pages'. There is a different action for each type of page, for example PageAction handles semi-static pages, QueryAction handles queries, DocumentAction displays documents. They know a little bit about specific service types. Based on the 'CGI' arguments passed in to them, they construct requests for the system, and put together the responses into data for the page. This data is returned to the Receptionist, which may transform it to HTML. The various actions are described in more detail in Section 3.9.

3.2 Start up configuration

We use the Tomcat web server, which operates either stand-alone in a test mode or in conjunction with the Apache web server. The Greenstone LibraryServlet class is loaded by Tomcat and the servlet's `init()` method is called. Each time a `get/put/post` (etc.) is used, a new thread is started and `doGet()/doPut()/doPost()` (etc.) is called.

The `init()` method creates a new Receptionist and a new MessageRouter. Default classes (DefaultReceptionist, MessageRouter) are used unless subclasses have

been specified in the servlet initiation parameters (see Section 1.4). The appropriate system variables are set for each object (interface name, site name, etc.) and then `configure()` is called on both. The MessageRouter handle is passed to the Receptionist. The servlet then communicates only with the Receptionist, not with the MessageRouter.

The Receptionist reads in the `interfaceConfig.xml` file (see Section 1.6.2), and loads up all the different Action classes. Other Actions may be loaded on the fly as needed. Actions are added to a map, with shortnames for keys. Eg the QueryAction is added with key 'q'. The Actions are passed the MessageRouter reference too. If the Receptionist is a TransformingReceptionist, a mapping between shortnames and XSLT file names is also created.

The MessageRouter reads in its site configuration file `siteConfig.xml` (see Section 1.6.1). It creates a module map that maps names to objects. This is used for routing the messages. It also keeps small chunks of XML—serviceList, collectionList, clusterList and siteList. These are part of what get returned in response to a describe request (see Section 3.4.).

Each ServiceRack specified in the configuration file is created, then queried for its list of services. Each service name is added to the map, pointing to the ServiceRack object. Each service is also added to the serviceList. After this stage, ServiceRacks are transparent to the system, and each service is treated as a separate module.

ServiceClusters are created and passed the `<serviceCluster>` element for configuration. They are added to the map as is, with the cluster name as a key. A serviceCluster is also added to the serviceClusterList.

For each site specified, the MessageRouter creates an appropriate type of Communicator object. Then it tries to get the site description. If the server for the remote site is up and running, this should be successful. The site will be added to the mapping with its site name as a key. The site's collections, services and clusters will also be added into the static xml lists. If the server for the remote site is not running, the site will not be included in the siteList or module map. To try again to access the site, either Tomcat must be restarted, or a run-time reconfigure-site command must be sent (see Section 1.7).

The MessageRouter also looks inside the site's `collect` directory, and loads up a Collection object for each valid collection found. If a `collectionInit.xml` file is present, a subclass of Collection may be used. The Collection object reads its `buildConfig.xml` and `collectionConfig.xml` files, determines the metadata, and loads ServiceRack classes based on the names specified in `buildConfig.xml`. The `<serviceRack>` XML element is passed to the object to be used in configuration. The `collectionConfig.xml` contents are also passed in to the ServiceRacks. Any format or display information that the services need must be extracted from the collection configuration file. Collection objects are added to the module map with their name as a key, and also a collection element is added into the collectionList XML.

3.3 Message passing

There are two types of messages used by the system: external and internal messages. All messages have an enclosing `<message>` element, which contains either one or more requests, or one or more responses. In the following descriptions, the message element is not shown, but is assumed to be present. Action in Greenstone3 is originated by a request coming in from the outside. In the standard web-based Greenstone, this comes from a servlet and is passed into the Receptionist. This "external" type request is a request for a page of data, and contains a representation of the CGI style arguments. A page of XML is returned, which can be in HTML format or other depending on the output parameter of the request.

Messages inside the system ("internal" messages) all follow the same basic format: message elements contain multiple request elements, or multiple response elements. Messaging is all synchronous. The same number of responses as requests will be returned. Currently all requests are independent, so any requests can be combined into the same message, and they will be answered separately, with their responses being sent back in a single message.

When a page request (external request) comes in to the Receptionist, it looks at the action attribute and passes the request to the appropriate Action module. The Action will fire one or more internal requests to the MessageRouter, based on the arguments. The data is gathered into a response, which is returned to the Receptionist. The page that the receptionist returns contains the original request, the response from the action and other info as needed (depends on the type of Receptionist). The data may be transformed in some way — for the Greenstone servlet we transform using XSLT to generate HTML pages.

Actions send internal style messages to the MessageRouter. Some can be answered by it, others are passed on to collections, and maybe on to services. Internal requests are for simple actions, such as search, retrieve metadata, retrieve document text There are different internal request types: describe, process, system, format, status. Process requests do the actual work of the system, while the other types get auxiliary information. The format of the requests and responses for each internal request type are described in the following sections. External style requests, and their page responses are described in the Section about page generation (Section 3.9).

3.4 'describe'-type messages

The most basic of the internal standard requests is "describe-yourself", which can be sent to any module in the system. The module responds with a semi-predefined piece of XML, making these requests very efficient. The response is predefined apart from any language-specific text strings, which are put together as each request comes in, based on the language attribute of the request.

```
<request lang='en' type='describe' to=''/>
```

If the `to` field is empty, a request is answered by the MessageRouter. An example response from a MessageRouter might look like this:

```
<response lang='en' type='describe'>
  <serviceList/>
  <siteList>
    <site name='org.greenstone.gsdl1'
            address='http://localhost:8080/greenstone3/services/localsite'
            type='soap' />
  </siteList>
  <serviceClusterList>
    <serviceCluster name="build" />
  </serviceClusterList>
  <collectionList>
    <collection name='org.greenstone.gsdl1/
                 org.greenstone.gsdl2/fao' />
    <collection name='org.greenstone.gsdl1/demo' />
    <collection name='org.greenstone.gsdl1/fao' />
    <collection name='myfiles' />
  </collectionList>
</response>
```

This MessageRouter has no individual site-wide services (an empty `<serviceList>`), but has a service cluster called build (which provides collection importing and building functionality). It communicates with one site, `org.greenstone.gsdl1`. It is aware of four collections. One of these, `myfiles`, belongs to it; the other three are available through the external site. One of those collections is actually from a further external site.

It is possible to ask just for a specific part of the information provided by a describe request, rather than the whole thing. For example, these two messages get the `collectionList` and the `siteList` respectively:

```
<request lang='en' type='describe' to=''>
  <paramList>
    <param name='subset' value='collectionList'/>
  </paramList>
</request>

<request lang='en' type='describe' to=''>
  <paramList>
    <param name='subset' value='siteList'/>
  </paramList>
</request>
```

Subset options for the MessageRouter include `collectionList`, `serviceClusterList`, `serviceList`, `siteList`.

When a collection or service cluster is asked to describe itself, what is returned is a list of metadata, some display elements, and a list of services. For example, here is such a message, along with a sample response.

```
<request lang='en' type='describe' to='mgppdemo'/>
```

```
<response from="mgppdemo" type="describe">
  <collection name="mgppdemo">
    <displayItem lang="en" name="name">greenstone mgpp demo
    </displayItem>
    <displayItem lang="en" name="description">This is a
      demonstration collection for the Greenstone digital
      library software. It contains a small subset (11 books)
      of the Humanity Development Library. It is built with
      mgpp.</displayItem>
    <displayItem lang="en" name="icon">mgppdemo.gif</displayItem>
    <serviceList>
      <service name="DocumentStructureRetrieve" type="retrieve" />
      <service name="DocumentMetadataRetrieve" type="retrieve" />
      <service name="DocumentContentRetrieve" type="retrieve" />
      <service name="ClassifierBrowse" type="browse" />
      <service name="ClassifierBrowseMetadataRetrieve"
            type="retrieve" />
      <service name="TextQuery" type="query" />
      <service name="FieldQuery" type="query" />
      <service name="AdvancedFieldQuery" type="query" />
      <service name="PhindApplet" type="applet" />
    </serviceList>
    <metadataList>
      <metadata name="creator">greenstone@cs.waikato.ac.nz</metadata>
      <metadata name="numDocs">11</metadata>
      <metadata name="buildType">mgpp</metadata>
      <metadata name="httpPath">http://kanuka:8090/greenstone3/sites/
                                 localsite/collect/mgppdemo</metadata>
    </metadataList>
  </collection>
</response>
```

Subset options for a collection or serviceCluster include `metadataList`, `serviceList`, and `displayItemList`.

This collection provides many typical services. Notice how this response lists the services available, while the collection configuration file for this collection (Figure 5) described serviceRacks. Once the service racks have been configured, they become transparent in the system, and only services are referred to. There are three document retrieval services, for structural information, metadata, and content. The Classifier services retrieve classification structure and metadata. These five services were all provided by the GS2MGPPRetrieve ServiceRack. The three query services were provided by GS2MGPPSearch serviceRack, and provide different kinds of query interface. The last service, PhindApplet, is provided by the PhindPhraseBrowse serviceRack and is an applet service.

A `describe` request sent to a service returns a list of parameters that the service accepts and some display information, (and in future may describe the content type for the request and response). Subset options for the request include `paramList` and `displayItemList`.

Parameters can be in the following formats:

```
<param name='xxx' type='integer|boolean|string|invisible' default='yyy'/>
<param name='xxx' type='enum_single|enum_multi' default='aa'/>
  <option name='aa'/><option name='bb'/>...
</param>
<param name='xxx' type='multi' occurs='4'>
 <param .../>
 <param .../>
</param>
```

If no default is specified, the parameter is assumed to be mandatory. Here are some examples of parameters:

```
<param name='case' type='boolean' default='0'/>

<param name='maxDocs' type='integer' default='50'/>

<param name='index' type='enum' default='dtx'>
  <option name='dtx'/>
  <option name='stt'/>
  <option name='stx'/>
<param>

<!-- this one is for the text box and field list for the
simple field query-->
<param name='simpleField' type='multi' occurs='4'>
  <param name='fqv' type='string'/>
  <param name='fqf' type='enum_single'>
    <option name='TI'/><option name='AU'/><option name='OR'/>
  </param>
</param>
```

The type attribute is used to determine how to display the parameters on a web page or interface. For example, a string parameter may result in a text entry box, a boolean an on/off button, enum_single/enum_multi a drop-down menu, where one or many items, respectively, can be selected. A multi-type parameter indicates that two or more parameters are associated, and should be displayed appropriately. For example, in a field query, the text box and field list should be associated. The occurs attribute specifies how many times the parameter should be displayed on the page. Parameters also come with display information: all the text strings needed to present them to the user. These include the name of the parameter and the display values for any options. These are included in the above parameter descriptions in the form of `<displayItem>` elements.

A service description also contains some display information—this includes the name of the service, and the text for the submit button.

Here is a sample describe request to the FieldQuery service of collection mgppdemo, along with its response. The parameters in this example include their display information. Figure 9 shows an example HTML search form that may be generated from this describe response.

```
<request lang="en" to="mgppdemo/FieldQuery" type="describe" />
```

```
<response from="mgppdemo/FieldQuery" type="describe">
  <service name="FieldQuery" type="query">
    <displayItem name="name">Form Query</displayItem>
    <displayItem name="submit">Search</displayItem>
    <paramList>
      <param default="Doc" name="level" type="enum_single">
        <displayItem name="name">Granularity to search at</displayItem>
        <option name="Doc">
          <displayItem name="name">Document</displayItem>
        </option>
        <option name="Sec">
          <displayItem name="name">Section</displayItem>
        </option>
        <option name="Para">
          <displayItem name="name">Paragraph</displayItem>
        </option>
      </param>
      <param default="1" name="case" type="boolean">
        <displayItem name="name">Turn casefolding </displayItem>
        <option name="0">
          <displayItem name="name">off</displayItem>
        </option>
        <option name="1">
          <displayItem name="name">on</displayItem>
        </option>
      </param>
      <param default="1" name="stem" type="boolean">
        <displayItem name="name">Turn stemming </displayItem>
        <option name="0">
          <displayItem name="name">off</displayItem>
        </option>
        <option name="1">
          <displayItem name="name">on</displayItem>
        </option>
      </param>
      <param default="10" name="maxDocs" type="integer">
        <displayItem name="name">Maximum documents to return
        </displayItem>
      </param>
      <param name="simpleField" occurs="4" type="multi">
        <displayItem name="name"></displayItem>
        <param name="fqv" type="string">
          <displayItem name="name">Word or phrase </displayItem>
        </param>
        <param default="ZZ" name="fqf" type="enum_single">
          <displayItem name="name">in field</displayItem>
          <option name="ZZ">
            <displayItem name="name">allfields</displayItem>
          </option>
          <option name="TX">
            <displayItem name="name">text</displayItem>
          </option>
          <option name="TI">
```

Form Query

Granularity to search at — Section
Turn casefolding — on
Turn stemming — on
Maximum documents to return — 10

Search for — in field
All fields
All fields
All fields
TextOnly
Subject
Title
Search

Figure 9: The previous query service describe response as displayed on the search page.

```
              <displayItem name="name">Title</displayItem>
            </option>
            <option name="SU">
              <displayItem name="name">Subject</displayItem>
            </option>
            <option name="ORG">
              <displayItem name="name">Organization</displayItem>
            </option>
            <option name="SO">
              <displayItem name="name">Source</displayItem>
            </option>
          </param>
        </param>
      </paramList>
    </service>
  </response>
```

A describe request to an applet type service returns the applet HTML element: this will be embedded into a web page to run the applet.

```
<request type='describe' to='mgppdemo/PhindApplet'/>

<response type='describe'>
  <service name='PhindApplet' type='query'>
    <applet ARCHIVE='phind.jar, xercesImpl.jar, gsdl3.jar,
jaxp.jar, xml-apis.jar'
            CODE='org.greenstone.applet.phind.Phind.class'
            CODEBASE='lib/java'
            HEIGHT='400' WIDTH='500'>
      <PARAM NAME='library' VALUE=''/>
      <PARAM NAME='phindcgi' VALUE='?a=a&amp;sa=r&amp;sn=Phind'/>
```

```
    <PARAM NAME='collection' VALUE='mgppdemo' />
    <PARAM NAME='classifier' VALUE='1' />
    <PARAM NAME='orientation' VALUE='vertical' />
    <PARAM NAME='depth' VALUE='2' />
    <PARAM NAME='resultorder' VALUE='L,l,E,e,D,d' />
    <PARAM NAME='backdrop' VALUE='interfaces/default/>
    images/phindbg1.jpg'/>
    <PARAM NAME='fontsize' VALUE='10' />
    <PARAM NAME='blocksize' VALUE='10' />
    The Phind java applet.
  </applet>
  <displayItem name="name">Browse phrase hierarchies</displayItem>
 </service>
</response>
```

Note that the library parameter has been left blank. This is because library refers to the current servlet that is running and the name is not necessarily known in advance. So either the applet action or the Receptionist must fill in this parameter before displaying the HTML.

3.5 'system'-type messages

"System" requests are used to tell a MessageRouter, Collection or ServiceCluster to update its cached information and activate or deactivate other modules. For example, the MessageRouter has a set of Collection modules that it can talk to. It also holds some XML information about those collections—this is returned when a request for a collection list comes in. If a collection is deleted or modified, or a new one created, this information may need to change, and the list of available modules may also change. Currently these requests are initiated by particular CGI requests (see Section 1.7).

The basic format of a system request is as follows:

```
<request type='system' to=''>
  <system .../>
</request>
```

One or more actual requests are specified in system elements. The following are examples:

```
<system type='configure' subset=''/>
<system type='configure' subset='collectionList'/>
<system type='activate' moduleType='collection' moduleName='demo'/>
<system type='deactivate' moduleType='site' moduleName='site1'/>
```

The first request reconfigures the whole site—the MessageRouter goes through its whole configure process again. The second request just reconfigures the collectionList—the MessageRouter will delete all its collection information, and re-look through the collect directory and reload all the collections again. The third request is to activate collection demo. This could be a new collection, or a reactivation of an old

one. If a collection module already exists, it will be deleted, and a new one loaded. The final request deactivates the site site1—this removes the site from the siteList and module map, and also removes any of that sites collections/services from the static lists.

A response just contains a status message[5], for example:

```
<status>MessageRouter reconfigured successfully</status>
<status>Error on reconfiguring collectionList</status>
<status>collection:demo activated</status>
<status>site:site1 deactivated</status>
```

System requests are mainly answered by the MessageRouter. However, Collections and ServiceClusters will respond to a subset of these requests.

3.6 'format'-type messages

Collection designers are able to specify how their collection looks to a certain degree. They can specify format statements for display that will apply to the results of a search, the display of a document, entries in a classification hierarchy, for example. This info is generally service specific. All services respond to a format request, where they return any service specific formatting information. A typical request and response looks like this:

```
<request lang="en" to="mgppdemo/FieldQuery" type="format" />

<response from="mgppdemo/FieldQuery" type="format">
  <format>
    <gsf:template match="documentNode"><td><gsf:link>
      <gsf:metadata name="Title" />(<gsf:metadata name="Source" />)
      </gsf:link></td>
    </gsf:template>
  </format>
</response>
```

The actual format statements are described in Section 2.4. They are templates written directly in XSLT, or in GSF (GreenStone Format) which is a simple XML representation of the more complicated XSLT templates. GSF-style format statements need to be converted to proper XSLT. This is currently done by the Receptionist (but may be moved to an ActionHelper): the format XML is transformed to XSLT using XSLT with the config_format.xsl stylesheet.

3.7 'status'-type messages

These are only used with process-type services, which are those where a request is sent to start some type of process (see Section 3.8.4). An initial 'process' request to a 'process' service generates a response which states whether the process had successfully started, and whether its still continuing. If the process is not finished,

[5]TODO: add in error/status codes

Table 7: Status codes currently used in Greenstone3

code name	code value	meaning
SUCCESS	1	the request was accepted, and the process was completed
ACCEPTED	2	the request was accepted, and the process has been started, but it is not completed yet
ERROR	3	there was an error and the process was stopped
CONTINUING	10	the process is still continuing
COMPLETED	11	the process has finished
HALTED	12	the process has stopped
INFO	20	just an info message that doesn't imply anything

status requests can be sent repeatedly to the service to poll the status, using the pid to identify the process. Status codes are used to identify the state of a process. The values used at the moment are listed in Table 7[6].

The following shows an example status request, along with two responses, the first a 'OK but continuing' response, and the second a 'successfully completed' response. The content of the status elements in the two responses is the output from the process since the last status update was sent back.

```
<request lang="en" to="build/ImportCollection" type="status">
  <paramList>
    <param name="pid" value="2" />
  </paramList>
</request>

<response from="build/ImportCollection">
  <status code="2" pid="2">Collection construction: import collection.
command = import.pl -collectdir /research/kjdon/home/greenstone3/web/sites/
    localsite/collect test1
starting
  </status>
</response>

<response from="build/ImportCollection">
  <status code="11" pid="2">RecPlug: getting directory
/research/kjdon/home/greenstone3/web/sites/localsite/collect/test1/import
WARNING - no plugin could process /.keepme

*********************************************
Import Complete
*********************************************
* 1 document was considered for processing
* 0 were processed and included in the collection
* 1 was rejected. See /research/kjdon/home/greenstone3/web/sites/
    localsite/collect/test1/etc/fail.log for a list of rejected documents
Success
  </status>
</response>
```

[6] A more standard set of codes should probably be used, for example, the HTTP codes

3.8 'process'-type messages

Process requests and responses provide the major functionality of the system—these are the ones that do the actual work. The format depends on the service they are for, so I'll describe these by service.

Query type services TextQuery, FieldQuery, AdvancedFieldQuery (GS2MGSearch, GS2MGPPSearch), TextQuery (LuceneSearch) The main type of requests in the system are for services. There are different types of services, currently: `query`, `browse`, `retrieve`, `process`, `applet`, `enrich`. Query services do some kind of search and return a list of document identifiers. Retrieve services can return the content of those documents, metadata about the documents, or other resources. Browse is for browsing lists or hierarchies of documents. Process type services are those where the request is for a command to be run. A status code will be returned immediately, and then if the command has not finished, an update of the status can be requested. Applet services are those that run an applet. Enrich services take a document and return the document with some extra markup added.

Other possibilities include transform, extract, accrete. These types of service generally enhance the functionality of the first set. They may be used during collection formation: 'accrete' documents by adding them to a collection, 'transform' the documents into a different format, 'extract' information or acronyms from the documents, 'enrich' those documents with the information extracted or by adding new information. They may also be used during querying: 'transform' a query before using it to query a collection, or 'transform' the documents you get back into an appropriate form.

The basic structure of a service 'process' request is as follows:

```
<request lang='en'  type='process' to='demo/TextQuery'>
  <paramList/>
  other elements...
</request>
```

The parameters are name-value pairs corresponding to parameters that were specified in the service description sent in response to a describe request.

```
<param name='case' value='1'/>
<param name='maxDocs' value='34'/>
<param name='index' value='dtx'/>
```

Some requests have other content—for document retrieval, this would be a list of document identifiers to retrieve. For metadata retrieval, the content is the list of documents to retrieve metadata for.

Responses vary depending on the type of request. The following sections look at the process type requests and responses for each type of service.

3.8.1 'query'-type services

Responses to query requests contain a list of document identifiers, along with some other information, dependent on the query type. For a text query, this includes term frequency information, and some metadata about the result. For instance, a text query on 'snail farming', with the parameter 'maxDocs=10' might return the first 10 documents, and one of the query metadata items would be the total number of documents that matched the query.[7]

The following shows an example query request and its response.

Find at most 10 Sections in the mgppdemo collection, containing the word snail (stemmed), returning the results in ranked order:

```
<request lang='en'  to="mgppdemo/TextQuery" type="process">
  <paramList>
    <param name="maxDocs" value="10"/>
    <param name="queryLevel" value="Section"/>
    <param name="stem" value="1"/>
    <param name="matchMode" value="some"/>
    <param name="sortBy" value="1"/>
    <param name="index" value="t0"/>
    <param name="case" value="0"/>
    <param name="query" value="snail"/>
  </paramList>
</request>

<response from="mgppdemo/TextQuery" type="process">
  <metadataList>
    <metadata name="numDocsMatched" value="59" />
  </metadataList>
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2"
docType='hierarchy' nodeType="leaf" />
    <documentNode nodeID="HASH010f073f22033181e206d3b7.2.12"
docType='hierarchy' nodeType="leaf" />
    <documentNode nodeID="HASH010f073f22033181e206d3b7.1"
docType='hierarchy' nodeType="interior" />
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.2.2"
docType='hierarchy' nodeType="leaf" />
    ...
  </documentNodeList>
  <termList>
    <term field="" freq="454" name="snail" numDocsMatch="58" stem="3">
      <equivTermList>
        <term freq="" name="Snail" numDocsMatch="" />
        <term freq="" name="snail" numDocsMatch="" />
        <term freq="" name="Snails" numDocsMatch="" />
        <term freq="" name="snails" numDocsMatch="" />
      </equivTermList>
    </term>
  </termList>
</response>
```

[7] no metadata about the query result is returned yet.

The list of document identifiers includes some information about document type and node type. Currently, document types include `simple`, `paged` and `hierarchy`. `simple` is for single section documents, i.e. ones with no sub-structure. `paged` is documents that have a single list of sections, while `hierarchy` type documents have a hierarchy of nested sections. For `paged` and `hierarchy` type documents, the node type identifies whether a section is the root of the document, an internal section, or a leaf.

The term list identifies, for each term in the query, what its frequency in the collection is, how many documents contained that term, and a list of its equivalent terms (if stemming or casefolding was used).

3.8.2 'browse'-type services

Browse type services are used for classification browsing. The request consists of a list of classifier identifiers, and some structure parameters listing what structure to retrieve.

```
<request lang="en" to="mgppdemo/ClassifierBrowse" type="process">
  <paramList>
    <param name="structure" value="ancestors" />
    <param name="structure" value="children" />
  </paramList>
  <classifierNodeList>
    <classifierNode nodeID="CL1.2" />
  </classifierNodeList>
</request>

<response from="mgppdemo/ClassifierBrowse" type="process">
  <classifierNodeList>
    <classifierNode nodeID="CL1">
      <nodeStructure>
<classifierNode nodeID="CL1">
          <classifierNode nodeID="CL1.2">
    <classifierNode nodeID="CL1.2.1" />
    <classifierNode nodeID="CL1.2.2" />
    <classifierNode nodeID="CL1.2.3" />
    <classifierNode nodeID="CL1.2.4" />
    <classifierNode nodeID="CL1.2.5" />
          </classifierNode>
</classifierNode>
      </nodeStructure>
    </classifierNode>
  </classifierNodeList>
</response>
```

Possible values for structure parameters are `ancestors`, `parent`, `siblings`, `children`, `descendants`. The response gives, for each identifier in the request, a `<nodeStructure>` element with all the requested structure put together into a hierarchy. The structure may include classifier and document nodes.

Structural info can also be requested in the `paramList`, and will be returned in a `<nodeStructureInfo>` element. (See the section on DocumentStructureRetrieve messages.) Possible values for info parameters are `numSiblings`, `siblingPosition`, `numChildren`.

3.8.3 'retrieve'-type services

Retrieval services are special in that requests are not explicitly initiated by a user from a form on a web page, but are called from actions in response to other things. This means that their names are hard-coded into the Actions. DocumentContentRetrieve, DocumentStructureRetrieve and DocumentMetadataRetrieve are the standard names for retrieval services for content, structure, and metadata of documents. Requests to each of these include a list of document identifiers. Because these generally refer to parts of documents, the elements are called `<documentNode>`. For the content, that is all that is required. For the metadata retrieval service, the request also needs parameters specifying what metadata is required. For structure retrieval services, requests need parameters specifying what structure or structural info is required.

Some example requests and responses follow.

Give me the Title metadata for these documents:

```
<request lang="en" to="mgppdemo/DocumentMetadataRetrieve" type="process">
  <paramList>
    <param name="metadata" value="Title" />
  </paramList>
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2"/>
    <documentNode nodeID="HASH010f073f22033181e206d3b7.2.12"/>
    <documentNode nodeID="HASH010f073f22033181e206d3b7.1"/>
    ...
  </documentNodeList>
</request>

<response from="mgppdemo/DocumentMetadataRetrieve" type="process">
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2">
      <metadataList>
        <metadata name="Title">Putting snails in your second pen</metadata>
      </metadataList>
    </documentNode>
    <documentNode nodeID="HASH010f073f22033181e206d3b7.2.12">
      <metadataList>
        <metadata name="Title">Now you must decide</metadata>
      </metadataList>
    </documentNode>
    <documentNode nodeID="HASH010f073f22033181e206d3b7.1">
      <metadataList>
        <metadata name="Title">Introduction</metadata>
      </metadataList>
```

```
    </documentNode>
  </documentNodeList>
</response>
```

One or more parameters specifying metadata may be included in a request. Also, a metadata value of `all` will retrieve all the metadata for each document.

Any browse-type service must also implement a metadata retrieval service to provide metadata for the nodes in the classification hierarchy. The name of it is the browse service name plus `MetadataRetrieve`. For example, the ClassifierBrowse service described in the previous section should also have a ClassifierBrowseMetadataRetrieve service. The request and response format is exactly the same as for the DocumentMetadataRetrieve service, except that `<documentNode>` elements are replaced by `<classifierNode>` elements (and the corresponding list element is also changed).

Give me the text (content) of this document:

```
<request lang="en" to="mgppdemo/DocumentContentRetrieve" type="process">
  <paramList />
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2" />
  </documentNodeList>
</request>

<response from="mgppdemo/DocumentContentRetrieve" type="process">
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2">
      <nodeContent>&lt;Section&gt;
       &lt;/B&gt;&lt;P ALIGN=&quot;JUSTIFY&quot;&gt;&lt;/P&gt;
       &lt;P ALIGN=&quot;JUSTIFY&quot;&gt;190. When the plants in
       your second pen have grown big enough to provide food and
       shelter, you can put in the snails.&lt;/P&gt;
      </nodeContent>
    </documentNode>
  </documentNodeList>
</response>
```

The content of a node is returned in a `<nodeContent>` element. In this case it is escaped HTML.

Give me the ancestors and children of the specified node, along with the number of siblings it has:

```
<request lang="en" to="mgppdemo/DocumentStructureRetrieve" type="process">
  <paramList>
    <param name="structure" value="ancestors" />
    <param name="structure" value="children" />
    <param name="info" value="numSiblings" />
  </paramList>
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2" />
  </documentNodeList>
</request>
```

```
<response from="mgppdemo/DocumentStructureRetrieve" type="process">
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2">
      <nodeStructureInfo>
        <info name="numSiblings" value="2" />
      </nodeStructureInfo>
      <nodeStructure>
        <documentNode nodeID="HASHac0a04dd14571c60d7fbfd"
                docType='hierarchy' nodeType="root">
          <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4"
                  docType='hierarchy' nodeType="interior">
            <documentNode nodeID="HASHac0a04dd14571c60d7fbfd.4.2"
                   docType='hierarchy' nodeType="leaf" />
          </documentNode>
        </documentNode>
      </nodeStructure>
    </documentNode>
  </documentNodeList>
</response>
```

Structure is returned inside a `<nodeStructure>` element, while structural info is returned in a `<nodeStructureInfo>` element. Possible values for structure parameters are as for browse services: `ancestors`, `parent`, `siblings`, `children`, `descendants`, `entire`. Possible values for info parameters are `numSiblings`, `siblingPosition`, `numChildren`.

3.8.4 'process'-type services

Requests to process-type services are not requests for data—they request some action to be carried out, for example, create a new collection, or import a collection. The response is a status or an error message. The import and build commands may take a long time to complete, so a response is sent back after a successful start to the command. The status may be polled by the requester to see how the process is going.

Process requests generally contain just a parameter list. Like for any service, the parameters used by a process-type service can be obtained by a describe request to that service.

Here are two example requests for process-services that are part of the build service cluster (hence the addresses all begin with 'build/'), followed by an example response:

```
<request lang='en'  type='process' to='build/NewCollection'>
  <paramList>
    <param name='creator' value='me@home.com'/>
    <param name='collName' value='the demo collection'/>
    <param name='collShortName' value='demo'/>
  </paramlist>
</request>
```

```
<request lang='en'  type='process' to='build/ImportCollection'>
  <paramList>
    <param name='collection' value='demo'/>
  </paramlist>
</request>

<response from="build/ImportCollection">
  <status code="2" pid="2">Starting process...</status>
</response>
```

The `code` attribute in the response specifies whether the command has been successfully stated, whether its still going, etc (see Table 7 for a list of currently used codes). The pid attribute specifies a process id number that can be used when querying the status of this process. The content of the status element is (currently) just the output from the process so far. Status messages, which were described in Section 3.7, are used to find out how the process is going, and whether it has finished or not.

3.8.5 'applet'-type services

Applet-type services are those that process the data for an applet. A request consists only of a list of parameters, and the response contains an `<appletData>` element that contains the XML data to be returned to the applet. The format of this is entirely specific to the applet—there is no set format to the applet data.

Here is an example request and response, used by the Phind applet:

```
<request type='query' to='mgppdemo/PhindApplet'>
  <paramList>
    <param name='pc' value='1'/>
    <param name='pptext' value='health'/>
    <param name='pfe' value='0'/>
    <param name='ple' value='10'/>
    <param name='pfd' value='0'/>
    <param name='pld' value='10'/>
    <param name='pfl' value='0'/>
    <param name='pll' value='10'/>
  </paramList>
</request>

<response type='query' from='mgppdemo/PhindApplet'>
  <appletData>
    <phindData df='9' ef='46' id='933' lf='15' tf='296'>
      <expansionList end='10' length='46' start='0'>
        <expansion df='4' id='8880' num='0' tf='59'>
          <suffix> CARE</suffix>
        </expansion>
        ...
      </expansionList>
      <documentList end='10' length='9' start='0'>
        <document freq='78' hash='HASH4632a8a51d33c47a75c559' num='0'>
          <title>The Courier - N??159 - Sept- Oct 1996 Dossier Investing
```

```
                in People Country Reports: Mali ; Western Samoa
          </title>
        </document>
        ...
      </documentList>
      <thesaurusList end='10' length='15' start='0'>
        <thesaurus df='7' id='12387' tf='15' type='RT'>
          <phrase>PUBLIC HEALTH</phrase>
        </thesaurus>...
      </thesaurusList>
    </phindData>
  </appletData>
</response>
```

3.8.6 'enrich'-type services

Enrich services typically take some text of documents (inside <nodeContent> tags) and returns the text marked up in some way. One example of this is the GatePOSTag service: this identifies Dates, Locations, People and Organizations in the text, and annotates the text with the labels. In the following example, the request is for Location and Dates to be identified.

```
<request lang="en" to="GatePOSTag" type="process">
  <paramList>
    <param name="annotationType" value="Date,Location" />
  </paramList>
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd">
      <nodeContent>
        FOOD AND AGRICULTURE ORGANIZATION OF THE UNITED NATIONS
        Rome 1986
        P-69
        ISBN 92-5-102397-2
        FAO 1986
      </nodeContent>
    </documentNode>
  </documentNodeList>
</request>

<response from="GatePOSTag" type="process">
  <documentNodeList>
    <documentNode nodeID="HASHac0a04dd14571c60d7fbfd">
      <nodeContent>
FOOD AND AGRICULTURE ORGANIZATION OF THE UNITED NATIONS
<annotation type="Location">Rome</annotation>
        <annotation type="Date">1986</annotation>
        P-69
        ISBN 92-5-102397-2
        FAO <annotation type="Date">1986</annotation>
      </nodeContent>
    </documentNode>
```

```
  </documentNodeList>
</response>
```

3.9 Page generation

A 'page' is some XML or HTML (or other?) data returned in response to an external 'page'-type request. These requests originate from outside Greenstone , for example from a servlet, or Java application, and are received by the Receptionist. As described below in Section 3.9.1, the requests are XML representations of Greenstone URLs. One of the arguments is action (a). This tells the Receptionist which Action module to pass the request to.

Action modules decode the rest of the arguments to determine what requests need to be made to the system. One or more internal requests may be made to the MessageRouter. A request for format information from the Collection/Service may also be made. The resulting data is gathered together into a single XML response, `<page>`, and returned to the Receptionist.

The page format is described in Section 3.9.2. The XML may be returned as is, or may be modified by the Receptionist. The various Receptionists are described in Section 3.9.3. The default receptionist used by a servlet transforms the XML into HTML using XSL stylesheets. Section 3.9.4 looks at collection specific formatting, in particular for HTML output. Sections 3.9.6 to 3.9.12 look at the various actions and what kind of data they gather.

3.9.1 'page'-type requests and their arguments

These are requests for a 'page' of data—for example, the home page for a site; the query page for a collection; the text of a document. They contain, in XML, a list of arguments specifying what type of page is required. If the external context is a servlet, the arguments represent the 'CGI' arguments in a Greenstone URL. The two main arguments are `a` (action) and `sa` (subaction). All other arguments are encoded as parameters.

Here are some examples of requests[8]:

```
<request type='page' action='p' subaction='about'
         lang='fr' output='html'>
  <paramList>
    <param name='c' value='demo'/>
  </paramList>
</request>

<request type='page' action='q' lang='en' output='html'>
  <paramList>
    <param name='s' value='TextQuery'/>
    <param name='c' value='demo'/>
```

[8]In a servlet context, these correspond to the arguments `a=p&sa=about&c=demo&l=fr`, and `a=q&l=en&s=TextQuery&c=demo&rt=r&ca=0&st=1&m=10&q=snail`.

```
    <param name='rt' value='r'/>
    <!-- the rest are the service specific params -->
    <param name='ca' value='0'/> <!-- casefold -->
    <param name='st' value='1'/> <!-- stem -->
    <param name='m' value='10'/> <!-- maxdocs -->
    <param name='q' value='snail'/> <!-- query string -->
  </paramList>
</request>
```

There are some standard arguments used in Greenstone, and they are described in Table 8. These are used by Receptionists and Actions. The GSParams class specifies all the general basic arguments, and whether they should be saved or not (Some arguments need to be saved during a session, and this needs to be implemented outside Greenstone proper — currently we do this in the servlet, using servlet session handling). The servlet has an init parameter `params_class` which specifies which params class to use: GSParams can be subclassed if necessary. The Receptionist and Actions must not have conflicting argument names.

Other arguments are used dynamically and come from the Services. Service arguments must always be saved during a session. Services may be created by different people, and may reside on a different site. There is no guarantee that there is no conflict with argument names between services and actions. Therefore service parameters are namespaced when they are put on the page, whereas interface (receptionist and action) parameters have no namespace. The default namespace is s1 (service1) — any parameters that are for the service will be prefixed by this. For example, the case parameter for a search will be put in the page as s1.case, and the resulting argument in a search URL will be s1.case. When actions are deciding which parameters need to be sent in a request to a service, they can use the namespace information.

If there are two or more services combined on a page with a single submit button, they will use namespaces s1, s2, s3 etc as needed. The s (service) parameter will end up with a list of services. For example, `s=TextQuery,MusicQuery`, and the order of these determines the mapping order of the namespaces, i.e. s1 will map to TextQuery, s2 to MusicQuery.

3.9.2 page format

The basic page format is:

```
<page lang='en'>
  <pageRequest/>
  <pageResponse/>
</page>
```

* show configuration and describe whats its used for

There are two main elements in the page: pageRequest, pageResponse. The pageRequest is the original request that came into the Receptionist—this is included so that any parameters can be preset to their previous values, for example,

Argument	Meaning	Typical values
a	action	a (applet), q (query), b (browse), p (page), pr (process) s (system)
sa	subaction	home, about (page action)
c	collection or service cluster	demo, build
s	service name	TextQuery, ImportCollection
rt	request type	d (display), r (request), s (status)
ro	response only	0 or 1 - if set to one, the request is carried out but no processing of the results is done currently only used in process actions
o	output type	XML, HTML, WML
l	language	en, fr, zh ...
d	document id	HASHxxx
r	resource id	???
pid	process handle	an integer identifying a particular process request

Table 8: Generic arguments that can appear in a Greenstone URL

the query options on the query form. The pageResponse contains all the data that has been gathered from the system by the action. The other two elements contain extra information needed by XSLT. Config contains run-time variables such as the location of the gsdl home directory, the current site name, the name of the executable that is running (e.g. library)—these are needed to allow the XSLT to generate correct HTML URLs. Display contains some of the text strings needed in the interface—these are separate from the XSLT to allow for internationalization.

The following subsections outline, for each action, what data is needed and what requests are generated to send to the system.

Once the XML page has been put together, the page to return to the user is created by transforming the XML using XSLT. The output is HTML at this stage, but it will be possible to generate alternative outputs, such as XML, WML etc. A set of XSLT files defines an 'interface'. Different users can change the look of their web pages by creating new XSLT files for a new 'interface'. Just as we have a sites directory where different sites 'live' (ie where their configuration file and collections are located), we have an interfaces directory where the different interfaces 'live' (ie their transforms and images are located there). The default XSLT files are located in interfaces/default/transforms. Collections, sites and other interfaces can override these files by having their own copy of the appropriate files. New interfaces have their own directory inside interfaces/. Sites and collections can have a transform directory containing XSLT files. The order in which the XSLT files are looked for is collection, site, current interface, default interface.[9] [TODO: describe a bit more?? currently only can get this locally]

[9]this currently breaks down for remote sites - need to rethink it a bit.

3.9.3 Receptionists

The receptionist is the controlling module for the page generation part of Greenstone . It has the job of loading up all the actions, and it knows about the message router it and the actions are supposed to talk to. It routes messages received to the appropriate action (page-type messages) or directly to the message router (all other types). Receptionists also do other things, for example, adding to the page received back from the action any information that is common to all pages.

There are different ways of providing an interface to Greenstone , from web based CGI style (using servlets) to Java GUI applications. These different interfaces require slightly different responses from a receptionist, so we provide several standard types of receptionist.

Receptionist: This is the most basic receptionist. The page it returns consists of the original request, and the response from the action it was sent to. Methods preProcessRequest, and postProcessPage are called on the request and page, respectively, but in this basic receptionist, they don't do anything.

TransformingReceptionist: This extends Receptionist, and overwrites postProcessPage to transform the page using XSLT. An XSLT is listed for each action in the receptionists configuration file, and this is used to transform the page. First, some display information, and configuration information is added to the page. Then it is transformed using the specified XSLT for the action, and returned.

WebReceptionist: The WebReceptionist extends TransformingReceptionist. It doesn't do much else except some argument conversion. To keep the URLs short, parameters from the services are given shortnames, and these are used in the web pages.

DefaultReceptionist: This extends WebReceptionist, and is the default one for Greenstone3 servlets. Due to the page design, some extra information is needed for each page: some metadata about the current collection. The receptionist sends a describe request to the collection to get this, and appends it to the page before transformation using XSLT.

By default, the LibraryServlet uses DefaultReceptionist. However, there is a servlet init-param called `receptionist` which can be set to make the servlet use a different one.

3.9.4 Collection specific formatting

get format info, transform gsf-¿xsl. transform xml-¿html

configuration params are passed in to the transformation

3.9.5 CGI arguments

3.9.6 Page action

PageAction is responsible for displaying kinds of information pages, such as the home page of the library, or the home page of a collection, or the help and pref-

erences pages. These pages are not associated with specific services like the other page types. In general, the data comes from describe requests to various modules. The different pages are requested using the subaction argument. For the 'home' page, a 'describe' request is sent to the MessageRouter—this returns a list of all the collections, services, serviceClusters and sites known about. For each collection, its metadata is retrieved via a 'describe' request. This metadata is added into the previous result, which is then added into the page. For the 'about' page, a describe request is sent to the module that the about page is about: this may be a collection or a service cluster. This returns a list of metadata and a list of services.

To get an external html page embedded into a greenstone collection, i.e. a two frame page, with the top frame containing the collection header and navigation bar, and the second frame containg the external page, use subaction html. A url would look like a=p&sa=html&c=collname&url=externalurl

3.9.7 Query action

The basic URL is a=q&s=TextQuery&c=demo&rt=d/r. There are three query services which have been implemented: TextQuery, FieldQuery, and AdvancedFieldQuery. These are all handled in the same way by query action. For each page, the service description is requested from the service of the current collection (via a describe request). This is currently done every time the query page is displayed, but should be cached. The description includes a list of the parameters available for the query, such as case/stem, max num docs to return, etc. If the request type (rt) parameter is set to d for display, the action only needs to display the form, and this is the only request to the service. Otherwise, the submit button has been pressed, and a query request to the TextQuery service is sent. This has all the parameters from the URL put into the parameter list. A list of document identifiers is returned. A followup query is sent to the MetadataRetrieve service of the collection: the content includes the list of documents, with a request for some of their metadata. Which metadata to retrieve is determined by looking through the XSLT that will be used to transform the page. The service description and query result are combined into a page of XML, which is returned to the Receptionist.

3.9.8 Applet action

There are two types of request to the applet action: a=a & rt=d and a=a & rt=r. The value rt=d means "display the applet." A describe request is sent to the service, which returns the <applet> HTML element. The transformation file applet.xsl embeds this into the page, and the servlet returns the HTML.

The value rt=r signals a request from the applet. A process request containing all the parameters is sent to the applet service. The result contains an appletData element, which contains a single element - this element is returned directly to the applet, in XML. No transformation is done. Because the AppletAction doesn't know or care anything about the applet data, it can work with any applet-service

pair.

Note that the applet HTML may need to know the name of the `library` program. However, that name is chosen by the person who installed the software and will not necessarily be "library". To get around this, the applet can put a parameter called "library" into the applet data with a null value:

```
<PARAM NAME='library' VALUE=''/>
```

When the AppletAction encounters this parameter it inserts the name of the current library servlet as its value.

3.9.9 Document action

DocumentAction is responsible for displaying a document to the user. The display might involve some metadata and/or text for a document or part of a document. For hierarchical documents, a table of contents may be shown, while for paged documents (those with a single linear list of sections), next and previous page buttons may be shown. These different display types require different information about the document. Depending on the arguments, DocumentAction will send requests to several services: DocumentMetadataRetrieve, DocumentStructureRetrieve and DocumentContentRetrieve.

A basic display, for example, Title and text, involves a metadata request to get the Title, and a content request to get the text. Hierarchical table of contents display requires a structure request. If the entire contents is to be displayed, the parameter `structure=entire` would be sent in the request. Otherwise, parameters `structure=ancestors`, `structure=children` and possibly `structure=siblings` may be used, depending in the position of the current node in the document. These return a hierarchical structure of nodes, containing ancestor nodes, child nodes and sibling nodes, respectively. For paged display, the structure is not actually needed. A structure request is still sent, but this time it requests some information, rather the structure itself. The information requested includes the number of siblings and the current position of the current node, or the number of children (if the current node is the root of the document).

Metadata may be requested for the current node, or for any nodes in the structure, and content also. The metadata and content are added into the appropriate nodes in the structure hierarchy, and this is returned as the page data.

3.9.10 XML Document action

XMLDocumentAction is a little different to the standard DocumentAction. It operates in two modes, `text` and `toc`. In `text` mode, it will retrieve the content of the current document node using a DocumentContentRetrieve request. In `toc` mode, it retrieves the entire table of contents for the document using a DocumentStructureRetrieve request. Either mode may also retrieve metadata for the current section or each section in the table of contents.

Table 9: Configure CGI arguments

arg	description
a=s	system action
sa=c|a|d	type of system request: c (configure), a (add/activate), d (delete/deactivate)
c=demo	the request will go to this collection/servicecluster instead of the message router
ss=collectionList	subset for configure: only reconfigure this part. For the MessageRouter, can be serviceClusterList, serviceList, collectionList, siteList. For a collection/cluster, can be metadataList or serviceList.
sn=demo	
st=collection	

3.9.11 GS2Browse action

GS2BrowseAction is for displaying Greenstone2 style classifiers.

3.9.12 System action

SystemAction allows for manual reconfiguration of various components at run-time. There is no interactive web-page displaying the options, it merely turns a set of CGI arguments into an XML system request. The response from a system request is a message which is displayed to the user.

3.10 Other code information

Greenstone has a set of Utility classes, which are briefly described in Table 10.

Table 10: The utility classes in org.greenstone.gsdl3.util

Utility class	Description
CollectionClassLoader	ClassLoader that knows about a collection's resource directory
DBInfo	Class to hold info from GDBM database entry
Dictionary	wrapper around a Resource Bundle, providing strings with parameters
GDBMWrapper	Wrapper for GDBM database. Uses JavaGDBM
GSConstants	holds some constants used for servlet arguments and configuration variables
GSEntityResolver	an EntityResolver which can be used to find resources such as DTDs
GSFile	class to create all Greenstone file paths e.g. used to locate configuration files, XSLT files and collection data.
GSHTML	provides convenience methods for dealing with HTML, e.g. making strings HTML safe
GSParams	contains names and default values for interface parameters
GS2Params	a subclass of GSParams which holds default service parameters too, necessary for the gs2 style interface.
GSPath	used to create, examine and modify message address paths
GSStatus	some static codes for status messages
GSXML	lots of methods for extracting information out of Greenstone XML, and creating some common types of elements. Also has static Strings for element and attribute names used by Greenstone .
GSXSLT	some manipulation functions for Greenstone XSLT
GlobalProperties	Holds the global properties (from global.properties)
MacroResolver	Used with replace elements in collection configuration files, replaces a macro or string with another string, metadata or text from a dictionary
GS2MacroResolver	MacroResolver for GS2 collections, that uses the GDBM database
Misc	miscellaneous functions
MyNodeList	A simple implementation of an XML NodeList
OID	class to handle Greenstone (2) OIDs
Processing	Runs an external process and prints the output from the process
SQLQuery	contains a connection to a SQL database, along with some methods for accessing the data, such as converting MG numbers to and from Greenstone OIDs.
XMLConverter	provides methods to create new Documents, parse Strings or Files into Documents, and convert Nodes to Strings
XMLTransformer	methods to transform XML using XSLT
XSLTUtil	contains static methods to be called from within XSLT

4 Developing Greenstone3 : Adding new features

[TODO: finish this section]

4.1 Creating and using new services

There are three parts to adding new services to Greenstone3: defining the new service, specifying that it should be loaded, and using it. If you are talking to Greenstone using the SOAP interface, then the firsttwo parts are all that need to be done. If you are using the Greenstone servlet interface, then you may need to do work for the third part, depending on what kind of new service it is. If you are adding a service of a type that is already present, for example, a new query service, then the query action can just use your new service as is (assuming it is set up in the

same way as the standard query services). However, if it is a new type of service that the interface and actions don't know about, you willl need to add a new action or modify an existing one so that your service is actually used.

4.1.1 Creating the service

You will need to write a new Java class which inherits from `org.greenstone.gsdl3.service.ServiceRack` (or a subclass of this). The class will need to implement at least the `configure`, `process<ServiceName>` and `getServiceDescription` methods. There is a dummy class called `MyNewServicesTemplate.java` in `greenstone3/resources/java` which describes these methods and what needs to be done.

`ServiceRack.java` handles the main `process` method. If the request type is 'describe', then it will send back a copy of short_service_info, which contains a list of services. If there request type is describe, but for a particular service, then it will call `getServiceDescroption` for that service. For a format request, it will send any format element found in format_info_map for that service. For a processing request to a service, then the `process<ServiceName>` method will be called.

Once the class is written, it needs to be compiled up and either included in one of the existing jar files, or added in as a jar file to `greenstone3/web/WEB-INF/lib` or a class file to `greenstone3/web/WEB-INF/classes`.

4.1.2 Loading the service

To have the library load in your new service, it needs to be specified in a configuration file somewhere. For a collection service, add a new `<serviceRack>` element to the collection's `buildConfig.xml` file. This element should contain any information that the class needs to configure its service(s). For a site-wide service, add the `<serviceRack>` element to the site's `siteConfig.xml` file, either in the `serviceRackList` or as part of a `serviceCluster`.

4.1.3 Using the service

If you are using the SOAP web service, then you can send an XML request directly to the service. The 'address' of the request will be the service name if it is a site-wide service, cluster-name/service-name if it is site-wide but belonging to a cluster, or collection-name/service-name if it belongs to a collection. You will need to know the format of the XML request and response that the service expects and returns.

If you want to access your new service through the current servlet interface that uses actions, then whether you need to do more work or not depends on whatkind of service you have implemented. If you have written a new query or browse service, for example, that has teh same request and response format as the existing services, then you don't need to do anything else. Your collection can just use the new query service straight away. If the service is of an existing type, but needs soemthing

different in the request/response format, then you may need to modify an existing action to supply or use the new information. If the service is of a completely new type, then you will probably need a new action to talk to the service and display the results.

4.2 creating new actions/pages

4.3 new interfaces

It is easy to create new interfaces to Greenstone3. Here we are talking about interfaces other than those to display in typical browser.

Handheld devices: Use the standard servlet setup, but with a different set of XSLT files to format the pages for small screens, or use WML.

Java GUI Interface: There are couple of alternatives. Depending on what you want to display in the GUI, you could talk to either a Receptionist or a MessageRouter. The library classes can be set up and compiled into the GUI program. Talking to a Receptionist will give you access to pages of XML. It is likely that the standard Receptionist class would be used - this doesn't transform the data to HTML. Queries such as "give me the home page of a collection" and "do the following search" can be issued. All the data needed for the result view is returned. Queries are quite simple, but are limited to what kinds of Actions are available in the library. Talking to a MessageRouter requires a bit more effort on the part of the GUI program, but results in greater flexibility. The kinds of queries that can be issued are individual units of action, such as "describe yourself", "search", "retrieve the content for this document". More than one request may need to be made for a particular feature of the GUI. However you can ask for any combination of data available in the system, you are not relying on Actions. What you will implement though, may be a lot like the Action code in terms of request sequences.

Interfaces in other programming languages: Because the communication is all XML based, other interfaces can talk to the Java library if a communication protocol is set up. This could be done using SOAP for example. Like for Java GUI interfaces, the program could talk to a Receptionist or to a MessageRouter. e.g. Java interface. where you can interface to. MR vs Receptionist. different receptionists. e.g., handheld - using servlet, transforming recpt, but new set of XSLT Java program other program - talk to recpt but just get back XML data for pages. Java gui - just talk to MR, do all processing itself.

Remote interfaces: remote interfaces can be set up in the same way as above, using a communication protocol between the interface, and the library program.

4.4 New types of collections

The standard type of collection is built with the Greenstone2 Perl collection building system. There are many options to this, but it is conceivable that these options don't meet the needs of all collection builders. Greenstone3 has an ability to use any type of collection you can come up with, assuming some Java code is provided.

There are four levels of customization that may be needed with new collections: service, collection, interface XSLT, and action levels. We will use the example collections that come with Greenstone to describe these different levels.

Firstly, new service classes need to be written to provide the functionality to search/browse/whatever the collection. If the services have similar interfaces and functionality to the standard services, this may be all that is needed. For example, MGPP collections were the first to be served in Greenstone3 . When we came to do MG collections, all we had to do was write some new service classes that interacted with MG instead of MGPP. Because these collections used the same type of services, this was all we had to do. The format of the configuration files was similar, they just specified MG serviceRack classes rather than MGPP ones.

The XML Sample Texts (gberg) collection, however, was done quite differently to the standard collections. New services were provided to search the database (built with Lucene) and to provide the documents and parts of documents (using XSLT to transform the raw XML files). The collectionConfig file had some extra information in it: a list of the documents in the collection along with their Titles. Because the standard collection class has no notion of document lists, a new class was created (org.greenstone.gsdl3.collection.XMLCollection). This class is basically the same as a standard collection class except that it looks for and stores in memory the documentList from the collectionConfig file.

To tell Greenstone to load up a different type of collection class, we use another configuration file: `etc/collectionInit.xml`. This specifies the name of the collection class to use. Currently, this is all that is specified in that file, but you may want to add parameters for the class etc.

```
<collectionInit class="XMLCollection"/>
```

The display for the collection is also quite different. The home page for the collection displays the list of documents. To achieve this, the describe response from the collection had to include the list, and a new XSLT was written for the collection that displayed this. Collection XSLT should be put in the transform directory of the collection[10].

Document display is significantly different to standard Greenstone . There are two modes of display: table of contents mode, and content mode. Clicking on a document link from the collection home page takes the user to the table of contents for the collection. Clicking on one of the sections in the table of contents takes them to a display of that section. To facilitate this, not only do we need new XSLT files , we also needed a new action. XMLDocumentAction was created, that used two subactions, toc and text, for the different modes of display.

The Receptionist was told about this new action by the addition of the following element to the interfaceConfig.xml file:

```
<action name='xd' class='XMLDocumentAction'>
  <subaction name='toc' xslt='document-toc.xsl'/>
```

[10]These are currently only used when running Greenstone in a non-distributed fashion, but it will be added in properly at some stage

```
  <subaction name='text' xslt='document-content.xsl'/>
</action>
```

XSLT files are linked to subactions rather than the action as a whole. The collection supplies the two XSLT files written appropriately for the data it contains.

All links that link to the documents have to be changed to use the xd action rather than the standard d action. These include the links from the home page, and the links from query results.

Querying of the collection is almost the same as usual. The query service provides a list of parameters, does the query and then sends back a list of document identifiers. The standard query action was fine for this collection. The change occurs in the way that the results are displayed—this is accomplished using a format statement supplied in the collectionConfig file inside the search node.

```
<search>
  <format>
    <gsf:template match="documentNode">
      <xsl:param name="collName"/>
      <xsl:param name="serviceName"/>
      <td>
        <b><a href="{$library_name}?a=xd&amp;sa=text&amp;c={$collName}&
            amp;d={@nodeID}&amp;p.a=q&amp;p.s={$serviceName}">
             <xsl:choose>
               <xsl:when test="metadataList/metadata[@name='Title']">
                 <gsf:metadata name="Title"/>
               </xsl:when>
               <xsl:otherwise>(section)</xsl:otherwise>
             </xsl:choose>
           </a>
         </b> from <b><a href="{$library_name}?a=xd&amp;sa=toc&amp;
           c={$collName}&amp;d={@nodeID}.rt&amp;p.a=q&amp;p.s={$serviceName}">
         <gsf:metadata name="Title" select="root"/></a></b>
      </td>
    </gsf:template>
  </format>
</search>
```

Instead of displaying an icon and the Title, it displays the Title of the section and the title of the document. Both of these are linked to the document: the section title to the content of that section, the document title to the table of contents for the document. Because these require non-standard arguments to the library, these parts of the template are written in XSLT not Greenstone format language. As is shown here it is perfectly feasible to write a format statement that includes XSLT mixed in with Greenstone format elements.

The document display uses CSS to format the output—these are kept in the collection and specified in the collections XSLT files. The documents also specify DTD files. Due to the way we read in the XML files, Tomcat sometimes has trouble locating the DTDs. One option is to make all the links absolute links to files in the collection folder, the other option is to put them in Greenstone 's DTD folder `$GSDL3SRCHOME/resources/dtd`.

4.5 The gs2 Interface

The library seen at `http://www.greenstone.org/greenstone3/nzdl` is like a mirror to `http://www.nzdl.org`—it aims to present the same collections, in the same way but using Greenstone3 instead of Greenstone2 . It uses a new site (nzdl) with a new interface (nzdl) which is based on the gs2 interface. The web.xml file had a new servlet entry in it to specify the combination of nzdl site and nzdl interface.

The site was created by making a directory called nzdl in the sites folder. A siteConfig file was created. Because it is running on Linux, we were able to link to all the collections in the old Greenstone installation. The convert_coll_from_gs2.pl script was run over all the collections to produce the new XML configuration files.

The gs2 interface was created to be used by this site (and is now a standard part of Greenstone). In many cases, creating a new interface just requires the new images and XSLT to be added to the new directory(see Sections 1.4 and 2.5). This gs2 interface required a bit more customization.

The standard Greenstone3 navigation bar lists all the services available for the collection. In Greenstone2 , the navigation bar provides the search option, and the different classifiers. This is not service specific, but hard coded to the search and classifiers. The XSLT that produces the navigation bar needed to be altered to produce this. The standard receptionist (DefaultReceptionist) gathers a little bit of extra information for each page of XML before transforming it: this is the list of services for the collection and their display information, allowing the services to be listed along the navigation bar. This is information that is needed by every page (except for the library home page) and therefore is obtained by the receptionist instead of by each action. The nzdl interface uses the classifier list that comes in the ClassifierBrowse service description to display teh list of classifiers.

The nzdl interface extends the gs2 interface to provide a different looking home page and an extra static 'gsdl' page.

5 Distributed Greenstone

Greenstone is designed to run in a distributed fashion. One Greenstone installation can talk to several sites on different computers. This requires some sort of communication protocol. Any protocol can be used, currently we have a simple SOAP protocol.

more explanation..

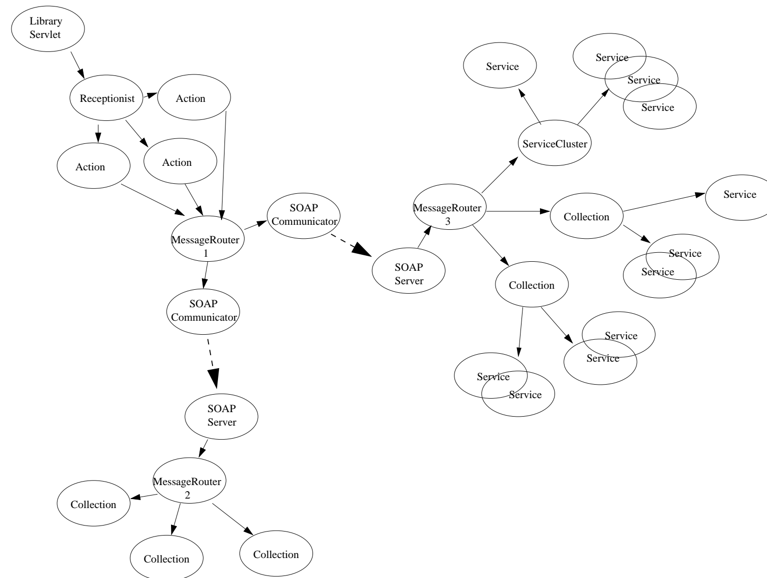

Figure 10: A distributed digital library configuration running over several servers

We have used Apache Axis SOAP implementation. This is run as a servlet in Tomcat. Axis is set up during installation of Greenstone. For more details about SOAP in Greenstone, see Appendix C. Debugging soap is described in Appendix C.1.

5.1 Serving a site using soap

A web service for localsite comes with Greenstone. However, it is not deployed by default. To deploy it, run run ant deploy-localsite. If you want to set up web services for other sites, run ant soap-deploy-site. This will prompt you for the sitename (its directory name), and a siteuri - a unique identifier for the web service. Tomcat needs to be running for this to work, and you need to have installed the Greenstonesource code.

The ant target deploys the service for the site specified. A resource file (<sitename>.wsdd) is created which is used to specify the service. It can be found in $GSDL3HOME/resources/soap, and is generated from site.wsdd.template.

The address of the new SOAP service will be tomcatserver-address/greenstone3/services/sitename, for example, www.greenstone.org/greenstone3/services/localsite.

5.2 Connecting to a site web service

There are two ways to use a remote site. First, if you have a local site running, then the site can also connect to other remote sites. In the siteConfig.xml file, you need to add a site element into the siteList element.

For example, to get siteA to talk to siteB, you need to deploy a SOAP server on siteB, then add a `<site>` element to the `<siteList>` of siteA's `siteConfig.xml` file (in `$GSDL3HOME/sites/siteA/siteConfig.xml`).

In the `<siteList>` element, add the following (substituting the chosen site uri for siteBuri):

```
<site name="siteBuri"
  address="http://localhost:8080/greenstone3/services/siteBuri"
  type="soap"/>
```

(Note that localhost and 8080 should be changed to the values you entered when installing Greenstone3. Localhost will only work for servers on the smae machine.).

If you have changed the siteConfig.xml file for a site that is running, it will need to be reconfigured. Either restart Tomcat, or reconfigure through a URL: e.g. `http://localhost:8080/greenstone3/library?a=s&sa=c`. Several sites can be connected to in this manner.

The second option is if you have a receptionist set up on a machine where you have no site, and you only want to connect to a single remote site. Instead of using site_name in the servlet initialisation parameters (in $GSDL3HOME/WEB-INF/web.xml), you can specify remote_site_name, remote_site_type and remote_site_address. A communicator object will be set up instead of a MessageRouter and the receptionist will talk to the communicator.

A Using Greenstone3 from CVS

Greenstone3 is also available via CVS. You can download the latest version of the code. This is not guaranteed to be stable, in fact it is likely to be unstable. The advantage of using CVS is that you can update the code and get the latest fixes.

Note that you will need the Java 2 SDK, version 1.4.0 or higher, and Ant (Apache's Java based build tool, http://ant.apache.org) installed.

To check out the Greenstone code, use:

```
cvs -d :pserver:cvs\_anon@cvs.scms.waikato.ac.nz:2402/usr/local/
          global-cvs/gsdl-src co -P greenstone3
```

If you need it, the password for anonymous CVS access is `anonymous`. Note that some older versions of CVS have trouble accessing this repository due to the port number being present. We are using version 1.11.1p1.

Greenstone is built and installed using Ant (Apache's Java based build tool, http://ant.apache.org). You will need a Java Development Environment (1.4 or higher), and Ant installed to use Greenstone. You can download Ant from `http://ant.apache.org/bindownload.cgi`. Make sure that the environment variables JAVA_HOME and ANT_HOME are set.

In the `greenstone3` directory, you can run `'ant'` which will give you a help message. Running `'ant -projecthelp'` gives a list of the targets that you can run — these do various things like compile the source code, startup the server etc.

The `README.txt` file has up-to-date instructions for installing from CVS. Briefly, for a first time install, run `'ant prepare install'`.

The file `build.properties` contains various parameters that can be set by the user. Please check these settings before running the installation process. The install process will ask you if you accept the properties before starting. For a non-interactive version of the install, run `'ant -Dproperties.accepted=yes install'`

To log the output in build.log, run `'ant -Dproperties.accepted=yes -logfile build.log install'`

Compilation includes Java and C/C++. On Windows, you will need to have Visual Studio or equivalent installed. Please check the `compile.windows.c++.setup` property in build.properties — make sure it is set to the setup script of Visual Studio.

Note: `gs3-setup` sets the environment variables `GSDL3HOME, GSDL3SRCHOME, CLASSPATH, PATH, JAVA_HOME` and needs to be done in a shell before doing collection building etc.

To run the library, use the `gs3-server.sh/bat` shell scripts.

B Tomcat

Tomcat is a servlet container, and Greenstone3 runs as a servlet inside it.

The file `$GSDL3SRCHOME/packages/tomcat/conf/server.xml` is the Tomcat configuration file. A context for Greenstone3 is given by the file `$GSDL3SRCHOME/packages/tomcat/conf/Catalina/localhost/greenstone3.xml`. This tells Tomcat where to find the web.xml file, and what URL (`/greenstone3`) to give it. Anything inside the context directory is accessible via Tomcat[11]. For example, the index.html file that lives in `$GSDL3HOME` can be accessed through the URL `localhost:8080/greenstone3/index.html`. The gs2mgdemo collection's images can be accessed through `localhost:8080/greenstone3/sites/localsite/collect/gs2mgdemo/images/`.

Greenstone sets up Tomcat to run on port 8080 by default. To change this, you can edit the tomcat.port property in build.properties. If you do this before installing Greenstone, then running 'ant install' will use the new port number. If you want to change it later on, shutdown tomcat, run 'ant configure', then when you restart tomcat it will use the new port.

Note: Tomcat must be shutdown and restarted any time you make changes in the following for those changes to take effect:

- `$GSDL3HOME/WEB-INF/web.xml`
- `$GSDL3SRCHOME/packages/tomcat/conf/server.xml`
- any classes or jar files used by the servlets

On startup, the servlet loads in its collections and services. If the site or collection configuration files are changed, these changes will not take effect until the site/collection is reloaded. This can be done through the reconfiguration messages (see Section 1.7), or by restarting Tomcat.

We have disabled following symlinks for the greenstone servlet. To enable it, edit `$GSDL3SRCHOME/packages/tomcat/conf/Catalina/localhost/greenstone3.xml` and set 'allowLinking' to true.

By default, Tomcat allows directory listings. To disable this, change the 'listings' parameter to false in the default servlet definition, in Tomcat's web.xml file (`$GSDL3SRCHOME/packages/tomcat/conf/web.xml`):

We have set the greenstone context to be reloadable. This means that if a class or resource file in web/WEB-INF/lib or web/WEB-INF/classes changes, the servlet will be reloaded. This is useful for development, but should be turned off for production mode (set the 'reloadable' attribute to false).

Tomcat uses a Manager to handle HTTP session information. This may be stored between restarts if possible. To use a persistent session handling manager, uncomment the `<Manager>` element in `$GSDL3SRCHOME/packages/tomcat/conf/server.xml`. For the default manager, session information is stored in the work directory:

[11] can we use .htaccess files to restrict access??

`$GSDL3SRCHOME/packages/tomcat/work/Catalina/localhost/greenstone3/SESSIONS.ser`. Delete this file to clear the cached session info. Note that Tomcat needs to be shutdown to delete this file.

B.1 Proxying Tomcat with apache

Instead of incorporating servlet support into your existing web server, an easy alternative is to proxy Tomcat. The `http://www.greenstone.org/greenstone3` site uses apache to proxy Tomcat. ProxyPass and ProxyPassReverse directives need to be added to the Virtualhost description for the www.greenstone.org server.

```
<VirtualHost xx.xx.xx.xx>
ServerName www.greenstone.org
...
ProxyPass /greenstone3 http://puka.cs.waikato.ac.nz:8080/greenstone3
ProxyPassReverse /greenstone3 http://puka.cs.waikato.ac.nz:8080/greenstone3
</VirtualHost>
```

In our example, the Greenstone3 servlet can be accessed at `http://www.greenstone.org/greenstone3/library`, instead of at `http://puka.cs.waikato.ac.nz:8080/greenstone3/library`, which is not publically accessible.

B.2 Running Tomcat behind a proxy

Almost everything works fine when Tomcat is running behind a proxy. The only time this causes trouble is if the servlet itself needs to make external HTTP connections. We do this in the infomine demo collection for example. One of the service classes sends HTTP requests to the infomine database at riverside. Since this is going through the proxy, a username and password is needed. It is not sufficient to prompt the user for a password because they are unlikely to have a password for the particular proxy that Tomcat is using. What we have done at present is to put a proxy element in the siteConfig.xml file. Here you have to enter a suitable username and password for the proxy server. Unfortunately these are entered in plain text. And the file is viewable via the servlet. So we need a better solution.

C SOAP

Greenstone uses the Apache Axis SOAP implementation for distributed communications. Axis runs as a servlet inside Tomcat, and SOAP web services can be deployed by this Axis servlet. The Greenstone installation process sets up Axis for Tomcat, but does not deploy any services.

To deploy the SOAP service for localsite, run `ant deploy-localsite`.

To deploy a SOAP service for other sites, run `ant soap-deploy-site`

This will prompt you for the sitename (the site's directory name), and a unique URI for the site. It creates a new SOAPServer class for the site (`$GSDL3SRCHOME/src/java/org/greenstone/gsdl3/SOAPServer<sitename>.java`), creates a resource file for deployment (`$GSDL3SRCHOME/resources/soap/<sitename>.wsdd`), and then tries to deploy the service.

Information about deployed services is maintained between Tomcat sessions— you only need to deploy something once. To undeploy a site, use `ant soap-undeploy-site`.

The axis services can be accessed at `localhost:8080/greenstone3/index.jsp`.

C.1 Debugging SOAP

If you need to debug the SOAP stuff for some reason, or just want to look at the SOAP messages that are being passed back and forth, you can use the TCP monitor. This intercepts messages coming in to one port, displays them, and passes them to another port. To run it, type:

```
java -cp $GSDL3HOME/WEB-INF/lib/axis.jar
org.apache.axis.utils.tcpmon
```

The listen port is the port that you want the monitor to be listening on. It should 'act as' a Listener, with target hostname 127.0.0.1 (localhost), and target port the port that Tomcat is running on (8080). You need to modify the address used to talk to the SOAP service. For example, if you want to monitor traffic between the gateway site and the localsite SOAP server, you will need to edit gateway's siteConfig.xml file and change the port number (in the site element) to whatever you have chosen as the listen port.

For example, in the Admin panel of TCPMonitor the Target Hostname might be 127.0.0.1, and the Target Port # 8080. Set the Listen Port # to be a different port, such as 8070 and click Add. This produces a new tab panel where you can see the messages arriving at port 8070 before being forwarded to port 8080. You then need to set your test request from your SOAP application to arrive at port 8070 and you will see copies of the messages in the new tab panel.

D Tidying up the formatting for imported Greenstone2 collections

D.1 Format statements: Greenstone2 vs Greenstone3

The following table shows the Greenstone2 format elements, and their equivalents in Greenstone3

Table 11: Greenstone3 equivalents of Greenstone2 format statements

Greenstone2	Greenstone3
[Text]	<gsf:text/>
[num]	<gsf:metadata name='docnum'/>
[link][/link]	<gsf:link></gsf:link> or <gsf:link type='document'></gsf:link>
[srclink][/srclink]	<gsf:link type='source'></gsf:link>
[icon]	<gsf:icon/> or <gsf:icon type='document'/>
[srcicon]	<gsf:icon type='source'/>
[Title] (metadata)	<gsf:metadata name='Title'/> or <gsf:metadata name='Title' select='current'/>
[parent:Title]	<gsf:metadata name='Title' select='parent' />
[parent(All):Title]	<gsf:metadata name='Title' select='ancestors'/>
[parent(Top):Title]	<gsf:metadata name='Title' select='root' />
[parent(All': '):Title]	<gsf:metadata name='Title' select='ancestors' separator=': ' />
[sibling(All': '):Title]	<gsf:metadata name='Title' multiple='true' separator=': ' />
{Or}{[dc.Title], [dls.Title], [Title]}	<gsf:choose-metadata> <gsf:metadata name='dc.Title'/> <gsf:metadata name='dls.Title'/> <gsf:metadata name='Title'/> </gsf:choose-metadata>
{If}{[parent:Title], [parent:Title], [Title]}	<gsf:choose-metadata> <gsf:metadata name='Title' select='parent'/> <gsf:metadata name='Title'/> </gsf:choose-metadata>
{If}{[Subject], <td>[Subject]</td>}	<gsf:switch> <gsf:metadata name='Subject'/> <gsf:when test='exists'> <td><gsf:metadata name='Subject'/></td> </gsf:when></gsf:switch>

D.2 Cleaning up macros

Here we show some of the replace items that have been used for Greenstone2 collections.

Getting rid of silly backslashes:

```
<replace scope='text' macro="\\?\\\(" text="\("/>
```

Macro resolving using resource bundles and metadata:

```
<replace scope='metadata' macro="_magazines_" bundle="NZDLMacros"
   key="Magazines"/>
<replace scope='all' macro='_thisOID_' metadata='archivedir'/>
<replace macro="_httpcollimg_"
   text="sites/localsite/collect/folktale/index/assoc"/>
```

Fixing up broken external links:

```
<replace macro="_httpextlink_&amp;rl=1&amp;href="
   text="?a=d&amp;c=folktale&amp;s0.ext=1&amp;d="/>
<replace macro="_httpextlink_&amp;rl=0&amp;href="
   text="?a=p&amp;sa=html&amp;c=folktale&amp;url="/>
```

These two examples show how to deal with Greenstone2's external link macros. The first one is for a 'relative' external link. In this case, the links are like URL's but they actually refer to Greenstone internal documents. So the Greenstone3 link is to the document, but with parameter s0.ext signifying that the d argument will need translating before retrieving the content. The second example is a truly external link. This is translated into a HTML type page action, where the URL is presented as a frame along with the collection header in a separate frame.

Sometimes we need to add in macros to be resolved in a second step:

```
<replace macro="_iconpdf_" scope="metadata"
   text="&lt;img title='_texticonpdf_' src='interfaces/default/images/ipdf.gif'/&gt;"/>
<replace macro="_texticonpdf_" scope="metadata" bundle="interface_gs2"
   key="texticonpdf"/>
```