Greenstone 3: A modular digital library.

Katherine Don, George Buchanan and Ian H. Witten

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{kjdon, grbuchan, ihw}@cs.waikato.ac.nz

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 capitalise 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 internationalisation, 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 gsdl3/docs/manual directory).

This documentation consists of several parts. Section 1 covers greenstone installation, how to access the library, and some administration issues. Section 2 looks at using the sample collections, creating new collections, and how to make small customisations 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, while Section 4 describes the collection building process. Section 5 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 6 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, and a comparison of greenstone 2 and greenstone 3 format statements.

1 Greenstone installation and administration

This section covers where to get Greenstone 3 from, how to install it and how to run it. The standard method of running Greenstone is as a Java servlet. We provide the Tomcat servlet container to serve 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 Greenstone, Tomcat must be started up, and then it can be accessed via a web browser.

1.1 Get and install Greenstone

Greenstone is available from `http://www.greenstone.org/greensone3`. There are currently two distributions: a self-installing tar for Linux, and a Windows executable.

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

1.1.1 Linux

Download the latest version of the self-installing tar file, gsdl3-x.xx-unix.sh, and run it in a shell (./gsdl3-x.xx-unix.sh). Greenstone will be installed into a directory called gsdl3 inside the current directory. The install script will prompt you for the name of your computer and what port to run tomcat on (the defaults being localhost and 8080). Once Greenstone has been installed, you can start the library by running ./gsdl3/gs3-launch.sh, and opening up a browser pointing to localhost:8080/gsdl3 (or different computer name and port).

1.1.2 Windows

Download the latest Windows executable, gsdl3-x.xx-win32.exe, and double click it to start the installation. You will be prompted for your computer name and port number to run Tomcat on (defaults are localhost and 8080). Once Greenstone is installed, you can access the library by selecting Greenstone 3 Digital Library in the Start menu.

1.1.3 Accessing the library in a browser

Once you have started up the library (see the previous sections for OS dependent instructions), you can access it in a browser at http://localhost:8080/gsdl3 (or http://your-computer-name:your-chosen-port/gsdl3). This gets you to a welcome page, with three links: one to run a test servlet (this allows you to check that Tomcat is running properly), one to run the standard library servlet using the site `localsite`, and one to run a library servlet using the site `soapsite`. This site

uses a SOAP connection to communicate with localsite, and demonstrates the library working in a distributed fashion. See Section 6 for details about how to run Greenstone distributedly.

1.2 How the library works

The standard library program is a Java servlet.

Other types of interfaces can be used, such as Java GUI programs. See Section 5.3 for details about how to make these.

1.2.1 Restarting the library

The library program (actually tomcat) can be restarted by ... (** put a mechanism in each install program **).

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

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

Note: stdout and stderr for the servlets both go to
`$GSDL3HOME/comms/jakarta/tomcat/logs/catalina.out`

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. Under Linux, these will eventually be installed into appropriate system directories. The second part shows stuff used by one person/group—their sites and interface setup (see Section 1.4). etc. There can be several sites/interfaces per installation.

1.4 Sites and interfaces

local gs stuff (sites and interfaces) vs installed stuff (code)
where they live, whats the difference, what each contains.

A site is comprised of a set of collections and possibly services. An interface 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 for the servlet case. One greenstone installation can have many sites and interfaces. One instantiation of a servlet uses one site and one interface. Sites and interfaces can be matched up in different ways. For example, a single site might be served with two different interfaces. This provides different modes of access to the same content. eg HTML vs WML, or perhaps providing completely different look and feel for different audiences. A

Table 1: The Greenstone directory structure

directory	description
gsdl3	The main installation directory—gsdl3home can be changed to something more standard
gsdl3/src	Source code lives here
gsdl3/src/java/	java source code
gsdl3/src/cpp/	c/ cpp source code—none yet
gsdl3/packages	Imported packages from other systems e.g. MG, MGPP
gsdl3/lib	Shared library files
gsdl3/lib/java	Java jar files
gsdl3/resources	any resources that may be needed
gsdl3/resources/java	properties files for java resource bundles - used to handle all the language specific text This directory is on the class path, so any other Java resources can be placed here
gsdl3/resources/soap	soap service description files
gsdl3/resources/dtd	Greenstone has trouble loading DTD files sometimes. They can go here
gsdl3/bin	executable stuff lives here
gsdl3/bin/script	some Perl building scripts
gsdl3/bin/linux	Linux executables for e.g. MGPP
gsdl3/bin/windows	windows executables for e.g. MGPP
gsdl3/comms	Put some stuff here for want of a better place—things to do with servers and communication. e.g. soap stuff, and tomcat servlet container
gsdl3/docs	Documentation :-)
gsdl3/web	This is where the web site is defined. Any static html files can go here. This directory is the Tomcat root directory.
gsdl3/web/WEB-INF	The web.xml file lives here (servlet configuration information for tomcat)
gsdl3/web/WEB-INF/classes	Servlet classes go in here
gsdl3/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
gsdl3/web/sites/localsite	One site - the site configuration file lives here
gsdl3/web/sites/localsite/collect	The collections directory
gsdl3/web/sites/localsite/images	Site specific images
gsdl3/web/sites/localsite/transforms	Site specific transforms
gsdl3/web/interfaces	Contains directories for different interfaces - an interface is defined by its images and XSLT files
gsdl3/web/interfaces/default	The default interface
gsdl3/web/interfaces/default/images	The images for the default interface
gsdl3/web/interfaces/default/transforms	The XSLT files for the default interface

Table 2: Greenstone servlet initialisation parameters

name	sample value	description
gsdl3_home	/research/kjdon/gsdl3	the base directory of the gsdl3 installation
site_name	localsite	the name of the site to use
interface_name	default	the name or the interface to use
library_name	library	the web name of the servlet
default_lang	en	the default language for the interface
receptionist_class	NZDLReceptionist	(optional) specifies an alternative Receptionist to use
messagerouter_class	NewMessageRouter	(optional) specifies an alternative MessageRouter to use

standard interface may be used with many different sites—provides 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 initialised will be loaded up and presented to the user. Collections require valid configuration files, but apart from this, nothing needs to be done to the site to use new collections. Collection is added while tomcat is running will not be picked up: you can either restart the server, or send a configuration request to the servlet: these are described in Section 1.6.

There are two Greenstone sites that come with the distribution: localsite, and soapsite. localsite has several demo collections, while soapsite has none. soapsite specifies that a soap connection should be made to localsite. Getting this to work involves setting up a soap server for localsite: see Section 6 for details.

Each site and interface has a configuration file which specifies parameters for the site or interface—these are described in Section 1.5.

The file `$GSDL3HOME/web/WEB-INF/web.xml` contains the setup 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 three servlets specified in web.xml (these correspond to the three links in the welcome page for greenstone): 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 two are Greenstone library servlets, *library*, which serves localsite, and *library1* which serves soapsite. Both of these servlets use the standard interface (called *default*).

The initialisation parameters used by the library servlets are shown in Table 2. This is where you define what site and interface each servlet uses. Any number of servlets can be specified here. See Appendix B for more details about Tomcat.

1.5 Configuring a greenstone installation

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

configuration files; these are discussed in Section 2.3. The configuration files are read in when the system is initialised, 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 CGI-type command to the library. There are a series of CGI-type commands that can be sent to the library to induce reconfiguration of different modules, including reloading the whole site. This removes the need to shutdown and restart the system to reflect these changes. These commands are described in Section 1.6.

1.5.1 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, instead they are determined by the contents of the site's collections 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].

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 are running on the same machine. 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/soap/servlet/rpcrouter`.

1.5.2 Interface configuration file

The interface configuration file `interfaceConfig.xml` lists all the actions that the interface knows about at the start (but other ones can be loaded dynamically). If the interface uses servlets, it specifies what short name each action should use for the action CGI 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 subaction.

[1] Currently, sites live inside the Tomcat gsdl3 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 http://.

```
<siteConfig>
  <localSiteName value="org.greenstone.localsite"/>
  <httpAddress value="http://localhost:8080/gsdl3/sites/localsite"/>
  <serviceClusterList/>
  <serviceRackList/>
  <siteList/>
</siteConfig>

<siteConfig>
  <localSiteName value="org.greenstone.gsdl1"/>
  <httpAddress value="http://localhost:8080/gsdl3/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:8090/soap/servlet/rpcrouter"
      type="soap"/>
  </siteList>
</siteConfig>
```

Figure 1: Two sample site configuration files

```
<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'/>
    </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'/>
  </actionList>
</interfaceConfig>
```

Figure 2: Default interface configuration file

This makes it easy for developers to implement and use different actions and/or XSLT files without recompilation. The server must be restarted, however.

1.6 Run-time re-initialisation

should this section go in here, cos its kind of adminy, or go into the user stuff, cos you need to do it after building a collection???

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 CGI-type 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 in CGI-arguments into the URL, there is no nice web form yet to do this.

The CGI arguments are entered after the `library?` part of the URL. There are three types of commands: configure, activate, deactivate[2]. 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

[2]There is no security for these commands yet in Greenstone, so the deactivate/delete command is disabled

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`.

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 arguments in a bit more detail.

2 Using Greenstone 3

Once you have greenstone 3 installed, you can access the sample collections. The installation comes with some example collections, and Section 2.1 describes these collections and how to use them. Section 2.2 describes how to build your own 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 ...

In the standard interface that comes with Greenstone3[3], 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 'about' page. The standard page banner 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 about page. The top right has buttons to link to the library home page, help pages and preference pages. All the available services are arrayed along a navigation bar, along the bottom of the banner. Click on a name to access that service. 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 search or browse page where you accessed the document from.

describe the colls that the sample installation comes with
brief description of what a collection is.
how to get around the collection, services etc.
querying vs browsing
use the demo colls that come with greenstone - one gs2 coll, one gs3 coll, tei coll??

2.2 Building a collection

There are two ways to get a new collection into Greenstone 3. The first is to build it using the greenstone 3 building process. The second way is to import a greenstone 2 collection.

[3] of course, this is all customisable

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

The following two sections describe how to create a collection from scratch, and how to import a greenstone 2 collection. Once a collection has been built, the library server needs to be notified that there is a new collection. This can be accomplished in two ways[4]. 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, there is a CGI command to reload a collection which can also load a new one. Use the CGI arguments `a=s&sa=a&st=collection&sn=collname`—this tells the library program to reload the collname collection.

2.2.1 Creating a collection from scratch

Building Greenstone 3 collections is done using the `gs3build` script, whilst the files that control how the building is done are found inside the `etc` subdirectory of `gsdl3/web/sites/localsite/collect/[collectionname]`. There are a number of considerations in building a collection: including what documents appear in the collection, how they are indexed for searching, which classifications are used for browsing, etc. All these aspects are controlled by files within the collection's directory.

Firstly, the documents that comprise the collection should be placed in the import subdirectory. At present, only documents in this directory will appear in the collection.

The basic means of finding documents in Greenstone is search. The etc/collectionConfig.xml file controls which indexes are created to support search. By default, a collection will simply index the text of each document in the collection using the MG search engine. Alternative choices include selecting other search engines, indexing individual fields of documents (e.g. the document title) and indexing documents by section.

Search indexes appear as individual `<index>` elements within the `<search>`element of the `collectionConfig.xml` file, and classifications as individual `<classifier>` elements within the `<browse>` element. In each case, some choices are made using attributes of the element itself, and some through child elements.

Indexes can alter which search engine to use for that index, the level at which the index should be built (e.g. document, section or paragraph) and the text over which it should be built (e.g. the document text, titles alone, author names, etc.). Section-level indexes allow a reader to recall part of a document (for instance, a chapter) rather than the entire document. However, Greenstone 3 must be able to identify the internal structure of the document to achieve this. The degree to which

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

structure can be found varies from file format to file format.

Each index also must have a unique name, which is used to identify it within Greenstone The name is given as an attribute of the <index> element. The “type” indicates which search engine to use for the index. This attribute can contain either ’mg’ or ’mgpp’. If the “type” attribute is not given, the default indexer is mg.

The other choices are described using child elements of <index>. The <level> tag indicates the index level and the <field> tag the text to be used. The <level> tag can contain one of document, section or paragraph, while the <field> tag can contain “text” or the name of a metadata field. If the <level> tag is omitted, the default setting is to index by document, and if the <field> tag is omitted, the default setting is to index the document text.

Example index tags include:

To index only the title of each separate document in the collection:

```
<index name="dtt">
  <level>document</level>
  <field>dc:title</field>
  <displayItem name=’name’ lang="en">entire documents</displayItem>
  <displayItem name=’name’ lang="fr">documents entiers</displayItem>
  <displayItem name=’name’ lang="es">documentos enteros</displayItem>
</index>
```

...in this case the <field> tag refers to the “title” metadata item, found in the Dublin Core namespace. The mg search engine would be used on this index.

Alternatively, to index the full document texts by section:

```
<index name="stx" type=’’mgpp’’>
  <level>section</level>
  <displayItem name=’name’ lang="en">entire documents</displayItem>
  <displayItem name=’name’ lang="fr">documents entiers</displayItem>
  <displayItem name=’name’ lang="es">documentos enteros</displayItem>
</index>
```

...or...

```
<index name="stx" type=’’mg’’>
  <level>section</level>
  <field>text</field>
  <displayItem name=’name’ lang="en">entire documents</displayItem>
  <displayItem name=’name’ lang="fr">documents entiers</displayItem>
  <displayItem name=’name’ lang="es">documentos enteros</displayItem>
</index>
```

...in the first example, the <field> tag is not explicitly defined, and would default to ’text’, whereas it is explicitly set to ’text’ in the second example. Note the different indexer selected for these two indexes. As they are of the same name, they should not appear in the same collectionConfig.xml file.

Moving onto <classifier> items, the format is broadly similar to <index> items, but with a couple of different choices. Firstly, each classifier should have a “name” and “type” attribute as with <index> tags. In the case of <classifier>

items the “type” attribute identifies the type of classifier it is. At present, this should either be “Hierarchy” or “AZList”.

The remaining choices for the classifier should follow as child elements of the <classifier> element. The <file> element should contain the name of the file that describes the classifier as its “URL” attribute. The format of this file will be described later - it will vary from classifier type to classifier type. The <field> element identifies the name of the field to index. More than one <field> element may appear if two or more metadata fields are to be used with the classifier. Finally, the <sort> item identifies another metadata field which the items within one classifier node are to be ordered. Unlike the <index> element, the <classifier> element does not have default, assumed values for its children.

Metadata for documents can be added using metadata.xml files. These files have already been used in Greenstone 2, and the format is the same in Greenstone 3. 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, or in one of its child directories, file will be selected. The filename tag encloses the regular expression as text, eg:

```
<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 the existing value of the metadata item will be deleted.

A sample metadata.xml file can be found in the gs3test collection. However, here is an example fragment of that metadata.xml file:

```
<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>
```

Here, only one file pattern is found in the file set. However, the `Description` tag contains a number of separate metadata items. Note that the `Title` metadata does not have the accumulate metadata. This means that when the title is assigned to a document, its existing `Title` information will be lost.

Whereever possible, the Greenstone 3 will import and use options from a Greenstone 2 `collect.cfg` file. However, it is strongly recommended that a proper `collectionConfig.xml` file is used wherever possible.

To build a collection execute `gs3build.sh -collect collectionname`. The process will run, placing the new indexes in the `building` subdirectory of the collection's directory.

The building directory should be renamed to index, and a buildConfig.xml file added to it. See Section 2.4 and look at the other collections' buildConfig files for examples.

[TODO: need to describe namespaces somewhere? need to generate the buildConfig file automatically.]

2.2.2 Importing a greenstone 2 collection

Greenstone 3 can also serve Greenstone 2 collections. If you have a Greenstone 2 collection[5], you can copy it into the collect directory of the site you are using. Or make a link to it from the collect directory if your OS supports that. The Greenstone 3 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 Greenstone 2 software.

The conversion script is `convert_coll_from_gs2.pl`. To run it, you need to specify the path to the collect directory, and the collection name. For example,

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

The script attempts to create gs3 format statements from the old greenstone 2 ones. The conversion may not always work properly, so if the collection looks a bit strange under greenstone 3, you should check the format statements. Format statements are described in Section 2.5.

Once again, to have the collection recognised by the library servlet, you can either restart tomcat, or load it manually by sending the arguments `a=s&sa=c&c=collname` to the library servlet.

[5]For information about the Greenstone 2 software, and how to build collections using it, visit `www.greenstone.org`

2.3 Collection configuration files

Each collection has two, or possibly three, configuration files, `collectionConfig.xml` and `buildConfig.xml`, and optionally `collectionInit.xml` that give metadata, display and other information for the collection.[6] The first 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. The second is produced by the build-time process and includes any metadata that can be determined automatically. It also includes configuration information for any ServiceRacks needed by the collection.

2.3.1 collectionInit.xml

This optional file specifies a new collection class if the standrad one is not to be used. The only syntax so far is the class name:

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

Section 5.6 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 config files may not be needed.

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. This includes the collection metadata such as title and description, and also includes what indexes and browsing structures should be built. The format of `collectionConfig.xml` is still under consideration. However, Figure 4 shows the parts of it that have been defined so far. (Since collection building at this stage is still done using Greenstone2 Perl scripts and the old `collect.cfg` file, we have only defined the format for the parts of `collectionConfig.xml` that are used by the runtime-system.)

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

configuration files need to be encoded in utf-8.

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. If languages other than English are used, the configuration file should be encoded in utf-8. The

[6] `siteConfig.xml` and `interfaceConfig.xml` is new for Greenstone3, while `collectionConfig.xml` and `buildConfig.xml` replace `collect.cfg` and `build.cfg` in Greenstone2.

```
<collectionConfig xmlns:gsf="http://www.greenstone.org/configformat"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <metadataList>
    <metadata name="creator">greenstone@cs.waikato.ac.nz</metadata>
  </metadataList>
  <displayItemList>
    <displayItem name="smallicon" lang="en">mgppdemosm.gif</displayItem>
    <displayItem name="description" lang="fr">C'est une collection pour
      demonstration du logiciel Greenstone. Elle contient une petite
      partie du projet de bibliotheques humanitaires et de developpement
        (11 livres).</displayItem>
    <displayItem name="description" lang="en">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 name="name" lang="en">greenstone mgpp demo</displayItem>
    <displayItem name="icon" lang="en">mgppdemo.gif</displayItem>
  </displayItemList>
  <search>
    <index name="idx"/>
    <format>
      <gsf:template match="documentNode">
        <td valign='top'><gsf:link><gsf:icon/></gsf:link></td>
        <td><gsf:metadata name='Title' select='ancestors'
          separator=': '/>: <gsf:link><gsf:metadata name='Title' />
          </gsf:link></td>
      </gsf:template>
    </format>
  </search>
  <browse>
    <classifier name="CL1"/>
    <classifier name="CL2"/>
    <classifier name="CL3"/>
    <classifier name="CL4">
      <format>
        <gsf:template match="documentNode">
          <br /><gsf:link><gsf:metadata name='Keyword' />
            </gsf:link></gsf:template>
      </format>
    </classifier>
  </browse>
  <display/>
</collectionConfig>
```

Figure 4: Sample collectionConfig.xml file (mgppdemo collection)

<search> and <browse> elements give some formatting information about the indexes and classifiers. <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.

The <display> element contains optional formatting information for the display of documents. Templates that can be specified here include documentHeading, DocumentContent, and other information that could be specified (in a yet to be decided format) are things such as whether or not to display the cover image, table of contents etc.

2.4 buildConfig.xml

The file buildConfig.xml is produced by the collection building process, and contains metadata and other information about the collection that can be determined automatically, such as the number of documents it contains. It also includes a list of ServiceRack classes that are required at runtime 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 5 shows an example. This configuration file specifies that the collection should load up 3 ServiceRacks: GS2MGPPRetrieve, GS2MGPPSearch, and PhindPhraseBrowse. The contents of each <serviceRack> element are passed to the appropriate ServiceRack objects for configuration. The collectionConfig.xml file 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 changes in collectionConfig.xml to take effect in the collection without rebuilding being necessary.

2.5 Formatting the collection

format statements. and displayItem stuff. advanced collection design.

Part of collection design involves deciding how the collection should look. Greenstone 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 greenstone, 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. Some commonly overwritten 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 xmlns:gsf="http://www.greenstone.org/configformat">
  <metadataList>
    <metadata name="numDocs">11</metadata>
  </metadataList>
  <serviceRackList>
    <serviceRack name="GS2MGPPRetrieve">
      <defaultLevel name="Sec" />
      <classifierList>
<classifier name="CL1" content="Subject" />
<classifier name="CL2" content="Title" horizontalAtTop="true" />
<classifier name="CL3" content="Organization" />
<classifier name="CL4" content="Keyword" />
      </classifierList>
    </serviceRack>
    <serviceRack name="PhindPhraseBrowse" />
    <serviceRack name="GS2MGPPSearch">
      <defaultLevel name="Sec" />
      <levelList>
<level name="Doc" />
<level name="Sec" />
<level name="Para" />
      </levelList>
      <fieldList>
<field shortname="ZZ" name="allfields" />
<field shortname="TX" name="text" />
<field shortname="TI" name="Title" />
<field shortname="SU" name="Subject" />
<field shortname="ORG" name="Organization" />
<field shortname="SO" name="Source" />
      </fieldList>
      <searchTypeList>
<searchType name="plain" />
<searchType name="form" />
      </searchTypeList>
      <defaultIndex name="idx" />
      <indexList>
<index name="idx" />
      </indexList>
    </serviceRack>
  </serviceRackList>
</buildConfig>
```

Figure 5: Sample buildConfig.xml file (mgppdemo 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 and dt.

Since XSLT is written in XML, we can use XSLT to transform XML into XSLT. Greenstone provides a simplified set of formatting commands, written in XML, which will be transformed into proper XSLT. Table 4 shows the set of 'gsf' (greenstone format) elements. If you have come from a Greenstone 2 background, Appendix D shows Greenstone 2 format elements and their equivalents in Greenstone 3.

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. '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.

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 Title 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

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: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 children sections of the current section
descendents	All the descendent sections

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

This will display the dls.Title metadata if available, otherwise it will use the dc.Title metadata 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 organisation it came from.

```
<gsf:switch metadata='Organization' preprocess='toLower;stripSpace'>
  <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>
```

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).

```
<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>
  </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 6: Places for format statements

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 overwritten. 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 6 shows the positions that templates can occur.

The user specifies a `<gsf:template>` for what they want to format—these can match `documentNode` or `classifierNode` (for node in a classification hierarchy).

Table 6: Formatting options

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

The template above is now represented as:

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

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.

For local collections[7] whole XSLT files can be overridden. A collection can have a transform directory. 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 about page for the collection, you can put a new about.xsl into the collections transform directory, and this will be used instead. This is what we do for the Gutenberg sample collection.

2.6 Customising the interface

The interface can be customised in several ways. adding a new interface, adding a new language,
changing the look and feel for an interface vs a site vs a collection

what needs a tomcat restart?

2.6.1 Changing the interface language

The interface language can be changed by going to the preferences page, and choosing a language from the list. The list lists (:-)) all languages in which the interface has been defined so far.

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 resources/java. Each interface has one named interface_name.properties

[7] and eventually remote collections

(where name is the interface name). 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.

You can tell Greenstone about a new language by ... currently in interfaceConfig.

2.6.2 Modifying an existing interface

Most of an interface is defined by XSLT files, which are stored in $GSDL3HOME/web/interfaces/interface-name/transform. These can be changed and the changes will take affect 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.) 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, 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).

2.6.3 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/web/interfaces, the name of this directory becomes the interface name. Inside, it needs images and transform directories, and an interfaceConfig.xml file. Any XSLT may be overridden for a

new interface by putting the replacement in the new transform directory. If the appropriate XSLT file is not there, the one from the default interface will be used - this enables just overriding a few XSLT files as needed.

To use a new interface, the tomcat web.xml must be edited: either change the interface that a current version of the servlet 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.

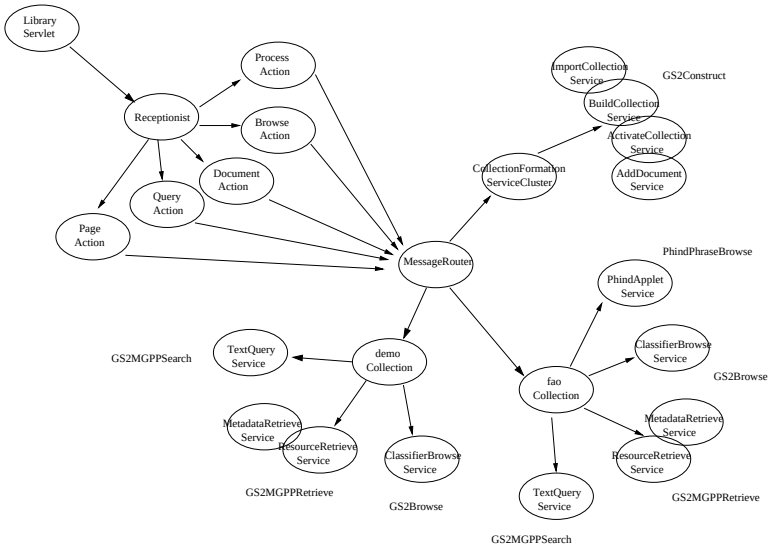

Figure 7: A simple stand-alone site.

3 Developing Greenstone 3: Run-time system

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
accessing the javadoc

3.1 Overview of modules??

A Greenstone3 'library' system consists of many components: MessageRouter, Receptionist, Actions, Collections, ServiceRacks etc. Figure 7 shows how they fit together in a stand-alone system.

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. 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 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. Services provide the core functionality for the system, e.g. searching, retrieving documents, building collections etc.

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 for example. There is a hierarchy of different Receptionist types, which is described in Section 3.8.1.

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 3.8.

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, 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`. 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 what get returned in response to a describe request (see Section 3.5.). 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-sites commands must be sent (see next section).

The MessageRouter also looks inside the site's `collect` directory, and loads up a Collection object for each valid collection found.

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

Action in Greenstone 3 is originated by a request coming in from the outside. In the standard web-based greenstone, this comes from a servlet 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 to the request. Messages inside the system 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.

When a page request comes in to the Receptionist, it looks at the action attribute to determine which action to send it to. The response is returned from the action.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 servlet greenstone we transform using XSLT to generate html pages which get returned to the servlet.

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 request types: describe, process, system...

The message formats for each request type, and the response formats for each module are described in the following section.

3.4 an attempt at an API: message formats

3.4.1 external− >action

request: These are the special 'external'-style messages. Requests originate from outside Greenstone, for example from a servlet, or java application. They 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'/>
    <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 -->
```

[8]In a servlet context, these correspond to the URLs `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`.

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 7: Generic arguments that can appear in a Greenstone URL

```
    <param name='q' value='snail'/> <!-- query string -->
  </paramList>
</request>
```

The Receptionist routes the message to the appropriate Action (determined by looking up its shortname− >Action object map). The actions determine what information is needed from the server and retrieves it, making one or more internal requests to the MessageRouter. This information is gathered together into a single response, and returned to the Receptionist. The Receptionist may process the result further, depending on what type of Receptionist is it.

3.5 '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/soap/servlet/rpcrouter'
            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>
```

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/gsdl3/sites/
                                localsite/collect/mgppdemo</metadata>
    </metadataList>
  </collection>
</response>
```

The subset parameter can also be used in a describe request to a collection, to retrieve just the `metadataList` or `serviceList`.

This collection provides many typical services. Notice how this response lists the services available, while the collection configuration file for this collection (Figure 4) 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, some display information, (and in future may describe the content type for the request and response).

Parameters can by 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 8 gives 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">
      <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>
```

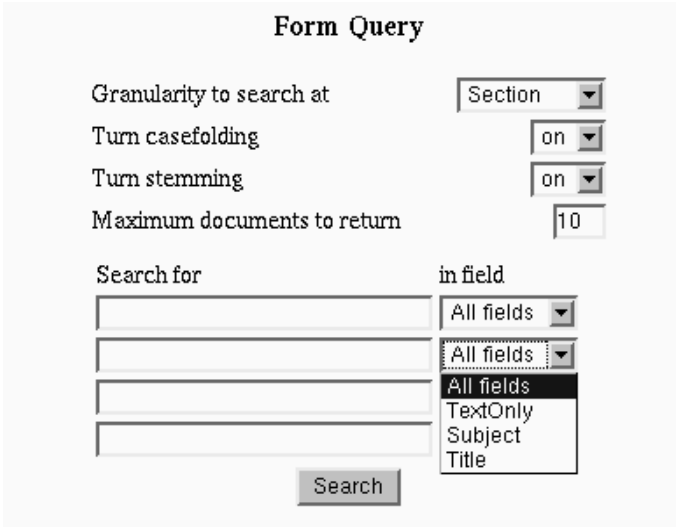

Figure 8: The previous query service describe response as displayed on the search page.

```
      </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.

3.5.1 '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 they are initiated by particular CGI parameters (see Section 1.6).

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, for example:

```
<response from="">
  <status>collectionList reconfigured successfully</status>
</response>
```

At some stage, an error or status code should be included.

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 2.5. They are templates written directly in XSLT, or in GSF, which stands for Greenstone Format, and 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 3.7.5). The initial response states whether the process had successfully started, and whether its still continuing. If the process is not finished, 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 8[9].

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">
```

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

Table 8: Status codes currently used in Greenstone 3

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 code="2" pid="2">Collection construction: import collection.
command = import.pl -collectdir /research/kjdon/home/gsdl3/web/sites/
    localsite/collect test1
starting
  </status>
</response>

<response from="build/ImportCollection">
  <status code="11" pid="2">RecPlug: getting directory
/research/kjdon/home/gsdl3/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/gsdl3/web/sites/
    localsite/collect/test1/etc/fail.log for a list of rejected documents
Success
  </status>
</response>
```

3.7.1 process 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.

3.7.2 '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.[10]

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>
```

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

```
    <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>
```

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).

3.7.3 '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`, `descendents`. 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.

3.7.4 '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 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`, `descendents`. Possible values for info parameters are `numSiblings`, `siblingPosition`, `numChildren`.

3.7.5 '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 8 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 are described in Section 3.7, are used to find out how the process is going, and whether it has finished or not.

3.7.6 '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>
```

3.7.7 '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. *** TODO ****

```
<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>
```

3.8 Page generation

**** REDO ********

* talk general first: get data, get format info, transform gsf-¿xsl. transfrom xml-¿html

* state saving. the XSLT files assume that arguments are saved somehow. This needs to be implemented outside Greenstone proper - we do this in the servlet, using something or other.

URL-style requests are received by the Receptionist. Based on the arguments, a page of data must be returned to the servlet. As described in Section 3.4.1, 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 CGI-arguments to determine what requests need to be made to the system. System requests are received by the Message-

Router, which answers them one by one, either itself or by passing them on to the appropriate module.

Once the data needed from the system has been accumulated, it is put into a 'page' of XML. The page is transformed to its output form, currently HTML, via XSLT transformations, and returned to the user.

The basic page format is:

```
<page>
  <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, 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.[11]
TODO describe a bit more?? currently only can get this locally

3.8.1 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

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

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 greenstone 3 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.

NZDLReceptionist: (do we want to talk about this?) This is an example of a custom receptionist. For a look-alike nzdl.org system, even more information is needed for each page, namely the list of classifiers available from the ClassifierBrowse service.

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

3.8.2 CGI arguments

The arguments used by the page come from several sources. Receptionist uses a couple, actions use some and services. the receptionist and actions are treated as a whole so must not have conflicting arguments. GSParams class specifies all the general basic arguments, and whether they should be saved or not. servlet has an init parameter params_class, that specifies which params class to use - if subclass it. actions or receptionist may specify some new ones

services may be created by different people, may be on a different site. cant guarantee no conflict with action params, or even with other services. so service

params are namespaced when they are put on the page. interface (recept and action) params will have no namespace) the default namespace is s1 (service1) - any parameters that are for the service will be prefixed by this. e.g. the case parameter for a search will be put in the page as s1.case. The actions must now look for all the s1 parameters to send to the service.

if there are two or more services combined on a page with a single submit button, they will use s1, s2, s3 etc as needed. the s parameter (service) will end up with a list e.g. s=TextQuery,MusicQuery, and the order of these determines the mapping order of the namespaces, ie s1 will be TextQuery, s2 MusicQuery.

also talk about saving arguments - save ones that GSParams says to save, and any service ones should always save.

3.8.3 Page action

* kind of info pages. other actions are associated with specific services. * uses describe requests to modules Depending on the subaction argument, different pages can be generated. 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. The page is transformed using `home.xsl`. 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, and the result is transformed using `about.xsl`.

3.8.4 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 (Formatter object??). The service description and query result are combined into a page of XML, which is transformed using `basicquery.xsl` to produce the html page.

3.8.5 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. The result is returned directly to the applet code, in XML. The other parameters are sent to the service untransformed, and the result is passed directly back to the applet. Applet action can therefore work with any applet whose service understands the messages.

Here are two examples of requests generated by the Applet action, along with their corresponding responses.

The first request corresponds to the URL arguments `a=a & rt=d & sn=Phind & c=mgppdemo`, which translate to "display the Phind applet for the mgppdemo collection".

The second request corresponds to the arguments `a=a & rt=r & sn=Phind & c=mgppdemo & pc=1 & pptext=health & pfe=0 & ple=10 & pfd=0 & pld=10 & pfl=0 & pll=10`—this indicates a request to the service itself. The extra arguments (not a, sa, sn, c) are simply copied into the request as parameters. The response is in a form suitable for the applet, placed inside `<appletData>` in a standard Greenstone message. AppletAction returns the contents of appletData to the browser, i.e. to the applet itself.

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 Applet action encounters this parameter it inserts the name of the current library servlet as its value.

3.8.6 Document action

DocumentAction sends a query to the DocumentRetrieve service of the collection requesting the text of the specified document. At this stage no additional information is obtained, but in future stuff like Title and table of contents would be needed to make the display nicer.

3.8.7 System action

SystemAction allows for manual reconfiguration of various components at runtime. 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.

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	

3.8.8 Some class info - where should this go??

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

Utility class	Description
ConfigVars	holds the servlet startup variables, including library name, site name, interface name, default language
Dictionary	wrapper around a Resource Bundle, providing strings with parameter
GSCGI	class to map between short name CGI arguments and long name request parameters
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
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
Misc	miscellaneous functions
OID	class to handle Greenstone (2) 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 Collection building architecture

**** GEORGE **** how building actually works
the building structure/architecture
modules API

5 Developing Greenstone 3: Adding new features

5.1 Creating new services

*inherit from ServiceRack - abstract base class. this handles the main process method, determines the service name and request type. if request type is describe, and to is empty, it returns a list of services (short_service_info) which is initialised in the configure method. a describe request to a particular service results in getServiceDescription being called, which must be supplied by the subclass. other request types (process) get sent to processXXX methods, where XXX is the service name.

* what methods are expected

*service type responses expected

*a browse type service must also implement servicenameMetadataRetrieve service.

* should a metadata retrieval service advertise what metadata is available??

standard service type vs new service type - standrad needs some xml response syntax.

5.2 creating new actions/pages

5.3 new interfaces

e.g. java interface. where you can interface to. MR vs Receptionist. diff receptionists. egs, 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.

5.4 Adding new classifiers

*** GEORGE ***

5.5 Adding new plugins

*** GEORGE ***

5.6 New types of collections

There are two types of standard Greenstone collections: collections built with the Greenstone 3 building system, and collections that are imported from Greenstone 2. There are many options to collection building but it is conceivable that these options don't meet the needs of all collection builders. Greenstone 3 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 customisation 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, the Greenstone 2 MGPP collections were the first to be served in Greenstone 3. When we came to do Greenstone 2 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 nzmaps collection used the same level of customisation, just implementing new services and fitting all the extra display elements into the standard query/display framework using javascript.

The 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[12].

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.

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

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

```
<action name='xd' class='XMLDocumentAction'>
  <subaction name='toc' xslt='document-toc.xsl'/>
  <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 may all the links absolute links to files in the collection folder, the other option is to put them in Greenstone's DTD folder gsdl3/resources/dtd.

5.7 The NZDL mirror site

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 Greenstone 3 instead of Greenstone 2. It uses a new site and a new interface. The web.xml file had a new servlet entry in it to specify the combination of nzdl site and interface.

The site was created by making a directory called nzdl in the sites folder. A siteConfig file was created. Because its 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.

A new interface, also called nzdl, was created in the interfaces directory. 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 2.6). This setup also required a bit more customisation.

The standard Greenstone navigation bar lists all the services available for the collection. In Greenstone 2, the navigation bar provided the search option, and the different classifiers. This is not service specific, but hard coded to the search and classifiers. The XSLT that produced the navigation bar needed to be altered to produce this. But also, a new Receptionist was needed. The standard receptionist (DefaultReceptionist) gathers a little bit of extra info 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 needed a bit more information than this: for the ClassifierBrowse service, if there was one, the list of classifiers and their display elements must be obtained. So a new Receptionist was written that inherited from DefaultReceptionist, and added this new info into the page.

One of the servlet initialisation parameters is the receptionist class: this was added to the servlet definition in the web.xml file so that the LibraryServlet would load up the right receptionist class.

6 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, however we have only implemented a simple SOAP protocol.

more explanation..

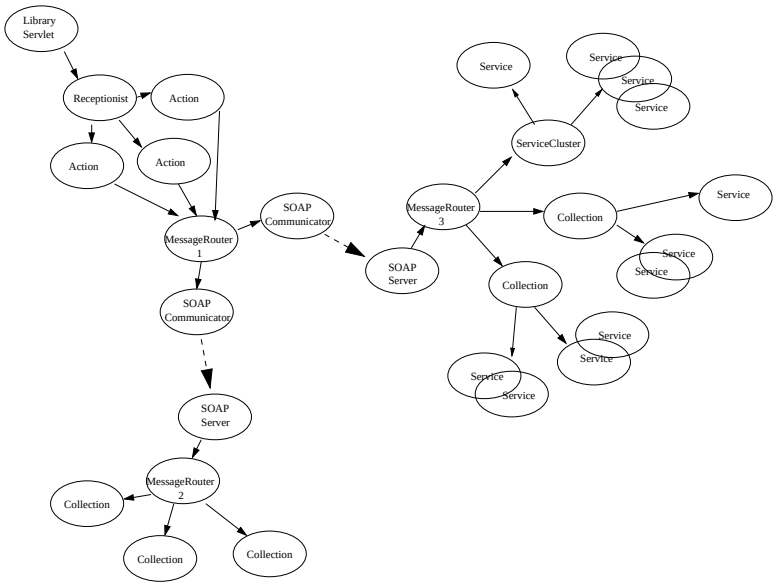

Figure 9: A distributed digital library configuration running over several servers

We have used Apache SOAP for Java. This is run as a servlet in Tomcat. If you have obtained Greenstone through CVS, you will need to install soap separately, describe in Appendix C.1. Debugging soap is described in Appendix C.2.

6.1 Serving a site using soap

what do we have to do?? resource file format, deploy the service etc.

A Using Greenstone 3 from CVS

*** need to make sure building stuff is in here ***

Greenstone 3 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.

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 gsdl3
```

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.

The software needs to be compiled and installed. The installation procedure has been semi-automated. The following sections describe installation under Linux and windows.

A.1 Linux install

An install.sh script is provided to compile and install Greenstone3. What you need to do is:

```
cd gsdl3
source gs3-setup.sh
./gs3-prepare.sh
./gs3-configure.sh
./gs3-compile.sh
./gs3-finalise.sh
```

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

To startup or shutdown the library (includes the Tomcat server and MYSQL server), the commands are:

```
./gs3-launch.sh ./gs3-launch.sh -shutdown
```

A.2 Windows install

Make sure that the following environment variables are set: JAVA_HOME (where the JAva 2 SDK is installed); PATH (should include the CVS program, and %JAVA_HOME%\bin). The following commands should be run in a DOS prompt.

Run gs3-setup.bat
Run gs3-prepare.bat
Using Wordpad, edit the gsdl3\comms\jakarta\tomcat\conf\server.xml file. Add

```
<!-- GSDL3 Service -->
<Context path='/gsdl3' docBase='@gsdl3home@\web' debug='1'
reloadable='true'/>
```

above the line: `<!-- Tomcat Root Context -->`

Still using Wordpad, edit the gsdl3\comms\jakarta\tomcat\conf\web.xml file. set the value of the 'listings' parameter to false.

Run make.bat
Run make.bat install.
Run gs3-finalise.bat

To run Greenstone, run gs3-launch.bat. This will start the Tomcat server in a new DOS window (stop it by closing the window), and open a broser window showing the Greenstone 3 homepage.

B Tomcat

Tomcat is a servlet container. It is used to serve a Greenstone site using a servlet.

The file `$GSDL3HOME/comms/jakarta/tomcat/conf/server.xml` is the Tomcat configuration file. The installation process adds a context for Greenstone3 servlets (`$GSDL3HOME/web`)—this tells Tomcat where to find the web.xml file, and what URL (`/gsdl3`) to give it. Anything inside the context directory is accessible via Tomcat[13]. For example, the index.html file that lives in `$GSDL3HOME/web` can be accessed through the URL `localhost:8080/gsdl3/index.html`. The demo collection's images can be accessed through `localhost:8080/gsdl3/sites/localsite/collect/demo/images/`.

Tomcat runs by default on port 8080—this can be changed in server.xml, in the `<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 --><Connector>` element. The siteConfig files also need changing if Tomcat's port is changed: `<httpAddress>` for the site, and `<address>` for a remote site both use this.

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

- `$GSDL3HOME/web/WEB-INF/web.xml`
- `$GSDL3HOME/comms/jakarta/tomcat-tomcat-4.0.1/conf/server.xml`
- any classes or jar files used by the servlets

Note: stdin and stdout for the servlets both go to `$GSDL3HOME/comms/jakarta/tomcat/logs/catalina.out`

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 1.6), or by restarting Tomcat.

We have set up Tomcat to follow symlinks. To disable this feature, remove the `<Resources>` element from the gsdl3 context in `$GSDL3HOME/comms/jakarta/tomcat/conf/server.xml`:

```
<Context path="/gsdl3" docBase="$GSDL3HOME/web" debug="1"
reloadable="true">
<Resources allowLinking='true'/>
</Context>
```

We have set up tomcat to disallow directory listings for everything in the docBase directory. To turn this back on, you need to edit Tomcat's default web.xml file ($GSDL3HOME/comms/jakarta/tomcat/conf/web.xml):

In the default servlet definition, change the 'listings' parameter to true.

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

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 `$GSDL3HOME/comms/jakarta/tomcat/conf/server.xml`. For the default manager, session information is stored in the work directory: `$GSDL3HOME/comms/jakarta/tom` Delete this file to clear the cached session info.

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/gsdl3
ProxyPassReverse /greenstone3 http://puka.cs.waikato.ac.nz:8080/gsdl3
</VirtualHost>
```

In our example, the greenstone 3 servlet can be accessed at `http://www.greenstone.org/greenstone3/` instead of at `http://puka.cs.waikato.ac.nz:8080/gsdl3/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

C.1 Setting up SOAP from CVS

If you have obtained greenstone through CVS, you will need to install the SOAP stuff by running:

```
install-soap.bash
```

This unpacks the soap distribution, adds a SOAP context to Tomcat's server.xml configuration file, and creates the file `src/java/org/greenstone/gsdl3/SOAPServer.java` from `src/java/org/greenstone/gsdl3/SOAPServer.java.in` (it has a place where gsdl3home needs to be added). It also tries to deploy the SOAP service, but this often doesn't work. You may need to run from a shell the following command:

```
java org.apache.soap.server.ServiceManagerClient
  http://localhost:8080/soap/servlet/rpcrouter deploy
  resources/soap/localsite.xml
```

You can also deploy a service through the website. If Tomcat is not running, start it up (see **??**).

The SOAP servlet can be accessed at `http://localhost:8080/soap`. You should see a welcome page. Click on "Run the admin client". This enables you to list, deploy and undeploy SOAP services.

To deploy the SOAPServer for localsite:

Click on "deploy" and edit the following fields in the deploy form:

ID:	org.greenstone.localsite
Scope: (choose Session or Application)	Request—new instantiation for each request Session—same instantiation across a session Application—only uses one instantiation
Methods:	process
Java Provider / Provider Class:	org.greenstone.gsdl3.SOAPServer

Now click the "deploy" button at the bottom of the page. If the service has been deployed, it should appear when you click on the left hand "List" button.

Information about deployed services is maintained between Tomcat sessions—you only need to deploy it once. To get the library1 servlet talking to the SOAP server, you need to shutdown and restart Tomcat (see **??**). You should see more collections when you run the library1 servlet.

C.2 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, use a program called TcpTunnelGui. This intercepts messages coming in to one port, displays them, and passes them to another port. To run it, type:

```
java org.apache.soap.util.net.TcpTunnelGui 8070 localhost
8080
```

8070 is the port that TcpTunnelGui listens on, and 8080 is the port that it sends the messages onto—the port that Tomcat is using. You need to modify Greenstone to talk to port 8070 when it wants to talk to Tomcat, so that the messages go through TcpTunnelGui. This is specified in the `<site>` element of the soapsite site configuration file (`$GSDL3HOME/web/sites/soapsite/siteConfig.xml`).

```
<site name="org.greenstone.localsite"
      address="http://localhost:8080/soap/servlet/rpcrouter"
      type="soap"/>
```

Note that `http://localhost:8080/soap/servlet/rpcrouter` is the address for talking to the Tomcat SOAP servlet services.

Table 11: Greenstone 3 equivalents of Greenstone 2 format statements

Greenstone 2	Greenstone 3
[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 Format statements: Greenstone 2 vs Greenstone 3

The following table shows the Greenstone 2 format elements, and their equivalents in Greenstone 3