XBOOKS

Developer’s Guide

pdf [1], ps [2]

Extreme! Computing Lab
Indiana University

December 31, 2002
## Contents

1. **Introduction** ................................................................. 3
   1.1 Document organization .................................................. 5

2. **Server-side** ........................................................................ 6
   2.1 xbook server ................................................................. 6
      2.1.1 Specification .......................................................... 6
      2.1.2 Differences from XCAT-SP .......................................... 6
      2.1.3 Implementation ....................................................... 6
         2.1.3.1 To be changed .................................................. 6
         2.1.3.2 To be completed ............................................... 7
   2.2 xbook manager .............................................................. 7
      2.2.1 Specification .......................................................... 7
      2.2.2 Differences from XCAT-SP .......................................... 7
      2.2.3 Implementation ....................................................... 7
         2.2.3.1 To be changed .................................................. 8
         2.2.3.2 To be completed ............................................... 8
      2.2.4 Dispatcher .............................................................. 8
         2.2.4.1 Specification .................................................... 8
         2.2.4.2 Differences from XCAT-SP ...................................... 8
         2.2.4.3 Implementation .................................................. 8
            2.2.4.3.1 To be changed ............................................ 8
            2.2.4.3.2 To be completed ........................................ 8
      2.2.5 Registry ................................................................. 8
         2.2.5.1 Specification .................................................... 8
         2.2.5.2 Differences from XCAT-SP ...................................... 8
         2.2.5.3 Implementation .................................................. 8
            2.2.5.3.1 To be changed ............................................ 9
            2.2.5.3.2 To be completed ........................................ 9
      2.2.6 Engine ..................................................................... 9
         2.2.6.1 Specification .................................................... 9
         2.2.6.2 Differences from XCAT-SP ...................................... 10
         2.2.6.3 Implementation .................................................. 10
         2.2.6.4 To be changed .................................................. 10
         2.2.6.5 To be completed ............................................... 10
         2.2.6.6 Jython Engine .................................................... 10
            2.2.6.6.1 Specification .............................................. 10
            2.2.6.6.2 Differences from XCAT-SP ................................ 10
            2.2.6.6.3 Implementation .......................................... 10
               2.2.6.6.3.1 To be changed ....................................... 11
               2.2.6.6.3.2 To be completed ................................... 11
      2.3 xbook repository .......................................................... 11
         2.3.0.1 Specification ..................................................... 11

1
2.3.0.2 Implementation .................................................. 11
    2.3.0.2.1 To be changed ............................................. 12
    2.3.0.2.2 To be completed .......................................... 12

3 Client-side ......................................................... 13
    3.1 Introduction ................................................... 13
    3.2 Portlet .......................................................... 13
        3.2.1 Specification ............................................. 13
        3.2.2 Differences from XCAT-SP ............................. 13
        3.2.3 Implementation ......................................... 13
            3.2.3.1 Viewing subxbooks ............................... 15
            3.2.3.2 To be changed .................................... 16
            3.2.3.3 To be completed .................................. 16
    3.3 Command-line .................................................. 17
        3.3.1 Specification ............................................. 17
        3.3.2 Differences from XCAT-SP ............................. 17
        3.3.3 Implementation ......................................... 17
            3.3.3.1 To be changed .................................... 17
            3.3.3.1.1 Input variables ............................... 17
            3.3.3.1.2 Methods ....................................... 17
            3.3.3.2 To be completed .................................. 17

4 Directory Service .................................................. 18
    4.1 Specification .................................................. 18
    4.2 Differences from XCAT-SP ................................. 18
    4.3 Implementation ............................................. 18
    4.4 To be changed .............................................. 19
    4.5 To be completed ............................................ 19

5 Writing xbooks .................................................... 20
    5.1 Xbook scripts ................................................ 20
        5.1.1 Jython scripts ........................................ 20
            5.1.1.1 Modules ......................................... 21
        5.1.2 Output Restrictions and Requirements ............ 22
    5.2 Properties .................................................... 23
    5.3 Registry ....................................................... 23
    5.4 Running the xbook manager .............................. 24
    5.5 Portlet client ............................................... 24
    5.6 Command-line client ....................................... 24
    5.7 Examples ..................................................... 24
        5.7.1 Form .................................................... 25
        5.7.2 Gram .................................................... 25

6 Case Study: Grappa ................................................. 26

7 Future Directions .................................................. 28
    7.1 HTML Server .................................................. 28
    7.2 Resource Management ....................................... 28
    7.3 Virtual Data .................................................. 28

Bibliography .......................................................... 29
Chapter 1

Introduction

The Computational Grid [3] (or Grid for short) promises to be a powerful platform for computationally-intensive and data-intensive applications. Such applications come from biology [4], chemistry [5], and physics [6] communities or virtual organizations [7]. Grid-enabling these applications oftentimes becomes a herculean effort, utilizing several Grid software packages [7, 8, 9, 10, 11, 12] to leverage the power and achieve performance on the Grid. Furthermore, these applications are already fairly large and complex with hundreds to thousands of input parameters; Grid-enabling these applications only increases this size and complexity due to additional setup/configuration procedures. This is a challenge for Grid application developers because they often have to expend significant additional programming effort in order to make their applications “user-friendly” for other members of their community. Oftentimes, this amounts to achieving a fine balance of constructing an user interface that reveals enough detail so that users can see the Grid “at work” but not too much detail that users get discouraged and walk away.

The three main services that an user interface provides to end-users to facilitate running their Grid-enabled applications on a particular set of resources are launching, monitoring, and archiving. The steps involved in launching an application are 1) collect all of the data/information needed to execute an application, 2) format the data/information into a request that be sent to Grid resources and/or a scheduler, and 3) send the request(s). Monitoring the launched Grid application or job involves checking generic job status information such as what resources are being used and how the job is distributed among them. Monitoring information can also be application-specific (e.g., summary graphs) to relate more intuitive progress information and can also relate quality of the job results. Finally, archiving involves storing a record of the job and its results so that it can be discovered by other collaborators.

In practice, user interfaces that perform the above functions are most often implemented as web portals or Grid portals. The advantage of Grid portals is that only the server running the Grid portal needs to have access to Grid tools and services (which are often not easy to deploy). Users of the Grid portal can then access the portal through any machine with just a web browser.

There are numerous Grid portal products currently available varying in their functionality and focus. However, most Grid portal products do provide a common set of core of services (e.g., credential management, file transferring, and resource management). While these portals may utilize the same underlying Grid APIs [13], very few share the same code which implements the interfaces inside their portal. One reason is that traditional web portal building mechanisms are fairly low-level which seems to foster monolithic designs that make it difficult to take a component from one portal and plug it into another. However, recent developments of commodity web portal frameworks aim to change this.

A web portal framework provides a generic set of high-level portal services such as account management and customization of the appearance and layout of a portal (even on a per user basis). The functionality of portal is determined by portal plug-ins called portlets. A portal provider will install a number of relevant portlets for a community and those users may load any number of portlets into their account. Each portlet is loaded into a pane and the user can arrange and nest the panes into a layout which best serves them. Existing implementations of web portal frameworks which conform to a portlet API include Jakarta Jetspeed [17] and IBM Websphere [18].

This emergence of web portal frameworks is causing a paradigm shift in Grid portal development. Programmers can now decompose their portals into portlets and share the generic components with other portal projects thus minimizing code replication. The xportlets project [19] is one such effort. However, programmers still need to construct the application-specific aspect of their Grid portal.
Traditionally, Grid portals which are customized for a particular application are referred to as science portals. The goal for a science portal is to provide end-users (i.e., application scientists) with access to an intuitive interface for executing their applications on the Grid. In previous work [20], we found that active notebooks were an effective abstraction for providing the application-specific part of a science portal. Simply put, an active notebook contains HTML forms which describe an application-specific interface and scripts to process the forms when submitted by the user. These scripts are Jython scripts and implement the application-specific aspects of launching, monitoring, and archiving jobs. These active notebook executes inside a portal framework called the XCAT Science Portal [20] which performs notebook archiving make it possible to re-execute or run similar jobs. However, the XCAT Science Portal framework suffers from the same monolithic portal design as noted earlier. Therefore, the goal of this work is to redesign the active notebook concept using commodity web portal framework technology herein referred to as a xbook.

A xbook is a self-contained application-specific user interface which works in conjunction with a xbook portlet to plug itself into a science portal. A xbook is served by a xbook server and may be discovered through a xbook directory service.

The following steps detail how a user would interact with the xbook portal architecture and is illustrated in Figure 1.1:

1. A user logs into a science portal where they have loaded a xbook portlet. The user contacts a xbook directory service to search for particular xbook of interest.
2. User finds a particular xbook called xbook-1 they want to use. The user retrieves a reference to the xbook server that is hosting xbook-1.
3. User contacts the xbook server via the xbook portlet and requests xbook-1.
4. The xbook server launches a xbook manager to service the user’s request.
5. The xbook manager sends the xbook portlet a HTML form. The user fills out the form and presses the submit button. The xbook portlet packages up the form values and send them to the xbook manager. The xbook engine invokes the xbook handler with the form values. This process can repeat until the xbook has enough information to launch the application.

Figure 1.1: xbook architecture.
6. The xbook manager launches the application. The values used to launch the application are archived so the user can come back later to review the results or optionally launch a similar job (see Section 2.3).

7. While the xbook is active, the user can interact and query the xbook instance for basic information about application status and any more complex application-specific information as defined by the xbook.

The rest of the document describes this xbook architecture.

1.1 Document organization

In the following sections, we will describe each component of the xbook architecture in full detail. Sections 2, 3, and 4 are structured into the following parts:

**Specification:** This section will describe the desired functionality of the component and how it will interact with other xbook-related components.

**Differences from the XCAT-SP:** For backwards references purposes, we will describe how the proposed architecture differs from the XCAT-SP architecture.

**Implementation:** We have implemented a partial prototype of this architecture. This section will describe how much of the specification is implemented and reference specific code files.

**To be changed:** I’ve made changes to the architecture specification since the initial prototype implementation. This section will specifically detail what needs to be changed in order to comply to the architecture specification.

**To be completed:** This section will detail what needs to be added to the current prototype in order to make it fully compliant with the architecture specification.

Then in Section 5, we describe how to write xbooks. Finally, we conclude in Section 7 with some future directions.
Chapter 2

Server-side

2.1  xbook server

2.1.1  Specification

An xbook server is an application-specific application factory web service [21] which serves one or more xbooks. It launches a xbook manager for each portal request it receives to handle the request. The xbook server will run in one of the following modes:

web service: All processes run under the account of the user who is running the xbook server. This is the currently supported mode of the AFWS.

gatekeeper: Application processes are run under the account associated with the proxy passed from portal.

2.1.2  Differences from XCAT-SP

The xbook architecture differs from the XCAT-SP in that it utilizes a commodity web portal framework; this gives it a modular design and also user management (i.e., a multi-user portal). Furthermore the xbook management is handled completely outside of the portal. One advantage of our this is that the client is not constrained to be a portal request only; for example, we can now also provide command-line clients (see Section 3.3.1).

2.1.3  Implementation

The xbook server is not currently integrated into the prototype. When integrated, one thing to watch for is startup time. In the current prototype, we use the xbook manager as the xbook server because of time limitations. Thus in the current prototype, a xbook server is not launched every time a user request is received. When the prototype is changed, we need to make sure the launch time via GRAM is reasonable for a GUI response time.

2.1.3.1 To be changed

One thing that will need to be addressed for any long-term running application factory is credential expiration. Here are a couple options:

- generate a long-term proxy credential
- generate a long-term proxy credential and store it in the myproxy server. Store the myproxy username and password in a file and use that to renew the certificate whenever the cached proxy expires.
- generate a separate portal certificate and a passphrase-less key. Whenever the proxy credential expires, just refresh the proxy.
2.1.3.2 To be completed
Use Gopi’s application factory portlet so that we can launch a xbook manager from the portal.

2.2 xbook manager

2.2.1 Specification
The architecture of the xbook manager is shown in Figure 2.1. The servicing of a xbook request is decomposed into the following steps:

1. The xbook manager receives the forwarded request from the xbook server and dispatches it to an engine which can appropriately handle the request. In Figure 2.1, this is represented by a dispatcher entity.
2. The dispatcher looks up the xbook in its registry to find out its type.
3. The type of the xbook determines which xbook engine will handle the request.
4. The dispatcher passes the request to the appropriate xbook engine (in this case engine-1).
5. engine-1 locates the xbook in the xbook repository and handles the request.
6. The request results in the launch of an application.

![Figure 2.1: xbook server architecture](image)

2.2.2 Differences from XCAT-SP
In the XCAT-SP, the request is handled by servlet which invokes the jython interpreter to handle the request.

2.2.3 Implementation
The xbook manager code is in the package `xbooks.server.manager`. It is a XCAT component and it's code is divided into 3 files:

- `XbookManagerService.java` The interface to describe the services the xbook manager provides.
- `XbookManagerPortInterface.java` The interface to describe the xbook manager port.
- `XbookManagerPort.java` The port implementation.
2.2.3.1 To be changed
Nothing comes to mind.

2.2.3.2 To be completed
Nothing comes to mind.

2.2.4 Dispatcher

2.2.4.1 Specification
The dispatcher is the front-end of the xbook server. It takes an user request for a xbook and looks it up in the registry. Using the registry entry, the dispatcher reads the xbook properties file (located using the xbook path) and reads in the map file for that xbook (i.e., who is authorized to view it). See Section 5.2 for more information about xbook property files. If the requester’s DN is not in the map file, the request is denied. Otherwise, the request is dispatched to the appropriate xbook engine which will then handle the request.

2.2.4.2 Differences from XCAT-SP
In the XCAT-SP, there was only one supported "engine" (which in the new framework is called JythonEngine). Thus, there was no need for a dispatcher.

2.2.4.3 Implementation
This is the startUp method of XbookManagerPort. It references the registry (see Section 2.2.5) to find the name of the xbook engine which can handle the xbook request. The engine must be a subclass of Engine and is assumed to be in the same package as Engine, under xbooks.server.engine (we use the newInstance() method from the Class object to create a new engine instance; see the XbookManagerPort.java file).

2.2.4.3.1 To be changed Nothing comes to mind.

2.2.4.3.2 To be completed Currently, we don’t check user credentials. Once the XCAT and XSOAP libraries are updated to Alek’s latest (see Section 3.2.3.3 for more information), we can get the the delegated proxy credential and check it against the map file (see Section 2.3.0.2). The map file contains user dn patterns so we need to use a regular expression library to perform the pattern matching. If they don’t match, we should raise an AuthorizationException (this exception needs to be written too).

2.2.5 Registry

2.2.5.1 Specification
A registry is a table (i.e., file) which maps the name of the xbook to its type. The type tells the dispatcher which xbook engine to use.

2.2.5.2 Differences from XCAT-SP
This is the similar to the XCAT-SP registry which was a XML file that mapped notebook names to their relative paths.

2.2.5.3 Implementation
Currently, this is Java properties file (i.e., read in using PropertyFile) with the following format:

   <xbook path prefix>: <engine type> <engine args>
The <xbook path pre£x> is a relative path to the location of the directory containing the xbook. It is relative to the root of the xbook repository which is read in from a con£guration £le upon xbook server startup. The <engine type> is the name of the engine class which will be instantiated to handle the request (see Section 2.2.4.3. The <engine args> is engine-speci£c. For example, the JythonEngine has 2 arguments: the python handler script name and the python class name (see Section 2.2.6.6).

2.2.5.3.1 To be changed This should be changed to a XML £le since this is will need to be published to the xbook directory service and should readable by other entities (i.e., PropertiesFile is speci£c to java).

2.2.5.3.2 To be completed Just the change noted above.

2.2.6 Engine

2.2.6.1 Speci£cation

A xbook engine is the entity that the xbook server will invoke to execute a particular type of xbook. In other words, it acts as an intermediary between the portlet client and the xbook. An engine can de£ne its own speci£cation for a xbook but must provide the following interface:

initialize(xbookPath, engineArgs) This routine is called when the engine is £rst instantiated for a particular xbook (i.e., is the engine constructor). The user should stick any engine setup code here (e.g., initializing a interpreter). the xbookPath is the location of the xbook and engineArgs are any con£guration information that is read in from the registry.

start(out) This routine will also be called after the engine is instantiated (i.e., after initialize). This will call the xbook initialization method. Any output which the xbook wants displayed to the user should be written to the out stream. If the user expects user input, html input tags should be written to out to collect them.

stop(out) This routine will be called with the engine is shutdown. Any cleanup code should be put here (i.e., shutting down an interpreter). Any output which the xbook wants displayed to the user should be written to the out stream.

execute(command, formValues, out) This routine will be called when the user presses a submit button from the portlet and can be called multiple times. The name of the submit button and any form parameters are sent as input. The engine will then call the appropriate xbook method to handle the input. Any output which the xbook wants displayed to the user should be written to the out stream. If the user expects further user input, html input tags should be written to out to collect them.

Figure 2.2 illustrates a sample xbook session with one xbook method being called.

![Diagram of xbook session](image-url)
The engine itself must be written in Java but the xbook itself can be written in any desired scripting language. In our first implementation, we target jython but engines can also be written to handle xbooks in python, csh, perl, etc.

A xbook session is considered active when the xbook sends a keep_alive hidden input form parameter in the out stream. During an active xbook session, the engine is responsible for maintaining state between a multi-stage request (i.e., calling execute multiple times). In the jython engine, this translates to executing the jython scripts under the same interpreter.

The jython engine is further described in Section 2.2.6.6.

2.2.6.2 Differences from XCAT-SP

In the XCAT-SP, there was one engine which was implemented as a servlet that ran a jython interpreter. The code was contained in the src/java/servlets/an/scripting directory. The handler contained only one function, main, which was invoked when the user pressed the submit button. xbooks differs from this model by allowing multi-stage requests (i.e., the output can be another form). Many Grid applications have hundreds to thousands input parameters so this is an important feature that allows users to solicit user input parameters in stages so as not to overwhelm their users. Another advantage is that this model supports monitoring actions (i.e., the output can be a form which allows users to request job status information).

2.2.6.3 Implementation

The engine code is contained in the package xbooks.server.engine. All engines should be a sub-class of Engine which supports the interface described in the specification. The jython engine is contained in JythonEngine.java.

2.2.6.4 To be changed

Nothing comes to mind.

2.2.6.5 To be completed

Nothing comes to mind.

2.2.6.6 Jython Engine

2.2.6.6.1 Specification

A jython engine is a sub-class of engine that provides an implementation for handling xbooks written in jython [22].

2.2.6.6.2 Differences from XCAT-SP

See Section 2.2.6.2.

2.2.6.6.3 Implementation

The jython engine code is defined in the xbooks.server.engine package in JythonEngine.java. It implements the Engine interface as follows:

initialize(xbookPath, engineArgs) It records the location of the xbook, xbookPath. It also expects 2 engineArgs: the name of the jython file to interpret (i.e., the xbook implementation), which contains a class definition referred to as the xbook handler, and that xbook handler class name.

start(out) Creates the jython interpreter and then creates a xbook handler object. Next, it calls the xbook handler’s initialize routine. So far, the initialize routine has been used to read in a html form and send the output to the xbook client.

stop(out) Calls the xbook handler’s finalize routine and then destroys the jython interpreter.

eexecute(command, formValues, out) Calls the xbook handler’s do<command> routine (e.g., if command is ‘Submit’, ‘doSubmit’ is called’).

For example, suppose the following entry exists in the registry:
JythonEngine would expect handler.py to contain the definition of a class named SampleHandler. This class, at a minimum, needs to define initialize and finalize. It will also (most likely, but not required) define one or more doXXX routines where XXX is an command request from the xbook client.

2.2.6.6.3.1 To be changed Nothing comes to mind.

2.2.6.6.3.2 To be completed Nothing comes to mind.

2.3 xbook repository

2.3.0.1 Specification

The purpose of the xbook repository is to store xbooks so they can be served by a xbook server. The repository also archives user xbook sessions for the following reasons:

- Users want to keep a record of the jobs they have launched so they can revisit their results or share with others.
- Users want to be able to re-execute a job or execute a similar job.
- Users want to be able to monitor their job and store pointers to their data.

An archived xbook session is called the subxbook of a xbook. During a xbook session, we store all of the user’s input parameters and the output that is sent back to the xbook client. The output is referred to as a page and can contain status information about the job it launched, URLs to data, meta-data, etc. A page can also contain another form (e.g., to monitor job status). Therefore, each xbook can emit multiple pages and are stored in sequence in order to be browsable by the xbook client. When we archive a page that contains a html form, we edit the page so that the default input parameters correspond to the input parameters the user used during the session. A subxbook also contains the xbook handler script and is a executable xbook as well. This allows users to re-execute a xbook with the same or similar parameters.

Each xbook also contains a properties file. This a xml file containing meta-data which will be published into a xbook directory service. It will also contain a map file which controls access as to what users can execute the xbook. The map file is just a list of user DNs.

The xbook repository is implemented over the file system. Figure 2.3 shows the layout of a sample xbook repository. The repository contains n xbooks labeled xbook-1 to xbook-n. The contents of xbook-1 are a html form which is the starting page that is sent to the user when they request the xbook, the handler script which contains the xbook implementation, the properties file, and the archive which stores xbook sessions. The archive contains m sessions or subxbooks. The first subxbook of xbook-1 is also shown, subxbook-1. It contains everything that xbook-1 did plus the pages collected during the session.

2.3.0.2 Implementation

The root of the xbook repository is read in from a configuration file when the xbook manager is started. Each xbook in the repository contains an entry in the registry file (see Section 2.2.5). Each xbook directory contains a properties.xml file with the following format:

The xbook client reads in this file to find out the xbook directory structure (i.e., how many subxbooks). The parameters are also replicated here (also stored in results) in case another entity wants easy access to them.

Archiving is done via jython libraries. This allows flexibility during development as archiving can be turned on and off by commenting out the archive function call. Also, some user requests will not needs archiving (e.g., a monitor job request).
2.3.0.2.1 To be changed  To support multi-stage request, the format for result should looks like this:

```xml
<result>
  <page>output1.html</page>
  <page>output2.html</page>
  ...
</result>
```

2.3.0.2.2 To be completed

- We have not incorporated authentication in xbooks yet. The map file is just a list of user dn patterns that can be read by the xbook manager and matched against the user’s to permit or deny their request (see Section 2.2.4.1).
Chapter 3

Client-side

3.1 Introduction

Users can request xbooks from the xbook server through a xbook client. We provide two xbook clients: a portlet and a command-line client. The portlet is a GUI client and is targeted to the end-user who needs to be guided through the process of how to invoke a xbook. While the command-line client is targeted to Grid programmers who often need to invoke a xbook with a specific configuration in a single keystroke during debugging and development. Each client is described in the following sub-sections.

3.2 Portlet

3.2.1 Specification

A portlet is a web application component that can be plugged into a portal framework. The portlet will provide the following services:

- browse a xbook directory service
- invoke and navigate a xbook
  - display xbook properties
  - retrieve the user’s proxy credential from the portal framework
  - retrieve the user’s resource file from the portal framework
  - package up xbook input values and call xbook manager methods to invoke a xbook
  - navigate a sub-xbook

3.2.2 Differences from XCAT-SP

In the XCAT-SP, there is no concept of a separate client and server. Therefore, the XCAT-SP was essentially limited to one type of client. In the new design, we can have multiple types of clients thus increasing its usability.

3.2.3 Implementation

The xbook client code is in the package xbooks.client.portlet. It is a subclass of VelocityPortletAction and the templates are in the top level xbook CVS directory under templates. The portlet registry file is in the conf directory under xportlets-xbooks.xreg and should be listed in the portlet directory listing under xportlets : xbooks.
Currently, the portlet connects directly to the xbook manager. When it is first loaded, the user is presented with a login screen that prompts the user for the machine and port that the xbook manager is running on. It also prompts for the name of the particular xbook. See Figure 3.1.

Once loaded, the properties of the xbook are displayed: name, description, version. The user has the option to view the xbook (i.e., run the xbook) or view one of its subxbooks which are displayed in a drop-down menu. See Figure 3.2.

When the user chooses to view the xbook, this sends a request to the xbook manager to start up the engine for that xbook (i.e., invokes the startUp method). The xbook specific screen is then returned to the user. Figure 3.3 shows the athena xbook screen.

The 'back to xbook' button will take the user back to the xbook properties screen while the 'start over' button will take the user back to the xbook login screen. The 'submit' button will trigger the invocation of one of the xbook’s methods (see Figure 2.2). The portlet routine that handles the 'submit' button is the doSubmit routine. It first finds out which of the xbook method names it should call by searching for an input variable named xbook.action. Then it filters all of the input variables for xbook input variables (i.e., any variable whose name begins with the prefix xbook.form). This filtering process is needed because otherwise we would be passing back Jetspeed specific variables to the user’s scripts. The xbook input variables are packaged into a hash table and the portlet calls the xbook manager’s callMethod method. The output returned from the callMethod routine is put into the RESULT context variable and displayed via the velocity template. The output can also be an other form (i.e., a multi-stage request). Therefore, we cannot shutdown the xbook manager until we know the xbook request is finished. This is determined by checking the value of a input variable named xbook.keep_alive. If this is set to false, we call the xbook manager’s shutDown routine.

Note: the xbook manager object is stored in temporary storage because the portlet is re-created for each portlet action. Therefore, we needed a way to store the xbook manager connection. Since temporary storage is per user session, there should be no conflict with the any other of the user’s session or other users sessions.
3.2.3.1 Viewing subxbooks

Currently, when a user decides to view a subxbook (i.e., an archived xbook session), they choose the subxbook version in the menu box and click the ‘View subxbook’ button (see Figure 3.3). This will bring up the properties screen for that subxbook just as it did for the xbook (see Figure 3.4 for an example). The screen is similar to the xbook in that the user can view the subxbook (i.e., run the subxbook) or view this subxbook’s subxbooks. The difference is that the user can also view the output of the subxbook (i.e., the output of the xbook session) and there are some additional properties (e.g., date).

In the case the user decides to view the subxbook, it will display the xbook specific interface. If the xbook session was archived using one of the xbook archiving methods (see Section 2.3.0.2), the default values in any forms will be those chosen when the user executed this xbook (this facilitates the execution of similar jobs). When the user wants to view the output of the subxbook, the output that was returned to the user when they executed the xbook is fetched (see Figure 3.5 for an example). The output can contain links to data or can be a form that can be used, for example, to check the status of a job.

![Figure 3.3: Snapshot of athena xbook.](image)

![Figure 3.4: Snapshot of portlet subxbook properties screen.](image)

15
3.2.3.2 To be changed

- Currently, error handling is not very robust. If there is an error in the xbook on the xbook manager (e.g., syntax error in a script), the portlet returns nothing. It should report the error that the xbook manager caught.

- When viewing a xbook, the user can click on the 'back to xbook' or 'start over' buttons. This means they are exiting the xbook so we should call the xbook manager's `shutDown` method.

- The browsing aspects of the portlet are kind of clunky (especially with respect to viewing subxbooks) and could stand for some improvement. One thing that really needs to be changed is the location of the 'View output' button. It needs to be moved to the 'View xbook's screens where currently the navigation buttons are Submit, Back to xbook, and Start Over (see Figure 3.3). Since we can grab the number of pages of output this xbook has (see Section 2.3.0.1) from the properties file, we can display them as a page navigation bar. For example, just as you would see at the bottom of a google results page: `<previous> 1 2 3 4 <next>`. Thus if the user just wants to page through the output, it’s easy to do. And if one of the pages is a monitoring job page, the user can just click Submit and get the updated results.

3.2.3.3 To be completed

- Need to update to the latest XCAT and XSOAP jars so that we can pass proxy credentials to the xbook manager (which can then pass it to the xbook). Code is also needed to get the proxy credentials from the ProxyManager.

- Jetspeed will unload the portlet when a user’s session times out. This will kill the connection with the xbook manager which will kill the thread handling the request (right?). But ideally we would like to call the xbook manager's `shutDown` routine and exit properly. There should be a routine we can define within the `VelocityPortletAction` class which would be called when Jetspeed decides to unload the portlet. We should find out what that routine is and call the xbook manager’s `shutDown` routine.

- Currently the portlet connects directly to a xbook manager. It should be changed so that it first connects to a xbook directory service. Then once it has a "xbook url", it contacts the xbook server and launches a xbook manager. Finally, it connects to the xbook manager.
3.3 Command-line

3.3.1 Specification

When constructing xbooks, users (i.e., programmers) need an environment where they can quickly debug and develop their scripts. Often, users will be running a xbook under a particular configuration (i.e., a set of input parameters) when testing and debugging. Development can often be slowed if the user has to use a GUI to construct the configuration they need to test or debug a certain feature. Thus, we provide a command-line interface to xbook managers so that a xbook configuration can be described in a configuration file and invoked in a single command. The command-line interface also provides a way for users to load a particular proxy and resource file so that the portal environment can be emulated to provide a realistic debugging and testing environment.

Furthermore, command-line clients allow for xbooks to be executed in batch. This may sound counter-intuitive as why would you need a xbook at all if you’re launching a bunch of jobs in batch from the command-line? However, a xbook is more than just a GUI job launching mechanism. It’s also a mechanism for monitoring jobs plus recording meta-data and data locations. It is therefore useful for users to be able to launch a bunch of xbooks and then monitor their progress and browse the output from a GUI interface. An example of where this is useful is when a collaboration decides to run a large set of Monte-Carlo jobs. This will be launched by one person but the xbooks will be viewable by all members of the collaboration and so they can monitor job progress and browse output too. Then if one member finds a particular result interesting, they can run further, more refined results (i.e., subxbooks) underneath the xbook that produced it.

3.3.2 Differences from XCAT-SP

This provides the same purpose and function as XCAT-SP command-line client which was built using Cactus testing framework.

3.3.3 Implementation

Development on this client is still rather premature as the current client is targeted to a particular xbook. However it won’t take much effort to make this generic (as described in the following subsections). The code for the command-line client is in XbookPortletEmulator.java file located in the xbooks.client package. It takes 3 arguments (i.e., just like the portlet): xbook manager host name, xbook manager port, and xbook path.

3.3.3.1 To be changed

3.3.3.1.1 Input variables

Currently, the values of the xbook input variables are hard-coded (very bad!). We need to provide a mechanism by which input form values can be read in from a file, the simplest format being a java PropertiesFile format (i.e., name, value pairs):

```plaintext
<name of input form variable>: <value of input form variable>
<name of input form variable>: <value of input form variable>
...
<name of input form variable>: <value of input form variable>
```

3.3.3.1.2 Methods

Currently, we only execute one xbook method called Submit. We need to provide a mechanism by which a list of xbook methods that should be executed can be read into the client. Then the xbook client is responsible for executing the xbook methods in sequence and returning the output to the screen.

3.3.3.2 To be completed

See section above.
Chapter 4

Directory Service

4.1 Specification

In a production environment, an xbook server will be storing hundreds to thousands of xbooks, most being subxbooks (i.e., results). Therefore, we need a mechanism by which users can easily locate xbooks based on their properties (i.e., meta-data). The xbook directory service is such a mechanism by which users can publish their xbooks (i.e., their results). Users will send their xbook property files to the directory server along with a URI describing its location. Other users will then be able to query the directory service to locate an xbook and be redirected to its location. If the user is authorized to use the xbook, they will then be able to execute/view it. Optionally, users can also publish the actual xbook (e.g., a tar file) to the directory for others to download and deploy into their own xbook servers (i.e., if they are not authorized to execute the registered xbook or want to modify it).

Users should be able to query for xbooks based on the properties described in their properties file. Examples of this include:

- name
- description keywords
- version numbers
- date
- author
- input parameters

The querying interface uses a filtering methodology much like a Google advanced search. See Figure 4.1 for an example querying interface. When the results of the search are returned, users should be able to view the xbook’s full set of properties and run additional refining queries.

*Note: this idea is based on a "personal index" conversation with Beth many moons ago.*

4.2 Differences from XCAT-SP

This will perform a similar function as the webdav server does for the XCAT-SP. The main difference being the querying interface.

4.3 Implementation

Query for xbooks using the following filters:

- **Name**: athena
- **Description**
- **Keywords**
  - Version: $> 2.5$
  - Version: $< 5.0$
- **Date**
- **Author**
- **Input Variables**
  - event_generator = PythiaModule
  - and simulation = atlfast

Figure 4.1: Example xbook directory service querying interface.

### 4.4 To be changed

Nothing.

### 4.5 To be completed

Everything. We will be running relational queries on the property files. What database back-end is most appropriate?
Chapter 5

Writing xbooks

In this section, we will describe the process of creating a xbook using the existing prototype implementation. A xbook is a Grid portal plug-in, a mechanism by which an application-specific interface can be loaded into a Grid portal. This main component of the xbook is a handler script which will be called when the user invokes the xbook from the portal. This xbook handler script (or xbook script for short) provides an interface (i.e., a HTML form) for collecting input values from the user and defines actions for handling the user's input values. The strength of the xbook script will be derived from its usage of script libraries (or modules) which can be used to construct highly functional xbooks. The available xbook modules will depend on the xbook engine which executes the xbook; this is further described in Section 5.1. Another important component of a xbook is a properties file which is described in Section 5.2. A properties file is a XML file that describes the xbook (i.e., meta-data) in sufficient detail that it can be discovered through a xbook directory service. An xbook can contain other components but only the xbook script and properties file are required.

In Section 5.4, we will describe how to start up the xbook manager. Then in Sections 5.5 and 5.6 we will describe how to run the xbook clients. Finally, we conclude by describing the xbook examples distributed with the source code.

Note: we assume the reader has access to the xbook source code and has read the README file and performed the following actions:

- Compiled the executables using ant.
- Installed the xbook client into a Jetspeed portal along with the other xportlets.

5.1 Xbook scripts

5.1.1 Jython scripts

In Section 2.2.6, we described a xbook engine as an intermediary between the xbook client and the xbook. In this section, we describe the interface between the xbook jython engine and xbook.

When the jython engine is invoked by the xbook manager, it creates a jython interpreter. It then places the following object into the namespace (i.e., main):

- xbook is a instance of the class defined in Xbook.java from the package xbooks.server.engine. This object contains any meta-data about the xbook that needs to be passed into the xbook. Currently, this object is used to store the path of the xbook. The user can retrieve the path using the getPath() method. For example, path = xbook.getPath(); (Note: the xbook object should also be used to get the user's delegated proxy credentials once delegation is incorporated, e.g., a getProxy() method.)

The jython engine also retrieves information from the registry entry describing the xbook. It contains the following information:

- the name of the xbook script to interpret
• the name of the handler class defined in the xbook script

Then the jython engine first reads in the xbook handler script. This xbook handler script is a jython module and must contain a handler class definition and may contain other class and function definitions; it should not contain a main. Next, the jython engine creates an instance of the xbook handler class. This handler class will contain methods that will be mapped to user requests coming from the client. The handler class must contain an initialize and finalize method and any number of doXXX methods where XXX is the name of a action variable defined in the user interface. The following describes their mapping:

```python
.__init__(self, workingDir)  The jython engine will create the instance of the handler using handler = <handler name>(xbookPath). Therefore, the user must define a constructor to handle this. This will communicate the location of the xbook path and can be used inside the xbook scripts as the “working directory”.

initialize(self, out)  This function will be called when the user requests this xbook from the xbook client. Typically, initialize will return HTML to the java ByteArrayOutputStream, out. The HTML will typically contain a form definition with a number of input tags.

finalize(self, out)  This function will be called when the xbook session is complete. Section 3.2.3 describes how xbook completion is determined. Users may write cleanup messages to the java ByteArrayOutputStream, out.

doXXX(self, formValues, out)  When a form is submitted from the client, it contains a special input variable named xbooks.action. The value of this variable is the XXX of doXXX. Therefore, for each form output that is returned the client, there should be a corresponding doXXX routine to handle the results. The values from the form will be available in the hash table formValues. Since the input variables can contain multiple values, the values are stored as an array. Therefore, formValues is a hashtable of string arrays. The user can choose to output either plain HTML to out or another form (in which case, the will need to define another doXXX routine to handle that output).

(Restrictions on the output that can be returned from these functions is described in Section 5.1.2.)

It is expected that users will use these methods to launch, monitor, and archive their specific Grid application jobs. But really, the user can place any functionality they want inside these methods; there are no restrictions.

5.1.1.1 Modules

There are two types of modules that will be used from xbook scripts: xbook modules and Grid modules. A xbook module provides users with an interface to better utilize the xbook infrastructure. Currently, we provide 2 modules:

```python
xbooks.archive  This module exports one xbook archiving method called archive(xbook, html, formValues, scriptOutput). This method is designed to be called within a doXXX method. The parameters are described below:

xbook  the xbook object passed in from the jython engine

html  the html string containing the form that triggered this method.

formValues  the hash table formValues passed into the doXXX routine.

scriptOutput  the output being returned to the client

xbooks.activity.handler  (note: the module name needs to be changed to just xbooks.handler). This module provides 2 classes which can be used as superclasses for a simple handler class. The classes are defined below:

Root  Defines a skeleton handler class for a xbook which just defines one method doXXX routine called 'do-Submit'.

  __init__(self, workingDir)  Constructor which just stores the xbook path in a member variable called dir.

  initialize(self, out)  Does nothing. It’s a placeholder and should be overridden by subclass.

  finalize(self, out)  Does nothing. It’s a placeholder and should be overridden by subclass.
```
doSubmit(self, out, formValues)  Does nothing. It’s a placeholder and should be overridden by subclass.

readFile(self, fileName)  Utility routine which reads in a file from the xbook directory and returns it as a string.

**HTML**  Defines a skeleton handler class for a xbook which reads in a html file and defines the handler in it for a doXXX routine called 'doSubmit'. This is a subclass of Root.

initialize(self, out)  Reads in a html called "index.html" located in the xbook directory and returns it to the client. (note this routine is currently broken; should just need to replace the line

```python
    content = self.readFile( xbook.getPath() + "/index.html" );
```

with

```python
    content = self.readFile( self.dir + "/index.html" );
```

doSubmit(self, out, formValues)  Echoes back the values of the formValues to the output stream out.

archive(self, out, formValues, xbook)  Archives a doXXX into a xbook page. The parameters are:

out  The string output of the doXXX being returned to the client.

formValues  the hash table formValues passed into the doXXX routine.

xbook  the xbook object passed in from the jython engine. This argument should be removed and replaced with the usage of self.dir.

Essentially, this function can be used to archive a simple xbook (i.e., create a subxbook). It utilizes the properties file to track the number and names of the subxbooks it has created. It stores the number in a instances tag and the names in subxbook tags as described in Section 2.3.0.2. It constructs the name of a new subxbook using the formula

```python
    subxbook_name = version + "." + str(instances++);
```

The second set of modules are Grid modules. Since jython allows for Java classes to be directly imported in the scripts, we can draw upon any Grid Java APIs. Examples of this include CoG[13], XMessages [24], XCAT [25], Chimera [26]. To use any of these modules, the user will need to ensure the paths to these jars are listed in the python.path in either the xbook manager jython_engine.cfg configuration file or the local xbook jython_engine.cfg configuration file.

### 5.1.2 Output Restrictions and Requirements

In Section 5.1, we defined a number of methods that write HTML code to a java stream. This code will then be returned to a portlet client and displayed inside the portal framework. Depending on the implementation of the portal framework, there may be restrictions on what HTML tags can be included in the output. The following lists the output restrictions for portlets built using the Jetspeed portal framework [17].

- Jetspeed uses tables to format the layout of the portal. Therefore, the output that is returned by our xbook will be displayed inside one of the table cells. Thus, any HTML tags written to the portlet should be tags that are valid inside a table cell (i.e., body tags only). The xbook should not write any header tags (e.g., HTML, HEAD, TITLE) nor the BODY tag.

- Our portlet client is a subclass of VelocityPortlet which allows our portlet client to be plugged into Jetspeed. The VelocityPortlet is also form-based and therefore places our xbook output inside its own specially formatted FORM tags. Therefore, our xbook output should not contain FORM tags, only input tags.

Our xbook portlet also impose some additional restrictions:

- Input tags written by the xbook (i.e., xbook variables) need to be differentiated from internal Jetspeed input variables and internal xbook input variables. Therefore, we impose a naming restriction on the xbook input variables. A xbook variable’s name must be prefixed with ‘xbook.form.’. If a xbook input tag does not have this special prefix, it will be ignored by the portlet and thus the value will not be returned back to the xbook.
• As discussed in Section 3.2.3, the xbook portlet determines whether a xbook session is complete by checking the value of a variable called xbooks.keep_alive. Therefore, the xbook must write a hidden input tag which has a boolean value. True or '1' indicates that the xbook session is still active and the portlet should expect more output to come.

    <input type="hidden" name="xbooks.keep_alive" value="1">

While false or '0' indicates that this is the last output and the portlet can shutdown the xbook manager.

    <input type="hidden" name="xbooks.keep_alive" value="0">

• As described in Section 5.1, each xbook form must name the doXXX routine that will handle the input values. The xbook portlet reads in the name of the xbook method name from the form variable xbooks.action. Thus, if the user creates a method called doSubmit, they would express this in a hidden input tag as follows:

    <input type="hidden" name="xbooks.action" value="Submit">

5.2 Properties

Each xbook must define a properties file called properties.xml with the following format:

    <xbook>
        <name>xbook name</name>
        <description>the xbook description that will be published in the xbook directory service</description>
        <version>a version number</version>
        <date>if this is a subxbook, this is the date it was instantiated</date>
        <instances>number of sub-xbooks</instances>
        <subxbook>name of sub-xbook</subxbook>
        <results>name of output file</results>
        <parameter name="name of an xbook input parameter">
            <value>user’s value of the xbook parameters</value>
        </parameter>
        <mapfile>
            <dn>user dn</dn>
            ...
            <dn>userN dn</dn>
            </mapfile>
    </xbook>

Currently, only the following tags are required: xbook, name, description, version. See Section 2.3.0.2 for a more detailed discussion.

5.3 Registry

As described in Section 2.2.5, each xbook is registered into a registry file which gets read into the xbook manager. The format of the registry file is as follows:

    <xbook path prefix>: <engine type> <engine args>
Specifically, for a jython xbook, the entry is as follows:

```
<xbook path prefix>: JythonEngine <path to handler script> <handler name>
```

The xbook path prefix is relative to the xbook root directory. Thus if the xbook root directory is `/home/ssmallen/xbooks` and the xbook example is located in `/home/ssmallen/xbooks/example`. The entry would look as follows:

```
example: JythonEngine script.py ExampleHandler
```

where `script.py` is the name of the handler script in the xbook directory and `ExampleHandler` is the class name of the handler defined inside the file.

The name of the registry file is determined through the configuration variables `xbooks.cfg.dir` and `xbooks.registry.name`. Both variables should be defined in the xbook manager's configuration file.

### 5.4 Running the xbook manager

The xbook manager executable is defined in the `XbookManager.java` file in the `xbooks.server.manager` package. It takes 2 arguments:

1. the port the xbook manager will run off of
2. the path to a configuration file (see the `manager.properties.sample` file in the `conf` directory for an example).

In the top-level xbook source directory there is a simple script called `run.sh` which can be used to invoke the manager on port 1111.

```
./run.sh manager
```

Note: be sure to run `grid-proxy-init` first.

### 5.5 Portlet client

Once the portlet is installed into Jetspeed, it will appear in the portlet directory under `xportlets : xbooks`. Just load it into the pane of your choice. See Section 3.2 for more details.

### 5.6 Command-line client

The command-line client currently is not really usable currently. It needs some fixes which are described in Section 3.3. Once those are fixed, this subsection should be revised.

### 5.7 Examples

This section will detail the example xbooks distributed with the xbook source code: form and gram. To use, make sure the following entries are in your registry:

```
form: JythonEngine <xbooks src dir>/examples/form/script.py FormExample
gram: JythonEngine <xbooks src dir>/examples/gram/script.py GramExample
```

Also be sure that `python.path` in the `jython_engine.cfg` contains all of the CoG jars, the jython jar, and the path to the C python libraries.
5.7.1 Form

The form xbook is a very simple xbook which displays all the types of input tags that can be used to gather values from the user and then echoes the values back to the user. The form which the user will see is in the file index.html. The script which will handle it is in the file script.py. The script defines a class called FormExample which is a subclass of xbooks.activity.handler.HTML. It overloads only 2 routines: initialize and doSubmit. initialize just reads the index.html file and returns it to the out stream. It should be removed from the example when the handler script is fixed (see Section 5.1.1.1). The doSubmit routine searches for each xbook variable that was defined in index.html and echoes the value(s) back to out.

5.7.2 Gram

This xbook performs the equivalent of a GUI globus-job-run using CoG. The layout is similar to the form xbook in that it contains index.html and script.py. The handler class name is GramExample and overloads both the initialize and doSubmit methods. The doSubmit routine launches a GRAM job, a GramJobListener, and a GassListener (to receive the job output). Since the job request is an asynchronous call, we use a condition to wait to be notified by the GramJobListener for job completion. Once the job is complete, we get the output from the GassListener and exit.
Chapter 6

Case Study: Grappa

In this chapter, we discuss how to setup a demo of Grappa (Grid Application Portal for Physics Applications). This will install a Jetspeed portal and install the xbook athena (a small write-up of the athena notebook will be displayed once the portal is installed). Note: the athena xbook is stored in the CVS repository at codes/applications/xbooks/athena.

1. Login into a machine and create a directory called grappa-demo.
2. Untar/Unzip pacman into directory. E.g.,
   
   tar zxvf /u/ewww/xportlets/pacman/packages/pacman-2.066.tar.gz

   Optionally, you can download pacman from

   http://www.extreme.indiana.edu/xportlets/pacman/packages/pacman-2.066.tar.gz

3. Change to the pacman-2.066 directory.
4. Source the appropriate setup.[csh,sh] file.
5. Change back up to the grappa-demo directory.
6. Type

   pacman -get pacman

   and answer y to the question asking to start a new installation

7. Edit the caches file and replace

   http://www.extreme.indiana.edu/xportlets/pacman/cache

   with

   http://www.extreme.indiana.edu/xportlets/pacman/grappa-cache

8. Type

   pacman -get grappa-demo

   This step will install a lot of packages into this directory and may take a while depending on your machine and network connection.

10. Create a proxy credential by typing `grid-proxy-init`.

11. Type

   `portal-start`

12. Change to the `xbooks-0.1-src` directory.

13. Type

   `nohup sh run.sh manager &`

   This will start up the xbook manager on port 1111. Edit the `run.sh` script to change this or modify debugging output.

14. Get the URL printed out in step 11 and append `jetspeed`. E.g.

   `https://linbox1.extreme.indiana.edu:8444/jetspeed`

15. Log in to Jetspeed with username `turbine` and password `turbine`.

16. Click the `xBooks` link in the sidebar.

17. Type `athena` as the xbook and change the hostname and port accordingly.

   The administrator username is `admin` and password is `jetspeed`. These are the defaults and can be changed by logging as the admin. The portal has been setup to not require administrator approval of account requests. This can be changed in the JetspeedResources.properties file. See the Jetspeed documentation [17] for more information.
Chapter 7

Future Directions

7.1 HTML Server

When a job is launched from a xbook, that job can return information/meta-data files to the xbook. For example, currently the athena notebook, stores summary graphs into the xbook folder. Therefore, the xbook manager should be able to serve files. One way to do this is have the xbook manager implement a HTML server port that can serve files. Then xbooks can then store URLs to these files and so they are easily retrievable when a user is viewing the xbook. Alek is currently working on creating a HTML server port.

7.2 Resource Management

Since many xbooks will be launching a job, it is important that users have a way to manage a list of resources they have access to in a generic way. Liang has created a resource manager portlet to do this resource management. We need to provide a way for this information to be relayed to the xbook (i.e., just like the proxy will be).

7.3 Virtual Data

Note: some thoughts regarding virtual data and xbooks

Each activity in a xbook is a transformation. We can store a transformation id in an activity’s properties folder and then register all application instances as derivations in the VDC. When a user requests an activity, we can first check to see if there is already a derivation existing in the VDC and fetch the data if it is. If a archived activity is re-executed with exactly the same parameters, then we can also grab the data from Chimera. If an archived activity is executed with one or more different parameters, then we can register it as a transformation.
Bibliography

[1] Pdf version at devguide.pdf. 1
[2] Pdf version at devguide.ps. 1


[22] Jython webpage at http://www.jython.org. 10


[26] Ian Foster, Jens Voekler, Michael Wilde, and Yong Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In 14th International Conference on Scientific and Statistical Database Management, 2002. 22