SANDIA REPORT
SAND2009-7011
Unlimited Release
Printed October 2009

Parallel Digital Forensics Infrastructure

David P. Duggan, Dr. Lorie M. Liebrock

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-7011
Unlimited Release
Printed October 2009

Parallel Digital Forensics Infrastructure

David P. Duggan
Networked Systems Survivability and Assurance Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0672

Dr. Lorie M. Liebrock
Department of Computer Science
New Mexico Tech
Socorro, New Mexico 87801

Abstract

This report documents the architecture and implementation of a Parallel Digital Forensics infrastructure. This infrastructure is necessary for supporting the design, implementation, and testing of new classes of parallel digital forensics tools.

ACKNOWLEDGMENTS

Thanks to the computer science students at New Mexico Tech for their help in completing this project. Thanks also to Dr. Lorie Liebrock for guiding their work.

CONTENTS

FIGURES

This page was intentionally left blank.

NOMENCLATURE

DOE	Department of Energy
NMT	New Mexico Tech, a.k.a. New Mexico Institute of Mining and Technology
PDF	Parallel Digital Forensics
SNL	Sandia National Laboratories

This page was intentionally left blank.

1. INTRODUCTION

Digital Forensics has become extremely difficult with data sets of one terabyte and larger. The only way to overcome the processing time of these large sets is to identify and develop new parallel algorithms for performing the analysis. To support algorithm research, a flexible base infrastructure is required. A candidate architecture for this base infrastructure was designed, instantiated, and tested by this project, in collaboration with New Mexico Tech.

Previous infrastructures were not designed and built specifically for the development and testing of parallel algorithms. With the size of forensics data sets only expected to increase significantly, this type of infrastructure support is necessary for continued research in parallel digital forensics.

This report documents the implementation of the parallel digital forensics (PDF) infrastructure architecture and implementation.

1.1. Architecture

The process flow for this testing infrastructure was created prior to the initiation of the implementation work. It is shown below in Figure 1. The architecture used for implementation is shown in Figure 2.

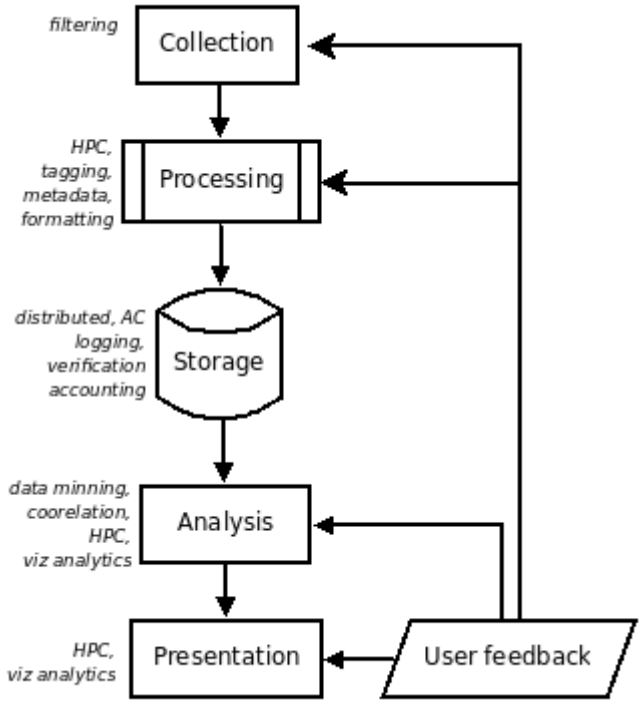

Figure 1 - PDF Process Flow

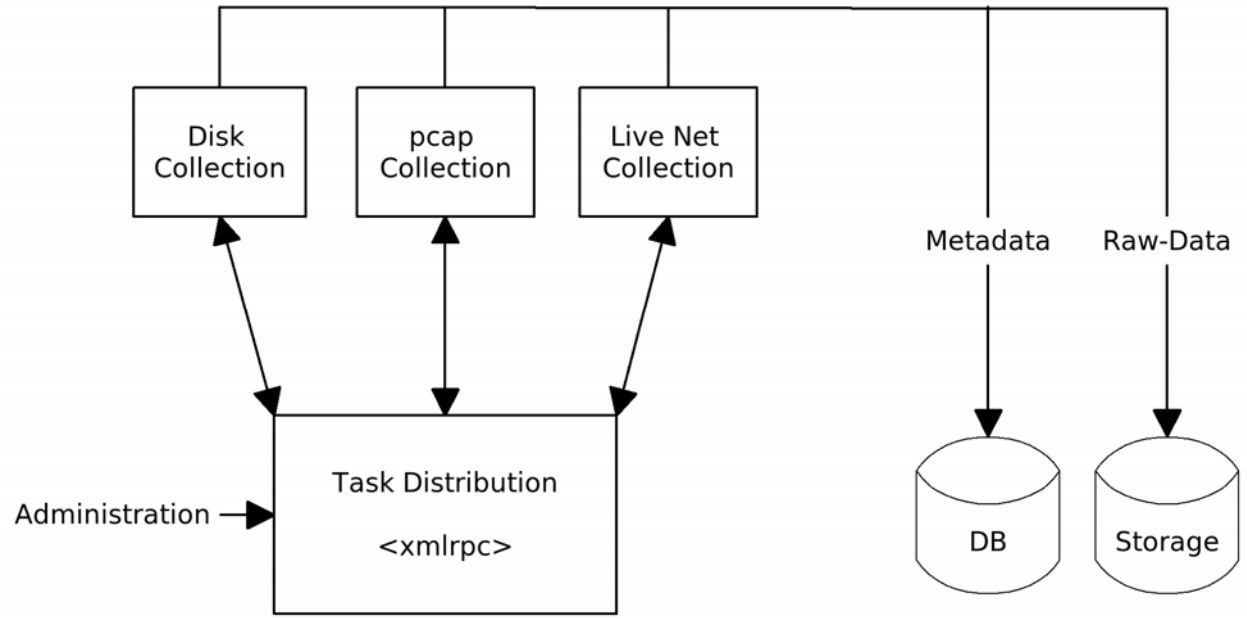

Figure 2 - PDF Architecture

A number of views of the system were developed and are shown in the following figures. Two different systems views are in Figure 3 and Figure 4.

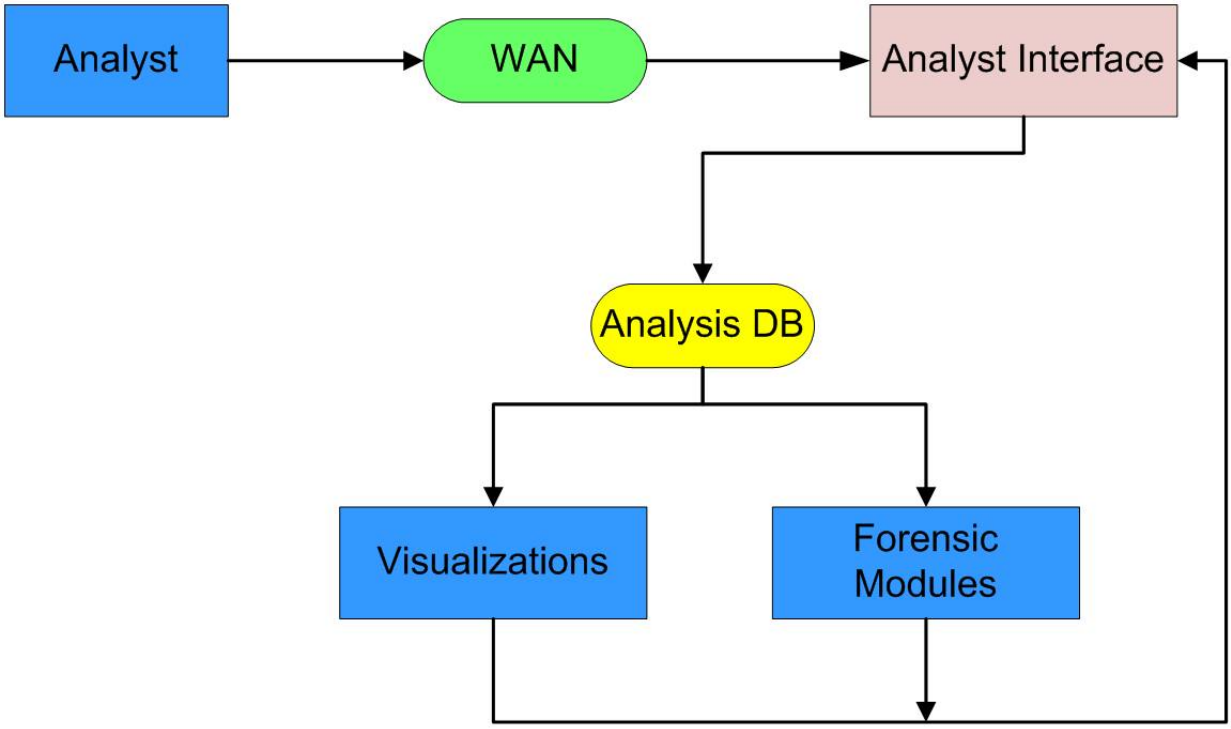

Figure 3 - Systems View A

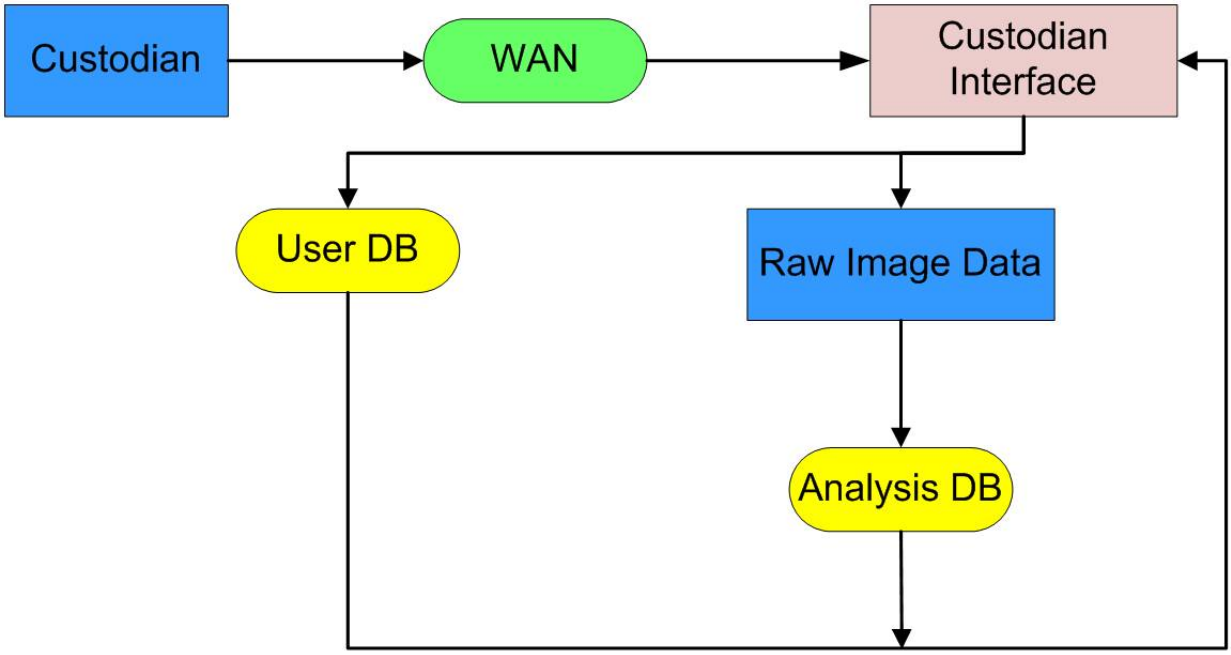

Figure 4 - Systems View B

The physical network view is shown in Figure 5. It shows the expected setup for usage of the infrastructure.

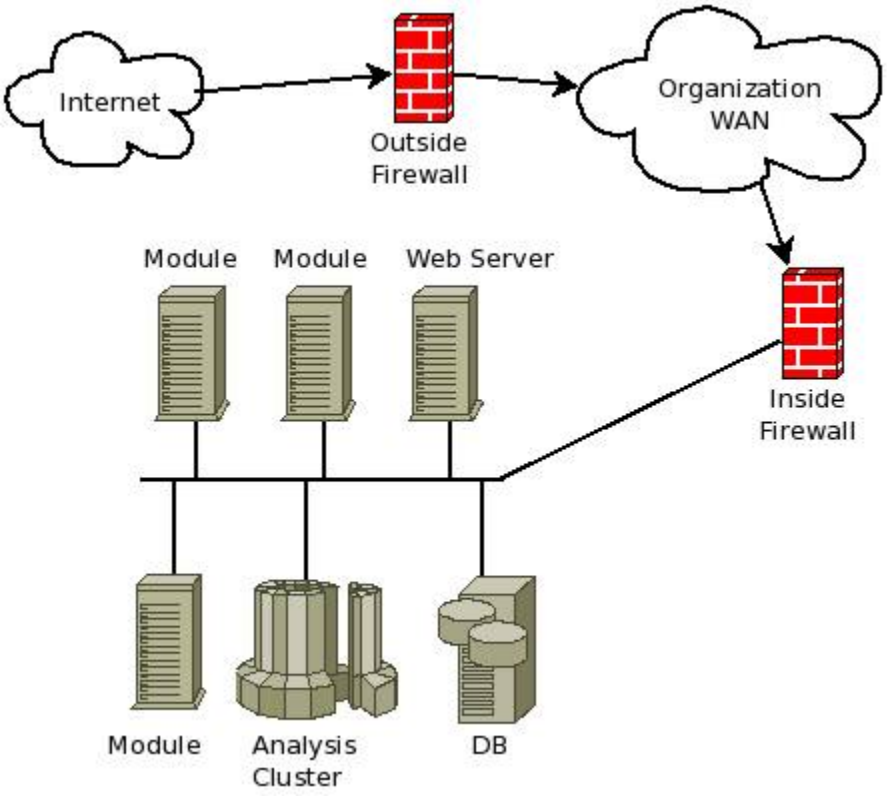

Figure 5 - Physical Network View

There are three database views shown. The first one shows the dataflow support functions, which is Figure 6. Second, all the interfaces and their interconnections are shown in Figure 7. Then, in Figure 8, there is an overview pictorial of the database structure. Colored lines in the figures signify where the links point to, pointing to the like colored row.

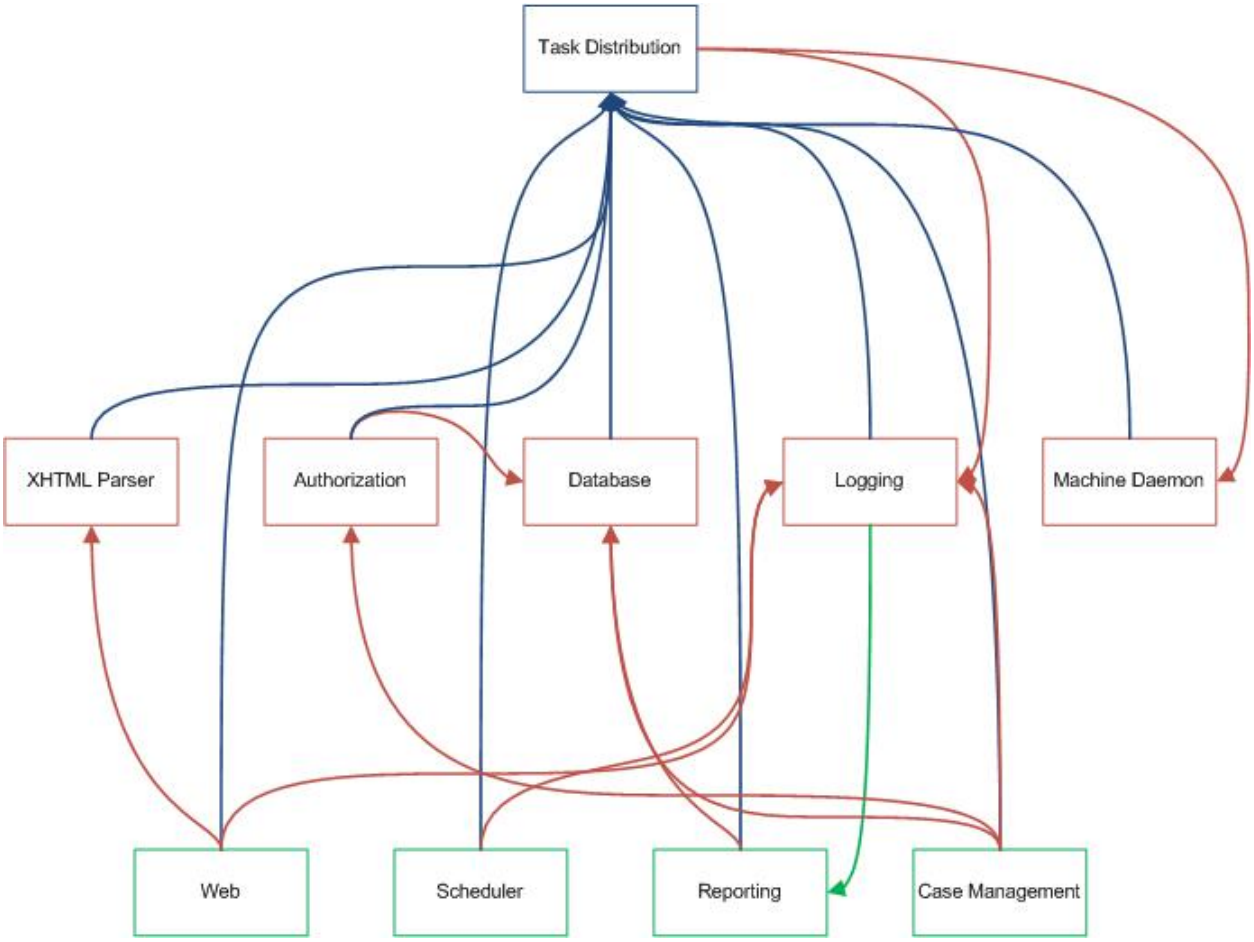

Figure 6 - Data Flow Support

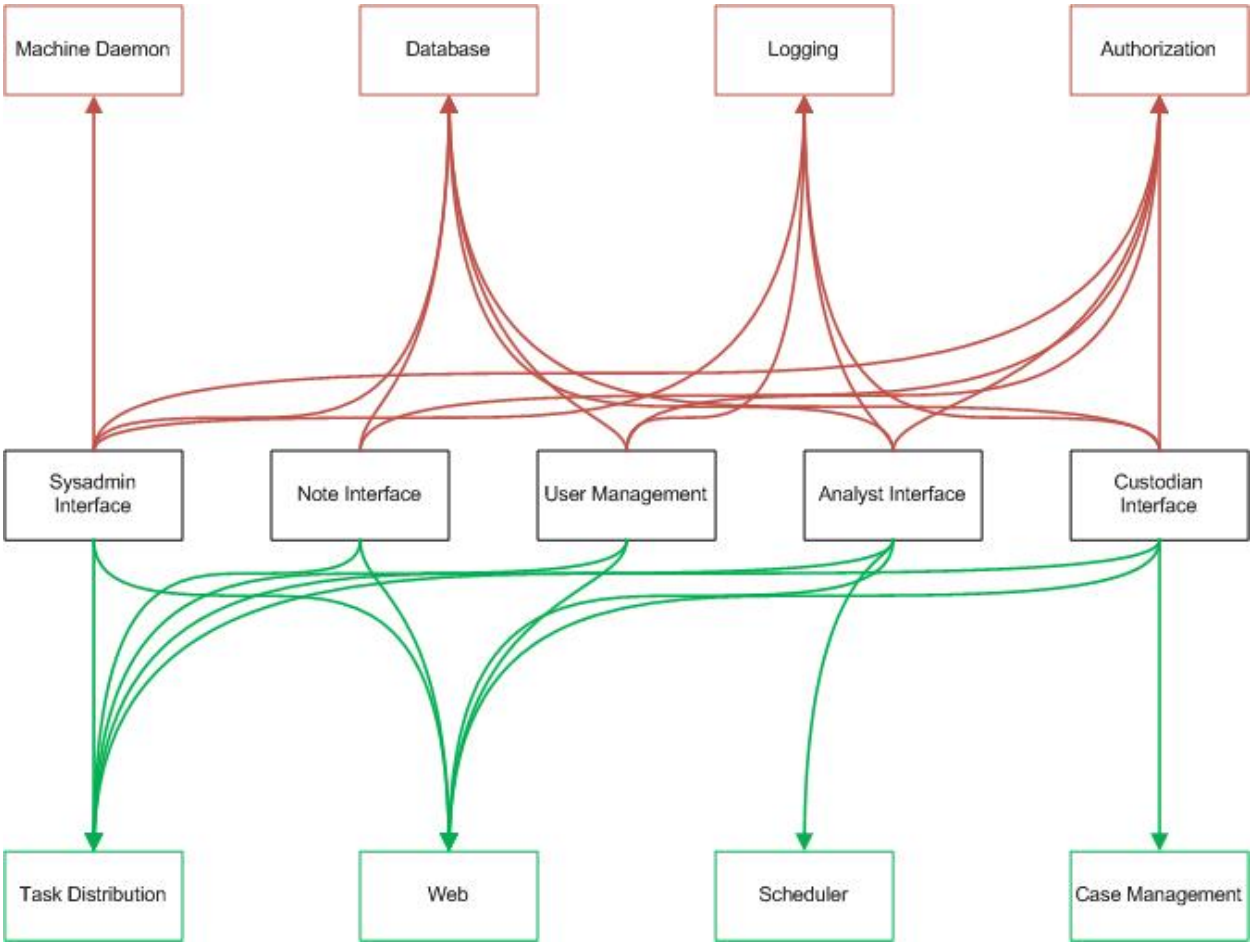

Figure 7 - Data Flow Interfaces

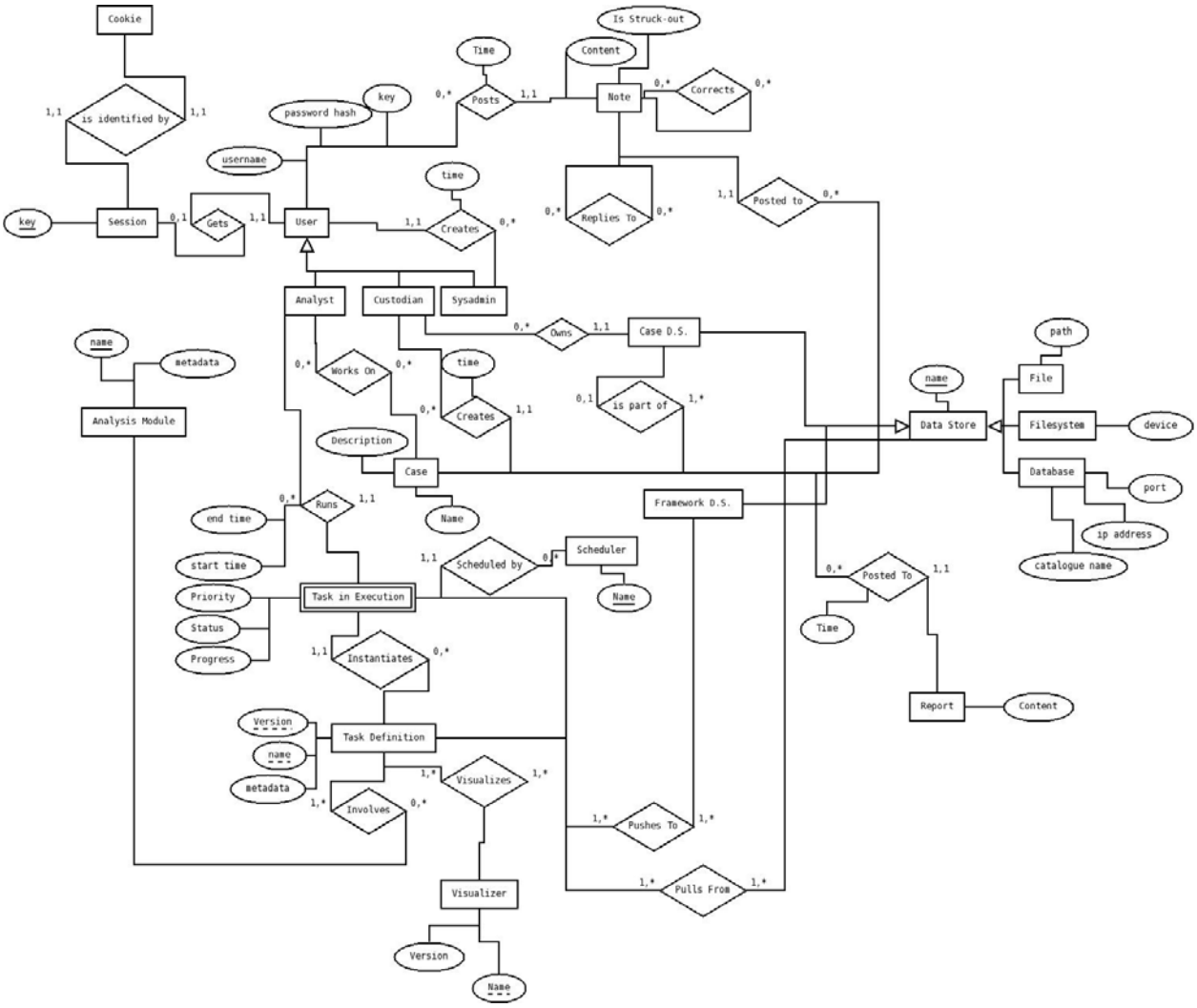

Figure 8 - Database Overview

Code module documentation is included in the appendix of this report.

1.2. Web Interface

The web interface for this infrastructure is shown in Figure 9 and Figure 10. All use of the infrastructure is performed through this web interface.

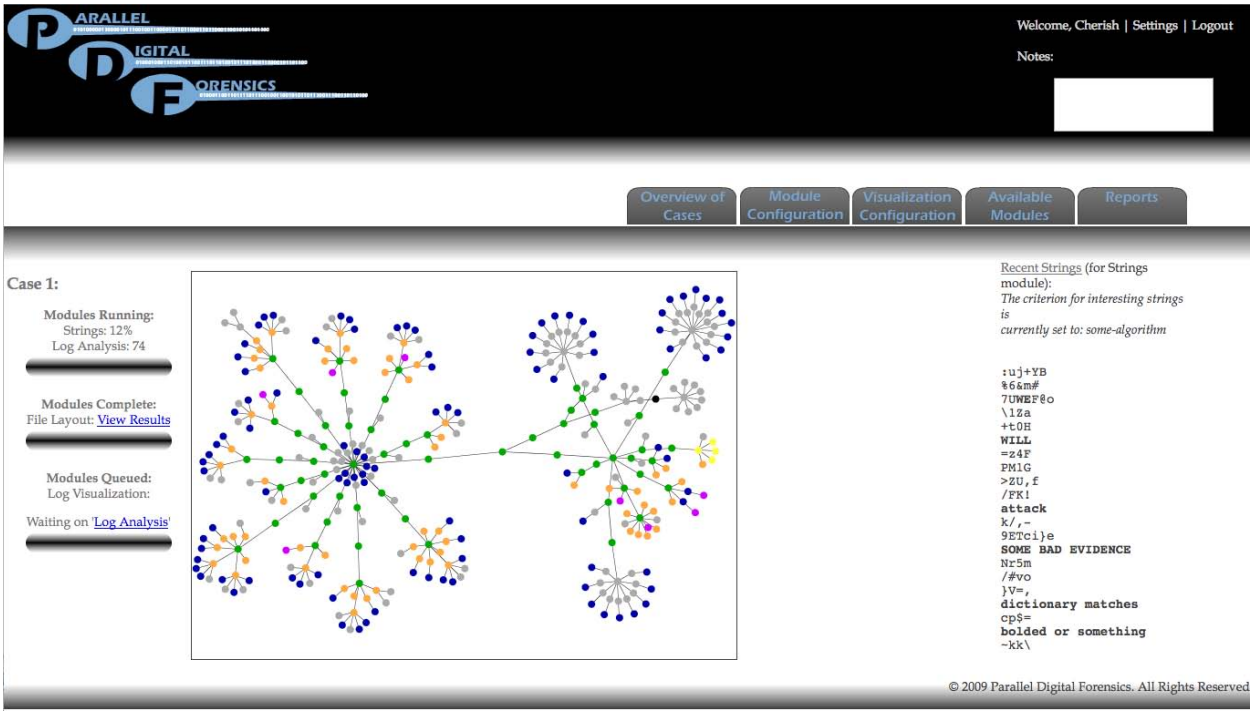

Figure 9 - Main Web Interface

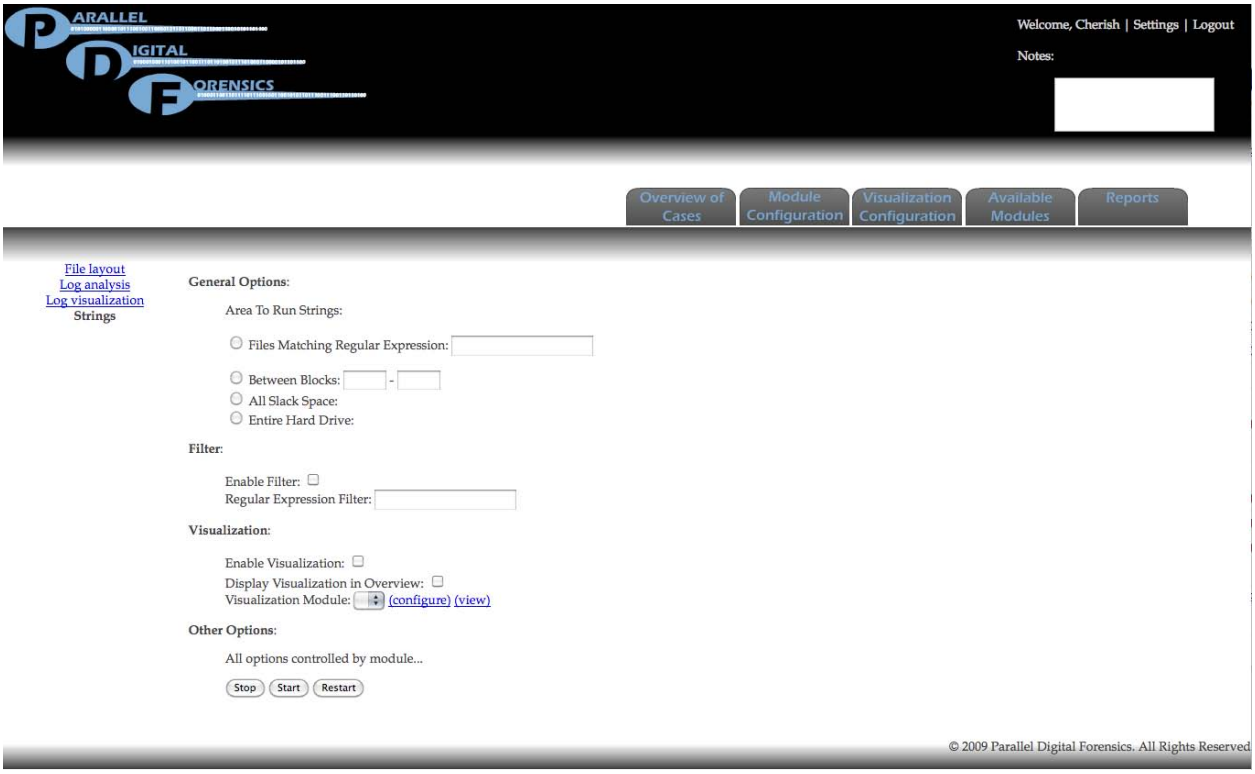

Figure 10 - Module Configuration Interface

This page was intentionally left blank.

REFERENCES

This page was intentionally left blank.

This page was intentionally left blank.

APPENDIX A: CODE MODULE DOCUMENTATION

Analyst Interface Module

==========

Module Overview:

This module provides a front-end interface to allow analyst users to carry out management of cases and personal user management. It primarily interfaces with the user and case management modules and task distribution module to allow these actions to be performed.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] action_called user_name function_specifics function_result

All actions that cause changes in the database should be logged after the database has been called.

Front-end Interface:

The interface has only one handle for the web module.
The handle is "analyst_interface", and is structured:

```
[... in version.xml for analyst interface module ...]
<interface>
   <handle dest="analyst_interface" header="analyst"
   footer="standard">analyst_interface_handle</handle>
</interface>
[... remainder of version.xml ...]
```

Module methods follow.

```
|
| analyst_interface_handle(cookie, fields)
|
| cookie -- this variable holds the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|   CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the Analyst
|  Interface is being addressed.  The presence or absence of additional,
|  action-specific fields will further determine the behavior of the method,
|  which will fit into one of three standard response modes, and one
|  special case mode:
```

| 1) Initial page view - This response mode occurs when fields indicating
| a completed form associated with the action are absent. The method
| replies with a generic, first-visit form of the appropriate page for
| the action.
| 2) Submission - This response mode occurs when a completed form has been
| submitted. The method triggers the appropriate action, and replies
| with a page indicating the success or failure of that action.
| 3) Erroneous input - this response mode occurs when some, but not all
| appropriate form data has been provided, or some input form data does
| not pass input validation. The returned page will be similar to the
| 'Initial page view" mode response, but with error messages and
| user-provided data added.
| A) Programmer error - This response mode occurs when none of the above
| three modes is appropriate, and indicates that there is a programmer
| problem, rather than a user problem.
|
| Not all actions will actually use all three traditional response modes -
| in some cases, an action will not have associated form data, and the
| 'Initial page view' and 'Erroneous input' modes do not make sense in
| those circumstances.
|
| For each call to the method, the user's cookie is checked for validity.
| In the case of an invalid cookie, the user will be redirected to the
| login page. Also, where appropriate, the Analyst's authorization for
| cases will be confirmed as well.
|
| The method supports the following contents in the 'action' field:
|_____
||None - present the home page
|_____
||'add_task' - add a new analysis task to the case.
|| Additional fields (for submission):
|| 'task_info' - a FieldStorage containing additional information
|| about the task.
|| Fields:
|| 'task_name'
|| 'module'
|| 'version'
|| 'datastore'
|| 'priority'
|| 'scheduler'
|| 'prerequisite'
|_____
||'generate_report' - retrieve a report from the database and show it to
|| the user.
|_____

||'manage_case' - display a page for controlling and adding tasks, as
|| well as generating reports.
|_____
||'pause_task' - instruct the scheduler to pause a task.
|| Additional fields (for initial page view):
|| 'task_id'
|_____
||'remove_task' - instruct case management to remove a task.
|| Additional fields (for initial page view):
|| 'task_id'
|_____
||'resume_task' - instruct the scheduler to resume a task
|| Additional fields (for initial page view):
|| 'task_id'
|_____
||'visualize' - request a visualizer's page for a given task
|| Additional fields (for initial page view):
|| 'task_id'
|| Additional fields (for submission):
|| 'task_id'
|| 'visualizer_name'

Authentication Module

==========

Overview of Module:

This module deals with all aspects of the system that needs to be authorized including user login, case login, cookies, and module authentication. Primarily this acts as an interface to the database.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] function_called user_name function_result

An example is as follows:

[2009/06/02 11:50:15] user_login John True

If the user failed to login then:

[2009/06/02 11:50:15] user_login John False

Format as will be stored in the config.xml file:

[$date] $function_called $user_name $function_result

All functions in this module should be logged accordingly.

Front-end Interface:

The interface will be designed with only one handle for the web module.
The handle will be "auth_interface", and will be structured:

```
[... in version.xml for sysadmin interface module ...]
<interface>
   <handle dest="auth_interface" header="standard"
   footer="standard">auth_interface</handle>
</interface>
[... remainder of version.xml ...]
```

Thus, a function auth_interface_handle needs to be created as follows:

```
_____________________________________________________________________
|
| auth_interface(cookie, fields)
|
| cookie -- this variable holds the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|   CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the Auth
|  Interface is being addressed.  The presence or absence of additional,
|  action-specific fields will further determine the behavior of the method,
|  which will fit into one of three standard response modes, and one
|  special case mode:
|  1) Initial page view - This response mode occurs when fields indicating
|    a completed form associated with the action are absent.  The method
|    replies with a generic, first-visit form of the appropriate page for
|                the action.
|  2) Submission - This response mode occurs when a completed form has been
|    submitted.  The method triggers the appropriate action, and replies
|    with a page indicating the success or failure of that action.
|  3) Erroneous input - this response mode occurs when some, but not all
|    appropriate form data has been provided, or some input form data does
|    not pass input validation.  The returned page will be similar to the
|    'Initial page view" mode response, but with error messages and
|    user-provided data added.
|  A) Programmer error - This response mode occurs when none of the above
|    three modes is appropriate, and indicates that there is a programmer
|    problem, rather than a user problem.
|
| On each call to a handler method, the cookie should be checked for
|   validity and that the user belongs to the sysadmin group.
|
| Not all actions will actually use all three traditional response modes -
```

```
| in some cases, an action will not have associated form data, and the
| 'Initial page view' and 'Erroneous input' modes do not make sense in
| those circumstances.
|
| The method supports the following contents in the 'action' field:
|_____________________________________________________________________
|| None - Display the login form
||  Simply displays a login form for the user to login
|_____________________________________________________________________
|| 'login' - Display the login page
||  Redirects to another page
|
| There is another interface function that creates the header for setting
|   the cookie for the client: auth_interface_headers
| auth_interface_headers(cookie, fields)
|_____________________________________________________________________
|| None - Returns a blank header
||____________________________________________________________________
|| 'login' - Check the login values
||  Form Fields:
||      'username' -- The user name as provided by the user
||      'password' -- The password as provided by the password
```

Interface Support Function:

```
_____________________________________________________________________
|
| auth_interface_login_page()
|
| Returns:
|   XHTML web page for the login interface.
```

Other Authentication Functions:

User Login:

Every time a user logs in using the web module that login information must be processed against the information stored in the database. The user's information is passed to the following function in this module:

```
_____________________________________________________________________
|
| user_login(user_name, password)
|
| user_name -- The user name as specified by the user and
|   captured by the web module.
| password -- A hashed password string specified by the user.
```

| Here is an example:
|
| user_login("Foo", "ddc35f88fa71b6ef142ae61f35364653")
|
| Returns:
| If login information matches from the database then their
| cookie values are retrieved and returned.
| If login failed then return false.

The database call is as follows:

database_module.db_authorization_user_login(user_name, password)

This should return the user's cookie values if successful.
Otherwise returns false.

User Logout:
When a user chooses to log out the user_logout function should be called.
This will terminate the session for the user.

|
| user_logout(user_name)
|
| user_name -- The user whose session will be terminated.
|
| Returns:
| True if the user was logged out
| Otherwise, an error string should be returned.

Case Authorization:
When a case is attempted to be viewed or edited the system must
ensure that the user has sufficient rights to that case. The
access control is stored in the database and must be retrieved
and confirmed. This function is defined as:

|
| case_authorization(user_name, case_id)
|
| user_name -- The user name that the user is logged into that
| the web module specifies.
| case_id -- The case id that the user is attempting to
| access.
|
| Here is an example:

|
| case_authorization("foo", 102)
|
| Returns:
| True if the user is authorized to access that case
| False otherwise.

The database call is as follows:

database_module.db_authorization_case_authorize(user_name, case_id)

This should return true if the user/case pair is found
Returns false otherwise.

Obtain List of Authorized Cases:
To give the user a selection of all authorized cases we need to retrieve
a list of all cases that they are authorized for.

|
| list_authorized_cases(user_name)
|
| user_name -- The name of the user to retrieve the list for
|
| Returns:
| A list of all case_ids that the user is authorized for
| An error message if error occurred

The database call is as follows:

database_module.authentication_list_authorized_cases(user_name)

This should return the list of case_ids that the user is authorized for.

Server Side Cookies:
In efforts to prevent unauthorized access cookies as given to the
user must be confirmed. The values for these cookies are stored
in the database. To do this the following function will be used:

|
| check_session(cookie)
|
| cookie -- The server side cookie, which must be a serialized SimpleCookie
|
| Returns:

| True if the user is logged in
| False otherwise

Also, this function must determine if the user's session has timed out. If the session has lasted for more than two hours with no activity, then the session should be terminated with user_logout and check_session should return false.

Communication Between Modules:

Communication between two modules will be encrypted via SSH tunnels. To facilitate this, every instance of a module must have a private key. The public keys generated from the private keys will be distributed from the task distribution module (via the get_file function).

For example, if the web module wants an encrypted connection to the logging module, it would call the following:

```
| td.get_file("logging", "public_key")
```

After retrieving the public key, it would need to create an SSH tunnel to that particular module.

Case Management Module

==========

Module Overview:

This module provides support to the interface modules giving access to general case management routines such as creating, modifying, and closing cases, adding and removing datastores, and listing the users assigned to a certain case. For the analysts, other tasks are needed including adding and removing tasks for a case and adding report filters.

Log Format Specification:

The log format that will be used for this module is as follows:

```
[YYYY-MM-DD hh:mm:ss] function_called case_id called_by_user
    function_specifics function_result
```

An example is as follows:

```
[2009/06/02 11:50:15] create_case 1234 John {'case_title':'Pirating'}
    True
```

Close case is the only function that does not have additional options, none should be provided instead:

```
[2009/06/02 11:50:15] close_case 1234 John none False
```

Format as will be stored in the config.xml file:

```
[$date] $function_called $case_id $called_by_user
    $function_specifics $function_result
```

All functions in this module should be logged accordingly.

Create Case:

In order to create a case all of the case information should be given at the time the function is called so that all of the information can be inserted into the database at once.

```
|
| create_case(cookie, case_details)
|
| cookie -- holds the session ID supplied to the calling interface
| case_details -- a dictionary containing all of the details of the case
|   that are to be recorded
|
| Returns:
|   The case ID if succeeded
|   Error message if failed
```

Once received the details should be processed and passed to the database module to be inserted into the database. Also the user that called this should be included in the details. The database call is as follows:

database_module.db_case_management_create_case(case_details)

Modify Case:

Cases need to have the option of being modified so that when new information is found out about the case it can be changed accordingly in the system. All aspects of the case should be able to be modified other than dates that have already passed and the name of the user that added the case.

```
|
| modify_case(cookie, case_id, case_properties)
|
| cookie -- holds the session ID supplied to the calling interface
| case_id -- the ID of the case to modify
| case_properties -- a dictionary containing the aspects of the case to
|   modify and the new values.
|
| Returns:
|   True if succeeded
|   Error message if failed
```

The case can be updated via the following database call:

database_module.scheduler_update_case(case_id, case_properties)

Close Case:

Once a case has been completed, report generated, and all issues of the case resolved it should be marked as closed. This can be done by the custodian or analyst users.

```
|
| close_case(cookie, case_id)
|
| cookie -- holds the session ID supplied to the calling interface
| case_id -- the id of the case to modify
|
| Returns:
|   True if succeeded
|   Error message if failed
```

After the session has been confirmed, mark the case as closed by calling the database module such that:

database_module.db_case_management_close_case(case_id)

Add or Remove Datastore to a Case:

When a datastore is added to the system it should be assigned to the case that it is for. Also removal of the datastore should be supported. This is done through the following functions:

```
|
| add_datastore(cookie, case, ip, port, path)
| remove_datastore(cookie, case, ip, port, path)
|
| cookie -- holds the session ID supplied to the calling interface
| case -- this is the case number that the datastore is to be or is
|   associated with.
| ip -- the IP to the machine daemon that is running for the datastore
| port -- the port on which the machine daemon for the datastore resides
| path -- the location on the datastore to the image
|
| Returns:
|   A datastore ID
```

The datastore change must be updated in the database so that all modules know of the new datastore or stop using the removed datastore. This is

done through the following call to the database module:

database_module.db_case_management_add_datastore(case, ip, port, path)
database_module.db_case_management_remove_datastore(case, ip, port, path)

Specify a Datastore Type:

Since there are several types of datastores which are each handled differently, there needs to be a function that allows the type to be assigned. This function is as follows:

```
|
| datastore_type(datastore_id, datastore_options)
|
| datastore_id -- the ID of the datastore that is returned by
|   add_datastore when created
| datastore_options -- a dictionary containing all of the datastore
|   options
```

Again the type of datastore needs to be entered into the database, this should be done such that:

database_module.db_case_management_datastore_type(datastore_id, datastore_options)

Add or Remove Case Task:

Each case can have a number of tasks that it needs to complete such as complete file carving, then searching the carvings for audio files. These tasks can be managed any time when the case is open. This will be used by the analyst. The functions should be formatted as follows:

```
|
| add_task_to_case(cookie, case_id, task_info)
| remove_task_from_case(cookie, case_id, task_info)
|
| cookie -- holds the session ID that is provided by the calling interface
| case_id -- the case number to which to add the tasks
| task_info -- a dictionary containing the details of the task to be added,
|   should contain the following keys:
|     task_name, module, version, datastore, priority, scheduler,
|     prerequisite
|
| Returns:
|   True if succeeded
|   An error message if failed
```

Add or Remove Report Filter:
For the generation of forensic reports tailored to the case a set of filters should be maintained for each case. Such filters would be when a certain type of file is found or a certain signature on the disk is found. These filters should be maintained in the database and should be added or removed through the following functions:

```
|
| add_filter_to_case(cookie, case_id, filter_dict)
| remove_filter_from_case(cookie, case_id, filter_list)
|
| cookie -- holds the session ID that is provided by the calling interface
| case_id -- the case number to which to add the tasks
| task_dict -- the dictionary of filters that are to be added, should be
|   formatted like: {'signature':'0b1c4d778e7f'} if a signature filter is
|   to be added or removed.
|
| Returns:
|   True if succeeded
|   An error message if failed:
|     If a filter in the list does not exist either in a predetermined list
|     of filters or in the list for the case (if removal) specify what
|     filters could not be added or removed and why
```

Custodian Interface Module

Overview of Module:
This module will provide a front-end interface to allow the custodian to carry out the user management, case management, and data store management actions. It will primarily interface with the user management, case management, and task distribution modules to allow these actions to be performed.

Log Format Specification:
The log format that will be used for this module is as follows:
[YYYY-MM-DD hh:mm:ss] action_called user_name function_specifics function_result

All actions that cause changes in the database should be logged after the database has been called.

Front-end Interface:
The interface will be designed with only one handle for the web module. The handle will be "custodian_interface", and will be structured:

```
[... in version.xml for custodian interface module ...]
<interface>
   <handle dest="custodian_interface" header="custodian"
   footer="standard">custodian_interface_handle</handle>
</interface>
[... remainder of version.xml ...]
```

Thus, a function custodian_interface_handle needs to be created as follows:

```
_____________________________________________________________________________
|
| custodian_interface_handle(cookie, fields)
|
| cookie -- this variable will hold the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|   CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the Custodian
|  Interface is being addressed.  The presence or absence of additional,
|  action-specific fields will further determine the behavior of the method,
|  which will fit into one of three standard response modes, and one
|  special case mode:
|  1) Initial page view - This response mode occurs when fields indicating
|    a completed form associated with the action are absent.  The method
|    replies with a generic, first-visit form of the appropriate page for
|               the action.
|  2) Submission - This response mode occurs when a completed form has been
|    submitted.  The method triggers the appropriate action, and replies
|    with a page indicating the success or failure of that action.
|  3) Erroneous input - this response mode occurs when some, but not all
|    appropriate form data has been provided, or some input form data does
|    not pass input validation.  The returned page will be similar to the
|    'Initial page view" mode response, but with error messages and
|    user-provided data added.
|  A) Programmer error - This response mode occurs when none of the above
|    three modes is appropriate, and indicates that there is a programmer
|    problem, rather than a user problem.
|
| On each call to a handler method, the cookie should be checked for
|   validity and that the user belongs to the custodian group.
|
| Not all actions will actually use all three traditional response modes -
|  in some cases, an action will not have associated form data, and the
|  'Initial page view' and 'Erroneous input' modes do not make sense in
```

```
|  those circumstances.
|
| The method supports the following contents in the 'action' field:
|_____________________________________________________________________
|| None or 'home' - present the home page
|_____________________________________________________________________
|| 'add_datastore' - Add a datastore to the system
||   This action has two pages, they will be outlined separately.
||   Page 1: Allows selection of node and path then searches for the case
||     Form Fields:
||       'node' - a combo box of all nodes in the system
||         The user should see 'node_name', 'node_ip' for each entry
||       'path' - the path to the datastore to be added, this should be on
||         the node itself
||       'case' - the name of the case to add the datastore to
||     This page directs to the second page.
||   Page 2: Gives a listing of cases that match, each has a add link
||     The add link should emulate a form with the following elements:
||       'case_id' - The id of the case of that row
||       'ip' - The ip address of the node the datastore resides on
||       'port' - The port that the machine daemon is communicating on
||       'path' - The path to the datastore
||   This should call a function from the case management module:
||     add_datastore(case_id, ip, port, path)
|_____________________________________________________________________
|| 'remove_datastore' - Remove a datastore from the system
||   This action has two pages, they will be outlined separately.
||   Page 1: Allows selection of node and datastore
||     Form Fields:
||       'node' - a combo box of all nodes in the system
||         The user should see '"node_name', 'ip"'
||       'datastore' - a combo box showing all datastores on that node
||         This needs to be updated when a new node is selected
||         The user should see '"path' for 'case_name"'
||     This page directs to the second page giving case_id, ip, port, path,
||       and confirm where confirm = False.
||   Page 2: Makes the user confirm to removing the datastore, has only
||     links Yes and No.
||     The Yes link should emulate the form with the elements:
||       'case_id' - A case that uses that datastore
||       'ip' - The ip address of the node the datastore resides on
||       'port' - The port that the machine daemon is communicating on
||       'path' - The path to the datastore
||       'confirm' - Ensuring the user has confirmed removal, should be
||         set to True
||     The No link should return the user to the previous page.
```

|| If confirmed this should call the case management module:
|| remove_datastore(case_id, ip, port, path)
|_____
|| 'datastore_type' - Change the type of a datastore
|| This action has two pages, they will be outlined separately.
|| Page 1: Allows the selection of a datastore
|| Form Fields:
|| 'node' - a combo box of all nodes in the system
|| The user should see "'node_name', 'ip'"
|| 'datastore' - a combo box showing all datastores on that node
|| This needs to be updated when a new node is selected
|| The user should see "'path' for 'case_name'"
|| This page directs to the second page giving datastore_id, ip, port
|| and path
|| Page 2: Allows the user to speicify the new type
|| Form Fields:
|| 'type' - This should show all possible types of datastores
|| This action should call the case management module:
|| datastore_type(datastore_id, datastore_options)
|_____
|| 'create_case' - Allows the user to create a case and define it initially
|| Form Fields:
|| 'case_name' - the user specified name of the case, does not have to
|| be unique
|| 'suspect_name' - the name of the suspect for the case
|| 'description' - a general description of the case
|| More fields may be necessary
|| This action should call the case management module:
|| case_management.create_case(cookie, case_details)
|_____
|| 'close_case' - Allows the user to close a currently open case
|| This action has three pages, they will be outlined separately.
|| Page 1: This allows the user to search for the case by name
|| Form Fields:
|| 'case_name' - The name of the case to search for, partial matches
|| should be shown
|| This page directs to the second page giving only case_name
|| Page 2: This allows the user to select one of the found cases
|| Each row should have a 'Close' link associated to the case_id for
|| that row, these links should emulate a form with the elements:
|| 'case_id' - the id of the case to close
|| 'confirm' - confirmation from the user, should be set to False
|| This page directs to the third page giving case_id and confirm
|| Page 3: This makes the user confirm to close the case, has only links
|| Yes and No.
|| The Yes link should emulate a form with the elements:

|| 'case_id' - the id of the case to close
|| 'confirm' - confirmation from the user, should be set to True
|| The No link should take the user back to the previous page
|| If confirmed this should call the case management module:
|| close_case(cookie, case_id)
|---
|| 'edit_case' - Allows the user to edit most aspects of a case, start_date
|| cannot be modified
|| This action has three pages, they will be outlined separately.
|| Page 1: This allows the user to search for the case by name
|| Form Fields:
|| 'case_name' - The name of the case to search for, partial matches
|| should be shown
|| This page directs to the second page giving only case_name
|| Page 2: This allows the user to select one of the found cases
|| Each row should have a 'Edit' link associated to the case_id for
|| that row, these links should emulate a form with the elements:
|| 'case_id' - the id of the case to edit
|| This page directs to the third page giving only case_id
|| Page 3: This allows the user to make changes to the selected case
|| Form Fields:
|| 'case_id' - the id of the case to edit
|| 'case_name' - the user specified name of the case, does not have to
|| be unique
|| 'suspect_name' - the name of the suspect for the case
|| 'description' - a general description of the case
|| More fields may be necessary
|| This action should call the case management module:
|| modify_case(cookie, case_id, case_properties)
|---
|| 'case_add_user' - Allows the user to add an analyst to a case
|| This action has three pages, they will be outlined separately.
|| Page 1: This allows the user to search for the case by name
|| Form Fields:
|| 'case_name' - The name of the case to search for, partial matches
|| should be shown
|| This page directs to the second page giving only case_name
|| Page 2: This allows the user to select one of the found cases
|| Each row should have a 'Add' link associated to the case_id for
|| that row, these links should emulate a form with the elements:
|| 'case_id' - the id of the case to add the analyst to
|| This page directs to the third page giving only case_id
|| Page 3: This allows the user to add an analyst to the selected case
|| Form Fields:
|| 'user_name' - the name of the analyst to be added to the case
|| This action should call the user management module:

|| add_user_to_case(cookie, user_name, case_id)
|_____
|| 'case_remove_user' - Allows a user to remove an analyst from a case
|| This action has three pages, they will be outlined separately.
|| Page 1: This allows the user to search for the case by name
|| Form Fields:
|| 'case_name' - The name of the case to search for, partial matches
|| should be shown
|| This page directs to the second page giving only case_name
|| Page 2: This allows the user to select one of the found cases
|| Each row should have a 'Remove' link associated to the case_id and
|| user_name for that row, these links should emulate a form with the
|| elements:
|| 'user_name' - the name of the analyst to remove from the case
|| 'case_id' - the id of the case to close
|| 'confirm' - confirmation from the user, should be set to False
|| This page directs to the third page giving user_name, case_id and
|| confirm
|| Page 3: This makes the user confirm to removing the user, has only
|| the links Yes and No.
|| The Yes link should emulate a form with the elements:
|| 'user_name' - the name of the analyst to remove from the case
|| 'case_id' - the id of the case to close
|| 'confirm' - confirmation from the user, should be set to True
|| The No link should take the user back to the previous page
|| If confirmed this should call the user management module:
|| remove_user_from_case(cookie, user_name, case_id)
|
| Returns:
| An XHTML fragment to be displayed to the user.

o Case management actions
   This custodian will be able to modify the settings of a given
   case that the custodian has control over. This includes
   adding a user to the case, removing a user from the case,
   closing a case, classifying a case and creating a new case.

   To retrieve the list of users in a case, the following function can be
   used:
      | case_management.list_users(cookie, case_id)
      |
      | Where cookie holds the session ID and case_id is the case ID
      |
      | This will return a tuple of user names

Database Module

==========

Overview of Database Module:

This module provides a layer of abstraction to the database. Instead of modules using SQL to query the database, they will call functions of this module which will then construct SQL queries and run them on the server.

This will centralize the database calls allowing for major changes to the database to occur without causing the need for major changes to PDF.

The structure of the functions within this module will be as follows:

```
_____________________________________________________________________
|
| db_module_name_query_name(var1, var2, ...)
|
| var1, var2, etc are all variables that will be used in the query.
|
| Returns:
|   These methods return a dictionary with two keys: "return" and "error".
|     "return" contains the return values.  A return value of False may
|       indicate an error condition.
|     "error" contains False if no error message is appropriate, and error
|       messages otherwise.
```

Specific Database Calls:

Analyst Interface Functions:

```
_____________________________________________________________________
|
| analyst_get_case_id(session_key)
|
| session_key - the session key found in the cookie for this session
|
| Returns:
|   the case ID currently associated with the session.  False indicates
|     that no case has been selected.
```

```
_____________________________________________________________________
|
| analyst_get_case_info(case_id)
|
| case_id - The ID of the case whose information should be retrieved.
|
| Returns:
```

| A dictionary containing information about the case in the following
| keys:
| 'id' - the case ID
| 'name' - the case name
| 'description' - the case description
| 'createdBy' - the name of the Custodian that created the case
| The following error messages may be returned:
| "Could not find a matching case." - Indicates that the supplied
| case ID is not currently in use.

|
| analyst_get_task_info(task_id)
|
| task_id - The ID of the task whose information should be retrieved.
|
| Returns:
| A dictionary containing information about the task in the following
| keys:
| 'id' - the task ID
| 'case' - the case the task is connected to
| 'owner' - the Analyst that launched the task
| 'scheduler' - the name of the scheduling algorithm used
| 'name' - the name of the task
| 'TDName' - the name of the Task Definition the task came from
| 'version' - the version of the Task Definition the task came from
| 'start' - a string representing the date and time at which the
| task was launched
| 'end' - a string representing the date and time at which the
| task finished, which may be empty
| 'metadata' - a string containing configuration information for the
| task
| 'status' - the current execution status of the task. Will be one
| of: "running", "finished", "paused", or "queued"
| 'nodes' - an integer containing the number of nodes that have
| been allocated to the task
| The following error messages may be returned:
| "Could not find a matching task." - indicates that the supplied
| task ID is not currently in use.

|
| analyst_set_case_id(session_key, case_id)
|
| session_key - the session key found in the cookie for this session
| case_id - the ID of the case to be associated with the session

```
|
| Returns:
|   True upon successful update, False otherwise.  False may indicate
|     that the new ID was the same as the old one.
```

Authentication Database Functions:

```
_____________________________________________________________________________
|
| authentication_add_cookie(cookie)
|
| cookie - a dictionary containing the following fields:
|          'user' - the name of the user the cookie belongs to
|          'sessionKey' - a key that uniquely identifies the session
|          'openTime' - the date and time at which the session began
|          'accessTime' - the date and time of the most recent activity
|                         for the session
|
| Returns:
|   True if the cookie was successfully added.
|   The following error messages may be returned:
|     "The user already has an open session." - the cookie could not be
|       added because the user already has a cookie
|     "The specified user does not exist." - the cookie could not be
|       added because the supplied user name does not match any
|       existing user

_____________________________________________________________________________
|
| authentication_case_authorization(user_name, case)
|
| user_name - a string containing the name of a user
| case - the ID of a case
|
| Returns:
|   True if the user is authorized for the case, False otherwise

_____________________________________________________________________________
|
| authentication_list_authorized_cases(user_name)
|
| user_name - a string containing the name of a user
|
| Returns:
|   A list containing the IDs of the cases for which the user is
|     authorized
```

```
|
| authentication_remove_cookie(cookie)
|
| cookie - a dictionary containing at least the key 'sessionKey' whose
|          associated value is a session key
|
| Returns:
|   True if the cookie was removed.
|   The following error messages may be returned:
|     "No matching cookie was found." - the cookie was not deleted
|                                       because the supplied session
|                                       key is not in use
```

```
|
| authentication_retrieve_cookie(info)
|
| info - a dictionary containing at least one of the following keys:
|        "user_name" - a string containing the name of a user, or None
|         "session_key" - a session key, or None
|
| Returns:
|   A dictionary containing the following keys:
|     'user' - the name of the user the session belongs to
|     'sessionKey' - the session key
|     'openTime' - the date and time the session was opened
|     'accessTime' - the date and time of the most recent activity for
|                    the session
|
|   The following error messages may be returned:
|     "Could not find a matching cookie." - no cookie was returned
|                                           because the supplied user
|                                           name or session key is not
|                                           in use
```

```
|
| authentication_retrieve_password(user_name)
|
| user_name - the name of the user whose password is to be retrieved
|
| Returns:
|   The requested password hash, if successful
|   The following error messages may be returned:
```

```
|    "Could not find user." - the supplied user name is not in use
```

```
|
| authentication_update_cookie_access(cookie, new_time)
|
| cookie - a dictionary containing at least the key "sessionKey" where
|         the value is the session key of the cookie whose access time
|          should be updated
| new_time - a string containing the new date and time of the most
|          recent activity for the session
|
| Returns:
|   True if successfully updated
|   The following error messages may be returned:
|     "No matching cookie was found." - no cookie was updated because
|                                the supplied session key is not
|                                in use
```

Case Management Database Functions:

```
|
| case_management_add_datastore(case_id, ds_name)
|
| case_id - the ID of the case to which the data store will be added
| ds_name - the name of the data store
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                       is most likely due to a primary key
|                       constraint, but it really shouldn't happen.
|     "Case does not exist." - Indicates that the supplied case ID is
|                         not currently in use.
|     "Data store does not exist." - Indicates the supplied data store
|                               name is not currently in use.
```

```
|
| case_management_add_task(case, owner, scheduler, name, td,
|                         version, start, metadata, status, nodes)
|
| case - the ID the case to which this task is to be added
```

```
| owner - the name of the Analyst creating the task
| scheduler - the name of the scheduling module to be used when
|           distributing work for the case
| name - the name of the task
| td - the name of the Task Definition the task is created from
| version - the version of the Task Definition the task is created from
| start - the date and time at which the task was created
| metadata - configuration information for the task
| status - one of the following: "running", "finished", "paused",
|         "queued"
| nodes - the number of nodes to be allocated to the task
|
| Returns:
|   The ID of the new task if successful
|   The following error messages may be returned:
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                           is most likely due to a primary key
|                           constraint, but it really shouldn't happen.
|     "Case does not exist."- Indicates that the supplied case ID is
|                             not currently in use.
|     "Creator does not exist." - the user named does not exist
|     "Scheduler does not exist." - the named scheduling module does not
|                                   exist
|     "Task definition does not exist." - no task definition with the
|                                         given name and version number
|                                         exists
|     "Invalid status." - the given status is not one of the allowed
|                         statuses

_____________________________________________________________________

|
| case_management_change_description(id, new_description)
|
| id - the ID of the case to be updated
| new_description - the new description of the case
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Update failed." - the given ID didn't match any case

_____________________________________________________________________

|
| case_management_change_name(id, new_name)
|
| id - the ID of the case to be updated
```

| new_name - the new name of the case
|
| Returns:
| True if successful
| The following error messages may be returned:
| "Update failed." - the given ID didn't match any case

|
| case_management_close_case(case)
|
| case - the ID of the case to be closed
|
| Returns:
| True

|
| case_management_create_case(case_info)
|
| case_info - a dictionary containing information about the case in the
| following fields:
| 'name' - the name of the case
| 'description' - a description of the case
| 'created_by' - the name of the Custodian that created the case
| 'creation_time' - the date and time the case was created
|
| Returns:
| The new case's ID if successful
| The following error messages may be returned:
| "Creator is not of the appropriate type." - the user named is not
| a Custodian
| "Creator does not exist." - the user named does not exist

|
| case_management_list_datastores(case)
|
| case - the ID of the case whose data stores should be listed
|
| Returns:
| A list of dictionaries, where each dictionary contains information
| about a datastore attacked to the case. Three types of
| dictionaries will appear in the list - File, Filesystem, Database.
| File keys:
| 'name' - the name of the file (not a path)

| 'path' - the path to the file
| Filesystem keys:
| 'name' - the name of the filesystem
| 'device' - the device which holds the filesystem
| Database keys:
| 'name' - the name of the database
| 'catalogue' - the name of the catalogue
| 'ip' - the IP address of the database
| 'port' - the port number of the database

|
| case_management_remove_datastore(case_id, ds_name):
|
| case_id - the id of the case from which the data store should be
| removed
| ds_name - the name of the data store to be removed
|
| Returns:
| True if successful
| The following error messages may be returned:
| "Deletion failed." - Indicates the supplied case/data store pair
| wasn't made in the first place.

|
| case_management_remove_task(id)
|
| id - the ID of the task to be removed
|
| Returns:
| True if successful
| The following error messages may be returned:
| "Deletion failed." - the supplied task ID is not in use

Notebook Database Functions:

|
| note_add_note(user, note, date, case_id, note_id, corrects)
|
| user - the user_name of the user doing the action
| note - the contents of the note to be inserted
| date - the current datestamp for the note
| case_id - the id of the case the note belongs to
| note_id - the id of the note that the adding note is a reply to, if

```
|          this note is not a reply this will be -1 corrects - the id
|          of the note that the adding note is correcting, if this
|          note is not a correction this will be -1 returns True on
|          success, False on failure
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                                 is most likely due to a primary key
|                           constraint, but it really shouldn't happen.
|     "Poster does not exist." - the named user could not be found.
|     "Case does not exist." - the identified case could not be found.
|     "A note cannot be a reply and a correction." - both note_id and
|                                                    corrects were
|                                                    supplied with
|                                                    non-negative values
```

```
|
| note_get_case_notes(case_id)
|
| case_id - the ID of the case for which notes should be retrieved
|
| Returns:
|   A list of dictionaries, each representing a note, where each
|     contains the following keys:
|     'id' - the ID of the note
|     'content' - the text contents of the note
```

```
|
| note_get_note(case_id, note_id)
|
| case_id - the id of the case that the note belongs to
| note_id - the id of the note to get information on
|
| Returns:
|   A dictionary containing the following keys:
|   'id' - the ID of the note
|   'poster' - the name of the user that posted the note
|   'content' - the text contents of the note
|   'date' - the date and time at which the note was posted
|   'isStruckOut' - a boolean which is the evaluation of the statement
|                   "This note has been struck out."
|   'postedTo' - the ID of the case to which this note was posted
```

```
|   'repliesTo' - the ID of the note to which this note is a reply
|   'corrects' - the ID of the note to which this note is a correction
```

```
|
| note_get_sub_notes(case_id, note_id)
|
| case_id - the id for the case to retrieve a list of sub notes for
| note_id - the id of the note to retrieve the sub notes for
|
| Returns:
|    A list of dictionaries, each representing a note, where each
|     contains the following keys:
|     'id' - the ID of the note
|     'content' - the text contents of the note
```

```
|
| note_strike_out(case_id, note_id, user)
|
| case_id - the id for the case to retrieve a list of sub notes for
| note_id is the id of the note to retrieve the sub notes for
|
| Returns:
|   True, if successful
```

Reporting Database Functions:

```
|
| reporting_insert_report_entry(date, content, case, user)
|
| date - a string containing the date and time the entry was made
| content - a string containing the message the report carries
| case - the ID of the case for which the report is made, or False
| user - the name of the user which this report is related to, or False
|
| Returns:
|   True if the report is successfully stored
|   The following error messages may be returned:
|     "Referenced case does not exist." - the supplied case ID is not in
|                                   use
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                           is most likely due to a primary key
```

```
|                    constraint, but it really shouldn't happen.
```

```
|
| reporting_retrieve_report(start, end, case, user)
|
| start - a string containing the date and time to begin retrieving
|        reports from
| end - a string containing the date and time to stop retrieving reports
|       from
| case - the ID of the case retrieved reports should be associated with,
|        or None if this should be ignored
| user - the name of the user for which reports should be retrieved, or
|        False
|
| Returns:
|   A list of dictionaries containing information about the reports
|     retrieved using the criteria specified, using the following keys:
|     'id' - the report ID
|     'content' - the message the report contains
|     'timestamp' - the date and time the report was made
|     'caseID' - the ID of the case the report is associated with
```

Scheduler Functions:

```
|
| scheduler_change_task_owner(task_id, new_owner)
|
| task_id - the ID of the task whose owner is to be changed
| new_owner - the name of the Analyst that is to be the new owner
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Could not find the matching task." - the supplied task ID is not
|                                  in use
```

```
|
| scheduler_get_task_datastores(task_id)
|
| task_id - the ID of the task for which to retrieve a list of data
|           stores
|
| Returns:
```

```
|   A list containing the names of all the data stores associated with
|     the task
|   The following error messages may be returned:
|     "Could not find the matching task." - the supplied task ID is not
|                                  in use

_____________________________________________________________________________
|
| scheduler_retrieve_case_tasks(case_id)
|
| case_id - the ID of the case for which to retrieve a list of tasks
|
| Returns:
|   A list of dictionaries containing information about the tasks
|     associated with the case, using the following keys:
|     'id' - the task ID
|     'case' - the case the task is connected to
|     'owner' - the Analyst that launched the task
|     'scheduler' - the name of the scheduling algorithm used
|     'name' - the name of the task
|     'TDName' - the name of the Task Definition the task came from
|     'version' - the version of the Task Definition the task came from
|     'start' - a string representing the date and time at which the
|               task was launched
|     'end' - a string representing the date and time at which the
|       task finished, which may be empty
|     'metadata' - a string containing configuration information for the
|                  task
|     'status' - the current execution status of the task.  Will be one
|                of: "running", "finished", "paused", or "queued"
|     'nodes' - an integer containing the number of nodes that have
|               been allocated to the task

_____________________________________________________________________________
|
| scheduler_retrieve_task(task_id)
|
| task_id - The ID of the task whose information should be retrieved.
|
| Returns:
|   A dictionary containing information about the task in the following
|     keys:
|     'id' - the task ID
|     'case' - the case the task is connected to
|     'owner' - the Analyst that launched the task
|     'scheduler' - the name of the scheduling algorithm used
```

```
|    'name' - the name of the task
|    'TDName' - the name of the Task Definition the task came from
|    'version' - the version of the Task Definition the task came from
|    'start' - a string representing the date and time at which the
|           task was launched
|    'end' - a string representing the date and time at which the
|          task finished, which may be empty
|    'metadata' - a string containing configuration information for the
|             task
|    'status' - the current execution status of the task.  Will be one
|            of: "running", "finished", "paused", or "queued"
|    'nodes' - an integer containing the number of nodes that have
|           been allocated to the task
|  The following error messages may be returned:
|    "Could not find a matching task." - indicates that the supplied
|                                task ID is not currently in
|                                use.
```

```
|
| scheduler_retrieve_task_definition(name, version)
|
| name - the name of the Task Definition to be retrieved
| version - the version number of the Task Definition to be retrieved
|
| Returns:
|  A dictionary containing information about the Task Definition,
|    using the following keys:
|    'name' - the name of the Task Definition
|    'version' - the version number of the Task Definition
|    'metadata' - information about how the modules that compose the
|             task work together
|    'inputType' - what kind of data store the task reads from (file,
|              filesystem, or database)
|    'outputDB' - the name used to reference the database the task
|             stores its results in
```

```
|
| scheduler_task_end_time(task_id, time)
|
| task_id - the ID of the task whose completion time is to be set
| time - the date and time the specified task was completed
|
| Returns:
|  True if successful
```

```
|   The following error messages may be returned:
|     "Could not find the matching task." - no task end time was updated
|                                  because the supplied task
|                                  ID is not in use
```

User Management Database Functions:

```
_____________________________________________________________________________
|
| user_management_add_user_to_case(user_name, case_id)
|
| user_name - the name of the user to be added
| case_id - the ID of the case to add the user to
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                       is most likely due to a primary key
|                       constraint, but it really shouldn't happen.
|     "Specified user does not exist." - the supplied user name is not
|                                  in use
|     "Specified case does not exist." - the supplied case ID is not in
|                                  use

_____________________________________________________________________________
|
| user_management_change_password(user, password)
|
| user - the name of the user whose password is to be changed
| password - the new password for the named user
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Could not find the matching user." - the supplied user name is
|                                  not in use

_____________________________________________________________________________
|
| user_management_change_type(user, new_type)
|
| user - the name of the user whose type is to be changed
| new_type - the new type of the named user
|
| Returns:
```

```
|   True if successful
|   The following error messages may be returned:
|   "Could not find the matching user." - the supplied user name is not
|                                  in use
|   "Specified user type does not exist." - the new supplied user type
|                                    is not known
```

```
|
| user_management_create_user(user)
|
| user - a dictionary containing information about the user to be
|        created in the following keys:
|         'name' - the name of the user
|         'password' - a hash of the user's password
|         'idKey' - the user's identification key
|         'userType' - the user's type
|         'createdBy' - the name of the user that created this one
|         'creationTime' - the date and time of the user's creation
|
| Returns:
|   True if successful
|   The following error messages may be returned:
|     "Insertion failed." - the database reported 0 rows inserted.  This
|                           is most likely due to a primary key
|                           constraint, but it really shouldn't happen.
|     "Specified user type is not allowed." - the supplied user type is
|                                     not known
|     "That user name is already in use." - Some other user already has
|                                    the supplied name
|     "Creator does not exist." - the supplied creating user is not
|                              known
```

```
|
| user_management_find_user(user, type, mode)
|
| user - the beginning of the name of the user to find
| type - the type of user to look for (this field is ignored if it does
|        not match one of the known user types)
| mode - must be one of 'any', 'end', 'exact', 'front', to indicate
|        where in returned user names the given string should appear
|
| Returns:
|   A list of dictionaries containing information about the relevant
|     users, using the following keys:
```

| "user_name" - the name of the user
| "user_type" - the type of the user

|
| user_management_remove_user(user_name)
|
| user_name - the name of the user to be removed
|
| Returns:
| True if successful
| The following error messages may be returned:
| "Could not find a matching user." - the supplied user name is not
| in use

|
| user_management_remove_user_from_case(user_name, case_id)
|
| user_name - the name of the user to be removed from the case
| case_id - the ID of the case the user is to be removed from
|
| Returns:
| True if successful
| The following error messages may be returned:
| "Could not find a matching authorization." - the named user isn't
| authorized for the
| specified case.

Logging Module

==========

Overview of Logging Module:

Logging is a fundamental part of digital forensics. The logging module will provide the capability to log all events that occur on the PDF system to disc as flat "log" files. Similar to logging in other programs, log rotation will not occur from within the logging module.

When an event occurs in PDF, that event should be logged. Each module will provide its own particular logging format string, and the logging module will write the compiled format string to that module's log file.

A distinction must be drawn between logging and reporting in PDF Reporting is compiling particular log entries, as well as notes

taken by the analyst, into a presentable format. This occurs within the reporting module. Logging, on the other hand, will log both the events that the reporting module will use as well as system events, such as program crashes, invalid access attempts, etc.

This module relies heavily upon XMLRPC, using it to both accept logs and determine information about how to log.

Retrieving Initial Variables:

The first step for the logging module is to determine a few of the variables that it is going to need.

The logging module needs to know:

- Which IP address/port to listen on
- Which directory to log to

These can be found by communicating, via XMLRPC, with the task distribution module. The location of that module will be given as command line arguments as such:

--td_ip <ip address> --td_port <port>

To determine configuration options, the following function should be used to communicate with the task distribution module:

```
_____________________________________________________________________
|
| get_configuration(module_name, (settings_list))
|
| module_name -- This is the name of the module you wish to retrieve
|   settings from
| settings_list -- This is a comma separated list of the settings to
|   retrieve.  For example,
|
| get_configuration("logging", ("ip", "port", "log_dir"))
```

Retrieving Format Strings:

To determine how to log for a particular module, a format string needs to be obtained from the task distribution module. This should occur similarly to above. The following XMLRPC command should be given:

get_configuration(module_name, ("log_syntax"))

This could return, for example,

[Auth failure] $remote_ip $username

Format Strings:

Format strings will be Templates in Python. This will allow for variable substitution as strings, and more complicated formatting can occur in the variables being passed to the logging module.

File Use:

Due to frequent writing to the log file handles should be maintained throughout periods of frequent writing. In effort to do this two functions will be needed:

```
|
| open_log(module_name)
| close_log(module_name)
|
| module_name -- The name of the module to open a log file for
```

The individual file handles should be maintained within a dictionary variable named file_handles and should be declared such that:

{"module_name": "file handle"}

The function open_log should ensure that a log is not currently opened for that module and that the log directory exists before attempting to open the log.

The function close_log should ensure that a log is currently opened for the module.

Receiving Logs:

The logging module must have a function, which is being listened for through XMLRPC, called "log". Its format should be as follows:

```
|
| log(module_name, log_variables)
|
| module_name -- The name of the module to log for
| log_variables -- A dictionary containing all variables needed for
|   logging. This dictionary will contain keys that are found in the
|   format string retrieved by get_configuration.
```

When this function is called via XMLRPC, the data in log_variables must be written to a flat file that is in the log_dir found above. The file should be appended and the data in log_variables must be

written out according to the format string. The file should be named "module_name.log". Before opening the file, ensure that there is no current file handle on it by checking in the file_handle dictionary variable. Assuming a handle is found simply write the processed string. If no handle exists proceed to open the file through:

open_log(module_name)

Then after the log has been written close the file with:

close_log(module_name)

Reporting:

To determine if a particular log needs to be sent to the reporting module, the following XMLRPC command should be given (again, to the task distribution module):

get_configuration(module_name, ("report_log"))

If the result is 1, then this log should be reported. Otherwise, this log should not be reported.

To report this particular log, an XMLRPC connection must be made to the reporting module. The address of the reporting module can be determined, similarly to above, by contacting the task distribution module.

Then, the following XMLRPC command should be sent to the reporting module:

```
|
| report(module_name, log_variables)
|
| The same variables used as inputs for the log function will be
| used.
```

Machine Daemon Module

Overview of Module:

This module is necessary for security purposes so that only this module will be run with root privileges. It will give such access as adding and removing files to the general file system, starting and stopping services and modules, also stopping and starting tasks for cases. This API is under construction.

Log Format Specification:
The log format that will be used for this module is as follows:
[YYYY-MM-DD hh:mm:ss] function_called module_name function_result

An example is as follows:
[2009/06/02 11:50:15] start_module task_distribution True
If the module failed to start then:
[2009/06/02 11:50:15] start_module task_distribution False

Format as will be stored in the config.xml file:
[$date] $function_called $module_name $function_result

All functions in this module should be logged accordingly

Log:
To aid all of the functions in the machine daemon module to log the results.

```
|
| log(function, result)
|
| function -- The name of the function to log for
| result -- The result of the function execution
|
| Returns:
|   True if the log was successful
|   False if an error occurred
```

Response:
Aids the functions in the machine daemon module to return the proper return format.

```
|
| response(error, result)
|
| error -- The error response section of the return, False by default
| result -- The result response section of the return, False by default
|
| Returns:
|   A dictionary formatted as follows:
|       {'error':error,'return':result}
```

Start Module:
For the addition and upgrade of modules the new module must be started. This will be called as follows:

```
|
| start_module(module_name)
|
| module_name - The name of the module that is to be started
|
| Returns:
|   True if the module has been started
|   An error message if the module has not been started
```

Stop Module:

For the upgrade or removal of a module the old module must be stopped first. This will be called as follows:

```
|
| stop_module(module_name)
|
| module_name - The name of the module that is to be started
|
| Returns:
|   True if the module has be stooped
|   An error message if the module has not been stops
```

Copy Module:

When adding and upgrading modules, new files need to be copied from the download location into the proper place. This will be called as follows:

```
|
| copy_module(module_name, source_path)
|
| module_name - The name of the module that files are being copied for.
|   The module's directory should be determined based on this.
| source_path - The path to the files that have been downloaded and
|   extracted,  the current means available
|   are:
|       o {"string": "actual containing directory"}
|       o {"nfs": "server:/path"}
|
| Returns:
|   True if the files were copied successfully
|   An error message otherwise
```

Delete Module:

When upgrading and removing modules the old files must be deleted. This will be called as follows:

```
|
| delete_module(module_name)
|
| module_name - The name of the module to be deleted
|   The module's directory should be determined based on this.
|
| Returns:
|   True if the files were copied successfully
|   An error message otherwise
```

Start Task:

In order to complete analysis upon a case each task must be started. Also to give the ability to pause and stop analysis each task must also have the option to be stopped at any time. In order to keep track of the running processes the following should be cached in a dictionary:

{("module_name", "path"):pid}

Where module_name is the name of the analysis module, path is the path to the files that should be analyzed, and pid is the process ID for the running task.

```
|
| start_task(module_name, config_list)
| stop_task(module_name, config_list)
|
| module_name - The name of the module that the task is running from
|   The module's directory should be determined based on this.
| config_list - The original command options given to the module when
|   the task was started.
|
| This also needs to mount the nfs share of the 'path' that is given.
|
| Returns:
|   True if the task was started/stopped correctly
|   An error message otherwise
```

Setting up and Removing NFS Shares:

The machine daemon will need to establish NFS shares that other modules require. These are used for inter-module communication or for access to datastores.

```
|
| create_nfs(module_name)
| remove_nfs(module_name)
|
| module_name - The name of the module requesting that a nfs share be
```

| either created or removed.
|
| Returns:
| Dictionary containing the keys nfs_share (the full nfs path) and
| local_path (containing the full local path to the shared directory)
| An error message if it failed

Mount NFS Share:

The machine daemon needs to mount NFS shares for other modules so that the information they desire can be accessed. This will be done by mounting the share to a temporary place.

|
| mount_nfs(module_name, nfs_share)
|
| module_name - The name of the module requesting a nfs share to be
| mounted
| nfs_share - The full path of the NFS share that is desired.
| This needs to contain server://path
|
| This should use mount -t nfs $nfs_share /mount/$module_name_$rand_number
| Where rand_number is a randomly generated number
| It should also be made sure that the path is not being used already
|
| Returns:
| A string containing the local path to the mounted share
| An error message if it failed

Export NFS Share:

Some of the NFS shares should be exported so that other nodes have access to them.

|
| export_nfs(path_to_share, host)
|
| path_to_share - The local path of the NFS share to export
| host - The name of the node that the share should be exported for
|
| This should write "$path_to_share $host(ro)" to /etc/exports
| then execute /etc/init.d/nfs-kernel-server reload (or equivalent)
|
| Returns:
| True if the share was exported correctly
| An error message otherwise

Create Add Node Script:

For a node to be added to the system it first needs some of the modules to be installed upon it and make sure that it is registered.

| generate_script()
|
| This will create a script that will download all necessary modules to
| the node, run the currently needed modules, and register itself as a
| node on the system. Once the script is complete it should display the
| node's public key to the user so that they may enter it into the
| sysdamin interface.

Removing Node from the System:

When a node should be removed from the system the machine daemon module running on that node will be called, will remove its information from the task distribution module then clean up the files on the node for removal.

| remove_node()
|
| This will run a series of tasks which will find its node id from task
| distribution then remove all associated information by set_configuration
| and setting the information to be blank. Also all files that were
| installed by the add node script should be deleted.

Build SSH Tunnel:

In order to allow communications to be secure between nodes SSH tunnels should be established by the machine daemon.

| build_ssh_tunnel(destination_ip, destination_module, destination_module_version)
|
| desitination_node_ip - The ip of the node that the requesting module desires to communicate securely with.
|
| This needs to get the listening SSH port of the destination from the
| Task Distribution Module by:
| TD.get_configuration(destination_module destination_module_version, '$destination_ip_port')
|
| Returns:
| The details on the ssh tunnel so that it may be used by the node
| ip, port to connect to the tunnel
| An error if creating the tunnel failed

Notes Interface Module

Overview of Module:

This module will provide a front-end interface to allow analyst and custodian users to comment upon various aspects of particular cases or all cases at once. It will primarily interface with the database module.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] action_called user_name function_specifics function_result

The function specifics will contain the following based on action_called

give_strike: case_id,note_id

add_note: if the note added was a reply or correction

Only add_note and give_strike should be logged and only logged after the database has been called.

Front-end Interface:

The interface will be designed with only one handle for the web module. The handle will be "notes_interface", and will be structured:

```
[... in version.xml for notes interface module ...]
<interface>
   <handle dest="notes_interface" header="notes"
   footer="standard">notes_interface_handle</handle>
</interface>
[... remainder of version.xml ...]
```

Thus, a function notes_interface_handle needs to be created as follows:

```
_____________________________________________________________________________
|
| notes_interface_handle(cookie, fields)
|
| cookie -- this variable will hold the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|   CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the Sysadmin
|  Interface is being addressed.  The presence or absence of additional,
|  action-specific fields will further determine the behavior of the method,
|  which will fit into one of three standard response modes, and one
|  special case mode:
|  1) Initial page view - This response mode occurs when fields indicating
|    a completed form associated with the action are absent.  The method
```

```
|   replies with a generic, first-visit form of the appropriate page for
|             the action.
| 2) Submission - This response mode occurs when a completed form has been
|   submitted.  The method triggers the appropriate action, and replies
|   with a page indicating the success or failure of that action.
| 3) Erroneous input - this response mode occurs when some, but not all
|   appropriate form data has been provided, or some input form data does
|   not pass input validation.  The returned page will be similar to the
|   'Initial page view" mode response, but with error messages and
|   user-provided data added.
| A) Programmer error - This response mode occurs when none of the above
|   three modes is appropriate, and indicates that there is a programmer
|   problem, rather than a user problem.
|
| On each call to a handler method, the cookie should be checked for
|  validity and that the user belongs to either the custodian or
|  analyst groups and that the user is authorized to view the specified
|  case.
|
| Not all actions will actually use all three traditional response modes -
|  in some cases, an action will not have associated form data, and the
|  'Initial page view' and 'Erroneous input' modes do not make sense in
|  those circumstances.
|
| The method supports the following contents in the 'action' field:
|_____________________________________________________________________
|| None or 'home' - present the home page
||
|| This home page should give a listing of cases that the user is
||  authorized for.
|_____________________________________________________________________
|| 'manage_notebook' - display a page for managing the notes associated with
||  the given case
|| This page gives a listing of all notes for the case including the
||  following:
||    Creation Date
||    Number of Files in Note
||    The first 30-50 characters of the annotation
||    A link to view the note
|_____________________________________________________________________
|| 'view_note' - display the contents of a specified note
|| This page gives the details of a desired note, it has the following
||  non-user editable fields:
||    Date Posted -- The date that the note was created
||    View File -- A combo box listing all of the files associated with
||      the note, if one is selected the file should be viewed
```

|| If the file is to be viewed it will need to be placed in a web
|| accessible folder
|| Annotation -- The contents of the note
|| Links to add an annotation to this note and to add a file to this note
|| should be provided.
|_____
|| 'add_note' - give an interface that the user can use to add a note
|| This page allows the user to input a note into the system, it has the
|| following field:
|| Note -- A large textarea for note content
|| Also if the note is a follow up to another these fields will appear:
|| Reply -- A radio field linked to correction, shows the type of note
|| Correction -- A radio field linked to reply
|_____
|| 'give_strike' - interface allowing the user to strike out a note
|| This page allows the user to strike out a note indicating that something
|| was wrong with the note and a correction should be referenced. This
|| page gives the user a simple yes/no confirmation page.

Reporting Module

==========

Overview of Module:

This module works closely with the logging module and produces a forensics report upon the settings established for the specific case. All of the entries captured by the reporting module will be entered into the database to be retrieved at a later time. This should also be able to generate reports based upon the entries that were captured. These entries come directly from the logging module as each time a log entry is made the reporting module is queried and decides if that entry is significant to the case at hand.

Save Report Entry:

The capture function in this module is called regularly by the logging module. Once called it must query the database to find what log entries should be saved for the report and which should only be retained by the machine's log. If the entry should be saved for the report then that entry is placed into the database under that case name. The function is defined as follows:

_____
|
| report(module_name, log_variables)
|
| module_name -- Again the name of the module to log for
| log_variables -- A dictionary containing all variables needed for

| reporting, this dictionary will contain keys that are found in the
| format string retrieved by get_configuration
|
| Nothing is to be returned by this function, since it is independent.

In this function if the entry is to be saved the following database function needs to be called:

```
|
| database_module.insert_report_entry(date, log_entry)
| database_module.insert_report_entry(date, case_id, log_entry)
| database_module.insert_report_entry(date, user_name, log_entry)
| database_module.insert_report_entry(date, case_id, user_name, log_entry)
|
| date -- This is the current date and time as given in log_variables
| log_entry -- This is the formatted string that has format as specified
|   by the module through information gathered through the task
|   distribution module with the values from log_variables placed
| user_name -- A name specified by the user that a report should be
|   generated of the actions that they have performed
| case_id -- The case number that a report should be generated for
```

Generate Report:

Reports need to be generated from the entries that have been placed in the database. The contents of the report should be written to a file and the location of that file should be returned. This should be done with the following function:

```
|
| There are three ways for this function to be called:
| generate_report(cookie, user_name, case_id)
| generate_report(cookie, user_name)
| generate_report(cookie, case_id)
|
| user_name -- A name specified by the user that a report should be
|   generated of the actions that they have performed
| case_id -- The case number that a report should be generated for
|
| Returns:
|   A dictionary containing the means of accessing the file and the
|   specifics to access this particular file.  The current means available
|   are:
|       o {"string": "actual file"}
|       o {"nfs": "server:/path"}
```

| Error message as a string otherwise

To retrieve the reports the following database calls should be made:

database_module.db_reporting_retrieve_report(user_name, case_id)
database_module.db_reporting_retrieve_report(user_name)
database_module.db_reporting_retrieve_report(case_id)

These will return a list containing all of the report entries of the given arguments.

Scheduler Module

==========

Overview of Module:

This module is in charge of scheduling the analysis of possibly multiple cases at a time across any number of nodes in the system. Currently the options that this module needs to be able to carry out are:

Analysis completion, for each case a set of tasks should be declared this module will schedule those tasks to be completed.

Analysis targets, for each case there should be at least one datastore which contains what needs to be analyzed.

Number of nodes, each case should have a set number of nodes that it should be ran on. This will be determined by priority, 3-100% of nodes, 2-75% of nodes, 1-50% of nodes.

Type of schedule, currently aggregation and distribution should be supported allowing either a single analysis to be done on all data or independent analysis to be done on each piece of data.

Scheduler Algorithms, currently round robin, size balanced, and aggregation will be supported. This is on a task by task basis.

Running Analysis:

The specific types of analysis will be governed by the modules that have been installed. A such module would be file_indexing. These modules should be registered with the scheduler module when installed.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] function_called user_name case_id task_id function_result

An example is as follows:

[2009/06/02 11:50:15] user_login John 10 2 True

If the user failed to login then:

[2009/06/02 11:50:15] user_login John 10 2 False

Format as will be stored in the config.xml file:

[$date] $function_called $user_name $case_id $task_id $function_result

All functions in this module should be logged accordingly.

Start Analysis:

Once a case has been added and all information has been completed the tasks on that case can be started. This should be called by the Analyst interface to start processing. When starting a case the database should be queried for all of the tasks that belong to that case. Each task should be started if the prerequisites are met.

```
|
| start_analysis(case_id)
|
| case_id -- The id of the case that should be started.
|
| Returns:
|    True if the case was started
|    An error message if the case was not started
```

Stop Analysis:

To allow the analyst further control over the system the ability to stop analysis on a case should be given.

```
|
| stop_analysis(case_id)
|
| case_id -- The id of the case that should be started.
|
| Returns:
|    True if the case was stopped
|    An error message if the case was not stopped
```

Analysis Status:

Show the status of a specified case. This will return a XHTML string that gives the status of each task on the case in a table.

```
|
| analysis_status(case_id)
|
| case_id -- The id of the case that the status is requested for.
|
| Returns:
|    An XHTML string giving the status of each task on the case
|    An error message if the status cannot be determined
```

Retrieve Task Definition:

Retrieve a list of subtasks given the task definition.

```
|
| get_task_definition(task_id, td_name, td_version, recursion, cur_id)
|
| task_id -- The id of the task that this is for
| td_name -- The name of the task definition
| td_version -- The version of the task definition
| recursion -- A list to keep track of the number of recursion levels
| cur_id -- The current subtask id
|
| Returns:
|   A list of all subtasks given by the task definition selected.
```

Retrieve Subtasks:

Retrieve a list of subtasks given the parent task and task definition.

```
|
| get_subtask(main_task, case_id, task_id, recursion)
|
| main_task -- The name of the parent task definition, used for recursion
|   tracking.
| case_id -- The id of the case that the task is for.
| task_id -- The id of the task that this subtask is for.
| recursion -- A list that contains all of the previous task definition
|   names, if none are provided this is an empty list.
|
| Returns:
|   A list of all subtasks as dictionaries
```

Start/Stop/Pause/Resume Task:

To give more user control over the analysis pause and resume should be supported for the tasks that allow it.

```
|
| start_task(case_id, task_id)
| stop_task(case_id, task_id)
| pause_task(case_id, task_id)
| resume_task(case_id, task_id)
|
| case_id -- The id of the case that should be modified.
| task_id -- The id of the task to modify
|
| Returns:
|     True if the task was modified correctly
|     An error message if the operation could not be completed
```

Update Task:

This function will allow a task to be updated through status or progress. Once a task reaches a milestone (every 5-10%) or finishes it should call this. If a task reports 100 for progress it is assumed that the task is done.

```
|
| update_task(case_id, task_id, config_options, progress)
|
| case_id -- The id of the case that should be updated.
| task_id -- The id of the task to update.
| config_options -- The dictionary of configuration options that the task
|   was originally started with
| progress -- The new progress of the task, a simple integer 0 - 100
|   If 100 is reported the task will be marked as Done.
|
| Returns:
|     True if the task was updated correctly
|     An error message if updated failed
```

Run Task:

This function is responsible for placing tasks from the queue onto nodes.
It simply pulls from a queue of tasks in memory and assigns them to the next
available node.

```
|
| run_task()
|
| This does not take any arguments, all data it needs are global for the
|   scheduler module.
|
| Returns:
|   True if tasks could be started
|   False along with the reason why tasks could not be started
```

Assign A Process to Node:

Since this system should have multiple nodes, each of these should be managed separately. A dictionary of all nodes should be kept to show what nodes have been assigned to and what task has been assigned to them. The dictionary should have the module_name and config_options stored in it when the node is in use and should contain the string 'free' when not in use.

```
|
| assign_task(module_name, num_nodes, config_options)
|
| module_name -- The name of the module to be started on the node(s)
| num_nodes -- The number of node(s) that are to be assigned to
```

| config_options -- A dictionary of options that are the be passed to the module
|
| Here is an example:
| assign_task('index', 2, {'path':'nfs://test/case'})
|
| Returns:
| True if the process was successfully assigned
| An error message otherwise

Update the Number of Nodes on the System:

Due to nodes being added and removed there needs to be a method that forces the scheduler to check for new nodes and to see if any nodes have been removed from the system.

|
| update_nodes()
|
| This will query the Task Distribution Module for the max_node_num, then
| find all nodes from 1 to that number and request information on all of
| them. If information on a node is returned then it is attached to the
| system still. If a node that is not registered in the scheduler is found
| then that node will be added to the list of nodes. If a node is not
| found in that dictionary then it should be removed from the list and its
| task should be pre-pended to the queue.
|
| Returns:
| True if successful
| An error message otherwise

Sysadmin Interface Module

==========

Overview of Module:

This module will provide a front-end interface to allow sysadmin users to carry out management of the system, its modules and clusters in addition to user management. It will primarily interface with the user management and task distribution modules to allow these actions to be performed.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] action_called user_name function_specifics function_result

Only actions that cause changes to the system should be logged after the database has been called.

Front-end Interface:

The interface will be designed with only one handle for the web module.
The handle will be "sysadmin_interface", and will be structured:

```
[... in version.xml for sysadmin interface module ...]
<interface>
   <handle dest="sysadmin_interface" header="sysadmin"
   footer="standard">sysadmin_interface_handle</handle>
</interface>
[... remainder of version.xml ...]
```

Thus, a function sysadmin_interface_handle needs to be created as follows:

```
_____________________________________________________________________
|
| sysadmin_interface_handle(cookie, fields)
|
| cookie -- this variable holds the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|  CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the Sysadmin
|  Interface is being addressed.  The presence or absence of additional,
|  action-specific fields will further determine the behavior of the method,
|  which will fit into one of three standard response modes, and one
|  special case mode:
|  1) Initial page view - This response mode occurs when fields indicating
|    a completed form associated with the action are absent.  The method
|    replies with a generic, first-visit form of the appropriate page for
|              the action.
|  2) Submission - This response mode occurs when a completed form has been
|    submitted.  The method triggers the appropriate action, and replies
|    with a page indicating the success or failure of that action.
|  3) Erroneous input - this response mode occurs when some, but not all
|    appropriate form data has been provided, or some input form data does
|    not pass input validation.  The returned page will be similar to the
|    'Initial page view" mode response, but with error messages and
|    user-provided data added.
|  A) Programmer error - This response mode occurs when none of the above
|    three modes is appropriate, and indicates that there is a programmer
|    problem, rather than a user problem.
|
| On each call to a handler method, the cookie should be checked for
|  validity and that the user belongs to the sysadmin group.
|
| Not all actions will actually use all three traditional response modes -
```

```
|  in some cases, an action will not have associated form data, and the
|  'Initial page view' and 'Erroneous input' modes do not make sense in
|  those circumstances.
|
| The method supports the following contents in the 'action' field:
|-------------------------------------------------------------------------
|| None or 'home' - present the home page
||
|| User Management is not included as a method because it should be
||   referenced from the user management module. A link should be made
||   to that cgi script like user_management.cgi?action="home"
|-------------------------------------------------------------------------
|| 'add_node' - Add a new node to the cluster
||   Form Fields:
||     'name' - string that should correspond with a physical label
||       on the node (hopefully unique but does not need to be)
||     'ip' - string containing the IP to the new node
||     'public_key' - the public key to associate with that new node
||       These are also the names of the only form elements
||   This should call get_configuration to get the web path to the script
||     for the download link.
||   Once submitted the public key should be saved like:
||     task_distribution.put_file("sysadmin" "sysadmin", "'ip'_public_key",
||                                'public_key', 'global')
|-------------------------------------------------------------------------
|| 'remove_node' - Remove a node from the cluster
||   This action has 2 pages, they will be outlined separately.
||   Page 1: This lacks the fields name, ip, and confirm
||     This page contains no form elements only links to remove each node
||     Each link emulates a form with the element:
||       'name' - The name of the node to remove
||       'ip' - The ip of the node to remove
||       'confirm' - An indicator to remove the node, should be set to
||         False
||   Page 2: The value of confirm is False on input
||     This page contains no form elements, only the links Yes and No
||     The Yes link should emulate a form with the elements:
||       'ip' - The ip of the node to remove
||     The No link should return the user back to page 1.
||   If 'ip' is provided and 'confirm' is True then the node should be
||     removed through the scheduler module:
||       delete_node('ip')
|-------------------------------------------------------------------------
|| 'add_module' - Add a new module to the system
||   Form Fields:
||     'name' - The name of the new module, this name will be used for the
```

```
||            module_name
||    'version' - The version of the module being added
||    'url' - The url to the module to download
||      These are also the names of the only form elements
||  This references the task distribution module calling add_module:
||    add_module('name', 'version', 'url')
|-------------------------------------------------------------------------------
|| 'remove_module' - Remove a module from the system
||  This action has several pages, they will be outlined separately.
||  Page 1: This lacks the name and confirm fields from the form input
||    Form Fields:
||      'name' - The name of the module to remove
||  Page 2: This has a name field but lacks a confirm field from input
||    This page does not contain a form but links instead. Each remove
||      link should emulate a form with the elements:
||      'name' - The name of the module to remove, taken from input
||      'confirm' - An indicator to remove the module, should be set to
||        False from the link
||  Page 3: Again this has no form only two links, Yes and No
||    The Yes link should emulate a form with the elements:
||      'name' - The name of the module to remove, taken from input
||      'version' - The version of the module to remove
||      'confirm' - An indicator to remove the module, should be set to
||        True from the link.
||    The No link should emulate a form with only the 'name' element
||  If 'name' is provided and 'confirm' is True then the module should be
||    removed through the task distribution module:
||    remove_module('name', 'version')
|-------------------------------------------------------------------------------
|| 'upgrade_module' - Upgrade a module currently on the system
||  Form Fields:
||    'name' - The name of the module to upgrade
||    'version' - The new version of the module
||    'url' - The url to download the new module from
||  This should call a function from the task distribution module:
||    upgrade_module('name', 'version', 'url')
|
| Returns:
|  An XHTML fragment to be displayed to the user.
```

Support Functions:

```
 ------------------------------------------------------------------------------
|
| get_module_list()
|
| Returns:
```

| A full list of modules that are associated with the system.

|
| get_node_list()
|
| Returns:
| A full list of nodes registered to the system.

|
| add_node(name, ip, port, public_key)
|
| name -- The name of the node, this should be labeled on the physical node
| ip -- The ip address of the node
| port -- The port to the machine daemon module running on that node
| public_key -- The public key of the node as generated through the setup
| script
|
| Returns:
| True if the node was added
| Error message otherwise

|
| remove_node(name, ip)
|
| name -- The name of the node, this should be visible on the physical node
| ip -- The ip address of the node
|
| Returns:
| True if the node was removed
| Error message otherwise

|
| add_module(name, version, url)
|
| name -- The name of the module to be added
| version -- The version of the module that is being added
| url -- The external url to the module that is being added
|
| Returns:
| True if the module was added
| Error message otherwise

```
|
| remove_module(name, version)
|
| name -- The name of the module to remove
| version -- The version of the module to remove
|
| Returns:
|   True if the module was removed
|   Error message otherwise
```

```
|
| upgrade_module(name, old_version, version, url)
|
| name -- The name of the module to be upgraded
| old_version -- The version of the module to be upgraded
| version -- The version of the upgraded module
| url -- The external url to the upgraded module
|
| Returns:
|   True if the module was upgraded
|   Error message otherwise
```

User Management Module

==========

Module Overview:

This module will provide a front-end interface to allow users to manage their account and that will allow Sysadmin users to create, edit, and remove users. Adding and removing users on cases is not done through the interface but should be done by calling the appropriate function in this module. It will primarily interface with the task distribution and database modules.

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] function_called calling_user function_specifics function_result

The function_specifics will contain the following for the functions:

create_user: the contents of information_dict
disable_user: the user to be disabled
remove_user: the user that is to be removed
modify_user: the contents of information_dict
add_user_to_case: user_name, case_id
remove_user_from_case: user_name, case_id

An example is as follows:
[2009/06/02 11:50:15] add_user_to_case John Bob, 1234 True
If the module failed to add the user to the case then:
[2009/06/02 11:50:15] add_user_to_case John Bob, 1234 False

Format as will be stored in the config.xml file:
[$date] $function_called $calling_user $function_specifics
$function_result

The following functions should be logged:
create_user, disable_user, remove_user, modify_user, add_user_to_case
remove_user_from_case

Front-end Interface:
The interface will be designed with only one handle for the web module.
The handle will be "user_management_interface", and will be structured:

[... in version.xml for user_management interface module ...]
<interface>
<handle dest="user_management_interface" header="user_management"
footer="standard">user_management_interface_handle</handle>
</interface>
[... remainder of version.xml ...]

Thus, a function user_management_interface_handle needs to be created as follows:

```
|
| user_management_interface_handle(cookie, fields)
|
| cookie -- this variable holds the session ID for the current user
| fields -- This is a Python FieldStorage() object that holds all of the
|   CGI fields.
|
| This method looks for the 'action' field in 'fields' and uses the content
|  of that field to determine which action associated with the User
|  Management Interface is being addressed.  The presence or absence of
|  additional, action-specific fields will further determine the behavior of
|  the method, which will fit into one of three standard response modes, and
|  one special case mode:
|  1) Initial page view - This response mode occurs when fields indicating
|     a completed form associated with the action are absent.  The method
|     replies with a generic, first-visit form of the appropriate page for
|                 the action.
```

```
|  2) Submission - This response mode occurs when a completed form has been
|    submitted.  The method triggers the appropriate action, and replies
|    with a page indicating the success or failure of that action.
|  3) Erroneous input - this response mode occurs when some, but not all
|    appropriate form data has been provided, or some input form data does
|    not pass input validation.  The returned page will be similar to the
|    'Initial page view" mode response, but with error messages and
|    user-provided data added.
|  A) Programmer error - This response mode occurs when none of the above
|    three modes is appropriate, and indicates that there is a programmer
|    problem, rather than a user problem.
|
| On each call to a handler method, the cookie should be checked for
|   validity and that the user belongs to the user_management group.
|
| Not all actions will actually use all three traditional response modes -
|  in some cases, an action will not have associated form data, and the
|  'Initial page view' and 'Erroneous input' modes do not make sense in
|  those circumstances.
|
| The method supports the following contents in the 'action' field:
|_____________________________________________________________________________
|| None or 'home' - present the home page
||
|| This shows a the general interface allowing the user to select from a
|| list of options based on their user type.
|_____________________________________________________________________________
|| 'custodian_user_search' - Allow a custodian user to search for a user
||
||   Form Fields:
||     'user_search' - The name of the user to look for, if this is not
||       given then the initial search page is shown
||   This calls the database to find all users by the name given
|_____________________________________________________________________________
|| 'sysadmin_user_search' - Allow a sysadmin user to search for a user
||
||   Form Fields:
||     'user_search' - The name of the user to look for, if this is not
||       given then the initial search page is shown
||   This calls the database to find all users by the name given
||   This is different from custodian_user_search due to the different
||     options the users have
|_____________________________________________________________________________
|| 'update_password' - Allow any user to change their password
||
||   Form Fields:
```

```
||    'old_password' - The user specified old password
||    'password' - The new password as specified by the user
||    'confirm_password' - A confirmation of the new password
||  After ensuring that the old password is correct the new password is
||    checked against the confirmation and then the password is updated.
|_____________________________________________________________________
|| 'modify_user' - Allow a sysadmin user to modify a user
||
||  Form Fields:
||    'user_name' - The name of the user to modify
||    'user_type' - The new type of a user
||    'password' - A new password for a user
||    'confirm_password' - A confirmation of the new password
||  If only user_name is given then the general modify user page is shown.
||  Otherwise if user_type or password and confirm_password are given the
||  appropriate change in the database will occur.
|_____________________________________________________________________
|| 'create_user' - Allow a sysadmin user to create a user
||
||  Form Fields:
||    'user_name' - The name of the new user
||    'password' - The password for the new user
||    'confirm_password' - A confirmation of the password
||    'id_key' - A unique id key for the user
||    'user_type' - The type of the new user
||  Once the password is confirmed the user is created in the database.
|_____________________________________________________________________
|| 'remove_user' - Allow a sysadmin user to remove a user
||
||  Form Fields:
||    'user_name' - The name of the user to remove
||    'confirm' - A true/false option based on sysadmin's action at the
||      given confirmation page.
||  Before the user is removed the sysadmin is given a confirmation page
||  in order to reduce the chance that a user is removed by accident.
```

Support Functions:

Create User:

This first task is be able to create a new user through the web interface.
In order to do this the interface should call the following function:

```
 _____________________________________________________________________
|
| create_user(cookie, information_dict)
|
```

| cookie -- a pickled SimpleCookie object (can be called as a hash)
| information_dict -- A dictionary containing the values to create a user
|
| Returns:
| An XHTML string stating the status of creation of user.

This requires the following database function to be called:

database_module.db_user_management_create_user(information_dict)

This will return true if the creation was successful,
An error message otherwise.

Remove User:
Much like disable user, only the user name to delete needs to be given:

|
| remove_user(cookie, user_name)
|
| cookie -- a pickled SimpleCookie object (can be called as a hash)
| user_name -- the name of the user that is to be deleted
|
| Returns:
| An XHTML string stating the status of the removal of the given user

This requires the following database function to be called:

database_module.db_user_management_remove_user(user_name)

This will return true if the creation was successful,
An error message otherwise.

Modify User:
This is the function that modifies the users and gives user_management users
access to modify all other users.

|
| modify_user(cookie, information_dict)
|
| cookie -- a pickled SimpleCookie object (can be called as a hash)
| information_dict -- A dictionary containing the values to modify the user
|
| Returns:
| An XHTML string stating the status of modification and a table of the

| new values.

This requires the following database function to be called:

database_module.db_user_management_modify_user(information_dict)

This will return true if the creation was successful,
An error message otherwise.

User Modification for Other Users:
Since the above function allows the modification of any specific user, some limitation must be made for custodians and analysts to ensure they do not modify the information of other users. This is accomplished by a session ID that is given to the interface. At first run this function returns a listing of the user's information and from there access to user modification is available. If the user is a user_management then a list of users will be shown first then after the user_management selects a user it will list that user's information. It is defined as follows:

| user_preferences_interface(cookie, form)
|
| cookie -- a pickled SimpleCookie object (can be called as a hash)
| form -- a picked FieldStorage object holding the request
|
| Returns:
| An XHTML string for an interface allowing the user to change their
| stored information.

This function will show the user their information and allow them to change it through the given session ID. Once the user submits the information to be changed then modify user will be called to complete the change.

Adding and Removing Users from Cases:
These functions support the custodian interface allowing the custodian users to add and remove users from specific cases. These functions will contact the database to add or remove these users from the case.

| add_user_to_case(cookie, user_name, case_id)
| remove_user_from_case(cookie, user_name, case_id)
|
| cookie -- a pickled SimpleCookie object (can be called as a hash)

| user -- the user ID that should be added or removed
| case -- the case ID that the user should be added to or removed from

This requires the following database function to be called:

database_module.db_user_management_add_user_to_case(user_name, case_id)
database_module.db_user_management_remove_user_from_case(user_name, case_id)

This will return true if the creation was successful,
An error message otherwise.

Task Distribution Module

Overview of Module:
This module is designed to help facilitate communication between modules, distribute configuration and configuration related files for modules, and work as a software repository for all modules.

Specifically this module provides the system with an interface for configuring and/or installing other core modules of PDF by use of the sysadmin interface module. Through the sysadmin module other modules may be added/configured/removed and the other module's configuration files should not have to be edited manually.

Frequent requests should also be cached in RAM regularly to increase speed. This is useful for relaying configuration options quickly.

XML Configuration File Format:

```
<module name="module_name">
  <var1>var1data</var1>
  <var2>var2data</var2>
</module>
```

Log Format Specification:
The log format that will be used for this module is as follows:
[YYYY-MM-DD hh:mm:ss] function_called module_name function_specifics function_result

The function_specifics will contain the following for the functions:
set_configuration: the contents of set_variables
get_file: the contents of file_name
find_handle: the contents of handle
add_module: the version of the module being added
upgrade_module: the version of the module being upgraded
remove_module: the version of the module being removed

An example is as follows:

[2009/06/02 11:50:15] set_configuration web {'ip':'127.0.0.1','port':
'8005'} True

If the module failed to set the configuration then:

[2009/06/02 11:50:15] set_configuration web {'ip':'127.0.0.1','port':
'8005'} False

Format as will be stored in the config.xml file:

[$date] $function_called $module_name $function_specifics
$function_result

The following functions should be logged:

set_configuration, get_file, put_file, remove_file, get_handle,
add_module, upgrade_module, remove_module, write_module_list

Relay Configuration Options:

The task distribution module needs to be able to give the configuration of a specified module to another module that is requesting it if the configuration is set to world readable. This function should be named "get_configuration" and have format as follows:

```
|
| get_configuration(module_name, (settings_list))
|
| module_name -- The name of the module that configuration settings are
|   desired from.
| settings_list -- A comma separated list of settings desired from the
|   module. Here is an example of the format:
|
|   get_configuration("logging", ("ip", "port", "log_dir"))
|
| Returns:
|   A dictionary of the retrieved keys and their values.  For example,
|     {'ip':127.0.0.1, 'port':48098, 'log_dir':'/log/'}
|   Error message as a string otherwise
```

These requests should be cached in RAM as these requests typically occur often.

Set Configuration Options:

The task distribution module needs to have an option for setting or modifying the configuration of modules. This function should be named "set_configuration" and have format as follows:

|
| set_configuration(module_name, set_variables)
|
| module_name -- The name of the module that has settings that need to
| be modified.
| set_variables -- A dictionary variable that contains the name(s) of the
| settings to modify along with their new values.
|
| Returns:
| True on success
| False on generic error, but this should throw the specific error

Once invoked the function should directly modify the XML configuration appropriately.

Distribute Module's Files:

The task distribution module needs to be able to retrieve files for other modules if the file is marked as world readable in the module's configuration.

|
| get_file(caller_module_name, module_name, file_name)
|
| caller_module_name -- The name of the module that invoked the get_file
| call.
| module_name -- The name of the module to retrieve the file from
| file_name -- The name of the file, in the directory of the module, that
| should be retrieved.
|
| Returns:
| A dictionary containing the means of accessing the file and the
| specifics to access this particular file. The current means available
| are:
| o {"string": "actual file"}
| o {"nfs": "server:/path"}
| Error message as a string otherwise

This function will specify where to find the file, or, if the file is small enough and a text file, it will pass the entire file as an escaped string in the dictionary.

Save a File for a Module:

The task distribution module needs to allow files to be created for modules. The access control should be specified at this time.

```
|
| put_file(self_module_name, module_name, file_name, file_contents,
|         access_control)
|
| self_module_name -- The name of the calling module
| module_name -- The name of the module that the file is for
| file_name -- The name that the file should be saved as
| file_contents -- A dictionary containing the keys type and content
|   type -- the type of input, string or nfs
|   content -- the content of the file or the path to the file
|     If this is an nfs path only the path should be given, not including
|     name of the desired file
| access_control -- A list containing the modules that are allowed to
|   access the file, if this contains 'global' all modules have access
|
| Returns:
|   True on success
|   An error message otherwise
```

Remove a File from a Module:

In efforts to allow the system to do some better housekeeping the removal of files that have been created needs to be allowed. This function will allow only the owning module to remove files that have been placed under its control i.e. a file that was created using put_file to the sysadmin module can only be removed by the sysadmin module. This function should be named "remove_file" and have format as follows:

```
|
| remove_file(self_module_name, module_name, file_name)
|
| self_module_name -- The name of the calling module
| module_name -- The name of the module to remove the file from
| file_name -- The name of the file to remove
```

Relay Interface Handle Information:

The task distribution module needs to be able to grab handle information for the modules which will contain handles for the web module.

```
|
| get_handle(handle)
|
| handle -- A string containing the handle to search for
```

```
|
| Returns:
|   A dictionary containing information about the handle found.  The
|     information could contain, for example,
|       { "module": module_name, "function": function_to_call,
|         "pre-header": pre_header_name, "header": header_name,
|         "footer": footer_name }
|     Where module_name would be the module that registered the handle,
|     and header_name and footer_name are the two options in the XML file.
```

This function could easily be a front-end of sorts to a hash table of all of the available handles determined when a module is added, upgraded, or the task distribution module is loaded.

Add Modules:

Through the task distribution module other modules should be able to be added to the system. The function should be named "add_module" and be formatted as follows:

```
|
| add_module(module_name, module_version, module_download_url)
|
| module_name -- The name of the module that has settings that need to
|   be modified.
| module_version -- The version of the module to install
| module_download_url -- The location to download the module from, should
|   be possible to get it from an external and internal web based source
|
| Returns:
|   True on success
|   Error message as a string otherwise
```

Upgrade Modules:

While upgrading is similar to adding, a couple of additional steps are required to ensure a proper upgrade. The function should be named "upgrade_module" and have format as follows:

```
|
| upgrade_module (module_name, old_version, module_version,
|                 module_download_url)
|
| module_name -- The name of the module that has settings that need to
|   be modified.
| old_version -- The version of the old module
```

| module_version -- The version of the module to install
| module_download_url -- The location to download the module from, should
| be possible to get it from an external and internal web based source
|
| Returns:
| True if upgrade was successful
| String containing error message, including current version, otherwise

To start the system needs to be checked for the specified module, if it exists we need to ensure that the new version number is higher than the old version number to prevent downgrading. If an upgrade will occur the new module should be downloaded and installed then the module should be restarted.

Download Module:
This function helps with the adding and upgrading of modules by downloading the desired module from the given url to a local directory, extracts it so that it can be easily installed into the system.

|
| download_module(module_download_url)
|
| module_download_url -- The url to the desired module, this must be a valid
| url and the file must be a tar archive.
|
| Returns:
| The name of the directory that it was extracted to if the module was
| downloaded correctly
| An error message otherwise

Remove Modules:
In effort to have good control over the system the task distribution module should be able to remove modules by deleting the module's files and stop the running instances of that module. This function should be named "remove_module" and have format as follows:

|
| remove_module(module_name, version)
|
| module_name -- The name of the module that has settings that need to
| be modified.
| version -- The version number of the specific module to remove
|
| Returns:

| True on success
| Error message as a string otherwise

This should stop all running instances of the module, completely remove all files associated with the specified module, and remove all configuration file entries.

Get Module List:
The list of modules registered to the system are stored in Task Distribution's configuration file, to retrieve this list and convert it to a readily usable state it must be read in and processed. This function should be named "get_module_list" and have format as follows:

```
|
| get_module_list()
|
| Returns:
|   A list of modules and their versions. Each item in the returned list
|   is a list containing [module_name, version]
```

Write Module List:
This function should take a full list of the modules currently registered to the system and write them to Task Distribution's configuration file for safe storage. This function should be named "write_module_list" and have format as follows:

```
|
| write_module_list(module_list)
|
| module_list -- A list of modules and their versions. Each item should be
|   formatted such that: [module_name, version]
|
| Returns:
|   True if the list was written properly
|   An error message otherwise
```

Web Module

==========

Module Overview:
This is the module is a CGI script that is called by the web server to complete the tasks that the user commands. The primary modules this will communicate with are the Task Distribution and Authorization modules. This module is also designed to communicate with any module that it has a

handle for.

Handle Incoming Requests:
All incoming requests will be handled by this and the Task Distribution module must be contacted to carry out the request given by the user.

Currently, the requests will be structured as a standard GET/POST request. For example, the end user might be visiting URIs such as: http://location.of.pdf/pdf.cgi?handle=analayst_interface&view=home

The web module will then first authorize a server side cookie. This will be done through the authorization module.

Then, the web module will need to determine what handle to use, and then use the appropriate header, module/function, and footer.

Authorize User's Cookie:
When a user submits cookie values to maintain their session it must be confirmed by the authorization module. This can be done by using:
| authorization.check_session(cookie)

This function will check all of the values in the cookie to ensure that it is a valid cookie.

Determine handle to use:
The handle that will be used will be the handle specified in the "handle" field of the HTTP query. The mapping of this handle to a function can be determined by querying the task distribution module.
| task_distribution.get_handle(handle)
This will return a dictionary containing all of the handle information. From this information, the header and footer can be displayed, and the module and function to be called can be determined.

Get Header/Footer:
Since this module compiles the user interface it also needs to read the headers and footers for each page along with the page specific information. This is done through use of the parser module function:
| parser.parse_to_XHTML("web", header_supplied_by_handle_return)
For example:
| parse_to_XHTML("web", "login_header")
The headers/footers will be stored as xml interface files for the web module.

XHTML Parser Module
==========

Overview of XHTML Parser Module:

The purpose of this module is to take the XML files for interfaces and parse them into XHTML segments that can be displayed through the web interface. These XML files contain specific instructions on how to format any generic interface, including XHTML, using a predefined XSL.

This module will have two primary functions: "parse_to_XHTML" and "clear_cache". It will also contain a function that will be most useful in development, "verify_XML".

Log Format Specification:

The log format that will be used for this module is as follows:

[YYYY-MM-DD hh:mm:ss] function_called module_name interface_name function_result

Since clear_cache has many different ways to call it:

Place the arguments into their similarly named section, if a section was not given as an arguments place 'none' into its place.

An example is as follows:

[2009/06/02 11:50:15] clear_cache web sysadmin_interface True

If the module failed to retrieve the interface named sysadmin_interface then:

[2009/06/02 11:50:15] clear_cache none sysadmin_interface False

Format as will be stored in the config.xml file:

[$date] $function_called $module_name $interface_name $function_result

The following functions should be logged:

clear_cache, verify_xml

Connecting to Task Distribution Module:

Similar to the majority of the modules, the information required to connect to the task distribution module will be provided on the command line as arguments to the parser module.

The format will be as follows:

--td_ip <ip address> --td_port <port>

Parsing to XHTML:

This is the function that provides the primary functionality of the module. This gathers the information on the specified module and the interface it is calling then produces an XHTML string. The format of the function is as follows:

```
|
| parse_to_XHTML(module_name, interface_name)
|
| module_name -- The name of the module the interface is desired from.
| interface_name -- The name of the interface that is to be prepared,
|   the interface files can be established from this.
|
| Returns:
|   Resulting XHTML string
```

Once this function is called the interface files must be found, this is done by connecting to the Task Distribution module and requesting the files like so:

```
get_file(module_name, "interface_name.xml")
```

After the file names and paths are established the name of the XML file should be cached to increase speed and then the XML file must be read and parsed. The result of the parsing should be returned with the function.

Clearing the Cache:

As can be seen in the previous step, a cache is needed here therefore we need a method of clearing the cache. Arguments to this function are optional. If there are arguments given there are a few different options. The possible function calls are as follows:

```
|
| clear_cache()
| clear_cache(module_name)
| clear_cache(module_name, interface_name)
|
| module_name -- The name of the module the interface is desired from.
| interface_name -- The name of the interface that is to be prepared,
|   the interface files can be established from this.
|
| If cache is to be cleared for all modules but for a specific interface
|   module_name should be given as -1.
|
| Returns:
|   True on success
```

If no arguments are given, then clear the cache for all modules and interfaces. If only a module or interface name is given, clear the

cache for those named. If a module and interface name is given, clear the cache for that module/interface combination.

This will return the status of the clearing of cache, true if the clearing succeeded and will return a descriptive error if it failed. No need to return an error if module_name or interface_name have no cache values associated.

Verifying XML:

The parser needs to be able to verify XML files to ensure that they follow the XSL for that interface version.

```
|
| verify_XML(module_name, interface_name)
|
| module_name -- The name of the module the interface is desired from.
| interface_name -- The specific interface to be verified.
|
| Returns:
|   True on success
|   False on failure
```

This function will mainly be used for development, i.e. to quickly ensure that a module's interface will work within the standards. However, it could also be used via the sysadmin interface to ensure that a module being installed is standards-compliant.

This page was intentionally left blank.

DISTRIBUTION

1 New Mexico Tech (electronic copy)
Computer Science Department
Attn: Dr. Lorie Liebrock
Socorro, NM 87801

1	MS0672	Robert L. Hutchinson	05629 (electronic copy)
1	MS0672	David P. Duggan	05629 (electronic copy)
1	MS0899	Technical Library	9536 (electronic copy)
1	MS0123	D. Chavez, LDRD Office	1011


Sandia National Laboratories