HOWTO: Create an OSCAR Package

Core OSCAR Team

January 22, 2004

Draft $Id: oscarpkg-howto.tex,v 1.15 2004/01/22 03:42:04 naughtont Exp $

Contents

1 Introduction **3**
1.1 Super Short Summary 3

2 Package Layout **3**
2.1 `config.xml` 4
2.2 RPMS & SRPMS 4
2.3 scripts 4
2.3.1 Package Setup 5
2.3.2 Configurator 6
2.3.3 Fixups without RPM modification 6
2.3.4 Setup after Clients Defined 6
2.3.5 Completing cluster configurations 7
2.3.6 Package Uninstall 7
2.4 testing 7
2.5 doc 7

3 Example Package **8**
3.1 A basic `config.xml` 8
3.2 Using the Configurator 8
3.2.1 post_configure 9
3.3 post_clients 10
3.4 post_install 11

4 OSCAR Package Downloader (OPD) **11**

5 OSCAR Database (ODA) **12**

6 OSCAR Configurator **12**

A Rules of Thumb **18**
A.1 Install Location 18
A.2 OSCAR Specific RPMS 18
A.3 `init.d` scripts 18
A.4 Generating Configuration Files 18

B Env-Switcher **18**
B.1 Example Switcher Module file 19

C Supported XML Tags **19**

List of Tables

1 Environment Variables 4
2 Package API Scripts 5
3 Outline of operations performed 5
4 OSCAR Package XML tags 20

1 Introduction

The Open Source Cluster Application Resources (OSCAR) toolkit is used to install and maintain computing clusters. The configuration and installation of software is central to cluster management. The approached used in OSCAR is to create a "package", which contains the necessary software as well as additional scripts for configuration on a standard OSCAR cluster. This enables software to be added to the installation framework with as little effort as possible by the package authors. This document will describe what comprises an *OSCAR Package*, giving examples for clarity. The intent is to provide a single document for those seeking to create packages for use with the OSCAR toolkit.

1.1 Super Short Summary

For the truely impatient here is a very short summary of what it takes to create an OSCAR package. The remainder of this document will describe things in more detail. The items marked *Optional* are just that, but are highly recommended.

- Make a directory based on the package's name, e.g., `pvm/`.
- Create the package directory structure, `<PKG_NAME>/{RPMS,SRPMS,scripts,testing,doc}`
- Add a `config.xml` meta file to the top-level package directory.
- Add a binary RPM to the `<PKG_NAME>/RPMS` directory for supported OSCAR distro(s).
- (*Optional*) Add the the Source RPM (SRPM) to the `<PKG_NAME>/SRPMS` directory
- (*Optional*) Add any configuration scripts as per OSCAR script API to `<PKG_NAME>/scripts`
- (*Optional*) Add package tests to `<PKG_NAME>/testing`, (special: `test_user, test_root`)
- (*Optional*) Add package documentation/license to `<PKG_NAME>/doc`, (special: `{user,install,license}.tex`)
- Tar/gzip the entire package directory and add to an OPD repository[1], e.g., `tar -zcvf <PKG_NAME>.tar.gz`.

2 Package Layout

As OSCAR evolved it became obvious that the mechanism to configure and install a cluster needed to be cleanly separated from the software that was to be installed. The approach taken was to create *OSCAR Packages*. The OSCAR Package layout is geared toward making things as simple as possible for package authors. So, in its simplest form an OSCAR Package is an RPM[2]. However, most software requires further configuration for a cluster environment so additional scripts, documentation, etc. may be added. The basic directory structure for an OSCAR Package is as follows.

`config.xml` – meta file with description, version, etc.
`RPMS/` – directory containing binary RPM(s) for the package
`SRPMS/` – directory containing source RPM(s) used to build the package
`scripts/` – set of scripts that run at particular times during the installation/configuration of the cluster
`testing/` – unit test scripts for the package
`doc/` – documentation and/or license information

For reference purposes, Table 1 contains a list of the currently recognized environment variables used by the OSCAR framework.

The packaging API provides authors the ability to make use of configuration scripts to setup cluster software outside of the RPM itself. The scripts fire at different stages of the installation process as detailed in Section 2.3. The packages may also include simple test scripts in the `testing/` directory, which are used to verify the software was properly installed (see Section 2.4). Lastly, an OSCAR Package Downloader (OPD) tool is provided to simplify acquisition of new packages (see Section 4).

[1] How things are added to OSCAR varies but most all packages can be obtained via on-line package repositories via OPD.
[2] A binary RPM compiled for an OSCAR supported Linux distribution.

Environment Variable	Description
OSCAR_HOME	Defines top-level directory for OSCAR installation
OSCAR_PACKAGE_HOME	Points to a packages directory (used with OPD)
OSCAR_HEAD_INTERNAL_INTERFACE	Contains the value provided to 'install_cluster *ethX*'

Table 1: Environment variables currently recognized/used by OSCAR.

2.1 `config.xml`

This XML file provides the package name (`<name>`), version number (`<version>`) and description information (`<description>`) as well as the list of RPMS (`<rpmlist>`) and their installation location (e.g., "oscar_server", "oscar_clients"). The file enables an author to convey constraints such as supported distribution (`<filter>`) or simple dependencies (`<requires>`) upon other OSCAR Packages, e.g., Env-Switcher. If this meta file is not included a simplistic default is used—install all files in `RPMS/` on all machines in cluster. The available XML elements for use in this file are listed in Appendix C, page 19 with a complete example given in Section 3.

The `<rpmlist>` contains the list of RPM names (without version numbers) to be installed for the package. There may be multiple instances of these `<rpmlist>`'s using the `<filter>` to differentiate where necessary. The `<filter>` constraints enable an author to express what `distribution`, `distribution_version` and `architecture` the associated `<rpm>`'s support. The `group` value specifies where the to install the `<rpm>`'s. Things are implicitly global if no constraints are provided, i.e., entire cluster for all distributions.

2.2 RPMS & SRPMS

The pre-compiled binary version of the software is provided in RPM format. The RPMS are placed, obviously enough, in the `RPMS` directory. The OSCAR Wizard copies all files from this `RPMS` directory to the `/tftpboot/rpm` directory.

If present, whichever `<rpmlist>` that fits the available distribution (based on `<filter>` constraints) is recorded in the OSCAR database for that particular package[3]. The list of supported distributions for each OSCAR release is typically stored in Table 1 of the installation document.

Notice: As of OSCAR-2.3 the `<filter>` tag does not yet support the `subdir` attribute, which is used to specify a directory for the `<rpm>`'s in the `<rpmlist>`. Due to this limitation, some packages use a `setup` script to copy the appropriate files to this `RPMS` area based on the values (distro_name, distro_ver) returned from Perl method `OSCAR::Distro::which_distro_server()`. To access this method from your `scripts/setup` file add the includes:

```
use lib "$ENV{OSCAR_HOME}/lib";
use OSCAR::Distro;
my ($distro_name, $distro_ver) = which_distro_server();
```

2.3 scripts

The OSCAR framework recognizes the set of scripts outlined in Table 2. A package author may use any/all of these as needed for application configuration. The order of operation during the OSCAR installation process is summarized in Table 3. The "Location" column indicates where the actual modification/operations take place[4]. Each phase of scripts is executed per package, with packages processed in alph-order.

Notice: All OSCAR API scripts must be re-runnable.

[3]The contents of `<rpmlist>` have no bearing on what is copied to `/tftpboot/rpm`. All files in `RPMS` are blindly copied.

[4]Note, the `post_install` scripts in practice often operate on both the server and client filesystems – regardless of what the original specifications suggested.

Seq#	Script Name	Description
1	`setup`	Perform any package setup
2	`pre_configure`	Prepare package configuration (dynamic user input)
3	`post_configure`	Process results from package configuration (user input results)
4	`post_server_rpm_install`	Perform "out of RPM" operations on server (limited cluster knowledge)
5	`post_client_rpm_install`	Perform "out of RPM" operations on client (limited cluster knowledge)
6	`post_clients`	Perform configurations with knowledge about cluster nodes (pre node install)
7	`post_install`	Perform final configurations with fully install/booted cluster nodes
	`post_server_rpm_uninstall`	(*NEW*) Runs on server's filesystem during OSCAR package uninstall
	`post_client_rpm_uninstall`	(*NEW*) Runs on client's filesystem during OSCAR package uninstall

Table 2: The set of available OSCAR API scripts listed in order of execution.

	Description	Location
1.	Install framework pre-requisites	Server filesystem
1.1.	Call the API scripts: `setup`	Server filesystem
1.2.	Read XML config files	Server filesystem
1.3.	Install server core RPMs	Server filesystem
1.4.	(*Optional*) Download additional packages	Server filesystem
1.4.1.	Call the API scripts: `setup`	Server filesystem
1.4.2.	Read XML config files	Server filesystem
1.5.	Select which packages to install	Server filesystem
1.6.	Call the API scripts: `pre_configure`	Server filesystem
1.7.	Configure the selected packages	Server filesystem
1.8.	Call the API scripts: `post_configure`	Server filesystem
1.9.	Install the server non-core RPMs	Server filesystem
2.	Call the API scripts: `post_server_rpm_install`	Server filesystem
3.	Install all the client RPMs	Client/`chroot`'d environment
4.	Call the API scripts: `post_client_rpm_install`	Client/`chroot`'d environment
5.	Define clients in the OSCAR/SIS database	Server filesystem
6.	Call the API scripts: `post_clients`	Server filesystem
7.	Push the images to the nodes	Server filesystem
8.	Call the API scripts: `post_install`	Server filesystem (access to clients)

Table 3: Note the `chroot`'d environment indicates the operations happen in the SIS image not on the actual machine. In step 8, the script runs on the Server but can affect the clients, e.g., via C3 commands.

2.3.1 Package Setup

The `setup` script executes before any packages are installed. This phase can be used to move files around in the package's directory or to do dynamic setup before the package `config.xml` scripts are processed. Once these XML files have been processed the available packages are then passed to the GUI where they are processed by the Selector panel. If any of the selected packages contain `pre_configure` scripts those are processed and then handed to the Configurator. After the Configurator has run any existing `post_configure` scripts are processed. At this point no package software has been installed and all that is know by the database is what packages were selected and their `config.xml` information (version, etc.).

Notice: If a user skips the "Configure Packages" step in the Wizard, the `pre_configure` and `post_configure` scripts will not be processed, i.e., the Configurator will not be run.

2.3.2 Configurator

Packages may obtain user input via a simple facility called the "Configurator". The package author writes a simple HTML Form style document that is presented to the user if the package is selected for installation. The standard multi-pick lists, radio button, checkbox fields are available. Typically default values are provided to simplify matters where possible for users.

To make use of this facility create a file in the top-level of the package's directory called `configurator.html`. After the package selection phase of the OSCAR Wizard all packages containing this file are processed by the Configurator. The results of this processing are written out in XML format to the top-level directory of the package in a file called `.configurator.values`. At this point the `post_configure` API scripts are fired so packages may read the results of the configuration phase. The Perl `XML::Simple` module is typically used for processing these results in conjunction with the `OSCAR_PACKAGE_HOME` environment variable. Alternatively, you can use the Perl subroutine `readInConfigValues` available in the `OSCAR::Configbox` module. A complete summary of the Configurator is available in Section 6, page 12. Also, a package example containing input and simple processing scripts for output is available in Section 3.2, page 8.

2.3.3 Fixups without RPM modification

The `post_server_rpm_install` and `post_client_rpm_install` are useful when you would like to leave an RPM untouched and perform "fixups" outside the RPM itself. The actual cluster nodes have not yet been defined so no information about number of nodes or names is available at this phase. As the name suggests these are performed on the server (after all server packages have been installed) and on the client (once the client has been "installed"[5]). This pair of scripts is pretty limited in use but helps some instances where RPMs would otherwise have to be modified.

2.3.4 Setup after Clients Defined

Once the cluster nodes have been defined the `post_clients` scripts are processed. The number of nodes, hostname/IPs and associated image are obtained in this "definition" phase. However, the nodes themselves have not yet been physically installed. This `post_clients` phase is when any package that needs knowledge about cluster nodes can query for the count, names, etc.

Notice: The method for obtaining this information is currently in flux but the following will provide a Perl hash containing hostname, domain name and IP for all nodes defined in the cluster, in numeric order.

```
use lib '/usr/lib/systeminstaller';
use SystemInstaller::Machine;

my %hash = get_machine_listing($image);   #Image can be null for default

foreach my $key (sort numerically keys %hash) {   #Key is nodenameN
    print $hash{$key}->{HOST}, ", ";
    print $hash{$key}->{DOMAIN}, ", ";
    print $hash{$key}->{IPADDR}, "\n--------\n";
}

# Special sort() sub-rtn (uses $a, $b instead of @_)
sub numerically
{
    $a =~ /(\d+)$/;  $A = $1;  #pickoff node num & save in local var
    $b =~ /(\d+)$/;  $B = $1;  #pickoff node num & save in local var
    ($A <=> $B);
}
```

[5]In the case of OSCAR this means that the SIS image (i.e., SystemImager image) has been built and client software is installed in this `chroot`'ed environment on the server node, but is not yet on the physical compute node hardware. This precludes the use of things like client specific environment variables or process managment tools e.g., `service`.

2.3.5 Completing cluster configurations

The final configuration script that fires is `post_install`. At this stage the nodes are completely installed and booted. It is typically assumed that they are accessible via C3 commands, e.g., `cexec` – parallel cluster execution. Any closing modifications are performed, such as restarting service or pushing out files, e.g., `cpush` account files.

2.3.6 Package Uninstall

The two "uninstall" scripts are run after the actual OSCAR package RPMS have been removed. The `post_server_-rpm_uninstall` and `post_client_rpm_uninstall` scripts are used to clean up any modifications or files added outside the RPM, i.e., via OSCAR API scripts. As with all other OSCAR API script, these scripts should be re-runnable.

> **Notice:** Currently (Nov. 2003) all package removal operations are explicitly listed in these uninstall scripts. This includes the RPMs themselves. When a package is uninstalled, these scripts contain all information/operations that are used to remove the package from the system.

2.4 testing

Tests are run for each package. The two scripts that are available for this testing are: `test_root` and `test_user`. These testing scripts may be written in any language so long as they are executable. When tests are run for the cluster, all `test_root` scripts are executed which perform any root level package tests.

> **Notice:** There are obvious security issues with this but currently all operations in the cluster installation are being performed by `root` so care is expected at all phases. The user tests are run as an actual user (`oscartst`) so those tests are slightly "less" dangerous and therefore most packages are using `test_user`.

The tests typically have PBS available and most of the `test_user` scripts simply setup and run a basic PBS job for the installed software, e.g., PVM, MPI's. Tests making use of PBS must be submitted via the `test_user` script. The `pbs_test` helper script is used to display results for the package's test as "PASSED" or "FAILED" based upon return codes. This script can be run in interactive or non-interactive mode. The arguments are detailed at the top of the `pbs_test` file and by using the '`--help`' option interactively.

As each `test_user` script is processed, the list of nodes are passed as command line arguments. The following is an excerpt taken from an example `test_user` script that submites a PVM job to PBS via `pbs_script.pvm`.

```
#!/bin/sh
cd $HOME
clients='echo $@ | wc -w'    # Get number of args (nodes)

$HOME/pbs_test $clients 1 $HOME/pvm/pbs_script.pvm "SUCCESSFUL" \
  $HOME/pvm/pvmtest 3 "PVM (via PBS)"

exit 0
```

(For further details see files in `$OSCAR_HOME/testing` and `$OSCAR_HOME/lib/OSCAR/Package.pm`.)

2.5 doc

This directory contains supplemental documentation for the package. There are a few pre-defined LaTeX files that may be incorporated into the overall OSCAR documentation if the package's classification is either *core* or *selected* [6]. These files are: `install.tex`, `user.tex` and `license.tex`. The first is added to the overall `install.pdf` and contains information related to the installation of the particular software package. The latter two files are incorporated into the `user.pdf`. The user information can be complete or simply pointers to obtaining more thorough documentation for the particular package. The license for all packages are listed in this document based on the contents of this `license.tex` file.

[6] That is to say the package is included in the main distribution tarball – not obtained via OPD.

3 Example Package

3.1 A basic `config.xml`

The following shows the meta file for a package called SSSlib.

> **Notice:** Due to the XML parser module used, currently multi-line elements like <`description`> must have the closing tag on the same line as the last line of text.

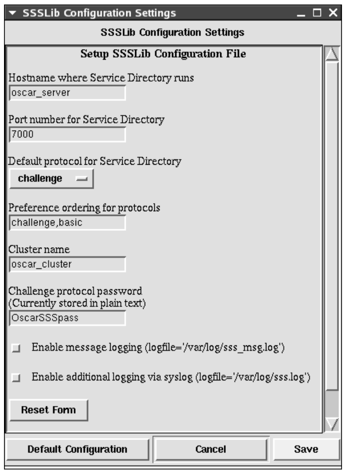
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<oscar>
  <name>SSSLib</name>

  <version>
    <major>0</major>
    <minor>95</minor>
    <subversion>2</subversion>
    <release>1</release>
    <epoch>1</epoch>
  </version>

  <class>third-party</class>

  <summary>SciDAC:SSS Infrastructure components</summary>
  <license>GPL</license>
  <group>Application/System</group>
  <url>http://www.scidac.org/ScalableSystems</url>

  <maintainer>
     <name>Narayan Desai</name>
     <email>narayan@anl.gov</email>
  </maintainer>

  <packager>
    <name>Thomas Naughton</name>
    <email>naughtont@ornl.gov</email>
  </packager>

  <description>The BCWG Infrastructure components for the SciDAC: Scalable
Systems Software.  These infrastructure components include the Service
Directory, Event Manager, etc. as well as the communication
library. </description>

  <rpmlist>
        <!-- explicitly stating 'architecture', often this is simply assumed ia32 -->
        <filter distribution="redhat" distribution_version="9" architecture="ia32"/>
        <filter distribution="redhat" distribution_version="8.0"/>
        <filter distribution="mandrake" distribution_version="9.0"/>
        <rpm>ssslib</rpm>
        <rpm>sss-clients</rpm>
        <rpm>sss-infrastructure</rpm>
        <rpm>ssslib-perl</rpm>
        <rpm>ssslib-python</rpm>
        <rpm>sss-nsm-client</rpm>
        <rpm>sss-old</rpm>
        <rpm>sss-schemas</rpm>
        <rpm>sss-validate</rpm>
        <rpm>sss-validate-python</rpm>
</rpmlist>

<rpmlist>
        <!-- explicitly stating 'architecture', often this is simply assumed ia32 -->
        <filter group="oscar_server" distribution="redhat" distribution_version="9" architecture='ia32'/>
        <filter group="oscar_server" distribution="redhat" distribution_version="8.0"/>
        <filter group="oscar_server" distribution="redhat" distribution_version="9.0"/>
        <rpm>sss-mayor-slave</rpm>
        <rpm>sss-servers</rpm>
  </rpmlist>
</oscar>
```

3.2 Using the Configurator

The following is an example `configurator.html` used to setup a configuration file. Default values are seeded so a user can simply click “Ok” if they like.

```
<html>
  <head>
    <title>SSSLib Configuration Settings</title>
  </head>

  <body>
    <form>
    <center>
    <h2>Setup SSSLib Configuration File</H2>
    </center>
    <br>

    <p> Hostname where Service Directory runs<br>
    <input name="sd_host" value="oscar_server"><br>

    <p> Port number for Service Directory<br>
    <input name="sd_port" value="7000"><br>

    <p> Default protocol for Service Directory<br>
    <select name="sd_protocol" size=1>
    <option selected>challenge
    <option>basic
    </select><br>

    <p> Preference ordering for protocols<br>
    <input name="prefs" value="challenge,basic"><br>

    <p> Cluster name<br>
    <input name="cluster" value="oscar_cluster"><br>

    <p> Challenge protocol password<br>
    (Currently stored in plain text)<br>
    <input name="password" value="OscarSSSpass"><br>

    <p> <input type="checkbox" name="msg_log" value="/var/log/sss_msg.log">
    Enable message logging (logfile='/var/log/sss_msg.log')<br>

    <p> <input type="checkbox" name="syslog_sss" value="/var/log/sss.log">
    Enable additional logging via syslog (logfile='/var/log/sss.log')<br>

    <p> <input type="reset" value='Reset Form'>
    </form>
  </body>
</html>
```

Below is an image of this HTML code as rendered by the Configurator:


SSSLib Configuration Settings

SSSLib Configuration Settings

Setup SSSLib Configuration File

Hostname where Service Directory runs
oscar_server

Port number for Service Directory
7000

Default protocol for Service Directory
challenge

Preference ordering for protocols
challenge,basic

Cluster name
oscar_cluster

Challenge protocol password
(Currently stored in plain text)
OscarSSSpass

Enable message logging (logfile='/var/log/sss_msg.log')

Enable additional logging via syslog (logfile='/var/log/sss.log')

3.2.1 post_configure

This is a simple `post_configure` example that process the results of the Configurator that are stored in the XML `.configurator.values` file. This reads those values to generate a configuration file `/etc/ssslib.conf`.

```
  #!/usr/bin/env perl
  # 'post_configure' -- reads Configurator result to setup '/etc/ssslib.conf'

  use XML::Simple;
  use Carp;

  my @fields = (sd_host, sd_port, sd_protocol, prefs, cluster, password,
                msg_log, syslog_sss);
  my $conf     = ">/etc/ssslib.conf";
  my $xml_data = "$ENV{OSCAR_PACKAGE_HOME}/.configurator.values";


  my $ref = XMLin($xml_data) or croak("Error: unable to open ($xml_data)");
  open(CONF, $conf) or croak("Error: unable to open ($conf)\n");

  print CONF "# SSSLib configuration file\n# Generated by OSCAR\n\n";
  foreach $key  (@fields) {
          if( defined($ref->{"$key"}) ) {
                  print CONF $key, "=", $ref->{$key}, "\n";
          }
  }
  print CONF "\n";

  close(CONF);
```

There is one important issue with multiline <select> elements. If the user selects only one <option>, XMLin() in its basic form above will return the value for the <select> element as a SCALAR. But if the user selects multiple <option>s, XMLin() will return the value as an ARRAY. To handle this case you have two options:

1. Use the Perl function `ref()` to determine if each hash value is an ARRAY or a SCALAR and handle it appropriately.

2. Call XMLin as `XMLin(FILENAME, forcearray => '1')`. This will force every hash value to be an array. Thus, you can either iterate through all hash values, or you can access the 0th element of the hash value for those inputs you know will have only one value.

3.3 post_clients

This is an example taken from C3 that builds the `/etc/c3.conf` using the information that was obtained after the nodes of the cluster have been defined.

```
#!/usr/bin/perl
# 'post_clients' -- generate the C3 configuration file '/etc/c3.conf'

use strict;
use Carp;
use lib '/usr/lib/systeminstaller';
use SystemInstaller::Machine;
use Data::Dumper;

my $image = shift;

my $c3_conf = "/etc/c3.conf";

my $hostname = `hostname`;
chomp($hostname);

my %hash = get_machine_listing($image);

open(OUT,">$c3_conf") or croak("Couldn't open $c3_conf");
print OUT "cluster oscar_cluster {\n";
print OUT "\t$hostname\n";
my $firstkey = 1;
foreach my $key (sort numerically keys %hash) {
    if ($firstkey)
      {
        $hash{$key}->{HOST} =~ /([a-zA-Z_\-]+)(\d*)/;
        print OUT "\tdead remove_line_for_0-indexing\n" if
          (defined($2) && ($2 != 0));
        $firstkey = 0;
      }
    print OUT "\t", $hash{$key}->{HOST}, "\n";
}
print OUT "}\n";
close(OUT);
```

```
# Sort hostnames numerically instead of an ascii sort.
# Special sort() sub-rtn (uses $a, $b instead of @_)
sub numerically
{
    $a =~ /(\d+)$/;  # pickoff number and
    my $A = $1;      #  save in local var

    $b =~ /(\d+)$/;  # pickoff number and
    my $B = $1;      #  save in local var

    ($A <=> $B)
}
```

3.4 post_install

These two examples show two very simple examples of OSCAR packages that are using this final phase script. The OPIUM package is used to synchronize system files across the cluster, e.g., account files like '/etc/passwd', etc. It takes a simple option to `--force` synchronization once all nodes have been built since the account information is not stored in the image that is used to build the machines.

```
    #!/bin/sh
    /opt/opium/bin/sync_users --force
```

The other package example making use of `post_install` is the NTP configuration package. It uses a C3 parallel execution command to restart the NTP daemon on all nodes of the cluster.

```
    #!/bin/bash
    # Post install action to restart ntpds.  This is necessary because when the
    # node is booting for the first time, other nodes are sometimes building,
    # causing network saturation.  This causes the ntpd to give up and try
    # later.

    . /etc/profile
    cexec /etc/init.d/ntpd restart
```

4 OSCAR Package Downloader (OPD)

The OSCAR Package Downloader (OPD) provides the capability to download and install OSCAR software from remote package repositories. A package repository is simply an FTP or web site. Given the ubiquitous access to FTP and web servers, any organization can host their own OSCAR package repository and publish their packages on it. There is no central repository; the OPD network was designed to be distributed such that no central authority is required to publish OSCAR packages. Although the OPD client program downloads an initial list of repositories from the OSCAR Working Group web site,[7] arbitrary repository sites can be listed on the OPD command line.

Since package repositories are FTP or web sites, any traditional FTP client or web browser can also be used to obtain OSCAR packages. Most users prefer to use the OPD client itself, however, because it provides additional functionality over that provided by traditional clients. OPD offers two interfaces: a simple menu-based mechanism suitable for interactive use and a command-line interface suitable for use by higher-level tools (or automated scripts).

Partially inspired by the Comprehensive Perl Archive Network (CPAN), OPD provides the following high-level capabilities:

- Automating access to a central list of repositories
- Browsing packages available at each repository
- Providing detailed information about packages
- Downloading, verifying, and extracting packages

While the job that OPD performs is actually fairly simple and could be performed manually, having an automated tool for these functions provides ease of use for the end-user, performs multiple checks to ensure that downloaded and extracted properly, and lays the groundwork for higher-level OSCAR package/retrieval tools.

[7]The centralized repository list is maintained by the OSCAR working group. Upon request, the list maintainers will add most repository sites.

5 OSCAR Database (ODA)

The database used in OSCAR is called ODA. The information about software packages and cluster nodes is stored in the database. There is both a command-line (`oda`) and Perl (`oda`) interface. There are several "shortcuts" defined to perform queries and mask the underlying data representation. The current DB engine is MySQL. See ODA(1), ODA(3pm).

Notice: More information should be added to this section. However, only a few packages are currently using ODA and are relatively sophisticated.

6 OSCAR Configurator

This section provides details on the syntax and usage of the OSCAR Configurator.

Writing the `configurator.html` File

To present configuration options to the user, the Configurator uses a simple HTML **Form**. While the basics of writing HTML files and HTML forms are discussed in detail in many books and online tutorials, this section seeks to describe the HTML form tags available to a package maintainer and how to use those tags to get information from the user.

Many of the basic HTML tags are supported in the OSCAR Configurator, including:

- various fonts and headers
- line and paragraph breaks
- horizontal (ruler) lines
- centered text
- images (GIFs and JPEGs)

There is also a decent assortment of HTML Form tags such as:

- input fields
- checkboxes
- radio buttons
- single- and multi-selection lists
- text boxes

All of the available tags are described in detail in below. There are some HTML tags that are NOT allowed in a configuration file. In particular, you may **not** use:

<`A HREF`> - A hypertext link to other places on the web. We don't want the user going off to another web page.

<`TABLE`>, <`TR`>, <`TD`> - While tables are parsed in without difficulty, they are not rendered as true talbes. Better to use <`PRE`> (preformatted text) for this purpose.

<`ISINDEX`> - The document cannot be searchable.

<`SUBMIT`>, <`BUTTON`>, & <`IMAGE`> - The Configurator has its own button to submit the values of the form to the program, so these elements are unnecessary.

<`FILE`> - The user is not allowed to upload a file.

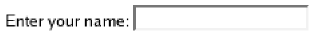
If any of these tags are present in your HTML document, they are removed prior to rendering so that your document will appear without the offending tags.

A Basic HTML Form Document

Since all of the configuration input from a user appears in an HTML Form, the minimum useful configuration file contains <`form`> and <`/form`> tags. Between these two tags, the package maintainer adds <`input`> tags to get input from the user. For example, if all you want from the user is his name, your configuration file might be as follows:

```
<form>
Enter your name: <input name="username">
</form>
```

The resulting output would look like this:



If you are HTML savvy, you may notice a few things missing from this example. Specifically, we omitted the main <`html`>...<`/html`> tags, and the <`head`><`/head`> and <`body`><`/body`> tags. For basic configuration files, these tags are optional. You may use them if you like (and some browsers would bonk if they were absent), but they are not required. Also, there are no attributes for the <`form`> tag. This is because the Configurator knows the action to take when the user saves the values, so no "action" label is needed. Also, there can be only one pair of <`form`><`/form`> tags in the configuration file, so there is no need for the "name" label.

A More Complete HTML Form

A typical HTML document consists of at least two separate sections, a "head" and a "body". The most important element of the "head" is the <`title`>...<`/title`> tag. This sets the title of the window and also acts as the text for the header of the configuration window. If you omit this tag or leave the value blank, it defaults to "Configuration".

The body is where the main part of the HTML document is stored. It includes both the Form and any other "standard" HTML elements you want to display. Many of these elements can appear inside or outside of the Form, so it is sometimes easiest to have a configuration form that looks like this:

```
<html>
  <head>
    <title>The Title Of Your Configuration Form<title>
  </head>

  <body>
    <form>
      All of your main HTML configuration goes here.
    </form>
  </body>
</html>
```

In HTML files, whitespace (including carriage returns) are usually compacted to a single space. So in the above example, the indentation and line spacing are shown simply to make the example easier to read. We could just have easily put everything on a single line and the output would be the same. This is important to know since if you want a true line break (say between paragraphs), you have to explicitly tell HTML by using the <`p`> (paragraph) tag. Also, the case of the tags is unimportant, so you could also use all uppercase letters for your tags if you find that to be more readable.

Basic (non-form) HTML Tags

There are many HTML tags which can be displayed by the Configurator that don't need to be in the form, but will be useful anyway. The function of each of these tags are not listed here. There are plenty of HTML reference books and online tutorials available. Use your favorite search engine and do a search on "HTML tutorial" to get a big listing.

HTML Form Tags

The main tags that you will need for your configuration file are <`form`> tags. These tags allow you to prompt the user for information to be entered via text boxes, check boxes, radio buttons, selection lists, etc. This section describes each tag in detail and provides examples.

Note that all of these tags must appear between a <`form`>...</`form`> tag pair. Otherwise your values will not get submitted correctly.

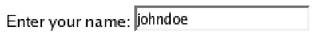
- <input type="text" name="VARIABLE" value="INITIAL" size="NUMCHARS" maxlength="MAXCHARS">

  A *text* element is a single line text input field in which the user can enter text. If you do not specify **type="text"**, then the <`input`> defaults to this type of text input field. Of the various attributes, only **name** is required. The **name** attribute designates the variable name for the data entered by the user. If you also specify the **value** attribute, that text appears in the text input field when it first appears. The **size** and **maxlength** attributes designate the number of characters that appear in the text box and the maximum length of the input text respectively.

  Example:

  ```
  <form>
  Enter your name:
  <input type="text" name="username" value="johndoe" size="20" maxlength="30">
  </form>
  ```

  Output:

  
  Enter your name: johndoe

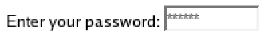
- <input type="password" name="VARIABLE" value="INITIAL" size="NUMCHARS" maxlength="MAXCHARS"> -

  A *password* element is a text input field in which each character typed is displayed as a * to conceal the actual value entered. In all other aspects, this element is the same as the *text* element.

  Example:

  ```
  <form>
  Enter your password:
  <input type="password" name="password" value="passwd" size="10" maxlength="20">
  </form>
  ```

  Output:

  
  Enter your password: ******

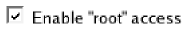
- <input type="checkbox" name="VARIABLE" value="RETURNVALUE" checked> -

  A *checkbox* is a toggle that the user can select (switch on) or deselect (switch off). As with all <`input`> elements, the **name** attribute is required and designates the variable name for the value returned by the check box. Usually, "ON" is returned when a check box has been checked by the user. If you specify the **value** attribute, that value is returned instead when the check box has been checked by the user. You can optionally specify the **checked** attribute to make the check box initially selected when first displayed. Otherwise, the check box is unselected when first displayed. Any check boxes which are not checked when the form is submitted do not get passed to the Configurator, ie. their values will be "" (the empty string).

  Example:

  ```
  <form>
  <input type="checkbox" name="rootaccess" value="YES" checked>
  Enable "root" access
  </form>
  ```

Output:



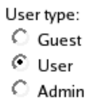
- <input type=”radio” name=”VARIABLE” value=”RETURNVALUE” checked> -

  A *radio* element is a radio button. A set of radio buttons consists of multiple radio buttons that all have the same **name** attribute. Only one radio button in the set can be selected at one time. When the user selects a button in the set, all other buttons in the set are deselected. If one radio button in a set has the **checked** attribute, that one is selected when the set is first displayed. Otherwise, none of the radio buttons are selected when first displayed (which may not be the desired functionality). As with all `<input>` elements, the **name** attribute is required and designates the variable name for the value returned by the radio button set. All radio buttons with the same **name** are in the same set, regardless of where they appear in the form. The **value** attribute is the value that is returned for the radio button set when the form is submitted. This default to “ON” which isn’t very useful for a set of radio buttons, so be sure to give each radio button its own **value**.

  Example:

  ```
  <form>
  User type:<br>
  <input type="radio" name="usertype" value="guest"> Guest<br>
  <input type="radio" name="usertype" value="user" checked> User<br>
  <input type="radio" name="usertype" value="admin"> Admin<br>
  </form>
  ```

  Output:



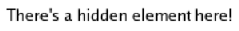
- <input type=”hidden” name=”VARIABLE” value=”RETURNVALUE”> -

  A *hidden* element is an invisible element whose main purpose is to contain data that the user does not enter. This data gets sent when the form is submitted. This allows you to always pass a certain name/value pair to the Configurator without input from the user. Both the **name** and **value** attributes are required.

  Example:

  ```
  <form>
  There’s a hidden element here!
  <input type="hidden" name="version" value="5.23">
  </form>
  ```

  Output:



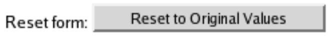
- <input type=”reset” value=”LABEL”> -

  When the user presses a *reset* button, all elements in the form are reset to the values that were present when the form was first displayed. Usually, the text of this button is “Reset”, but you can change this by specifying the **value** attribute.

  Example:

  ```
  <form>
  Reset form: <input type="reset" value="Reset to Original Values">
  </form>
  ```

  Output:



- <input type="reset"
- <textarea name="VARIABLE" cols="WIDTH" rows="HEIGHT"

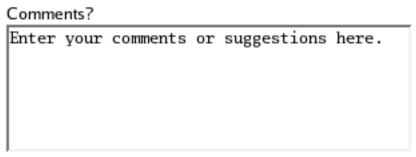
  wrap="OFF"—"HARD"—"SOFT">Text to display</textarea> -

  The *textarea* tag defines a multiline input field into which the user can enter text. As with <`input`> elements, the **name** attribute is required and designates the variable name for the text present in the box when the form is submitted. The width and height (in terms of characters) of the text box is given by the optional **cols** and **rows** attributes. By default, any text entered by the user is displayed "as is" meaning that if the line input by the user is longer than the width of the text box, the text will scroll off the screen. The only time a new row is started is when the user types a carriage return. This is also the behavior when you use the **wrap="OFF"** attribute. If you want to have the text word wrap automatically, use the **wrap="HARD"** or **wrap="SOFT"** attribute. (In standard HTML, these two attributes differ by whether or not the extra carriage returns generated by word wrapping get submitted in the text or not. For the Configurator, these extra carriage returns are never submitted, which is equivalent to **wrap="SOFT"**, so using **wrap="HARD"** generates the same behavior.) Between the two <`textarea`>...</`textarea`> tags, you can put optional "Text to display" when the textarea is first displayed.

  Example:

```
<form>
Comments?<br>
<textarea name="comments" cols=40 rows=5 wrap="SOFT">
Enter your comments or suggestions here.
</textarea>
</form>
```

  Output:



Comments?

```
Enter your comments or suggestions here.
```

- <select name="VARIABLE" size="LISTLENGTH" multiple> <option value="OPTIONVALUE"> <option value="OPTIONVALUE" selected> ... </select> -

  The *select* and *option* tags define a selection list. A selection list displays a list of options from which the user can select one (or more) items. If the **multiple** attribute is present, the user can select multiple items from the list at a time. Otherwise, only a single item can be selected at a time. As with **input** elements, the **name** attribute is required and designates the variable name for the value(s) selected in the list when the form is submitted. The optional **size** attribute indicates how many items are presented in the box before scrolling is necessary. This defaults to "10". If you set **size** to "1" and do not set the **multiple** attribute, you get a single element drop-down list.

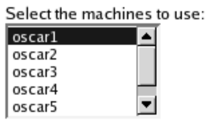
  To actually put items in the selection list, you use the *option* tag followed by the text you wish to appear in the list. You can make that option selected when the list is initially displayed by using the **selected** attribute. By default, the value that gets returned to the Configurator when an item is selected is the actual text of the item in the list. You can override this behavior by using the optional **value** attribute. When you set this value, it gets returned when that item in the list is selected.

  Example:

```
<form>
Select the machines to use:<br>
<select name="machinelist" size=5 multiple>
```

```
<option selected>oscar1
<option>oscar2
<option>oscar3
<option>oscar4
<option>oscar5
<option>oscar6
<option value="unlisted">unlisted machine
</select>
</form>
```

Output:



Note: when viewing the above example in a standard HTML browser, you might have to use the <`SHIFT`> or <`CTRL`> key in conjunction with clicking the mouse to select multiple items from the list. In the Configurator, this extra keypress is not required.

A Rules of Thumb

A.1 Install Location

The general suggestion is to install into a non-global directory location. Typically the `/opt` directory is used in OSCAR. This was based on recommendations from the Linux File System Hierarchy standard drafts. When installing into non-global directories, the recommended method for adding your application to `PATH` is via the Env-Switcher tool (aka Switcher) Appendix B, 18.

A.2 OSCAR Specific RPMS

When creating RPMs that are specific in some manner to OSCAR it is generally a good idea to modify the package's "Name:" (within the `.spec` file). This reduces conflicts with any mainstream versions of the package when using automated update tool for example. The suggested format is: `<name>-oscar-<ver>-<rel>.<arch>.rpm`, e.g., `lam-oscar-7.0.4-1.i586.rpm`.

Another tip when making modifications to an existing RPM is to make sure you update the "Packager:" information to reflect who did the re-packaging. Also it can be helpful to change the "Conflicts:" and "Provides:" information as follows.

- Conflicts: ¡Mainstream Packages Name/Version¿
- Provides: ¡Name of Mainstream Package/Version¿

This can be used to keep the mainstream items from accidentally working with the modified OSCAR specific versions via the "Conflicts:". Additionally this lets you satisfy circular depedencies for packages you don't rebuild via the "Provides:".

A.3 `init.d` scripts

It is a good idea to honor the `start`, `stop`, and `restart` targets. It is also helpful to have a `status` target to display whether the given application is running currently. Also, the scripts should behave properly when used in conjunction with the `service` command, e.g., `service sshd status`. The `service` command is used throughout the Red Hat distributions and may be part of the new LSB drafts (*not 100% sure on LSB*). The command runs in a stripped down environment which sometime caused problems if assuptions are made about the execution shell environment.

A.4 Generating Configuration Files

Often a default configuration file is included in the RPM itself. While this can be helpful, it often leads to scripts that have to modify the default config file. It is generally cleaner to just generate the file in one place.

Another thing to remember is that the hostname "oscar_server" should always resolve to the internal interface of the OSCAR cluster. This can be helpful when generating defaults.

One other note

B Env-Switcher

Managing the shell environment – both at the system-wide level as well as on a per-user basis – has historically been a daunting task. For cluster-wide applications, system administrations typically need to provide custom, shell-defendant startup scripts that, create and/or augment `PATH`, `LD_LIBRARY_PATH`, and `MANPAGE` environment variables. Alternatively, users could hand-edit their "dot" files (e.g., `$HOME/.profile`, `$HOME/.bashrc`, and/or `$HOME/.cshrc`) to create/augment the environment as necessary. Both approaches, while functional and workable, typically lead to human error – sometimes with disastrous results, such as users being unable to login due to errors in their "dot" files.

Instead of these models, OSCAR provides the **env-switcher** OSCAR package. **env-switcher** forms the basis for simplified environment management in OSCAR clusters by providing a thin layer on top of the Environment Modules package (Furlani & Osel). Environment Modules provide an efficient, shell-agnostic method of manipulating the environment. Basic primitives are provided for actions such as: add a directory to a `PATH`-like environment variables, displaying basic information about a package, and setting arbitrary environment variables. For example, a module file for setting up a given application may include directives such as:

```
setenv FOO_OUTPUT $HOME/results
append-path PATH /opt/foo-1.2.3/bin
append-path MANPATH /opt/foo-1.2.3/man
```

The **env-switcher** package installs and configures the base Modules package and creates two types of modules: those that are unconditionally loaded, and those that are subject to system- and user-level defaults.

Many OSCAR packages use the unconditional modules to append the `PATH`, set arbitrary environment variables, etc. Hence, all users automatically have these settings applied to their environment (regardless of their shell) and guarantee to have them executed even when executing on remote nodes via `rsh`/`ssh`.

Other modules are optional, or a provide one-of-many selection methodology between multiple equivalent packages. This allows the system to provide a default set of applications, that optionally can be overridden by the user (*without* hand-editing “dot” files). A common example in HPC clusters is having multiple Message Passing Interface (MPI) implementations installed. OSCAR installs both the LAM/MPI and MPICH implementations of MPI. Some users prefer one over the other, or have requirements only met by one of them. Other users wish to use both, switching between them frequently (perhaps for performance comparisons).

The **env-switcher** package provides trivial syntax for a user to select which MPI package to use. The `switcher` command is used to select which modules are loaded at shell initialization time. For example, the following command shows a user selecting to use LAM/MPI:

```
shell$ switcher mpi = lam-6.5.9
```

B.1 Example Switcher Module file

This example was taken from the PVM module file, which gets loaded unconditionally for all shells.

```
# PVM modulefile for OSCAR clusters (based on LAM modulefile)

proc ModulesHelp { } {
  puts stderr "\tThis module adds PVM to the PATHand MANPATH."
  puts stderr "\tAdditionally the PVM_ROOT and PVM_ARCH are set."
}

module-whatis   "Sets up the PVM environment for an OSCAR cluster."

setenv PVM_RSH  ssh
setenv PVM_ROOT /opt/pvm
setenv PVM_ARCH LINUX

append-path MANPATH /opt/pvm/man

append-path PATH /opt/pvm/lib
append-path PATH /opt/pvm/lib/LINUX
append-path PATH /opt/pvm/bin/LINUX
```

C Supported XML Tags

The following table contains the currently recognized XML tags along with a brief description. The syntax used for XML elements and attributes:

Table syntax:	`<element>`	implies	`<element>...</element>`
	`<element/>`	implies	`<element ... />`
	`attr=`	implies	`<element attr= ...`

Require	Elements	Sub Elements	Description
Y	`<xml>`		Standard XML opening element e.g., `<?xml version="1.0" encoding="ISO-8859-1"?>`
Y	`<oscar>`		Top-level OSCAR Package element
Y	`<name>`		Name of the software package
Y	`<version>`		Version of the software package
		`<major>`	Major version of the software package
		`<minor>`	Minor version of the software package
		`<subversion>`	Sub-version of the software package
		`<release>`	Release number of the software package
		`<epoch>`	Release number of the software package
N	`<class>`		Classification of package: 'core', 'selected', or 'third-party'(default)
N	`<installable>`		Boolean integer install flag, *Configurator* modifiable (1=install OR 0=not-install)
Y	`<summary>`		Brief one-line description of package
N	`<license>`		Software license for the package
N	`<group>`		RPM-style software group, e.g., Application/System
N	`<url>`		Homepage for software package
N	`<packager>`		Packager of OSCAR package
		`<name>`	Name of package author
		`<email>`	Email address of package author
N	`<maintainer>`		Original author of software
		`<name>`	Name of package author
		`<email>`	Email address of package author
N	`<vendor>`		Vendor name for package
Y	`<description>`		Brief description of the package
N	`<rpmlist>`		List RPMS included in the rpmlist
		`<filter/>`	Filter applied to all RPM in list
		`<rpm>`	Name of RPM (without version)
N	`<filter/>`		Filter Attributes used on RPMLIST
		`group=`	Attribute specifying target location for RPMS e.g., "oscar_server" or "oscar_clients"
		`distribution=`	Attribute specifying supported distribution e.g., "redhat", "mandrake", "suse", "rhas" see also: *lib/OSCAR/Distro.pm*
		`distribution_version=`	Attribute specifying supported distribution version e.g., "7.3", "9", "2.1AS"
		`architecture=`	Architecture the RPM was compiled to support e.g., "ia32", "ia64"
N	`<requires>`		Express requirement/dependency
		`<type>`	Specifying dependence type: 'package' (OSCAR pkg name)
		`<name>`	Name for dependence analysis
N	`<provides>`		Expresss capabilities/synonym for package name
		`<type>`	Specifying dependence type: 'package' (OSCAR pkg name)
		`<name>`	Name for dependence analysis
N	`<conflicts>`		Expresss conflicts for dependence analysis
		`<type>`	Specifying dependence type: 'package' (OSCAR pkg name)
		`<name>`	Name for dependence analysis
N	`<package>`		Package specific namespace, to define own tags
N	`<download>`		Location to obtain package (see also: OPD)
		`<uri>`	Full Universal Resource Identifier, to download pkg
		`<size>`	Integer size (bytes) for tarball specified by `<uri>`
		`<md5sum>`	Value of MD5 sum for tarball specified by `<uri>`
		`<sha1sum>`	Value of SHA1 sum for tarball specified by `<uri>`
N	`<oda>`		OSCAR Database (ODA) defines for package
		`<shortcut>`	ODA shortcut defined by a package

Table 4: Overview of existing XML tags for OSCAR Packages.