

# The OurayCM User Manual

<b>1. INTRODUCTION.....</b>	<b>1</b>
<b>2. CORE CONCEPTS.....</b>	<b>2</b>
<b>3. ARCHITECTURE.....</b>	<b>4</b>
OURAYCM DEPLOYMENT CONFIGURATIONS.....	5
<b>4. NOTATION.....</b>	<b>6</b>
THE OCM COMMAND LINE.....	6
FILE AND DIRECTORY NOTATION.....	6
BRANCH NOTATION.....	6
VERSION NOTATION.....	6
<b>5. GETTING STARTED.....</b>	<b>8</b>
MINIMUM RECOMMENDED COMPUTER.....	8
INSTALLATION INSTRUCTIONS.....	8
USING OCM.....	8
<i>Creating a Repo.....</i>	8
<i>Starting a Server.....</i>	9
<i>Connecting a Workspace.....</i>	9
<i>Adding many files at once.....</i>	13
ADDITIONAL IMPORTANT OCM COMMANDS.....	13
GENERAL GUI USAGE.....	14
<i>Mouse.....</i>	14
<i>Scrollbars.....</i>	14
<i>Text editing.....</i>	15
<i>Text Select/Copy-paste.....</i>	15
<i>Keyboard Navigation.....</i>	15
WHERE TO GO FROM HERE.....	15
<b>6. SAVING.....</b>	<b>16</b>
FINDING THE CHANGES.....	16
NEW ENTRIES.....	17
<i>Add.....</i>	17
<i>Delete.....</i>	17
<i>Skip.....</i>	17
<i>Ignore.....</i>	17
SPECIFYING RENAMES.....	17
<i>The Rename GUI.....</i>	18
<i>The renames file.....</i>	18
<b>7. UPDATE/PULL AND CONFLICT RESOLUTION.....</b>	<b>19</b>
FILE CONFLICTS.....	19
LINE CONFLICTS.....	19
LOCATION (NAMING) CONFLICTS.....	20
<b>8. REVERTING.....</b>	<b>21</b>
<b>9. VREFS AND RELEASE MANAGEMENT.....</b>	<b>22</b>
RELEASE LABELING (CREATING VREFS).....	22
RELEASE MAINTENANCE TIPS.....	22
<b>10. REMOTE REPO.....</b>	<b>24</b>
IMPORTANT, PLEASE READ:.....	24
SETTING UP A REMOTE REPO.....	25

1. <i>Create remote branches</i> .....	25
2. <i>Create remote repo</i> .....	25
3. <i>Configure Ownership</i> .....	25
4. <i>Deploy remote repo</i> .....	26
<b>11. WORKSPACE SUPPORT FILES</b> .....	<b>27</b>
WORKSPACE CONFIG DATABASE.....	27
THE .OCM DIRECTORY.....	27
IGNORE FILES.....	28
<i>Ignore File Format</i> .....	28
<b>12. ADMINISTRATION</b> .....	<b>30</b>
RENEWAL.....	30
BACKUP.....	31
RESTORE.....	31
UNDOING REPO OPERATIONS.....	31
<i>Recovery of a backup version</i> .....	32
<i>Removing repo commits</i> .....	32
<i>Rebuilding the repo</i> .....	33
WORKSPACE NON-ATOMICITY.....	33
<b>13. ACKNOWLEDGMENTS</b> .....	<b>34</b>

Copyright© 2003-2004 Ouray Software, LLC. Permission to create copies of this document is hereby granted for the exclusive purposes of using and evaluating OurayCM. All other uses of this document, including modification or general redistribution, are prohibited.



# 1. Introduction

This manual is designed to give the uninitiated reader a solid, comprehensive but not overly-detailed understanding of how to use OurayCM. Chapters 1-5 are critical to read before attempting to use the tool; they are brief, but provide enough information for you to quickly be using the tool. It is recommended to at least browse through the rest of the manual.

We at Ouray Software have a firm belief not only in doing things the right way, but in taking due care in figuring out exactly what that right way is. We also know that we can't do everything all at once, and that we sometimes err in figuring out the right way to do something. For these reasons, we are quite attentive to customer feedback: it helps us prioritize forthcoming features and evaluate whether or not an existing feature should change. So feel free to send us an email to [support@ouraysoftware.com](mailto:support@ouraysoftware.com) and tell us what you would like to see in future versions of the tool. If your request is simple enough, you may find that we can turn your request around in as little as a week.

## 2. Core Concepts

The key concepts in OurayCM are those of the workspace, version, and branch.

A “workspace” is a directory hierarchy containing an organized set of files that the user edits and/or uses. The workspace stores the entire set of source code or other files the user is editing and that are required to build the project he is working on. The files typically represent the direct result of conscious work, versus the automatically generated files that are compiled from this source<sup>1</sup>. A workspace might be the directory `/home/joe/project/src`—the place where Joe's working copy of the project source exists, as opposed to `/home/joe/project/dst`—the place where he puts his compiled artifacts.

A workspace is simply an area on the disk; it is the starting point for CM operations, it is not in itself part of OurayCM—you have a workspace whether or not you're using OurayCM. Not all of the files in a given workspace are necessarily managed by OurayCM; some can be compiler artifacts or temporary files the user is working on but that are not seen by the rest of the team.

A “version” is a copy of the workspace as it existed at a given point (minus specifically ignored temporary files and artifacts), efficiently stored in OurayCM as a difference from the previous version. In OurayCM, both files and directories are versioned as part of the workspace. OurayCM also has a properly implemented directory and file rename facility.

A “branch” is a sequential set of durable versions. Having multiple branches allows different workspaces to evolve independently; this feature is needed when a single user needs to maintain different versions of his workspace in parallel (such as when maintaining different releases of a product) or when two or more users are working together.

Branches are organized hierarchically to reflect the hierarchical nature of typical parallel development, where a project is broken into tasks that are worked on in parallel, which themselves might be broken down into smaller tasks, and so on. The topmost branch is the “root” branch, also called the “main” branch. A “child” branch is a branch organized under another branch, its “parent” branch. “Ancestor” branches include the parent of a given branch, the parent's parent, etc., up to the ultimate ancestor: the root branch. “Descendant” branches include the children of a branch, the children's children, etc.

There are four different types of versions that can be created in a given branch:

- **Save.** A save version stores the state of an individual workspace. This is stored as a difference from the previous version. This version type allows the individual developer to record the state of his workspace without interfering with other developers.
- **Submit.** A submit version is created when a version from a descendant branch is published to the branch. This version type is used to publish changes to shared branches.

---

<sup>1</sup> A good principle of organization is to put all automatically-built artifacts in a completely different area (e.g., ‘dst’), instead of being strewn throughout the workspace. This makes it very simple to verify that one actually can build everything from the source—you simply delete ‘dst’ and rebuild. In case some of your files are not organized this way, OurayCM provides a means of ignoring various files in the workspace, e.g., \*.exe or \*.obj files should not usually be managed by the SCM system.

- **Update.** An update version is created when changes from an ancestor branch are incorporated into the branch. This version type is used for receiving changes published to shared branches by other developers.
- **Pull.** A pull version is created when changes from a non-ancestor branch are incorporated into the branch. This version type is used to receive changes from another branch in an ad-hoc way, e.g., when two developers are working in tandem for a short period and want to share changes with each other but not with the rest of the team.

A branch can be “connected” to a workspace. When a branch and workspace are connected, the user can use the “save” command to store the current state of the workspace in a version on the branch and the “update” and “pull” commands to make a new version that incorporates changes from other branches (and also apply the changes to the workspace).

A branch may be a “user” branch (a branch with a workspace connected to it) which stores versions for a particular user<sup>2</sup>, or a “shared” ancestor branch (a branch with descendants and no workspace connected) which stores versions published by its descendant branches. A branch can change from being a user branch to a shared branch and vice versa.

By virtue of providing a means for developers to record the state of their workspaces, OurayCM provides a detailed history of project activities for each developer and for the project as a whole. For example, if you want to know who added a given line of code, when, and on what computer, OurayCM can efficiently answer that question.

OurayCM also provides high change-flow visibility, providing an additional review aid for improving software quality. With this feature you can:

- Review all of your changes before committing to a save;
- Review the changes coming from an update or pull before accepting them;
- Review changes that will be submitted to a shared branch before publishing them.

---

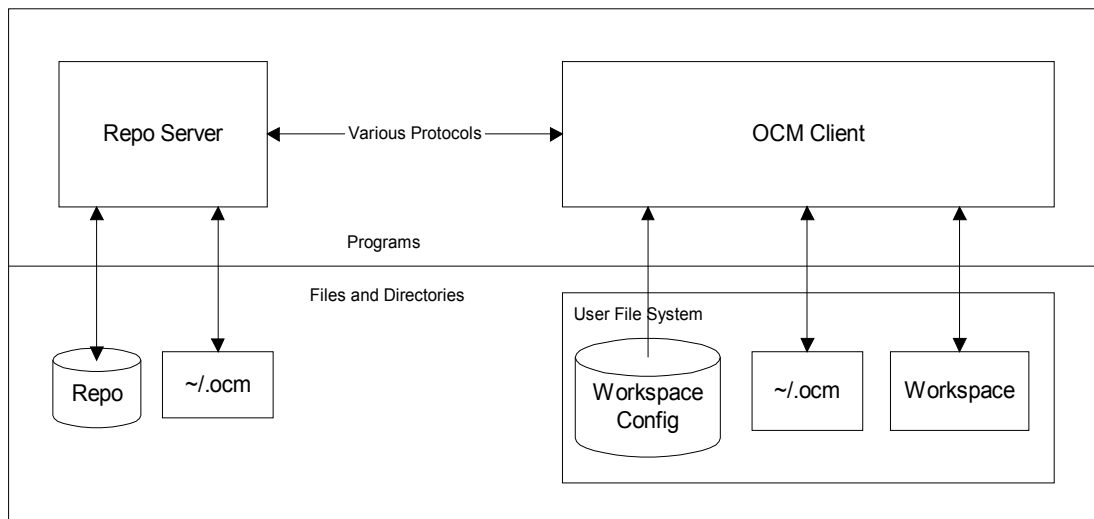
<sup>2</sup> In order to be able to independently keep track of his workspace versions as they evolve, each OurayCM user gets his own branch. This is a complicated thing to set up in some SCM systems, but it’s quite easy to set up and manage in OurayCM.

### 3. Architecture

A typical deployed OurayCM system consists of the following basic elements:

- The user workspaces, which contain the files the users work on using other applications, like compilers, editors, etc.
- The repository database, or “repo”, which contains, in essence, the history of the user workspaces.
- The workspace configuration database, a very small database specifying the connection of a particular workspace to a branch in the repo and some other workspace information. Both this database and the repository database exist as a directory on your file system with several files in it. The workspace configuration database is located at <workspace path>.ocm.
- An optional Repo Server, which provides various types of networked access to the repo.
- The OCM client, which provides the main user interaction, including saving new changes, getting updates from other branches, browsing the repo, etc.
- The .ocm directory (located in your home directory), that stores error logs and backup copies of workspace entries that OCM modifies. This directory may be deleted at will in order to conserve disk space, it is only used as a safety net and error analysis feature.

Figure 1 is a simplified schematic of how these elements relate:



**Figure 1: OCM's high-level architecture.**

This figure is merely a schematic; it does not indicate a particular deployment configuration. For example, the server is optional, since a client may access the repo directly. Also, multiple clients typically use the same repo; this figure only depicts a single client.

## ***OurayCM Deployment Configurations***

OurayCM provides various ways for the client to interact with the repo:

- **Local.** With this method, the client connects directly to the repo file with no use of networking protocols.

This method is appropriate for single-user configurations or when multiple individuals share the same physical machine.

- **Cache Server.** With this method, a cache server is run on the machine where the repo physically resides. Clients connect to the server over the network and synchronize a local cache with the server, thereby creating and maintaining an exact replica of the remote database on each client machine.

Once the client has been initially synchronized, the network bandwidth requirements of this type of use are typically very small; using OCM this way over a 56Kbps modem or over a high-speed network both work well. Since the cache is an exact replica, there is built-in data redundancy to provide protection against hard drive failure. *This is the recommended usage model for most team environments.*

- **Remote Repo.** With this method, the user(s) of the remote repo can have full SCM capability, even without a network connection to the main repo. Patches are used to synchronize the main repo with the remote repo either manually or via a synchronization server.

This method is recommended when the network is either non-existent or unreliable.

These are not mutually-exclusive types of use; they can be combined at will, depending on your needs. E.g., someone can use the repo locally while someone else uses it via the cache server.

**Note: DO NOT USE A REPO VIA A NETWORK MOUNTED FILE SYSTEM.**

Such use can cause performance and reliability problems. OurayCM is designed to interact with the repo files through the local file system.

## 4. Notation

There are several syntaxes one must be aware of in order to use OCM effectively:

- The OCM command line
- File and directory notation
- The RepoSpec
- Branch notation
- Version notation

### ***The OCM Command Line***

Access to all OurayCM functionality is provided via the program ‘ocm’. The commands are arranged hierarchically. For more information on the help system run 'ocm help'.

### ***File and directory notation***

In OCM, files and directories are referred to using the UNIX convention of the forward slash for the path separator. **On Microsoft Windows systems, forward slashes must be used, and the drive specifier should be included for absolute paths, for example: c:/project/src.**

### ***Branch Notation***

Since branches are organized hierarchically, just like directories, the simple UNIX file path notation is used for them. The difference from a file path is that with branches there are two ways to refer to branches: by name or by number. The branch number is a permanent number assigned to a branch contained within a given parent. Because the name may be reassigned at will, do not depend on the branch name for recovery purposes—use the numerical branch address instead.

For example, the following are legitimate BranchSpecs:

```
/tasks/bugfix423
/2/0
/2/windows
/joe/windows
```

### ***Version Notation***

A VersionSpec refers to a version, specified by a version number on a particular branch. The syntax is:

```
<BranchSpec>:<version number>
```

The version field can be any positive integer or a “-” which means the latest version on the branch. Examples of legitimate VersionSpecs are:

```
/tasks/bugfix423:33
```

```
/tasks/bugfix423:-  
/2/0:-  
/mdw/windows:42
```

## 5. Getting Started

In this section we give a brief overview of how to install and use OurayCM.

### ***Minimum Recommended Computer***

- 400MHz Pentium II (or equivalent on other platforms).
- 256MB RAM
- Ethernet card (a MAC address is required for licensing purposes)
- 30MB of hard drive space for OCM, plus enough room for your repository<sup>3</sup>

### ***Installation Instructions***

OurayCM exists as a single executable file on your disk, so the install process is very simple:

1. Download the appropriate release for your platform and copy to as many machines as desired. To download OCM, go to: <http://www.ouraysoftware.com/download>.
2. Extract the downloaded file and move the extracted executable (named ocm) to a suitable file location, preferably one that is already part of your shell's search path. On Windows platforms use WinZip ®, on others, use 'tar -xzf <downloaded file>'.

### ***Using OCM***

With the exception of the Remote Repo method, using OCM involves:

1. Creating a repo
2. Optionally starting a Cache Server (this step is not required if all users are working on the same machine).
3. Connecting individual workspaces to the repo and using OCM.

**NOTE FOR WINDOWS USERS:** The following example is tailored to UNIX platforms. For Windows platforms, add the drive specifier to the front of any absolute path, e.g., c:/project/repo. And set environment variables via “Control Panel | System | Advanced | Environment Variables”.

**NOTE FOR MAC USERS:** OurayCM is implemented as an X Window application on the Mac. To start the X Server, double click on the X11.app icon in / Applications/Utilities. By default, this will launch xterm. OurayCM commands may be entered in the xterm, or in a Terminal.app window.

### **Creating a Repo**

To create a new, empty repo called ‘myrepo’:

```
% ocm repo create myrepo
```

This repo will only be writable for a certain period before you must renew it. To renew, run 'ocm renewal request'.

---

<sup>3</sup> The initial repository is approximately the same size as your workspace.

See <http://www.ouraysoftware.com/pricing.html> for more details on the renewal process.

## Starting a Server

If you choose the Cache Server method (recommended for most team environments), you need to start a server. To start the Cache Server, run this on the machine where the repo directory physically resides:

```
% ocm repo cache_server myrepo
```

See 'ocm help cache\_server' for more information on this command.

You can run both the Cache Server and the Remote Server with the same repo file at the same time.

## Connecting a Workspace

A workspace connection specifies how a workspace on a user's disk relates to the repo. This includes what server method is used, the user ID, and the location of the workspace on the disk. The connection information is stored in the workspace configuration database.

### Single-user use

This scenario covers a single user using ocm with a single workspace. More complicated single-user use cases are possible, but those are not significantly different from multi-user use cases.

In this example, the user works directly on the main branch.

Here is a set of UNIX csh commands that will create a repo, connect a workspace under '/project/src' to its main branch (src should already exist), and save all of the files to the repo:

```
% ocm repo create /project/repo
% ocm ws connect /project/src / -f -r { local /project/repo }
% setenv OCM_WS /project/src
% ocm save
```

See the section "Saving" for further information on the save process.

Note that in this example we do not use a server. The 'ocm ws connect' command connects directly to the repo.

### Multi-user use

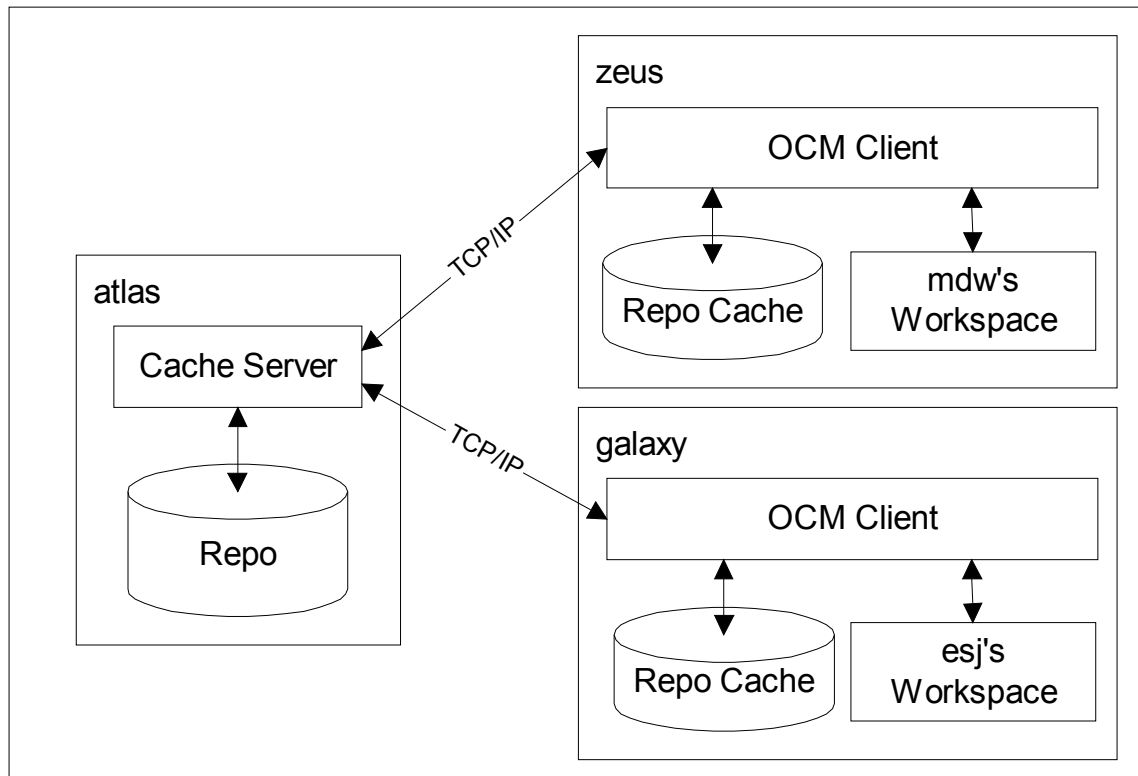
For this example we will demonstrate how to 2 users might set up and use OurayCM: Eric S. Jensen (esj), and Martin D. White (mdw). This example is easily extended to any number of users.

Using OurayCM with multiple users requires user IDs to be created and a branch to be created for each user<sup>4</sup>. Eric works on 'galaxy'; Martin works on 'zeus'. They both keep

---

<sup>4</sup> Unlike some other SCM systems, creating a branch per user in OurayCM adds little administrative overhead. For example, using a main branch with a branch per user in OurayCM is similar in complexity to using CVS with only the main branch.

their work in a directory named “/project” on their own machine. They will run the Cache Server on a machine named ‘atlas’. Here’s the final deployed configuration:



**Figure 2: Multi-user deployment example.**

To set this up we’ll need to do the following:

1. Create the repo on atlas
2. Create “esj” and “mdw” user IDs in the repo
3. Create a branch for each user
4. Start a Cache Server on atlas
5. Create the users’ workspaces on their machines using a Cache Server

After going through these steps we will also demonstrate how one typically uses the tool. (See the ‘ocm help’ for each command’s syntax.)

**Step 1 – Create the repo:**

```
atlas% ocm repo create /project/repo
```

This command creates the repo as a directory under the existing directory /project.

**Step 2 – Add users to the repo:**

```
atlas% ocm user create esj "E. Jensen" -r { local /project/repo }
atlas% ocm user create mdw "M. White" -r { local /project/repo -u esj }
```

(The reason why the last line has ‘-u esj’ in the RepoSpec is that once any user is added to the repo you are subsequently always required to specify your user ID when using that repo.<sup>5</sup>)

To confirm that the users were added correctly, run ‘ocm user show’:

```
atlas% ocm user show -r { local /project/repo -u esj }
```

You may also view and edit the users list by running 'ocm user edit'.

### Step 3 – Create a branch for each user:

```
atlas% ocm branch create /:- -n esj -r { local /project/repo -u esj }
atlas% ocm branch create /:- -n mdw -r { local /project/repo -u esj }
```

Here we have chosen to give each branch a name matching the user ID (the ‘-n’ argument), but any strings would have worked (they could have been, say, the name of the task each user is assigned to). The first argument to ‘branch create’, ‘/:-’, is a VersionSpec that refers to the latest version on the main branch—the version we are branching from.

### Step 4 – Start a Cache Server on atlas

```
atlas% ocm repo cache_server /project/repo
```

### Step 5 – Create the users’ workspaces

```
galaxy% ocm ws create /project/src /esj -r { cache /project/repo_cache -s
atlas -u esj }

galaxy% setenv OCM_WS /project/src

zeus% ocm ws create /project/src /mdw -r { cache /project/repo_cache -s
atlas -u mdw }

zeus% setenv OCM_WS /project/src
```

On each machine, ‘ocm ws create’ creates two directories under /project: the workspace (src) and the workspace configuration database (src.ocm). The final argument (/esj or /mdw) is the BranchSpec specifying what branch the workspace is created from and connected to (the ‘-n’ argument given to the ‘ocm branch create’ command in step 3).

Setting the environment variable OCM\_WS to the workspace location allows OCM to automatically find it. In the shell where the environment variable is set, the user can simply type, say, ‘ocm save’ to save the contents of their workspace. Alternately, the workspace location can be specified with the ‘-W’ option, which will override the OCM\_WS value. *The ‘-r’ option used in the various commands above is not required if the environment variable OCM\_WS or ‘-W’ flag is used to specify a connected workspace.*

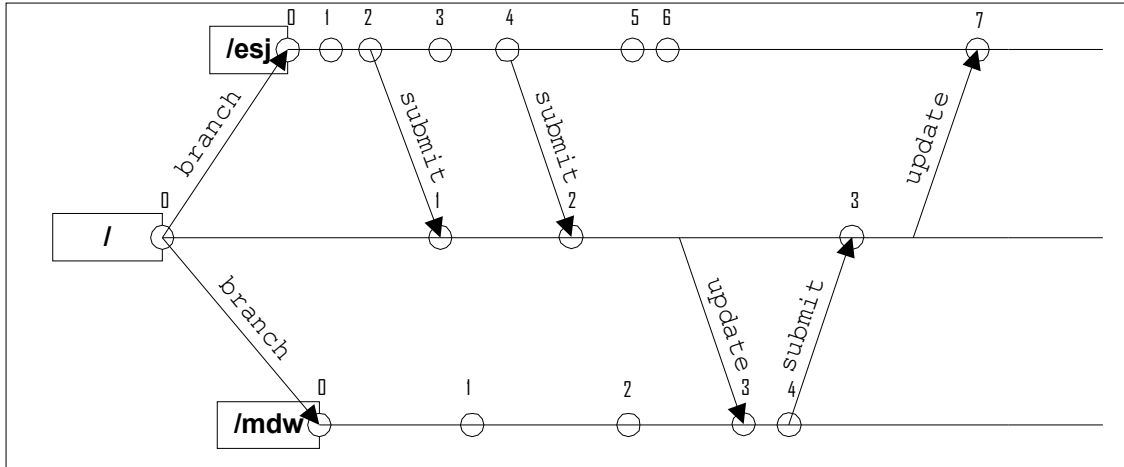
At this point this two-man team is set up and ready to go. Many users of OurayCM could do just fine with this simple configuration (simply add branches and users as necessary), though more complex branching and repository configurations are possible. For information about how to design a branch hierarchy, see [http://www.ouraysoftware.com/articles/branch\\_design.html](http://www.ouraysoftware.com/articles/branch_design.html).

### Using the system

Figure 3 depicts an example of the contents of the repo after a period of use:

---

<sup>5</sup> In case you forget your user ID, you can always use the “root” user to connect to the repo and show the users.



**Figure 3: Two individuals working in parallel.**

This diagram uses the following conventions:

- **A rectangle** with a line to the right denotes a branch. (A branch always comes with version 0).
- **A circle** denotes a version.
- **A circle with no arrow pointing to it** is a save version.
- **A circle with an arrow pointing to it** is an update, pull, submit, or branch version. The version type can be inferred from the arrows: an arrow going to version 0 on a branch indicates a branch version; an arrow going from descendant to ancestor indicates a submit version; and an arrow going from ancestor to descendant indicates an update version. In the above diagram the arrows are labeled for clarity.

Excluding the initial version on each branch (which is created by the branch creation operation) the versions shown here were created when users ran ‘ocm save’, ‘ocm update’, and ‘ocm submit’. It is also possible to create a pull version. For example, mdw can create a version on /mdw that incorporates changes directly from esj’s branch by running ‘ocm pull /esj:-’. (However, in this scenario a pull is not necessary and should not be used.)

In OCM, all versions are numbered starting from zero within each branch. A version address is formed by combining a branch address with a version number. So for example, here is a table of all the versions shown above (every circle in the diagram), in the order they appear on their respective branches. Recall that a ‘-’ in place of the version number means “latest version on that branch”.

VersionSpec	Type
/:0	Initial empty version, from repo creation
/:1	Submit from /esj:2
/:2	Submit from /esj:4
/:3 (/:-)	Submit from /mdw:4
/esj:0	Initial version on /esj, created from /:0

/esj:1, /esj:2	Save; /esj:2 was submitted, creating /:1
/esj:3, /esj:4, /esj:5, /esj:6	Save; /esj:4 was submitted, creating /:2
/esj:7 (/esj:-)	Update on /esj by combining /esj:6 with changes from /:3
/mdw:0	Initial version on /mdw, created from /:0
/mdw:1, /mdw:2	Save
/mdw:3	Update on /mdw by combining /mdw:2 with /:2
/mdw:4 (/mdw:-)	Save; submitted creating /:3

**Table 1: Every version from Figure 3.**

## Adding many files at once

When adding an existing source code set to OCM for the first time, if there are many files involved it is usually best to take your time and do it systematically. Here are some suggestions for helping you do that (see the section on “Saving” for more details):

1. Review your directory hierarchy and file naming. You can easily change it later because OCM has the rename feature, but you may want to do easy changes before entering the data into the tool.
2. Don't add inappropriate things to the repository. All you really should keep in there are things that are needed to build your application, source essentially; keeping huge binary files or compiler artifacts in the repository is usually not what you want.
3. Use the “Auto Add” feature on the Save GUI. This feature automatically detects the file types of new files, according to the following simplistic algorithm: If a file is greater than ¼ MB or has a null character, it is binary; otherwise, it is text. This guess is usually right, but be sure to review the automatic selections before committing them to the database.
4. If you have a huge number of files to deal with, use the “Skip” feature on the “Workspace” entry at the top of the Save GUI to skip all the entries, and then selectively add some files you're sure you want. Save what you've done every few minutes, and then re-run 'ocm save' to add some more items.

## ***Additional important OCM commands***

You can list all available OCM commands by typing ‘ocm all’. Commands that are routinely used but that were not introduced above:

- ocm renewal is a command set that allows you to renew your OCM license, a requirement for extended use of the repository. (See the Administration section for details).
- ocm browse starts a GUI that allows you to browse all the changes that have ever been made to the repository as well as other important information. You can also save various information to disk using the browser.
- ocm ws diff is a command that allows you to efficiently identify recent changes to your workspace.

- ocm copy is a command that lets you copy versions, in whole or in part, to your local file system.
- ocm rm and ocm mv are utilities for removing and moving files within the workspace while preserving file history.
- ocm vref edit starts a GUI that allows you to manage release labels (see “Vrefs and Release Management”)

Refer to 'ocm help' for more details on these and other commands.

## **General GUI Usage**

The OurayCM GUI, which consists of a number of different screens, is designed to work identically across all platforms. Most of its behavior will be familiar, but some is unique to OCM. This section covers its usage characteristics.

### **Mouse**

Mouse interaction involves the left mouse button, the roller, and the mouse movement itself:

- A left click activates various widgets, such as in pressing a button.
- Holding down the left button and moving the mouse (dragging) selects a region.
- There is no “double-click” behavior.
- The mouse roller, if present, activates scrolling.
- When the mouse is over a region that can be clicked, an outline or highlight is shown.

### **Scrollbars**

Various methods can be used to scroll in a region that has a scrollbar attached:

- Use the left mouse button to drag the scrollbar.
- Use the mouse roller.
- Use the keyboard. Up/Down arrows move the window up/down one item; page up/down moves up/down one page.

Using the roller and keyboard require that the scrolled area be selected. This is the only area of difference in behavior between platforms: On UNIX platforms, an area is selected simply by moving the mouse over it. On Windows you are (unfortunately) required to select the area by clicking on it with the left mouse button<sup>6</sup>.

Ouray scrollbars have a unique and useful feature. On other windowing systems, when the scrolled region becomes very large, the scrollbar becomes useless for fine-scrolling as even the smallest motion causes the region to move a large amount. With Ouray scrollbars if

---

<sup>6</sup>This difference will be eliminated in a future version of OurayCM.

the region becomes large, a new scrollbar shows up in green that allows one to continue to scroll finely.

We call the grey bar a “relative motion scrollbar”, since its motion is relative to the overall scrollable distance. The green bar, which shows up only when it can be useful, is called an “absolute motion scrollbar”, and always moves the screen by a certain number of items per number of pixels of mouse motion.

When visible, the absolute motion scrollbar comes with additional markers in the scrollbar trough to indicate its scrolling limits: part of the trough becomes grey instead of blue to indicate scrolling limits.

## **Text editing**

There are two types of editable text areas in the OurayCM GUI: text fields and free-form text. The text field cursor position can be set with the mouse or the arrow keys, and information entered from the keyboard.

For free-form text, such as comments, the OurayCM GUI at present has no built-in editor. Instead, when you want to edit these areas, click on the Edit button near the panel. This brings up a text editor in a new window. See 'ocm env' for details on configuring OCM to use your preferred text editor.

## **Text Select/Copy-paste**

Copy-paste follows the Windows convention:

1. Select a region by dragging with the left mouse button. Many areas can be selected in OurayCM, including labels.
2. Use ctrl-C to copy the region.
3. Use ctrl-V to paste the region.

## **Keyboard Navigation**

*This feature will be defined in a future release.*

## **Where to go from here**

The preceding should be all you need in order to start using OCM productively in simple software development scenarios. The rest of this manual provides information for a few other functions, and expanded detail for those already mentioned.

## 6. Saving

When you run ‘ocm save’, OCM analyzes your workspace to determine what changed since the last time a version was created from your workspace<sup>7</sup>. It then presents you with a GUI showing you the changes and allowing you to provide additional information and resolve any ambiguities. When you confirm the operation, OCM creates the save version.

This section will explain the mechanics of this process, which includes:

- Finding the changes
- Specifying what to do for new entries
- Specifying renames

### ***Finding the changes***

There are various methods by which an SCM tool could, in theory, detect workspace changes:

1. It could look at file system entry time stamps.
2. It could do a thorough file comparison with previous versions.
3. It could depend on the programmer to manually notify the system that a change occurred (e.g., via the editor). Neither this method nor method 1 is perfectly reliable, but both usually work well enough.
4. If the file system supports detecting and reporting changes, it could report them to the SCM system.

OCM implements the first two approaches. By default, the timestamp method is used. To select the thorough method, set the environment variable “OCM\_WS\_MD5CHECK”.

The timestamp method works like this: The timestamp, size, and MD5 sum of each workspace entry is listed in `$OCM_WS.ocm/md5cache`<sup>8</sup>. The timestamp and size of each file and directory is scanned and if it matches the stored values, the entry is assumed to be unchanged so the stored MD5 is used, otherwise the MD5 sum is calculated (and stored for the next operation).

The thorough method works like this: The MD5 sum for each entry in the workspace that is not presently being ignored is computed and compared with the MD5 sum stored in the repository. If that sum is different, then the entry was changed and potentially needs to be stored. If you have a fast computer and a medium-size project, this method works well. But it can obviously be slow when the workspace is very large, since it requires the MD5 computation for every operation.

---

<sup>7</sup> Do not edit your workspace while running the save or update GUIs. Because your file system is not a database, this can lead to partially-edited changes being processed.

<sup>8</sup>To reset the MD5 cache, simply delete this file. You may want to do this if some of the file times were artificially altered.

## ***New entries***

For each new entry that OCM detects, you are given various alternatives to specify what to do with it. The basic alternatives are: Add, Delete, Skip, and Ignore.

### **Add**

There are two basic types of entries that can be added to the database: directories and files. Directories are auto-detected. For files, you must specify whether the content should be treated as binary or text.

A text file is parsed into lines, and OCM applies diff/merge algorithms to it. Changes to text files are stored as differences from the previous version, and full OCM line history features are available. The text type should be used for source code and any other hand-edited text files.

A binary file is stored as an opaque stream of data, and new versions of that file are not stored as differences from previous versions, nor is merging supported.

### **Delete**

If you specify “Delete” for an entry, it is not added to the repository, and *is removed from your workspace*. A backup of this file is available in the `~/.ocm/log` directory—as a rule OCM will never delete or modify a file without putting a copy in the log directory.

### **Skip**

If you specify “Skip” for an entry, it will be skipped during that session, but in the next save session you will be able to re-specify its Save type.

### **Ignore**

If you specify “Ignore” for an entry, its existence is ignored by OCM (see “Workspace Support Files” for details). To unignore an entry, edit the ignore file, `$OCM_WS.ocm/ignore`.

## ***Specifying Renames***

OCM stores the complete history of an entry even if it is moved (i.e. renamed). Unfortunately, standard file systems do not allow 'ocm save' to detect whether an entry has been moved since the last version was stored. If you just look at the two states of the workspace, it appears as if one file was deleted and another was created. Storing the change that way won't cause any errors in workspace recovery, but it will disrupt the continuity of the change history and potentially cause merging problems.

To rectify this "rename problem", the OCM installation includes 'ocm mv', a file/directory move (rename) utility which also records the move in the `$OCM_WS.ocm/renames` file so that OCM automatically records what actually happened.

There is also a problem if a file is deleted and another file is created with the same name. To OCM, it will look like the original file changed. This is solved by 'ocm rm', a remove utility which records a file remove in \$OCM\_WS.ocm/renames.

If you use 'ocm mv' to perform all moves and 'ocm rm' to perform all removes, OCM will record the changes correctly with no further input from you. This is the recommended method. If you do not use these, you can still specify this information to OCM, in one of two ways: 1) Using the "Rename GUI"; 2) Editing the renames file manually.

## The Rename GUI

To specify renames using the GUI, press the "Rename" button on the Save GUI. The rename panel shows the entries in the last saved version, the entries in the current workspace, and the current contents of OCM's rename list.

To add to the rename list, select the former name of the entry in the saved entry list and the new name of the entry in the workspace list and press the "Mark rename" button.

To remove items from the rename list, select the item to remove and press the "Unmark" button.

Note that this does not change anything in the workspace; it only records renames and removes which previously occurred.

When you are finished editing the rename list, you can cancel the changes or apply them. Applying the changes saves the new list to \$OCM\_WS.ocm/renames and uses the list for the current Save. Either action will return you to the main Save GUI panel.

## The renames file

You can edit the text file \$OCM\_WS.ocm/renames with any text editor. OCM expects to find a list of "rename" or "remove" records each on their own line.

A rename line consists of three comma-separated fields:

1. The word "rename"
2. The old entry path (the path relative to the workspace directory)
3. The new entry path (the path relative to the workspace directory)

For example:

```
rename,/dir1/orig_file_name,/dir2/new_file_name
```

A remove line consists of two comma-separated fields:

1. the word "remove"
2. the old entry path (the path relative to the workspace directory)

For example:

```
remove,/directory/junk_file.txt
```

## 7. Update/Pull and Conflict Resolution

An “update” incorporates changes from a shared source branch into your branch and workspace. The default source branch is the parent, but any ancestor branch can be specified. A “pull” incorporates changes into your branch from a non-ancestor source branch. Both operations are equivalent from the point of view of potential conflicts that might happen during the operation, so in this section we will only discuss update, but generally, everything here also applies to pull.

When you run ‘ocm update’, OCM first checks whether there are new changes on the source branch. If there are no changes, it tells the user that there is nothing to do and exits. If there are changes, OCM determines whether or not there have been changes made to your workspace since your last save, and if so, performs a save<sup>9</sup> and then the update.

Inherent in working in parallel is the possibility that the changes that one programmer makes conflict with those of another. The conflicts can happen on various levels: semantic, file, line, and directory (a.k.a., location, or naming). A semantic conflict must be detected by reviewing the changes and/or by regression testing. OCM notifies you when the other types of conflicts occur.

### **File conflicts**

A file conflict happens if a file which was changed in your branch was also changed in the source branch.

If the file is binary, then the only possible resolution is to pick one of the files ("Here" for your version or "There" for the version in the source branch). The version of the file that was not selected is still available in the repo, but its changes are not present in your workspace and will not appear in future versions based on your current version.

If the file is text, you have the same options as for binary files ("Here" and "There") and in addition you can resolve the file conflict by merging the two files ("Merge"). If you do that, changes from both files are combined into a new version of the file which may contain line conflicts.

### **Line conflicts**

A line conflict occurs when you choose to merge two files that are in conflict, and the diff algorithm determines that the same line or section of lines was changed both on your branch and on the source branch. In this case, the files still are automatically combined, but the resulting file contains both sets of changes, as well as the original section (the way it was before either branch changed the file), along with markers that indicate what lines came from where. *If line conflicts happen, you should typically resolve them before saving again, and especially before submitting or updating again, but OCM does not require you to do this since it is conceivable that one might want to save a conflicted file.*

---

<sup>9</sup> The reason for saving is that an update will modify your workspace.

## ***Location (naming) conflicts***

Every entry in a workspace can be said to have a “location”—its file path, i.e., its existence in a given directory with a given name. A location conflict happens if there is an ambiguity when assigning a location to an entry on update.

There are three kinds of location conflicts. These conflicts are resolved using the update GUI:

1. **Name Collision** – you gave an entry the same path that was given to a different entry on the source branch. To resolve this, you must move (rename) or delete one of the entries.
2. **Name Split** – you gave an entry a new location, but the source branch gave it a different new location. To resolve this, pick one of the locations.
3. **Change vs. remove** – The source branch changed an entry that you removed. This is always automatically resolved as a delete—you can recover the source branch's version of the entry from the repo if the changes need to be kept.

## 8. Reverting

Reverting, the operation of restoring and reinstating older versions of entries, is seemingly trivial in principle, but there are nuances and variations that one should understand before attempting it.

The simplest revert scheme is straightforward in OurayCM. Since OCM holds all previous versions of entries, you can always revert to some past version by simply copying it out (using 'ocm copy' or the “Save” button on the File History panel of 'ocm browse'), overwriting the current version of that entry in your workspace, and running 'ocm save'. *However, doing this is potentially dangerous.* When you save the reverted version, it reverts not only your own changes, but any changes you may have received from others since that older version. The next time you submit you might wipe out changes someone else made. Also, any renames that happened since the revert must be manually accounted for. So use this method with care.

Since the copy method is not only potentially dangerous, but also a little inconvenient for most revert scenarios, OurayCM provides a Revert GUI, which you can start by running 'ocm revert'. This GUI presents you with the safe revert options, which is basically all the save versions that exist in your branch since the last time you updated. You can revert to any of these without worrying about losing someone else's information.

To use the GUI, find the version of the entry you want to revert to, and then select its checkbox. After you are done selecting entries, press “OK”.

Other revert scenarios than those presently handled by 'ocm revert' are possible, and will be implemented in future versions of OurayCM. For now, the copy method works fine as long as you take into account the possibility of accidentally throwing out someone else's changes.

## 9. Vrefs and Release Management

Managing releases in OurayCM is a very simple affair, consisting of two basic activities: Release labeling and release maintenance.

Release labeling involves associating a particular name to a particular version—the release labels are called vrefs in OurayCM. Release maintenance involves creating new versions based on the released version and managing these changes relative to development versions.

### ***Release labeling (creating vrefs)***

In OCM, every saved version of a workspace is forever available in the repository (OurayCM versions are “Durable”). So, any previous release is available as long as it was recorded in OurayCM (by running one of save, update, submit, or pull). Further, since each version can be uniquely referred to using its VersionSpec, there is no need for any other mechanism for recovering old releases, at least technically.

However, the VersionSpec is an OCM-specific notation, and is somewhat meaningless as far as release naming goes. For example, you may want to call your release “1.0”, but the VersionSpec for it might be “/2/1:321”. This is where the “version reference”, or “vref” feature of OCM comes in.

A “vref” allows you to associate an informative name of your own choosing with a particular version<sup>10</sup>. Vrefs are only side-information—they do not modify your versions in any way. You can organize these in a hierarchy. The easiest way to create and manage vrefs is with the vref editor:

```
% ocm vref edit
```

Since a vref stands for a particular version, it is an acceptable form of a VersionSpec, and so can be used in all commands that require a VersionSpec. See 'ocm syntax' for further details.

### ***Release maintenance tips***

OurayCM's branching features provide all the necessary tools for managing releases. You can create a new branch from any existing version, you can specify its parent to be any other branch in the repo, and you can transfer changes between any two branches. How you use these features is up to you; we can't really define that a particular organization is best for everyone. However, we can make some suggestions.

With OCM, you don't need to plan your release branching ahead of time, because new branches can be made based on any past versions. A typical pattern is to create a release from whatever branch is convenient, add a vref for it, and then later, if that release requires changes, to use branching for maintenance.

---

<sup>10</sup> It might be thought that an alternative to release labeling would be to create a branch for each release, and name the branch appropriately. Since branches are lightweight, this would not cause an undue overhead in the database, but it is not a recommended method. The purpose of a branch is to isolate changes and hold a set of versions, not to tag a particular version.

A suggested approach for organizing release maintenance branches is to create a top-level branch named “/releases” and put all of your maintenance branches there, with an appropriate name. With the branch command, you can independently specify the version to branch from and the parent of the new branch. I.e. the new branch can have a parent different than the original version's branch.

An important question arises once you start dealing with maintenance branches: how to get the changes made in a maintenance branch back into the other branches, or vice versa. There is no easy answer to this, because the respective branches might have evolved to the point that they are not very comparable from a physical source point of view, so automatic merging may be impractical. OCM provides two features that enable you to deal with this one way or the other: ‘ocm pull’ and the Version History tab on ‘ocm browse’.

If you think that the respective versions may be comparable, try using ‘ocm pull’ to get changes into or out of a maintenance branch. Inspect the changes on the pull GUI, and if they are appropriate, proceed. If the versions are not comparable enough, use the Version History tab on ‘ocm browse’ to identify the relevant changes and then manually apply them, or use the 'ocm copy' command to copy out particular versions and compare and merge using different methods.

## 10. Remote Repo

The remote repo feature of OurayCM allows you to perform SCM operations without a full-time network connection. A remote repo is a copy of the repo with some restrictions on what changes are allowed. A user with a remote repo retains almost full SCM capabilities, even though there is no direct connection to the main repo. Changes are transferred between the main repo and the remote repo using either the remote server (if a part-time network is available) or by using any file transfer media (if a network is not available).

When using a remote repo, branch ownership is important. All branches are marked as either “owned” (locally modifiable) or “unowned” (not locally modifiable). The branch data is always present whether a branch is owned or not, but a user is only allowed to modify branches that are owned in his repo.

The remote repo feature comes with limitations and is administratively a bit more complex than the Cache Server method, so it is recommended that you only use it if necessary. The remote user may not<sup>11</sup>:

- Edit the users list.
- Perform a renewal operation. The renewal operation must be done via a sync with the main server.
- Connect to remote branches, create branches or make any other changes outside of owned branch(es), except that a remote submit may be performed (see below).
- Edit vrefs.
- Submit to a remote branch, except through the remote server.

Before proceeding, please be sure to read and understand the following section:

### ***IMPORTANT, PLEASE READ:***

If you wish to use the remote repo feature, it is critical that you understand the concept of branch ownership, because you are the one who assigns it.

Branch ownership must be mutually exclusive: a given branch should only be owned, i.e., made modifiable, in one repo: if a given remote repo owns a branch, then neither the main repo nor any other remote repo should own that branch.

In the remote repo scenario there is generally no network available to automatically verify this condition, *so it is your responsibility to insure that only one repo has ownership of any given branch*. If you make inconsistent branch ownership settings, you can potentially make your repos unsyncable—i.e., if a branch is marked “owned” by more than one repo, you may need to manually incorporate remote changes back into the main repo and recreate your remote repo.

---

<sup>11</sup> Some of these restrictions may be relaxed in future versions of OurayCM.

## Setting up a remote repo

A remote repo is itself a repo, but without ownership of the topmost branch. To create a remote repo:

1. Create the branch(es) that will be used by the remote repo, if they don't already exist.
2. Create the remote repo from the main repo.
3. Configure ownership properties in each repo.
4. Deploy the remote repo, setting up servers and clients as appropriate; set up remote sync server if desired.

### 1. Create remote branches

This step is optional. You can create an ordinary branch in the main repo and change its ownership later. However, at this time you may want to consider your branch organization, taking into account the following ownership properties of branches:

- The main repo must own the topmost branch, `/'`—ownership of `/'` is what defines a repo as being “main” vs. “remote”. A remote repo may own any other branch.
- A branch may only be owned in one repo (see “IMPORTANT” above).
- A child branch may only be created if the repo owns the parent branch. Once created, the ownership of a child branch can be changed to remote.
- You can create as many branches as you want in a remote repo as long as their parent branches are owned by that repo.

### 2. Create remote repo

To create a remote repo from a main repo you copy the main repo<sup>12</sup> and then recursively give up ownership for all branches. On UNIX, you might copy a repo this way:

```
% cp -r repo remote_repo
```

To recursively take ownership from all branches on `remote_repo`, run this:

```
% ocm remote set -r { local remote_repo -u root } -R /
```

In this usage, the `-R` argument specifies to apply the changes recursively to all child branches; the `/'` specifies to apply them to the topmost branch.

### 3. Configure Ownership

In this step, we must take the ownership of specific branches from the main repo and transfer them to the remote repo. Use the `-R` flag as appropriate to apply the settings to all branch children. The `-t` flag specifies that the given repo should take ownership of the specified branches. Here are sample commands we would use if we wanted to remote the `esj` branch:

---

<sup>12</sup> Use any method that recursively copies the repo directory. You should shut down any servers before doing this, or otherwise be sure that it is not being modified during the copy operation.

```
% ocm remote set -r { local_repo -u root } /esj
% ocm remote set -r { local_remote_repo -u root } -t /esj
```

The actual command line you use will vary depending on your needs; see ‘ocm help remote set’ help for more details.

#### 4. Deploy remote repo

To deploy, copy the remote repo to the remote machine, and choose whether to use the synchronization server or the manual method. Both methods use the same underlying mechanism:

1. On the remote side, an update “request file” is formed, using information about what was last known to be in the main repo. The update request contains all the changes that have happened to the remote repo since it was created or since the last synchronization.
2. The request file is sent to the main repo and applied, forming a “reply file”, containing all the changes made to the main repo since the last synchronization.
3. The reply file is sent back to the remote repo and applied, bringing it up to date with the main repo.

The synchronization server is more convenient, but requires an available TCP/IP connection for certain operations (it does not need to be up all of the time—that would defeat the purpose of the remote repo). It also lets you perform a remote submit; if you use the manual method, you must arrange for the submit to be done at the main repo site. If you can use this method, it is recommended.

##### Synchronization server method

Setting up the sync server is very similar to setting up the cache server. For the example above, at the main repo site, you would start the server this way:

```
zeus% ocm repo remote_server repo -p 2181
```

Then, at the remote repo site, you connect your workspace this way:

```
% ocm ws connect /project/src /mybranch -r { remote /
project/remote_repo -u root -s zeus:2181 }
```

##### Manual method

When you must use a non-TCP/IP method to get changes back and forth, use the commands ‘ocm remote update\_request’, ‘ocm remote apply\_request’, and ‘ocm remote apply\_reply’ to accomplish the above 3 steps manually. To submit, use the ‘ocm version submit’ command, specifying what version to submit from where. This command must be run on the main repo (the repo that owns the branch being modified by the submit operation).

See the online help for these commands for further details.

## 11. Workspace Support Files

One of the design principles of OurayCM is: Don't litter the user's workspace with CM utility files. Instead of doing that, we have a few different areas that store utility information for OurayCM:

- The Workspace Config Database
- The .ocm directory
- The Project Ignore File

### ***Workspace Config Database***

The Workspace Config Database represents a “connection” between your branch and your workspace. It stores information about the connection, such as the server host and connection password, and contains utility information. This database is stored at the same level as your workspace directory, with the same name but “.ocm” appended (e.g., src.ocm). You usually will not need to do anything with this, but occasionally you may want to deal with the following files:

- ignore – contains your GUI-selected ignore choices. You may want to add regular expressions to this or delete previously-ignored items (see “The Project Ignore File” below for details on file format).
- md5cache – contains the cache of the MD5 sum for each entry in your workspace for the time-stamp-based change detection algorithm. You may need to delete this file, e.g., if the time on a given file or directory was artificially altered causing OCM to ignore your changes.
- renames – this stores the rename information created by the Rename GUI, 'ocm mv' and 'ocm rm'. You may need to edit this file directly if you had renamed files without using these utilities (See the above section on Save for more information about this file).

### ***The .ocm Directory***

The .ocm directory is stored in your home directory. It contains error log information and organized backups of all the files that OCM modifies. You may delete these logs whenever you wish; OCM will recreate them as necessary.

The “error.log” file is appended to whenever OCM exits abnormally, whether because of user interrupt or program fault. Debugging information related to the fault is stored there, and bug reports should include a copy.

The “log” directory contains directories with backups of the original files that were modified from certain OCM operations, such as 'ocm rm' and 'ocm revert'. This directory should be deleted from time to time.

## Ignore Files

When you perform a save, OCM analyzes your workspace to determine what is different from what is already in the repository. The “ignore file” feature of OCM allows specific files and file patterns to be ignored during this process, effectively making OCM blind to the existence of these files. The reason you would use such a feature is to exclude various irrelevant development artifacts from being stored in the database. Such files might include editor backup files, compiler artifacts, and other temporary files and directories.

There are two possible ignore files associated with a given workspace: an individual ignore file stored in the Workspace Config Database (see above), and the project ignore file.

The workspace ignore file is only associated with a particular workspace and is typically defined through use of the ignore feature in the Save GUI but you can edit it yourself.

The project ignore file is shared by all users, and defines a common set of ignore patterns and files. By default, this file is stored at the top level of your workspace, in a file named “ocm.ign”. If present, it is by default used. You can change this default name using the environment variable OCM\_IGNORE\_FILE (see 'ocm help env' for more details).

The combined list from the project ignore file and workspace ignore file is used by OurayCM to filter out files from the workspace.

## Ignore File Format

OCM uses a list of regular expressions that define the workspace path of files and directories that should be ignored. The “entry path” is the path name relative to the workspace directory. The syntax for an entry path is the same as that of UNIX paths. For example, if you have the file `/project/src/dir1/myfile.txt` in workspace `/project/src`, the workspace path is `/dir1/myfile.txt`.

The save algorithm iterates over all entries in the workspace and checks the workspace path of each against the newline-separated regular expressions in the ignore list. If there are any expressions that match, the entry is ignored by OCM (if a directory is ignored the contents are also ignored). For example, if you wanted to ignore the workspace file `/dir/myfile.txt`, this line of text would go into the ignore file:

```
^/dir/myfile.txt$
```

The ignore feature uses the Perl Compatible Regular Expression (PCRE) library written by Philip Hazel, which provides a powerful way to specify which files to ignore. Here are some example ignore file patterns:

Pattern	Meaning
<code>~\$</code>	Ignore any path with a ‘~’ at the end (editor backup)
<code>\.~cpp\$</code>	Ignore any path ending in “~.cpp” (editor backup)
<code>\.obj\$</code>	Ignore any path ending in “.obj” (compiler artifact)
<code>/CVS\$</code>	Ignore any path ending in ‘/CVS’
<code>\.tmp\$</code>	Ignore any path ending in “.tmp” (programmer temporary files, conventionally ending in “.tmp”).

More sophisticated patterns are possible. Please see the PCRE web page (<http://www.pcre.org>) for more information on syntax and behavior.

## 12. Administration

Administration in OurayCM is very simple. It is likely that the only things you will have to do administratively are:

- Create repos (covered above in “Getting Started”)
- Add users (covered in “Getting Started”)
- Remote Repo (covered in “Remote Repo”)
- Renewal
- Backup
- Restore

This section covers Renewal, Backup, Restore, and a few other technical points. The technical points are covered for the sake of completeness; we recommend skimming them but it is unlikely that you will ever have to worry about them. The Remote Repo is covered in its own section.

### ***Renewal***

In order to offer our customers the highest possible level of fairness and flexibility, OurayCM is priced using a usage-based model (see <http://www.ouraysoftware.com/pricing.html>). This model requires each repos usage to be periodically reported to Ouray Software. It is important to remember that unless it goes through this renewal process, a repo will revert into a read-only state: you will always be able to extract your old data and use any OurayCM features that don't modify the database (we give these features away for free under conditions of the EULA), but you will no longer be able to add new data.

To renew, run:

```
% ocm renewal request
```

This brings up a GUI with three tabs. In the Payment tab you will need to enter the information for the party responsible for payment—this is who we will send the usage bill to. You must also specify where to send the renewal, which may be different. In the Data tab you will see the information that will be sent to Ouray Software. The first two tabs contain the entirety of the message that will be sent. The Renewal Status tab shows you the renewal status of your repo.

You can choose to send us this information in two ways. The most convenient method is to click “Renew via WWW”. This automatically uploads the renewal information to us. If you do not have an Internet connection or want to review the file before sending it to us, click OK and a text file will be stored in the current working directory. Send this text file to [support@ouraysoftware.com](mailto:support@ouraysoftware.com).

After we receive your renewal request, we will process it into a renewal reply, which is another text file we send back to you<sup>13</sup>. When you get this file, apply it to your repo like this:

```
% ocm renewal apply reply.txt
```

Now your repo will be renewed for a month or so<sup>14</sup>. Note: When your project ends, then according to the EULA you are required to perform a renewal request in order to report your final usage information.

## **Backup**

Depending on your topology, you may not need to explicitly back up your main repo. Since the cache server creates exact replicas of the repo on individual machines, that may be all the redundancy you need.

Achieving full rigor in the backup process is easy. Just pick a machine that will have its hard drive backed up, and run 'ocm repo sync' on that machine right before backup. For example:

```
% ocm repo sync -r { cache repo_backup -s atlas -u root }
% ocm repo check repo_backup
```

The first time you run this the entire repo will be copied from the server; subsequent times only the differences will be applied to repo\_backup.

It is wise to run the 'repo check' command before every backup, as well as nightly on the server itself. If an error is detected (a non-zero exit status indicates an error), it indicates a faulty hard drive<sup>15</sup>. You should replace the hard drive and restore the repo from an uncorrupted version.

## **Restore**

If your server hardware fails, the best restore procedure is typically to use one of the caches from some developer's machine. Simply copy the repo (the repo is a directory containing one file named "db" and one named "idx") to the new server and restart the Cache Server. However, it must be the newest cache or the process won't work for everyone (anyone who has a newer cache will not be able to sync). To find the newest cache, check the size of the "idx" file for each cache; the largest is the newest.

## **Undoing repo operations**

Normally, there is no need to undo any repo operation—indeed, doing this would remove history, and a major purpose of an SCM tool is to keep track of history. But extreme scenarios exist. For example, you might accidentally add a 2GB file to the repo that you just don't want taking up room in the repo.

<sup>13</sup>A forthcoming feature will allow established customers to get their renewal reply automatically when they renew via WWW making this step unnecessary.

<sup>14</sup>The duration of the renewal period is negotiable. We like this to be shorter for new customers and longer for customers that have already been through a few cycles.

<sup>15</sup>Either it's a faulty hard drive or a stray program or person poking in random values into the database files.

There are three basic methods to undoing a repo operation:

1. Recovering a backed up version of the repo.
2. Removing repo commits
3. Rebuilding the repo using a filter to exclude certain entries (this method will be supported in a future version of OCM).

**It is critical to understand that any of these methods can invalidate a repo cache or remote repo.** This can imply a series of administrative steps to get your full system back online (you may need to undo each repo cache and remote repo, or get new copies). This is not necessarily a big ordeal, but it will cause an interruption in service. Also, before undertaking any of these steps, be sure to create a backup copy of the repo you are about to undo.

For each of these methods, be aware that they only apply to the repo itself—you'll still have to take care of your individual workspaces (e.g., don't forget to remove or ignore that 2GB file).

### **Recovery of a backup version**

This is the most basic method, and just requires going to a previously backed up version of the repo. This can invalidate a repo cache or remote repo—if the backup is older than either of these then you must re-create them (for the cache this simply means copying the backed up repo into the cache location).

### **Removing repo commits**

With this method, you can remove transactions one operation at a time. Again, this method can invalidate a repo cache or remote repo. To remove a transaction:

1. Make sure you have a backup of your original repo. Removing transactions is a destructive operation!
2. Use UNIX 'head -c' or an equivalent on the "idx" file in the repo directory to remove the last byte from the file. For example, if the idx file is 3620 bytes long, do this: 'head -c 3619 < idx > idx2; mv idx2 idx'.
3. Run any operation on the repo *locally* that can modify it (ocm browse is read-only and will not work for this purpose), e.g.: 'ocm user edit -r { local repo -u root }'. As 'ocm user edit' starts up, it will detect a corruption in the last transaction, assume that it failed, and delete it. *Don't add any users at this time; just cancel after it starts up.*
4. Repeat as many times as necessary, removing one transaction at a time. You can use 'ocm browse -r { local repo -u root }' to observe the results.

Note that you can remove transactions from an individual repo cache using the same method if network resources are too constrained to get a new copy of the cache to all the individual locations. Simply apply the above procedure to the repo cache until it is the same size as the repo, except that you may truncate the idx file down to the server idx file size in one operation.

It is recommended that you do not attempt any of this with a remote repo, and if one is involved, that you recreate it.

## **Rebuilding the repo**

When implemented, this method will allow you to specifically remove particular files while reconstructing the repo. No repo transaction will be undone; each transaction is redone, but leaving out entries you request.

## ***Workspace non-atomicity***

All operations on the repo are atomic transactions: if you kill OCM in the middle of an operation, it leaves the database in a coherent state. However, your local file system is not a database, and if you kill OCM while it is modifying your workspace, it could be left in an incoherent state. We minimize the time during which workspace operations occur so errors here are rare, but since it is possible for an error to occur, you should be aware of how to detect and recover from this case.

You can be suspicious of an incoherent workspace if OCM is killed during any operation that is modifying the workspace: 'ocm update', 'ocm pull', 'ocm branch create' (if using the `-w` option), 'ocm ws create', and 'ocm ws connect'.

Detection of an incomplete operation is simple. If you run 'ocm save' directly after one of these operations, it should not do anything, since you have not edited your workspace (if it brings up the GUI, observe the error and press cancel). If there are changes, then they are likely truncated or missing files.

Recovery is simple. In the case of an incomplete operation that creates the workspace ('ocm branch create', 'ocm ws create', or 'ocm ws connect'), remove the partially created workspace, and repeat the operation. In the case of an incomplete update or pull, use 'ocm revert' to get the latest version of your source and repeat the update or pull.

## 13. Acknowledgments

OurayCM is mostly implemented from scratch. It relies primarily on the native operating system (including the C library), but incorporates a few freely available open-source libraries, which Ouray Software gratefully acknowledges:

- Julian Seward's [bzip2](#), used for data compression
- Philip Hazel's [PCRE Library](#), used for regular expression processing
- L. Peter Deutsch's MD5 algorithm
- On Windows platforms, PowerDog's strptime
- [OpenSSL](#), exclusively used for the protection of our IP<sup>16</sup>: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (<http://www.openssl.org/>)"; "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)".

For specific copyright notices on some of the above, see <http://www.ouraysoftware.com/credits.html>.

---

<sup>16</sup>In order to more easily comply with US export regulations, we do not support user encryption at this time. However, if you have a special need for user-level encryption and are a U.S. customer, we can build you a special version of the tool that supports it, for a fee. Contact us at [support@ouraysoftware.com](mailto:support@ouraysoftware.com) for more details.