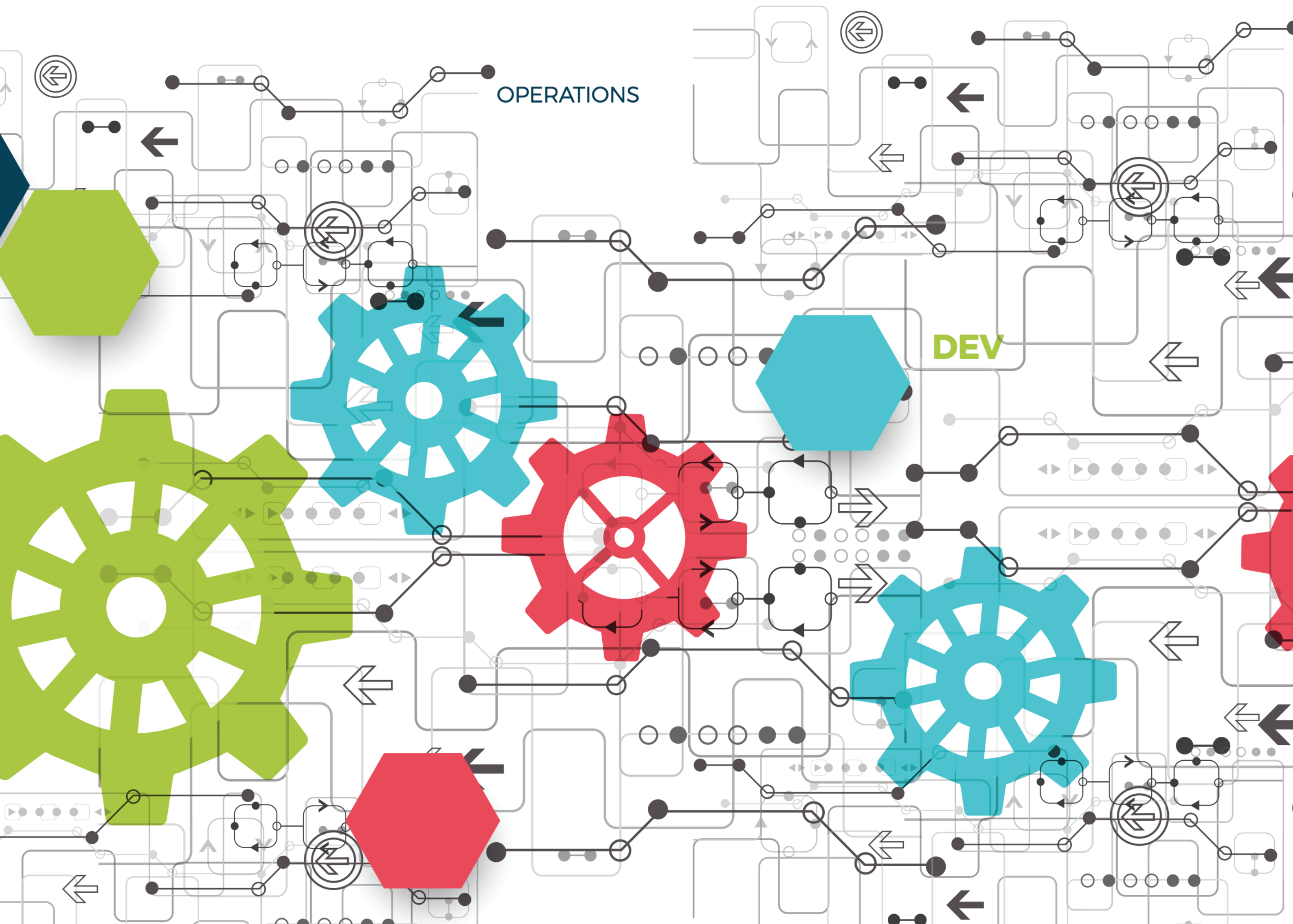
02 YEAR **B.Tech** Computer Science and Engineering in DevOps

Source Code Management

Additional Appendix

Mercurial



Version Control Systems | Mercurial

Introduction

Mercurial is a popular distributed version control system, that offers way to archive as well as to save older versions of source code. Mercurial came into existence in 2005 as an open-source version control system, as an alternative to the closed-source BitKeeper and was developed by Matt Mackall. Unlike SVN, which is a centralized version control system, Mercurial is a distributed version control system. That is, when you push changes to the repository, it will go to the local machine. Because of this, the process becomes very faster, since you're not constantly pushing to a remote server (although it can be set up that way).

Mercurial is built primarily in Python, which makes it cross-platform compatible. This is also one of the reasons that Mercurial is mostly used as a command-line tool, though there are GUI tools available. Mercurial has been the version control system used by big brands like Adium, Mozilla, Netbeans, Vim, Growl and so forth. Apart from these, a lot of individual developers use Mercurial to manage their code.

Features of Mercurial

Some of the important features of Mercurial, which make it the popular version control system, are as follows:

Distributed architecture

Most of the traditional version control systems like SVN are based on client-server architecture, where a central server holds the updates done to a project. Mercurial is completely distributed, where each developer has a local copy of the complete project. This way, the developer is not dependent on network or server access. Committing the code, branching and merging can be done faster.

Fast

The implementation and data structure of Mercurial are designed in a way that the tool is fast enough to handle multiple commits. Diffs can be generated between revisions or rolled back in much lesser time, usually in seconds. That's why it has been used in larger and complex projects like OpenJDK or NetBeans.

Platform independent

Mercurial is intended to be platform independent. Hence, a major portion of it is written in Python, and a small chunk in portable C for performance related reasons. Because of this reason, binary releases for Mercurial are available on all major platforms.

Extensible

Mercurial can be functionally extended using the official plugins that are shipped along with Mercurial or downloading from external sources or writing our own. Extensions are written in Python and can change the workings of the basic commands, add new commands and access all the core functions of Mercurial.

Easy to use

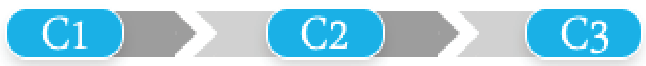
The command set of Mercurial is so simple that most of the SVN users will find it very easy. The basic interface of Mercurial is easy to learn and use.

Open Source

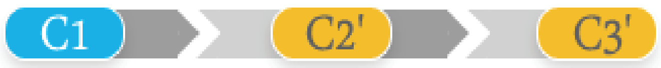
Mercurial is available free and is licensed under the terms of the GNU General Public License Version 2 or any later version.

Basic Concepts and Working Mechanism of Mercurial

1. Similar to Subversion, the revision history in Mercurial consists of a number of commits. They're termed as changesets in Mercurial.
2. In Subversion, there is a strict linear ordering of the commits and linear revision numbers are given to them. Because of this reason, revision N will have one child revision, N+1. This may be simple, but a central server is required to make sure that everybody agrees on the revision numbers.
3. The abovesaid process is generalized in Mercurial. Here, each changeset can have multiple children. For example, if developer A makes three commits.

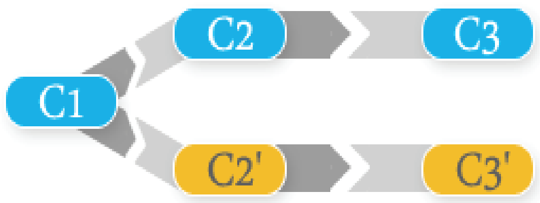
   

   The commit C3 with no children is a "head". It is also the newest changeset in the repository – and is called "tip". If developer A shares C1 with developer B and B starts their work from that, commits mad by B will build a repository like this:

   Here C3' is a head in B's repository and A will not have any clue about C2' and C3' yet.
4. If A pulls from B, or B pushes to A, the two repositories are compared. By default, all missing changesets are transferred. This is all there is to push/pull: compare two graphs of changesets and transfer the missing ones.

   After a pull from B, repository of A will look like this:

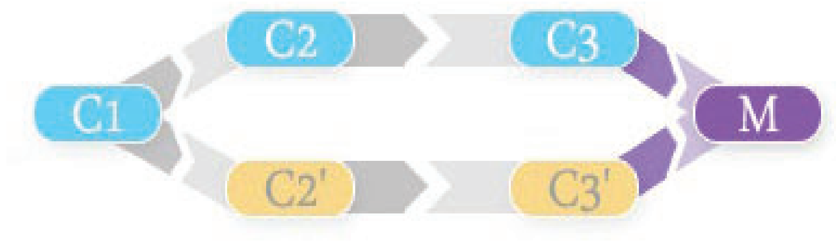
   

Therefore, C1 has two child changesets, and the repository now has two heads since the development has diverged.

The changeset C3' will be the new tip since it is the newest changeset in the repository.

5. Since the repository has two heads, someone should merge them -- otherwise the changes from two developers will not be combined.

   When merging with 'hg merge', the ideal way of combining the changesets need to be figured out. If the changes do not overlap the task becomes easy, otherwise we need to perform a three-way merge. The merge must be committed and this creates a changeset that looks like this:



   Note that the merge changeset M has two parents. If we fail to merge C3 and C3' and try to push, you get the 'new remote head' message and push will not get complete. The developer who created the two heads by pulling in some code should also normally do the merging.

6. The important points to note are:

   - "hg commit" adds a new node. The parent changesets of the new node is given by "hg parents"
   - "hg push" and "hg pull" transfer nodes in the graph between two repositories.
   - "hg update" updates the working copy to reflect a given node in the history graph. This also changes the parent changeset of the next commit, see "hg parents".

Installing Mercurial

Mercurial can be downloaded from the source. Alternatively, prebuilt binary packages of Mercurial are available for most of the popular operating systems. These packages can be found in the website - https://www.mercurial-scm.org/wiki/Download. These make it easy to start using Mercurial on your computer immediately. Package managers in Linux also are helpful in installing Mercurial in the system. Windows users will find Tortoisehg GUI easy to use.

Packages for common Linux, BSD and Solaris distributions can be installed from the system specific repositories as follows:

1. Debian/Ubuntu - $ apt-get install mercurial
2. Fedora - $ dnf install mercurial
3. Gentoo - $ emerge mercurial
4. Mac OS (homebrew) - $ brew install mercurial
5. FreeBSD

```
$ cd /usr/ports/devel/mercurial
$ make install
```

6. Solaris 11 Express - $ pkg install SUNWmercurial

Basic commands in Mercurial

Some of the basic commands in Mercurial are as follows:

Command	**Function**
hg init	To create a new repository
hg commit	To save your changes in the current repository
hg log	To see all changes in your repository
hg pull	To get all changes from another repository into the current one
hg push	To get all changes from your repository into another one
hg serve	To create an instant-webserver. People can see the history there and pull from it
hg merge	To join different lines of history

Getting started with Mercurial

Mercurial can be called by using the command hg. The command hg version is used to check if Mercurial is installed properly. This command also helps to find out the version of Mercurial that is being used.

```
$ hg version

Mercurial Distributed SCM (version 4.2)

(see https://mercurial-scm.org for more information)



Copyright (C) 2005-2017 Matt Mackall and others

This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Mercurial help

Mercurial offers a built-in help system. Help can be used to find our way out if we're stuck while working with the tool.

```
$ hg help init

hg init [-e CMD] [--remotecmd CMD] [DEST]


create a new repository in the given directory

Initialize a new repository in the given directory. If the given directory
does not exist, it will be created.

If no directory is given, the current directory is used.

It is possible to specify an "ssh://" URL as the destination. See 'hg help
urls' for more information.


Returns 0 on success.


options:


 -e --ssh CMD       specify ssh command to use

    --remotecmd CMD specify hg command to run on the remote side

    --insecure      do not verify server certificate (ignoring web.cacerts

                    config)


(some details hidden, use --verbose to show complete help)
```

The -v option is short for --verbose, and instructs Mercurial to print more information than it usually would. The -k or –keyword option will search through the help system for a specific keyword and fetch the details.

```
$ hg help -k init
Topics:
 config      Configuration Files
 glossary    Glossary
 merge-tools Merge Tools
 revisions   Specifying Revisions
 templating Template Usage


Commands:
init  create a new repository in the given directory
paths show aliases for remote repositories
```

Working with a Repository

A Mercurial repository contains in it all the files of the project, along with their revision histories. A repository is a directory tree in the filesystem. A repository can be deleted, modified or renamed at any time, using specific commands.

There are two major types of Mercurial commands in the context of repositories:

- The first type is for working on a local repository and helps us do any action on the repository. These are completely local commands.
- The second type of commands are for network operations. Using these commands, we can send changes to our code to another repository or retrieve changes from someone else.

Creating a new repository

New repository can be created simply by using the hg init command, which will be the working repository. For example:

$ hg init myproject

The above command is used to create a new repository by name ‘myproject’ in the current directory.

Adding Content to the Repository

The newly created repository doesn’t have any files in it currently. We’ll now add a file called ‘hello.c’ to the repository with some piece of code.

```
$ cat hello.c

int main()

{

    printf("hello world!\n");

}
```

Having done this, we can use the hg status command, which will give an overview of changed and unknown files in the repository.

```
$ hg status

? hello.c
```

In the above example, we just created a new file, but Mercurial is unaware of this, hence it displayed a question mark beside the file name. This is an untracked file. The command hg add is used to instruct Mercurial to start tracking the new file.

```
$ hg add hello.c
```

As we’ve instructed Mercurial how to handle the file, running the hg status will show something different now:

```
$ hg status
A hello.c
```

To know the changes made to the file

Now that we’ve modified the file, we might want to know the changes that we’ve made to the file. The command hg diff is used for this purpose.

```
$ hg diff
```

```
diff -r 000000000000 hello.c
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/hello.c Mon Dec 17 04:01:38 2018 +0000
@@ -0,0 +1,4 @@
+int main()
+{
+    printf("hello world!\n");
+}
```

Now that we’ve a file that’s being tracked by Mercurial, we can add it to history. The snapshot of the history is termed as a *changeset* in Mercurial, as it keeps the record of the changes to several files. This changeset can be created by adding the new file permanently to the history of the repository and we use the hg commit command for this purpose.

We used the hg status command to know what will happen when we make a commit. The 'A' beside the new file specifies that the new file will be added to the history.

```
$ hg commit -m 'Initial commit'
```

Changes are now committed, and so it will not be seen in the output of the hg status command.

Changing the file content

For example, if we change the contents of the hello.c file created earlier.

```
$ cat hello.c

int main()

{

    printf("goodbye world!\n");

}
```

The status and diff commands can be used again to know what has been changed in the repository.

```
$ hg status
```

```
M hello.c

$ hg diff

diff -r 9d16f03a559f hello.c

--- a/hello.c Thu Jan 01 00:00:00 1970 +0000

+++ b/hello.c Wed May 17 04:01:38 2017 +0000

@@ -1,4 +1,4 @@

 int main()

 {

-    printf("hello world!\n");

+    printf("goodbye world!\n");

 }
```

The ‘M’ beside the file name indicates that Mercurial has found out that we’ve modified the file hello.c. Before or after modifying we need not instruct Mercurial that we’re going to or we’ve done so. The command hg commit will take care of handling the change and will save the modifications done to the changeset.

Apart from the status and diff commands, the summary command is also used to know the changes happening in the working directory.

```
$ hg summary

parent: 0:9d16f03a559f tip

 Initial commit

branch: default

commit: 1 modified

update: (current)

phases: 1 draft
```

Creating a Local Copy of the Repository

The command for creating a local copy of the repository is hg clone. This command creates an identical copy of the working repository. For example:

```
$ hg clone https://bitbucket.org/bos/hg-tutorial-hello hello

requesting all changes

adding changesets

adding manifests

adding file changes

added 5 changesets with 5 changes to 2 files

updating to branch default

2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

The major advantage of using the clone command is that it allows us to create copies of repositories over a network. The location from where the repository is cloned is also recorded. If the clone operation succeeds, we'll have a local directory in the name specified in the command. A repository in Mercurial is self-contained and independent. It will have the local copy of the project's repositories and history. The cloned repository will remember the location of the repository it was cloned from, but will not have any communication with that, unless instructed. This ensures that the local repository will not affect any other developer's work.

It is important to keep a clean copy of the remote repository, from which we can create temporary clones for creating the sandboxes for each task that we do without affecting other's work. This will also ensure that the repository under modification is kept completely local until it's complete and is ready to integrate back. Local clones will not cost you anything even if modified or deleted. While making changes, it will not create any impact on the repository cloned.

```
$ hg diff

diff -r 2278160e78d4 hello.c

--- a/hello.c Thu Aug 16 22:16:53 2018 +0200
```

```
+++ b/hello.c Mon Dec 17 04:01:29 2018 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
      printf("hello, world!\");
+     printf("hello again!\n");
      return 0;
 }
```

History of Repositories

The hg log command is used to view the history of changes in the repository.

```
$ hg log
changeset:   4:2278160e78d4
tag:         tip
user:        john@smith.com
date:        Sat Sep 15 22:16:53 2018 +0200
summary:     Trim comments.

changeset:   3:0272e0d5a517
user:        john@smith.com
date:        Sat Sep 15 22:08:02 2018 +0200
summary:     Get make to generate the final binary from a .o file.

changeset:   2:fef857204a0c
user:        john@smith.com
date:        Sat Sep 15 22:05:04 2018 +0200
summary:     Introduce a typo into hello.c.

changeset:   1:82e55d328c8c
user:        abc@gmail.com
date:        Sun Aug 26 01:21:28 2018 -0700
summary:     Create a makefile
```

The output of hg log has four fields as follows:

```
changeset:   0:0a04b987be5a
user:        abc@gmail.com
date:        Sun Aug 26 01:20:50 2018 -0700
summary:     Create a standard "hello, world" program
```

- changeset

This field is indicated by a number, followed by a colon, followed by a hexadecimal (or hex) string, which identify the changeset uniquely. The hex string is a unique identifier: the same hex string will always refer to the same changeset in every copy of this repository. The number may be shorter and easier to type than the hex string, but it isn't unique: the same number in two different clones of a repository may identify different changesets.

- user

Refers to the person who created the changeset. This is a free-form field, but it most often contains a person's name and email address.

- date

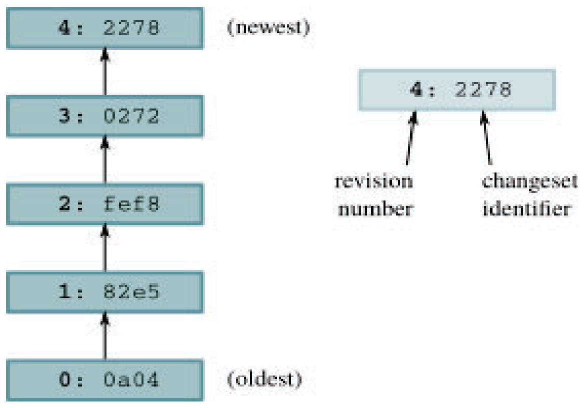
The timestamp of the changeset creation, and the time zone in which it was created. (The date and time are local to that time zone)

- summary

The first line of the text message that the creator of the changeset entered for describing the changeset.

- Some changesets, for example, the first in the list above, have a tag field. A tag is also a way to identify a changeset, by giving it an easy-to-remember name. (The tag named tip always refers to the newest change in a repository)

The following image illustrates the history of a repository.



As we saw above, the changeset field has a number and a hexadecimal string.

- The number, which is the revision number is a notation that is only valid in that repository.
- The hexadecimal string is the permanent, unchanging identifier that will always identify that exact changeset in every copy of the repository.

Propagating Changes to Other Repositories

Pulling Changes from another Repository

For this, we'll clone a temporary repository from the original repository that we created (without the changes).

```
$ cd ..

$ hg clone hello hello-pull

updating to branch default

2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

The command hg pull is used to bring in changes from one repository to another. But this command will blindly pull unknown changes into a repository. The hg incoming command will tell us what changes the hg pull command would pull into the repository, without actually pulling the changes in.

```
$ cd hello-pull

$ hg incoming ../my-hello

comparing with ../my-hello

searching for changes

changeset:   5:3358452fd7d5

tag:         tip
```

```
user:       test
date:       Thu Jan 01 00:00:00 1970 +0000
summary:    Added an extra line of output
```

The hg pull command will simply pull the changes in, but it will not be applied to the working directory. To update the working directory, we'll use the hg update command.

```
$ grep printf hello.c
    printf("hello, world!\");
$ hg update
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ grep printf hello.c
    printf("hello, world!\");
    printf("hello again!\n");
```

The hg update command can be used to update the working directory to the state at which it was in, at any revision in the history of the repository. To find out the revision of the working directory, we can use the hg parents command.

```
$ hg parents
changeset:   5:3358452fd7d5
tag:         tip
user:        test
date:        Thu Jan 01 00:00:00 1970 +0000
summary:     Added an extra line of output
```

To update the working directory to any particular revision, the revision number or the changeset ID is given to the hg update command.

```
$ hg update 2
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
```

```
changeset:   2:fef857204a0c
user:        def@gmail.com
date:        Sat Aug 15 22:05:04 2018 +0200
summary:     Introduce a typo into hello.c.

$ hg update
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   5:3358452fd7d5
tag:         tip
user:        test
date:        Thu Jan 01 00:00:00 1970 +0000
summary:     Added an extra line of output
```

Pushing Changes to another Repository

We can push changes from the repository that we're working to any other repository. For example:

```
$ cd ..
$ hg clone hello hello-push
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

The command hg outgoing will tell us what changes will be pushed to another repository.

```
$ cd my-hello
$ hg outgoing ../hello-push
comparing with ../hello-push
searching for changes
changeset:   5:3358452fd7d5
tag:         tip
user:        test
date:        Thu Jan 01 00:00:00 1970 +0000
```

```
summary:     Added an extra line of output
```

The command hg push does the actual push.

```
$ hg push ../hello-push
pushing to ../hello-push
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Similar to hg pull, the hg push command does not update the working directory in the repository that it's pushing changes into. Unlike hg pull, hg push does not provide a -u option that updates the other repository's working directory. This asymmetry is deliberate: the repository we're pushing to might be on a remote server and shared between several people. If we were to update its working directory while someone was working in it, their work would be disrupted.

For additional reading: https://www.mercurial-scm.org/guide