Chapter

8

Code Versioning

CHAPTER AUTHORS

Damien, Florian Catala

Koh Cher Guan

Soh Yuan Chin

Teo Wai Ming Steve

CONTENTS

1 Introduction to Version Control Systems

1.1 What is a version control system?

Version Control, also known as *Revision Control*, is the management of changes to computer files, often source code files. It is most commonly used in software development, where a team of people may change the same files. Every change made to the file is tracked, along with who made the change and a summary of the change, which can consist of why the change was made, what the changes are and/or any other information that are relevant to the change. Version Control is an important aspect of Software Configuration Management (SCM).

Version Control Systems (VCS) are software implementations that simplify and in some cases automate the process of Version Control. They can be broadly categorized into two categories; centralized and distributed.

Revision Control Systems and Version Control Systems mean the same thing. However, when the words version and revision are used on their own, revision refers to the internal versions, known only to the VCS and developers, while version refers to the version of the software known to the public. For example, *SomeSoftware* Version 2.0 is the version released to the public and it is revision 1337 in the VCS.

1.2 Why use a version control system?

Many of us are probably already doing version control without realizing it. For example, while working on a project inside a folder named `HelloWorld`, we have probably saved multiple copies of the folder as `HelloWorldStable`, `HelloWorldOld` or something along these lines. The more meticulous ones among us would have given it more meaningful names that contain a date or version, such as `HelloWorld_13032011` or `HelloWorldVersion2`. In addition, we would have probably done it using the classic **Copy and Paste**, followed by a **Rename**.

However, in the above scenarios, we ourselves perform the role of the VCS. We waste time thinking of what file names to use and from time to time we forget to save an important milestone. It is not uncommon to see `HelloWorldV3.java` in one folder and another `HelloWorldV3.java` in another folder, each with different file contents to add to the confusion.

> *"With another project, …, I found I had no less than 3 different versions of the code lying around on my hard drive. It took some investigation with diff in order to find out which one was the most current." – Fraser Hess, Developer of Harmony*[1]

The situation is even worse in team projects. Do you find yourselves emailing each other every time you make some changes? This problem becomes harder to manage as more people get involved in the project. To compound this problem further, what happens if there is a serious problem with one of the changes? Short of the person admitting his mistake, the only way to find out who is responsible for it would be to search through all the emails and their attached files and check each attachment individually.

With a proper VCS, all of the above problems can be alleviated. Projects, small or large, need a VCS to track changes, account for changes, allow multiple people to work at the same time and allow the developers to revert to an older working version if their latest version is problematic. In addition, Version Control Systems have the ability to duplicate your code into a separate area so you can work on it in isolation without affecting the current code. This is known as **branching**. When you are done with your work, you can then **merge** these changes from your branch into the current code. A VCS handles the tedious task of managing code changes for you so you have more time to do what is important; developing software.

[1] http://sweeterrhythm.com

In summary, a VCS has the following features:

- Accountability
- Branching and Merging
- Change Tracking
- Concurrency (Multiple users can work on the same project)
- Reversibility

1.3 A brief history of version control systems

We will take a look at a brief history of Version Control Systems (VCS). By looking at the background of VCS, we can better understand how much VCS has changed over the years, the motivation of development behind each VCS and the evolution from centralized systems to distributed systems.

Version Control Systems have come a long way since it was first introduced. One of the earliest VCS was the Source Code Control System (SCCS), developed by Marc J. Rochkind in 1972. It only works locally and only on individual files. Although merging was not supported, its file storage technique was used by later VCS and is key to advanced merging and versioning techniques. It was later written for Unix and it became the de-facto VCS for Unix until Revision Control System (RCS) merged.

RCS, developed in 1982 by Walter F. Tichy was an improved version of SCCS; it supported binary files and has better storage performance. Like SCCS, it was itself replaced by Concurrent Versions System (CVS) in 1986.

CVS was written by Dick Grune as a set of RCS scripts to operate on multiple files. It was re-released in 1990 as a client-server VCS, making it the first Centralized Version Control System (CVCS). However, it had many flaws, most notably, the lack of an atomic commit. To tackle this, Subversion (SVN) was written and sponsored by CollabNet in 2000. Its main goal, "CVS done right" was central to its purpose, and as it introduced many improvements over CVS, which made it the de-facto CVCS used nowadays.

Distributed Version Control Systems (DVCS) emerged in the 1990s with the commercial release of Sun Workshop TeamWare by Sun MicroSystems. It was the first DVCS and it was used internally by Sun for the development of the Solaris and Java platforms. It incorporated a lot of advanced features not found in earlier VCS, such as the local repository. DVCS took off and in 1998, BitKeeper was released by BitMover Inc, whose CEO Larry McVoy previously designed TeamWare.

BitKeeper was built upon many TeamWare concepts. It was most notably used as the SCM tool of the Linux Kernel project from 2002 to 2005. It was used by project founder Linus Torvalds himself, despite it being a proprietary product. It withdrew from the Linux Kernel project controversially.

To fill up the void left behind after BitKeeper's exit, two new DVCS was initiated around the same time, Git and Mercurial. Linus Torvalds wrote Git in 2005 within 2 months. His motivation for Git was to have it support a BitKeeper workflow instead of the CVS workflow which he disliked, while having high performance and strong safety measures against corruption of source code. It was used in the Linux kernel project after its release.

Likewise, Matt Mackall developed Mercurial with similar aims and motivation as that of Git. Written in Python, it focused on cross-platform interoperability and user-friendliness.

Till this day, many developers are still continuing to adopt DVCS over CVCS. This may be attributed to several reasons, which will be seen later.

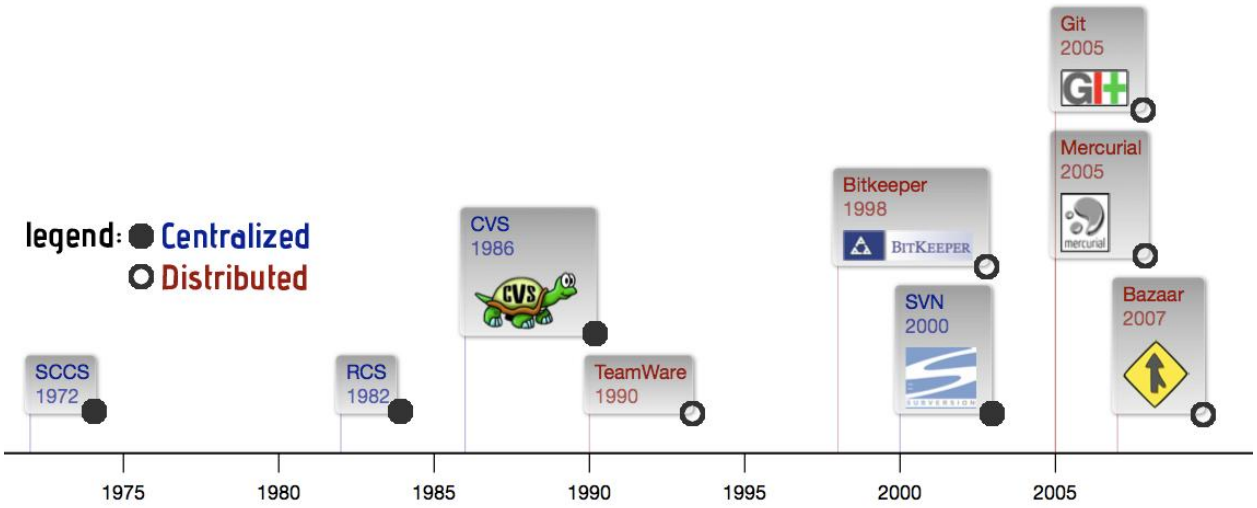

Figure 1: Timeline of the various VCS

2 Common Version Control Systems Terminologies

Entities

Repository (Repo)	The database where the files and historical data are stored, including the author of the changes and the summary of each change. Commonly called **repo** for short.
Working Copy	The local directory of your files.
Trunk	The primary location for code in the repo. Remember that a Version Control System allows for branching, the trunk is the one that usually contains the stable code that everyone is working on. Also called **master**, **main** or **default**.
Branch	A secondary location of the code in the repo. A repo can have multiple branches but usually only one trunk.
Revision	A revision is the set of changes whenever a check in is performed. Each revision is given a number. See Figure 2.

Actions

Check In (Commit)	Uploads a changed file or a set of changed files to the repository. Commonly known as **commit**.
Check Out	Downloads a file or a set of files from the repository (for the first time).
Add	Tells the Version Control System to track a file, a set of files or a directory. These tracked files do not go into the repository until the next check in.

Update	Synchronize the files in your working copy with the latest files from the repository. This is normally performed after a **check out**.
Revert	Discards all changes in the working copy and use a specified revision from the repository. See Figure 3.
Branch	The act of creating a branch. See Figure 4.
Merge	Apply the changes from one or more file(s) to another. For example, you can merge code from a branch into the trunk. See Figure 5.
Resolve	A **conflict** may occur when multiple changes to a file contradict one another. When that happens, the process of fixing these conflicts and checking in the fixed file is called **resolve**.
Tag	Label a revision for easy reference. See Figure 6.

Figure 2: New revisions are created whenever a commit is made

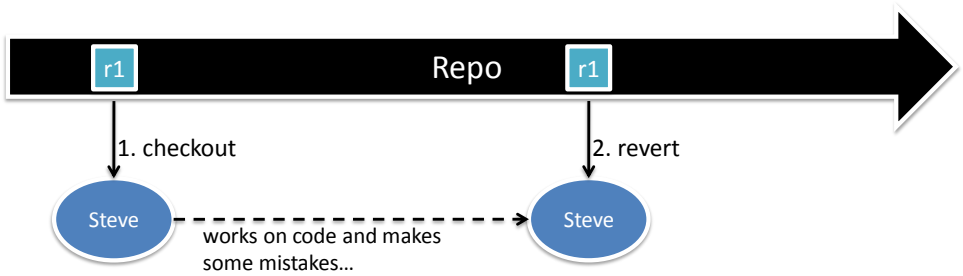

Figure 3: Reverting after making mistakes

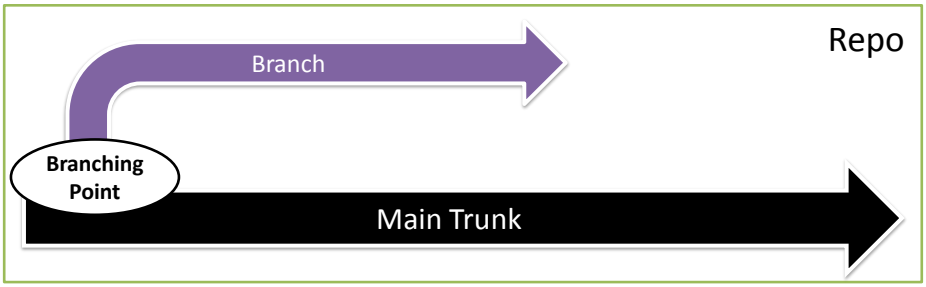

Figure 4: Branching

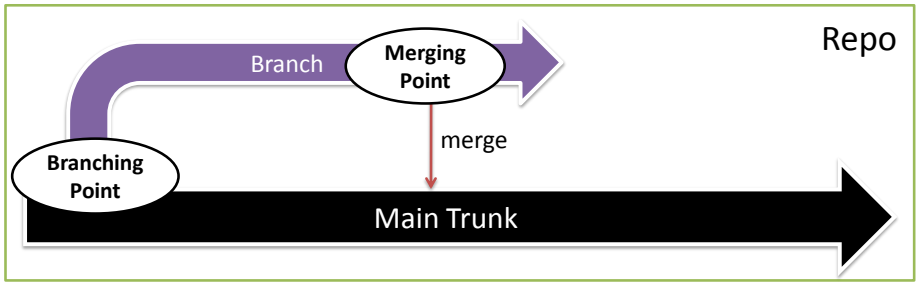

Figure 5: Merging after branching

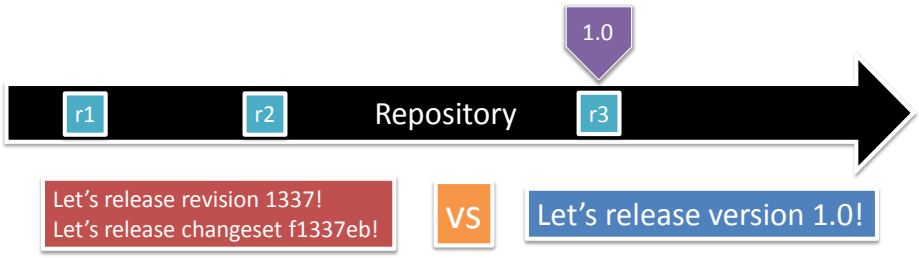

Figure 6: Tagging Revision 1337 as 1.0

3 Centralized Version Control Systems

3.1 Overview

In a Centralized Version Control System (CVCS), there is a single **centralized** repository, which everyone uses to check in, check out and basically perform all other Version Control Systems-related operations.

In Figure 7, we have a scenario that consists of 4 people working on a single project. It can be clearly seen that all 4 of them do not interact with one another. Instead, all 4 of them interact with just the centralized repository. In this scenario, the repository can be thought of as a sort of boss that everyone must report to. If Steve makes a change and he wants to share it with everyone, he must first commit his changes into the repository.

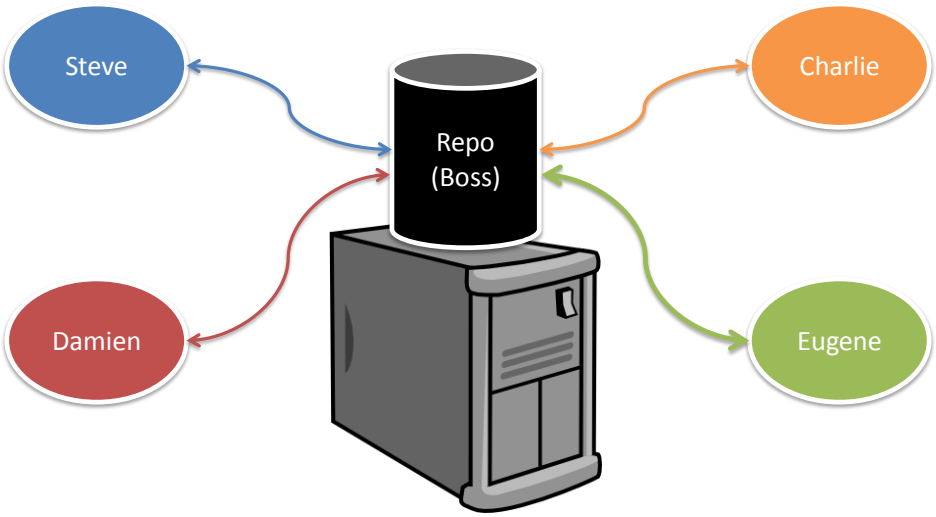

Figure 7: CVCS Overview

Some of the available CVCS available today are Subversion[2] (SVN), Concurrent Versions System[3] (CVS) and Perforce[4].

3.2 workflow

In a CVCS, every contributor must work off the master centralized repository.

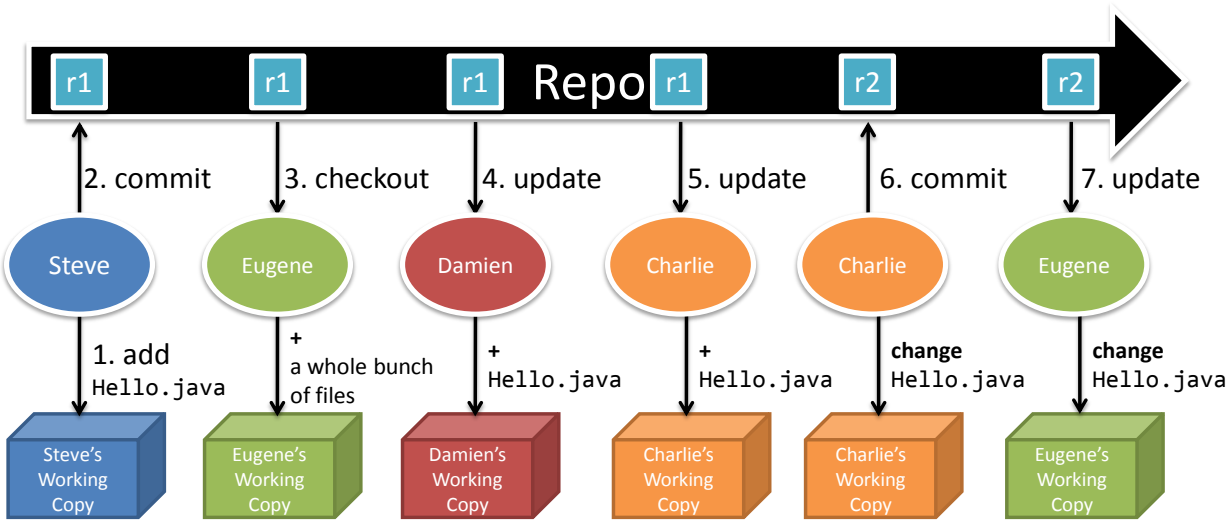

Figure 8: CVCS Workflow

Let us take a look at each step of Figure 8, which outlines a basic CVCS workflow. Firstly, Steve creates a new file called `Hello.java` and **adds** it using the Version Control System. At this point in time, the VCS will start tracking `Hello.java` but the file is not in his repository yet. Next, Steve **commits** his changes (in this case, the only change was to add `Hello.java`). At this point in time, the repository will have `Hello.java` inside.

[2] http://subversion.tigris.org
[3] http://www.nongnu.org/cvs
[4] http://www.perforce.com

Next, Eugene, who is new to the team, wants to start working on the project. To get the project files onto his computer, he first has to do a **checkout**. Upon completion of the checkout process, Eugene will then have all the project files in his working copy on his computer.

Following that, Damien and Charlie, who are both existing team members, want to synchronize the code in their working copy with the latest code from the repository. In order to do that, they execute the **update** action, which will bring their working copy up to date with the repository. In this case, they both discover they have one new file, `Hello.java`.

Charlie then makes some changes to `Hello.java` after discovering a bug with it. When he is done with his changes, he **commits** it. Next, Eugene, who performs an **update** after Charlie is done, receives the updated `Hello.java`. The changes from Charlie's `Hello.java` are automatically **merged** into Eugene's `Hello.java` to bring it up to date.

At this point in time, only Charlie and Eugene have the latest version of the code. In order for Steve and Damien to have the updated code as well, they should perform an **update** again.

4 Distributed Version Control Systems

4.1 New terminologies

Entities

Local Repository	A repository that is on the local computer.
Remote Repository	A repository that is hosted on a server. Also known as **hosted repository**.
Changeset	Distributed Version Control Systems tend to use the word changeset in place of revision. A revision has a revision number. Likewise, each changeset has a **changeset ID**, which is used to uniquely identify the changeset.

Actions

Clone	Makes a copy of an existing repository, normally a remote repository.
Push	Sends a (committed) change to another repository, normally a remote repository. Authorization is needed to push changes most of the time.
Pull	Grab a (committed) change from another repository, normally a remote repository.

4.2 Overview

In a Distributed Version Control System (DVCS), every developer has his or her own local repository. In addition, a remote repository is used for code hosting and sharing. It is perfectly fine to work with just the local repository on a project, without ever having to create a remote repository. This is vastly different from the centralized model in which there is only one central

repository that everyone must base his or her work on. Sometimes the word **decentralized** is used instead of distributed.

In addition, there can be any number of remote repositories where people share their changes by pulling and pushing to one another. DVCS have no forced hierarchy. You can decide if you want a "centralized" repository or if you want to have multiple repositories for different purposes (see Section 4.4).

In Figure 9, we have 4 developers, each with their own local repository as well as a remote repository which all of them can pull from and push to. In addition, one of the developers, Charlie, has another remote repository, which he uses to host his code privately without sharing with others.

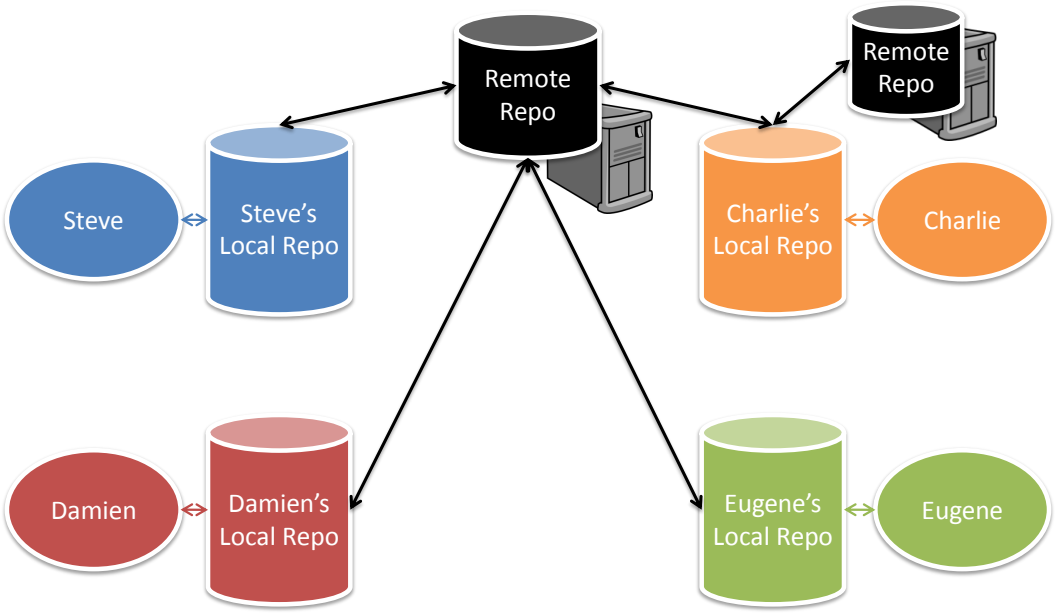

Figure 9: DVCS Overview

Some of the available DVCS available today are Git[5], Mercurial[6] and Bazaar[7].

4.3 Workflow

Unlike a CVCS, the workflow in a DVCS can be very different between two projects. This is due to the lack of a centralized repository. As a result, there can be various ways to organize a project (see Section 4.4). However, regardless of how the workflow is in a DVCS, there are a few concepts that every DVCS user must know.

Firstly, unlike centralized systems, distributed systems have the addition of a local repository. This means that all the contributors will have their own repository on their computers. As a result, most of the operations such as add, revert, update and commit are all done locally, without the need to interact with a remote repository which is hosted on some server.

Secondly, instead of checking out to start working on a project, users must first clone the project. Cloning basically creates a copy of a repository (see Figure 10). For existing projects, contributors will usually clone a copy of the remote repository into their own computer.

[5] http://git-scm.com
[6] http://mercurial.selenic.com
[7] http://bazaar.canonical.com

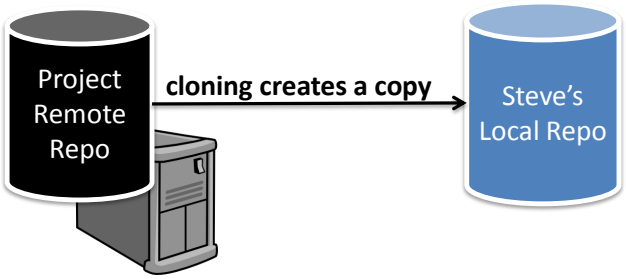

Figure 10: Cloning a repository

If all the Version Control System operations can be done locally, how do we actually send our changes to a remote repository? The answer comes in the form of **pulling** and **pushing**. Pushing means to upload your changes to another repository, usually a remote one, while pulling means to download the changes from another repository, again, usually a remote one.

In a typical scenario, a contributor will first work on just his or her local repository, without the need for an Internet connection. After numerous commits, the contributor is finally happy with his or her work and ready to share it with others. At this point in time, the contributor will **push** his or her code to his or her remote repository (see Figure 11). The rest of his team, eagerly waiting for this awesome new code, will proceed to **pull** it from the contributor's repository. Subsequently, if the team members are all happy with this new code, they will merge it into their own code and then **push** to their own repository (see Figure 12).

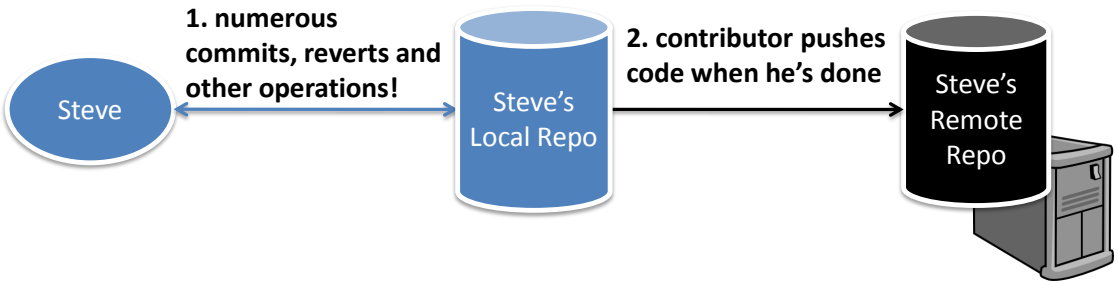

Figure 11: Steve (contributor) working on his changes

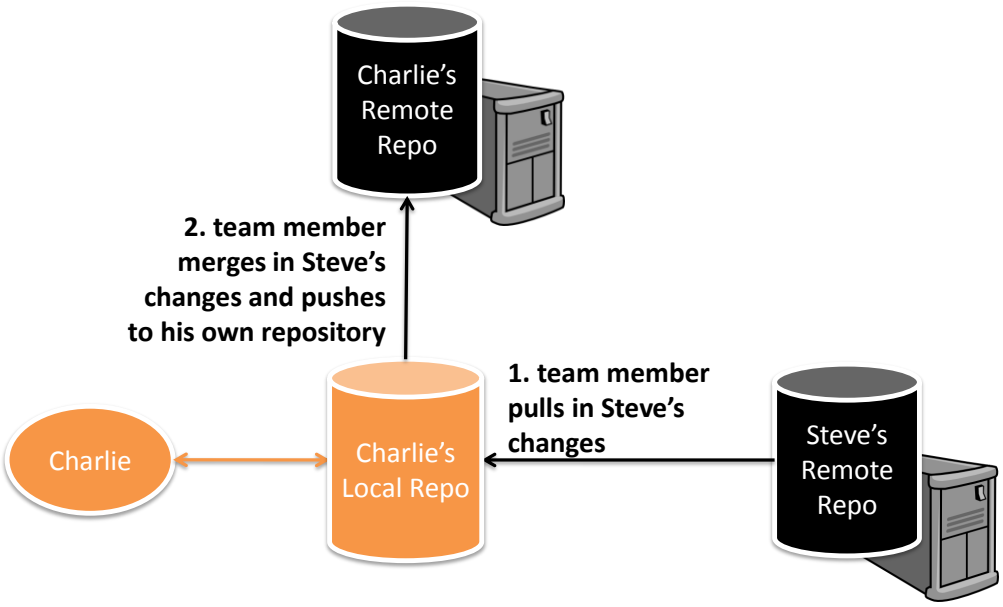

Figure 12: Charlie pulls Steve's changes

Notice that Charlie does not push directly to Steve's remote repository, or vice versa. Although this is possible (see Section 4.4.2), most of the time only the contributor has push rights to his or her own remote repository. In addition, it is important to note that a pull does **not** merge the changes into the repository, unlike a centralized system's update. In order to merge it in, you have to perform a pull, followed by a merge. Most DVCS come with commands that combine them both for ease of use.

4.4 Flexible organization

DVCS is more flexible than CVCS in how the members of a team and their code should interact. For example, you can assign a location to be the "central" repository, so it works just like a CVCS, or you can make it peer-to-peer, where everyone is equal. The following sub-sections illustrate some possible structures a DVCS might have.

4.4.1 Peer-to-Peer

In Figure 13, we have a peer-to-peer system. As seen from the diagram, all of the project members are peers and are able to share their changes to and from one another. Assuming that only the owner is allowed to push to his or her own remote repository, we will have a model where everyone is able to pull changes from one another without affecting one another's code. This is usually the case as allowing others to push to your own remote repository can result in problems, especially if they push bad code into your repository. This model is most commonly used in open source projects.

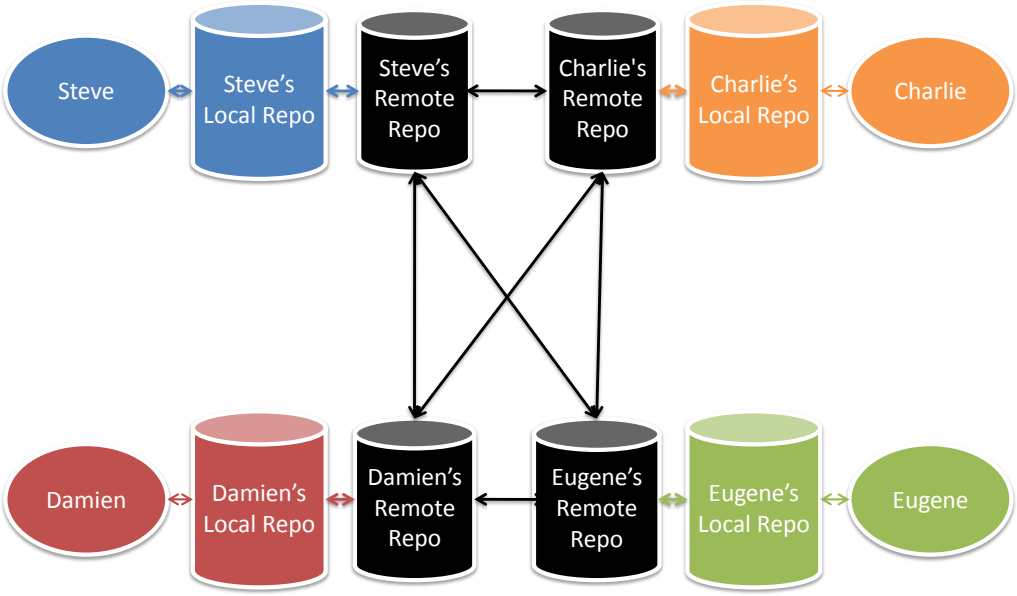

Figure 13: Peer-to-Peer

4.4.2 Shared Push

In Figure 14, we have a shared push model. In a shared push model, all the project contributors are given permission to push to the repository. Notice that there is only one remote repository. In fact, this is essentially the same as a CVCS with one exception, the addition of local repositories.

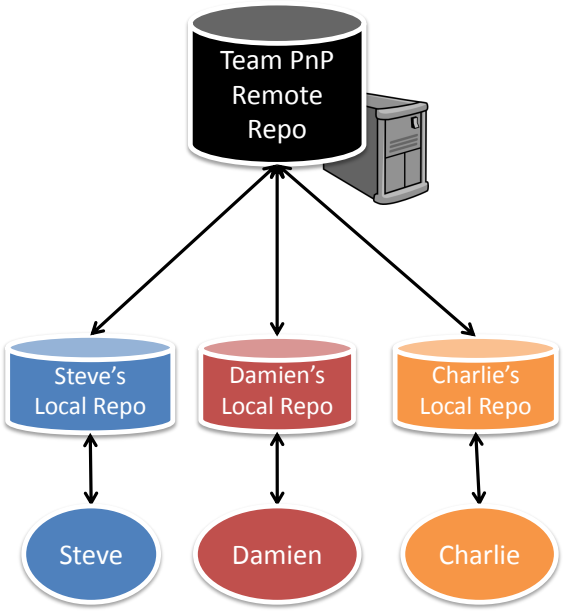

Figure 14: Shared Push

4.4.3 Pull-Only

Figure 15 shows a pull-only model. In a pull-only model, only a selected few are allowed to push into the "main" repository. In this case, only Damien is authorized to push to **Team's Remote Repo**. The rest of the contributors are only allowed to pull from the "main" repository, work on it and then push their changes into their own remote repository. In such a scenario, Damien is

also known as a **lieutenant**. This model allows for him to pull as required. For example, if Steve fixed a bug with his latest changes, while Charlie introduced a bug, Damien can choose to pull only from Steve. This model can be scaled up further (see Figure 16). This model is being employed in the Linux project.

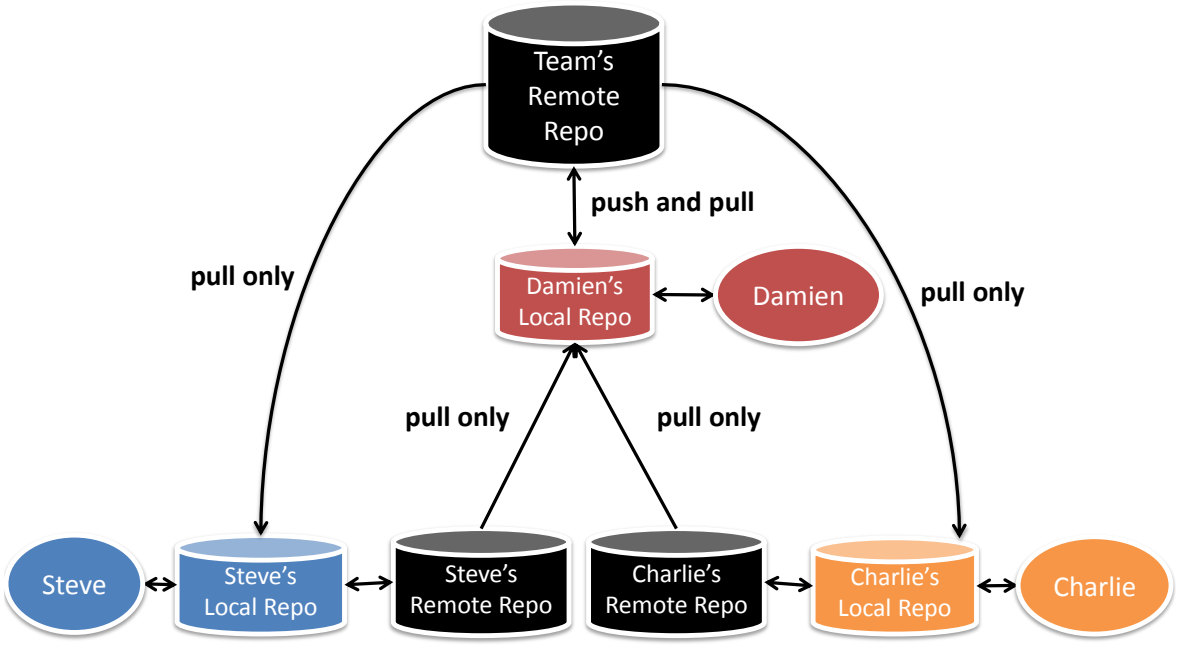

Figure 15: Pull-only model

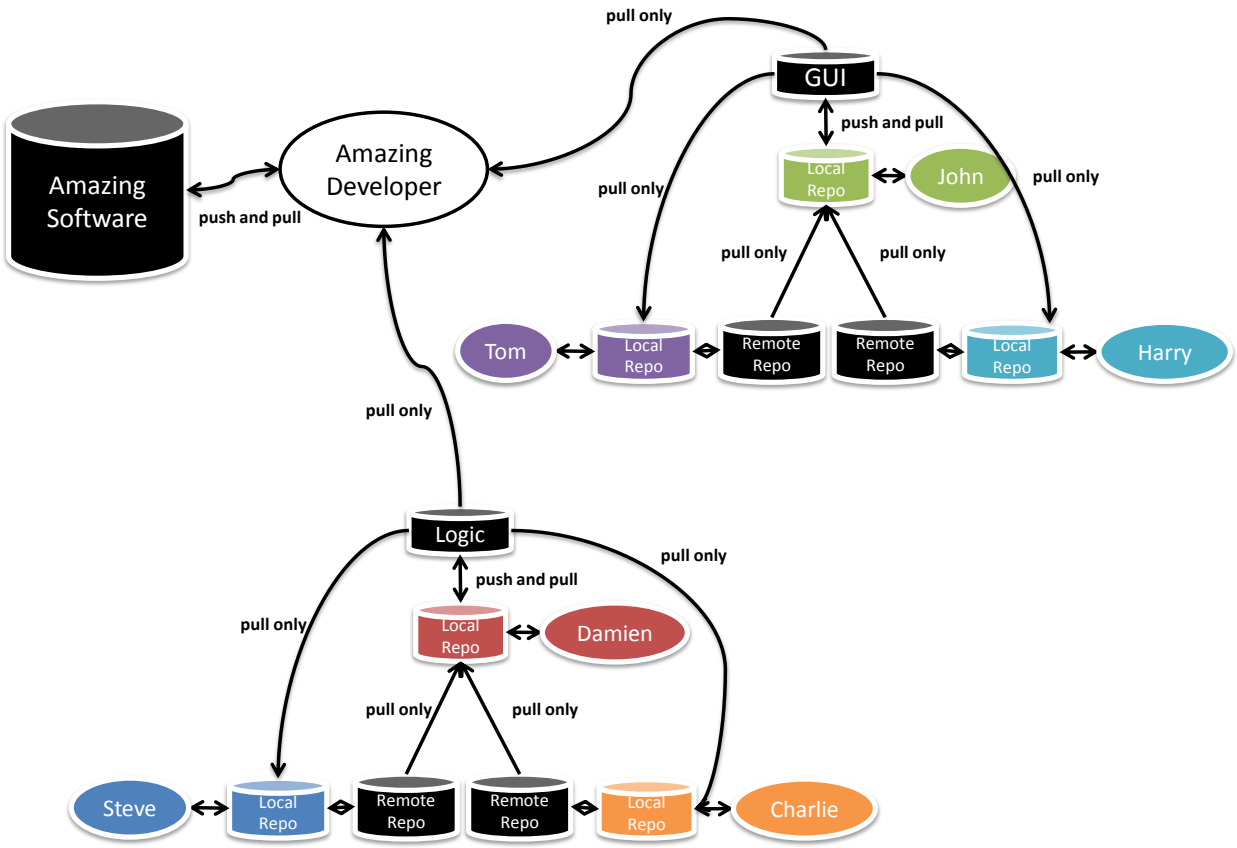

Figure 16: Scaled up pull-only model

As we can see, there are many ways to organize your code and team hierarchy using a DVCS. Instead of adapting to the DVCS workflow, you can make it adapt to your needs instead.

5 Centralized versus Distributed Version Control Systems

5.1 CVCS

Advantages

Simple workflow. Since the workflow in a CVCS is linear, it is familiar to most people and easy to get started with.

Implicit hierarchy. With a central repository, every contributor knows where to commit their changes to and where to get the latest version of the code.

Disadvantages

Very few offline operations. Most of the operations in a CVCS such as commit, revert and branch requires an Internet connection to the remote repository. As such, when you are on the move, it becomes almost impossible to make use of your CVCS.

Inflexible organization. Due to the forced structure of a central repository, you have to adapt the organization of your team to the CVCS, instead of the other way round.

Inefficient merging. Although CVCS come with branching and merging functions, most of the CVCS are largely inefficient with merging since they do not keep track of where each change came from.

5.2 DVCS

Advantages

The key advantages of a DVCS system are as follows:

Ability to work offline. Since most of the operations are performed on the local repository, with the exception of pulling and pushing changes, a developer can literally work offline until he or she is ready to share the code with others, or there is a need to retrieve code from others. This means you can make full use of a DVCS while in the train or on the plane.

Local repository. With a local repository, common operations like revert, commit and branch are faster since there is no need to depend on a possibly unstable Internet connection. In addition, it is precisely the inclusion of a local repository that allows a user to work offline.

Branching and merging is easier. The way DVCS handle branching and merging is much more efficient since they were built around sharing changes. Although different DVCS may have different ways of handling changes, fundamentally, all of them are able to keep track of where each change come from, making merging easier.

More flexibility in management. Since there is no implicit structure with DVCS, you can structure your project to suit your needs.

Scales better with more people. With a DVCS, any new project members can clone your existing remote repository and work off it, and you can decide whether to pull from them. There is no need to give them permission to your existing repository.

Disadvantages

No "latest" version. Since there is no central location, you do not know which of your team members has the latest version. This can be fixed by designating a central location that helps clarify where the latest "stable" release is.

No universal revision numbers. Every repository has its own revision numbers depending on the changes. Instead, people refer to change numbers, such as a1337bc, which are not elegant or easy to remember. However, you can tag releases with meaningful names.

6 ADVANCED TOPIC: BRANCHING AND MERGING

Each time we commit in Mercurial, it creates a new changeset. Mercurial assigns both a revision number and a changeset ID to each changeset. Both these terms can be used interchangeably. The important difference to note is that revision number should be used only within the local repository, while changeset IDs are unique and should be used when pulling or pushing changes across repositories. For example, after a commit and we run `hg log`, we might get the following:

```
changeset:   0:2ee237db8944
tag:         tip
user:        John Doe <johndoe@email.com>
date:        Fri Apr 01 16:38:52 2011 +0800
summary:     Added file
```

As we can see, from the part in bold, it is revision number **0** and the changeset ID is **2ee237db8944**.

In addition, you should get familiar with the concept of *head*. A head is basically a changeset with no children. A Mercurial repository can have multiple heads.

6.1 What is Branching?

Suppose you have been working on a project and decide to implement a radical new feature. However, you do not want your new feature to introduce any bugs into your existing code. As a result, you decide to make a copy of your current project and work on that copy instead, so that your existing project remains safe. That copy, in essence, is a **branch**. A branch is essentially a copy of project and thus it has a common history with the project. The act of creating a branch is known as **branching**.

Another scenario when a branch is required is when you want to work on a separate set of features for a team project. Suppose all the team members are working on a particular branch of code, e.g. **stable**, and you will like to work without affecting their code. In this scenario, you will create a branch based off stable, give it a name, e.g. **experimental** and work on it. This way, the changes you make to experimental will have no effect on stable at all and vice versa.

VCS have support for branching and thus can create the copy when you require it to do so, instead of you manually copying the folder. In addition, the VCS will know of the existence of the branch and basically keeps a record of all branches within a project (repository).

6.2 What is Merging?

Suppose you are in the first scenario in Section 6.1 and you have finished implementing and testing your radical new feature. At the same time, you have made some changes to your original project. You would like to have the new features in your original project. Manually, you will have to copy the code from one project to another. VCS come with a feature known as **merging**, which attempts to find out the differences among files and apply them to the file you want.

6.3 Branching and Merging in Mercurial

Different VCS handles branching differently. We will be taking a look at how branching is handled in Mercurial. In essence, Mercurial is able to handle branching in four different ways. The main purpose is to illustrate the different branching concepts.

1. Clones
2. Bookmarks
3. Named Branches

4. Anonymous Branches

6.3.1 Clones

Cloning is one way in which a branch can be created. Cloning can be done as follows:

```
hg clone <REPO> <BRANCH>
```

where `<REPO>` is the folder containing the original project and `<BRANCH>` is the name of the new folder which will become the branch. Once done, we will have two identical copies of the repository.

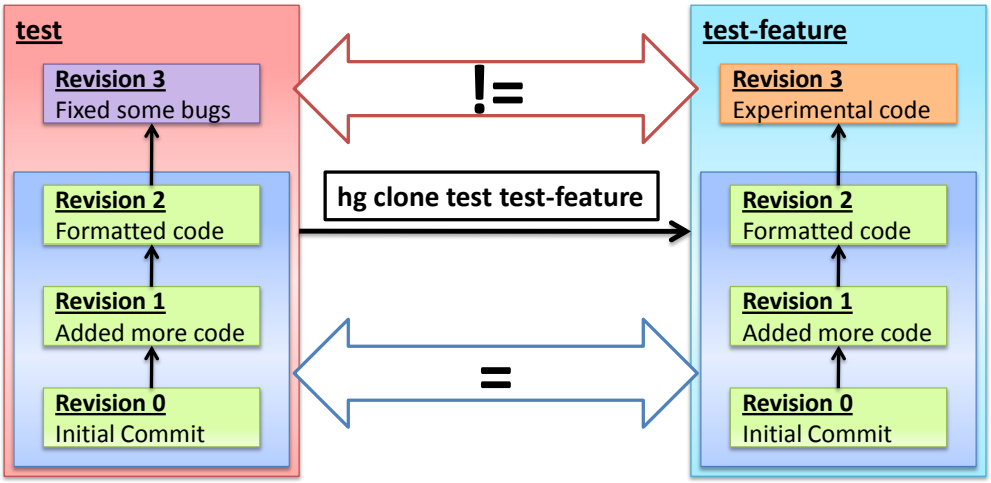

Figure 17: Branching with clones

In Figure 17, we can see that `test-feature` was created when `test` was at Revision 2. At this point of time, both `test` and `test-feature` are identical. Subsequently, some bugs were fixed in test while experimental code was added to `test-feature`. They are no longer identical after these modifications.

In order to merge changes in from one branch to another, we make use of either pushing or pulling to move changes from one branch to another, as we would if we wanted to share changes when working in a team.

Advantages

- Safest way of creating a branch. The branches, being separate repositories, are completely isolated, so what you do in one branch will not affect the other branch, until you push or pull.
- Deleting a branch is as simple as deleting the repository folder.

Disadvantages

- Slower than the other methods, since creating a branch by cloning literally means copying the current repository.
- To share the branches, they would have to be published as separate repositories. Thus, team members who are interested in all your branches will have to clone all your repositories. This is wasteful as well, since some if not most of the code in these separate repositories will be common.

Sidenote: The Mercurial development team themselves use this form of branching.

6.3.2 Bookmarks

Bookmarks are another way to do branching in Mercurial. The command to do that is:

```
hg bookmark <NAME>
```

to create a bookmark at the current revision, or:

```
hg bookmark -r <REV> <NAME>
```

to create a bookmark at a specific revision. By default, when multiple bookmarks point to the same changeset, they will all move forward together. It is possible to configure Mercurial to only track and update the currently active bookmark by adding the following lines in Mercurial's configuration file:

```
[bookmarks]
track.current = True
```

Once done, a bookmark that points to current or specified revision will be created. Bookmarks are essentially pointers to commits and are automatically updated when new commits are made. They are similar to tags, but bookmarks move along with changes whereas a tag is fixed to the specific revision.

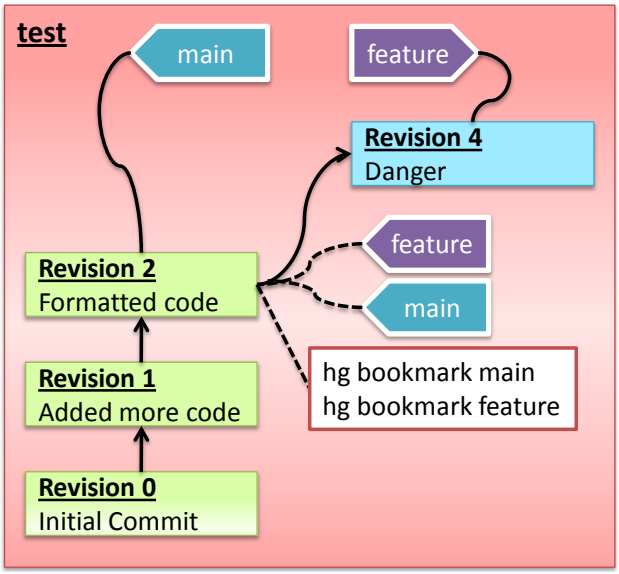

Figure 18: Branching with bookmarks

In Figure 18, upon running the two commands `hg bookmark main` and `hg bookmark branch`, two bookmarks are created. At this point in time both bookmarks are pointing to the same revision. To switch between bookmarks, we use the `hg update` command. For example, to switch to the `feature` branch, we will execute `hg update feature`. Following the branch switch, a commit was made. This commit belongs only to `feature` and not to `main`.

To merge changes using bookmarks, we first have to switch to the branch we want the changes to be merged into. For example, if we wanted to merge in the changes from `feature` to `main`, we will do the following:

```
hg update main
hg merge feature
hg commit –m "Merged changes from feature branch"
```

The above set of commands basically switches to the branch `main`, merges the changes from `feature` into `main` and finally commits the changes. For more information on bookmarks, see http://mercurial.selenic.com/wiki/BookmarksExtension.

Advantages

- Fast and lightweight (compared to cloning)
- Bookmarks are easily deleted

Disadvantages

- Once a bookmark is deleted, there is no longer any history of it. Giving your commits a proper commit message will ensure that a history of it is kept when you merge in changes from a bookmark to another.

6.3.3 Named Branches

The third way of creating a branch in Mercurial is via the form of named branches. The command to do so in Mercurial is:

```
hg branch <NAME>
```

Whenever a commit is made, it will be on the same branch as its parent, unless `hg branch <NAME>` is used before the commit so that the commit goes to the specified branch.

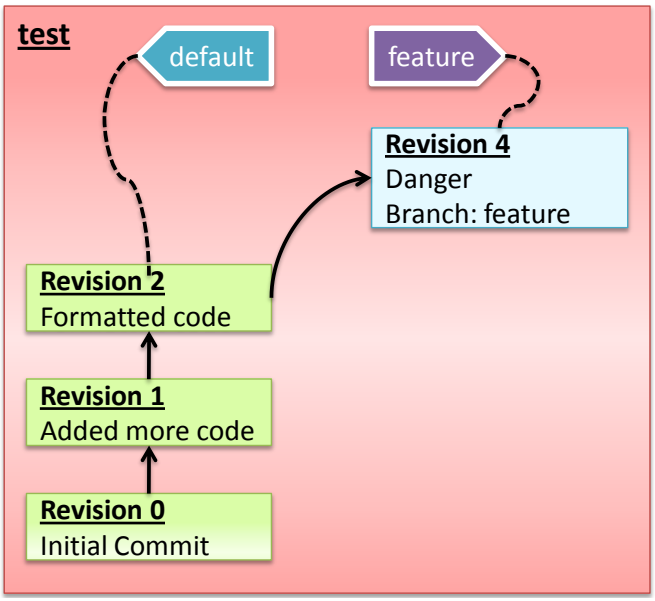

Figure 19: Branching with named branches

Although the named branches diagram (Figure 19) looks similar to that of bookmarks (Figure 18), it is important to note the following differences:

- The branch name is permanently recorded as part of the revision's metadata as seen in Revision 4 in Figure 19. This is unlike a bookmark where no reference to the bookmark can be seen in the metadata.
- These branches do not actually exist as a physical file on the disk, unlike a bookmark. When we request to switch to and use a branch, Mercurial calculates the revision on the fly.

In addition, the default branch in Mercurial is named `default`. Switching between branches is fast since named branches are just some extra metadata made on a commit. The time it takes to switch between branches is determined by the difference between them. In addition, you can tell which branch a commit was made, since the branch name is part of the commit's metadata. For example, running hg log on after a commit on branch `feature` is done will give us the following:

```
changeset:   0:2ee237db8944
branch:      feature
user:        John Doe <johndoe@email.com>
date:        Fri Apr 01 16:38:52 2011 +0800
summary:     Added file
```

As we can see, the branch name is a part of the commit's metadata. However, this means you cannot really delete branches, since the branch is technically just part of a commit's metadata. You cannot rename a branch as well and everyone who has used that commit of yours will have the same branch name as the one you specified. Altering or deleting a branch will mean altering older commits and thus is not doable. Branches can be marked as closed to indicate to Mercurial that these branches are no longer being worked on. To switch between branches, we make use of the `hg update <NAME>` command again. Similarly, to merge in changes, we first switch to the branch we want the changes to be merged into. For example, if we made some changes to feature, and would like to now merge the changes into default, we will do the following:

```
hg update default
hg merge feature
hg commit –m "Merged changes from feature branch"
```

Please see http://mercurial.selenic.com/wiki/Branch and http://mercurial.selenic.com/wiki/NamedBranches for more information on named branches in Mercurial.

Advantages

- Every revision on a branch has the branch name as part of its metadata. Useful for logging purposes.
- Easy and fast to switch between branches.

Disadvantages

- It is against the conventional idea of branches. From Mercurial themselves, *"The term branch is sometimes used for slightly different concepts. This may be confusing for new users of Mercurial."*
- Branches cannot be deleted.

6.3.4 Anonymous Branches

Anonymous branches are basically branches without names. They are basically heads in the Mercurial repository. You refer to the branches by revision number or changeset ID. This method works the best for short-lived branches, where you know that the branch will be integrated into the main development trunk very soon.

The concept of anonymous branches is best explained with a practical example. Suppose you have a new repository and you just added a new file `Hello.txt` which contains the line "Hello". After that, you decide to add and commit the file to your repository. Running `hg log` may result in the following:

```
changeset:   0:caba3ec1541b
tag:         tip
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:51:50 2011 +0800
summary:     Added Hello.txt
```

Next, you make some changes to `Hello.txt`, perhaps you added the line "World" and perform another commit. `hg log` may now show the following:

```
changeset:   1:a814a8d50cf4
tag:         tip
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:54:12 2011 +0800
summary:     Added line World

changeset:   0:caba3ec1541b
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:51:50 2011 +0800
summary:     Added Hello.txt
```

This is where it gets interesting. Suppose you want a branch that shows "Earth" instead of "World". You can perform `hg update -r 0` to switch to your very first commit, added in "Earth" and then commit. `hg log` will now show the following:

```
changeset:   2:1a5bc64dc907
tag:         tip
parent:      0:caba3ec1541b
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:57:31 2011 +0800
summary:     Added line Earth

changeset:   1:a814a8d50cf4
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:54:12 2011 +0800
summary:     Added line World


changeset:   0:caba3ec1541b
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:51:50 2011 +0800
summary:     Added Hello.txt
```

Running `hg heads` gives the following output:

```
changeset:   2:1a5bc64dc907
tag:         tip
parent:      0:caba3ec1541b
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:57:31 2011 +0800
summary:     Added line Earth
```

```
changeset:   1:a814a8d50cf4
user:        John Doe <johndoe@email.com>
date:        Wed Apr 06 18:54:12 2011 +0800
summary:     Added line World
```

We have arrived at a situation where there are two heads and they are basically anonymous branches. To switch between them, we have to specify `hg update –r <N>`, where N can be the revision number or the changeset ID. Similar to named branches, if we want to merge in changes, we first have to switch to the branch we want the changes to be merged into. An example would be as follows:

```
hg update –r 1
hg merge –r 2
hg commit –m “Merged in changes”
```

Advantages

- Fastest and easiest way to branch. There is no need for any naming of the branch and no need to delete the branch when you are done.
- Most suitable for short-lived feature branches that will not leave your local repository.

Disadvantages

- Since there is no descriptive name for the branch, good commit messages will have to be written.
- Anonymous branches are referred to only by their revision and changeset numbers. Thus, you will have to use `hg log` to find out the respective numbers each time you want to switch branches.

7 CONCLUSION

As we have seen, each of the different systems comes with their own advantages and disadvantages. The choice of which VCS to use largely depends on your needs. For example, if you are new to VCS and would like to put a simple project on it, a CVCS will be perfectly suitable for your needs. The workflow is simple to get used to and with a small project, the implicit hierarchy might be a good advantage.

On the other hand, if you need a lot of flexibility in management, you might want to use a DVCS instead. A strong advantage of DVCS is the inclusion of a local repository, which also leads to the ability to work offline. In fact, some detractors of DVCS are convinced that it is not so much the distributed part of DVCS but the ability to work offline that is attractive. It is also important to note that it is possible to model a DVCS to behave like a CVCS, as seen with the shared push model (see Section 4.4.2), but not the other way round.

> *“The slogan of Subversion for a while was ‘CVS done right’ or something like that and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right.”*
> *– Linus Torvalds*

With the huge number of proponents behind DVCS nowadays, including prominent people like Linus Torvalds, we are inclined to say that DVCS is better. However, the truth is, **using any form of Version Control System is better than not using it at all**. Even the older CVCS will help you to keep track of your file changes, allowing you to focus on developing your software.

For 'getting started' resources, see http://sites.google.com/site/cs4217jan2011team7/

8 BIBLIOGRAPHY

A Visual Guide to Version Control. http://betterexplained.com/articles/a-visual-guide-to-version-control/ (accessed 24 January, 2011).

Amit Bahree, Dennis Mulder, Shawn Cicoria, Chris Peiris, Nishith Pathak. *Pro WCF: Practical Microsoft SOA Implementation [Paperback].* Apress, 2007.

Chappell, David. *Introducing Windows Communication Foundation in .NET Framework 4.* March, 2010. http://msdn.microsoft.com/library/ee958158.aspx (accessed 9 March, 2011).

Corporation, Oracle. *The Java EE 5 Tutorial.* http://download.oracle.com/javaee/5/tutorial/doc/bnazq.html (accessed 10 March, 2011).

Eichengreen, Barry. *One Economy, Ready or Not: Thomas Friedman's Jaunt Through Globalization.* May/June, 1999. http://www.foreignaffairs.com/articles/55017/barry-eichengreen/one-economy-ready-or-not-thomas-friedman-s-jaunt-through-globaliz (accessed 6 March, 2011).

Erl, Thomas. *SOA Principles of Service Design.* Prentice Hall, 2007.

Goncalves, Antonio. *Beginning Java(TM) EE 6 with GlassFish(TM) 3: From Novice to Professional.* Apress, Inc., 2009.

Hårsman, J. *Choosing to branch in Mercurial.* 13 October, 2010. http://ghostinthecode.posterous.com/choosing-how-to-branch-in-mercurial (accessed 25 March, 2011).

Hewitt, Eben. *Java SOA Cookbook.* O'Reilly Media, 2009.

Intro to Distributed Version Control (Illustrated). http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/ (accessed 24 January, 2011).

Jane Laudon, Kenneth Laudon. *Essentials of Management Information Systems.* Prentice Hall, 2007.

Judith Hurwitz, Robin Bloor, Carol Baroudi, Marcia Kaufman. *Service Oriented Architecture for Dummies.* Wiley Publishing, Inc., 2007.

Kalin, Martin. *Java Web Services: Up and Running.* O'Reilly Media, Inc., 2009.

Klein, Scott. *Professional WCF Programming: .NET Development with the Windows Communication Foundation.* John Wiley & Sons, 2007.

Losh, S. *A Guide to Branching in Mercurial.* 30 August, 2009. http://stevelosh.com/blog/2009/08/a-guide-to-branching-in-mercurial/ (accessed 18 March, 2011).