Introduction to Git

Graeme Geldenhuys

2009-02-20

If you are a developer paying attention to recent trends, it is hard to ignore the recent surge of interest in distributed version control systems (DVCS). Git is one such open source system and is designed to handle anything from small to large projects with speed and efficiency in mind. This article is an introduction to Git, which will show you some basic Git commands and tools so you can get started working in your own Git repository.

Concepts and History

Lets start by giving a brief summary of what a Version Control System (VCS) is. A VCS is a system that tracks changes to files. The files and its changes are stored in a repository. The repository in simple terms is a database of changes. You also have a working tree which is the files or source code living in the operating system's file system. It is these files that the developer edits and the repository tracks.

There are two types or version control systems: centralized version control systems and distributed version control systems.

Centralized version control system uses a Client/Server approach. There is a single repository stored on the server which holds the complete history of the files. You then have multiple clients accessing the repository on the server. The clients each have a working tree and a single revision of the files. The main problems with a centralized version control system is that it is a single point of failure and can be a performance bottleneck if many clients access the server simultaneously or you have a slow network connection.

A distributed version control system is decentralized and any client can be a server. Each repository holds the complete history of the files. Disconnected operations can also be handled even if you are not connected to a network, because the client holds the repository and working tree. There is no single point of failure. If one server goes down, other clients can simply pull changes from other servers. Backups are trivial, because everybody has a complete repository with full history. Micro or local commits are also allowed and doesn't affect other users by breaking an automated build system. So you

can commit your local changes at the end of a working day, even though those feature are not 100% complete yet.

Git[1] is an example of a Distributed Version Control System (DVCS) created by Linus Torvalds[2]. The Linux Kernel project started using a commercial DVCS created by BitMove and was called BitKeeper. This was back in 2002. On the 6 April 2005 there was some disagreements and licensing issues, so BitMover revoked the free licenses given to the Linux Kernel project. Linus decided on trying the Monotone project as a repository, but it was extremely slow. It is said that Linus started working on Git while he was waiting for Monotone to import the Linux kernel history. On the 18 April 2005 Git was already able to merge branches. By 16 June 2005 Git became the official version control system for the Linux Kernel project. Since then many more developers started working on Git and improving it.

In the beginning it was very hard to use Git and there was hardly any documentation. The Git developers started noticing all the complaints from new users about this issue. On the 15 February 2007 Git version 1.5 was released. This was a huge usability effort, and probably the first real version of Git than new and experienced users could use. Git was a lot more user friendly and had decent documentation added. The latest stable version of Git is currently at version 1.6.1.3.

General VCS operations

There are four major categories of command operations in any Version Control System. These categories are Bootstap, Modify, Information and Reference.

Bootstap:

This covers commands to create a repository or get you working on a repository. It should have similar commands as follows:

- *init*: Some way to initializes the history database or repository.
- *checkout*: Gets the files of a specific commit out of the repository. This then allows you to work with those files.
- *switching branches*: Allows you to switch between branches so you can work on different files or features in your project.

Modify:

This category is for commands that modify the staging area or repository.

- *add*, *delete* or *rename*: As these names suggest, they allow you to add files to a repository, delete files from a repository or rename files in a repository.
- *commit*: This adds any changes that were staged into a repository.

Information:

This category of commands are for querying information from a repository.

- *status*: There will be some command to retrieve the status of the working tree. It will show things like what files have been added, deleted or modified.

1 Git homepage: http://git-scm.com/
2 Linus Torvalds - http://en.wikipedia.org/wiki/Linus_Torvalds

- *diff*: View differences between commits and will normally give you a patch output.
- *log*: Query the historical changes in the repository

Reference:

These are commands that manages or creates tags and branches.

- *tag*: Marks a specific commit with a tag indicating a important state in the repository.
- *branch*: Create or delete branches giving alternative paths of development.

So these are all general features you will find in most version control systems. In this article I will cover all these features, and show you how to execute them using the Git version control system.

Installation

Currently I am using Ubuntu Linux 7.10 on my work PC which includes Git version 1.5.2.5 in the *apt* repositories. I want to demonstrate newer features of Git, as some of those features were only introduced in laters versions of Git. So rather that use the Git release from the apt repository, I chose to install the latest stable Git from source code. So for this article I am using version 1.6.1.3 which is the latest stable release.

First we need to download the latest source tarball, unpack it and configure it for compilation. Please note that in this article I prefix an actual command as typed into a terminal with a $ sign prefix. So, to get the source and configure it we do the following in a terminal window:

```
$ cd /tmp
$ wget
http://kernel.org/pub/software/scm/git/git-1.6.1.3.tar.bz2
$ tar xvfj ./git-1.6.1.3.tar.bz2
$ cd git-1.6.1.3/
$ ./configure
```

Now to compile Git you can execute the following command.

```
$ make all
```

This will build the Git executables, but exclude the building of the documentation. The documentation requires a few external tools like *asciidoc* and *xmlto* which are not always available on all distros. If you do have the required dependencies, you can build the documentation by executing the following command:

```
$ make doc
```

You don't always need to install Git to be able to run it. You can run it directly from the directory you compiled it in. Git requires various paths to be able to find it's own tools and scripts. The following environment variables need to be setup for this to work.

```
GIT_EXEC_PATH=`pwd`
PATH=`pwd`:$PATH
GITPERLLIB=`pwd`/perl/blib/lib
export GIT_EXEC_PATH PATH GITPERLLIB
```

Alternatively, if you want to install the newly compiled version of Git you need to execute the following as root user.

```
$ make install install-doc install-html
```

The default installation is to `/usr/local/` directory. For more information on the installation options and instructions, see the `INSTALL` file included in the Git tarball.

Now to confirm that the installation went well we can change to a different directory - for example the users $HOME directory and execute the following command to see if we have the correct version.

```
$ git --version
git version 1.6.1.3
```

Overview of Git

If you are currently using other source control systems like SubVersion or CVS, then Git needs some further explanation simply because it is slightly different to other systems and what you will be used to. The follows is a quick overview of the terminology used when working with Git and to help you understand the information later on in this article.

In Figure 1 we show a history graph of a sample repository. Each square block represents a commit with a revision number assigned to them. We use the letters *A* through *G* for simplicity and to show the order in which they occurred, when in actual fact Git generates a unique 40-hexdigit SHA1 hash for each commit. No two commits in a repository will ever have the same SHA1 value. The arrows represent the relationship with another commit. They point to one or more parent commits. The three trapezoids pointing to commits D, E and G represent branches. The default development path is normally called `master`. Any alternative paths of development are called *branches*. In this case we have three branches called `topic_a`, `testing` and `release`.

The elongated pentagons with the text `v1.0` and `v1.2` represent *Tags* in the repository. A tag is like a read-only branch and indicates an important state in the repository - like when you created a release for your software. Branches differ from Tags in the sense that Branches follow the path of development. If we were working in the `topic_a` branch and added a new commit H, then the `topic_a` branch will advance and point to H and not G. On the other hand, a tag gets assigned to a specific commit and will forever point to that exact commit.

In Git there is a concept of a *HEAD*. A HEAD is a pointer to a Branch (most of the time) and represents the current commit checked out in your working tree. If you look at Figure 1 again, you will see we have `HEAD` pointing to `topic_a`, which means we have commit G checked out in our working tree.

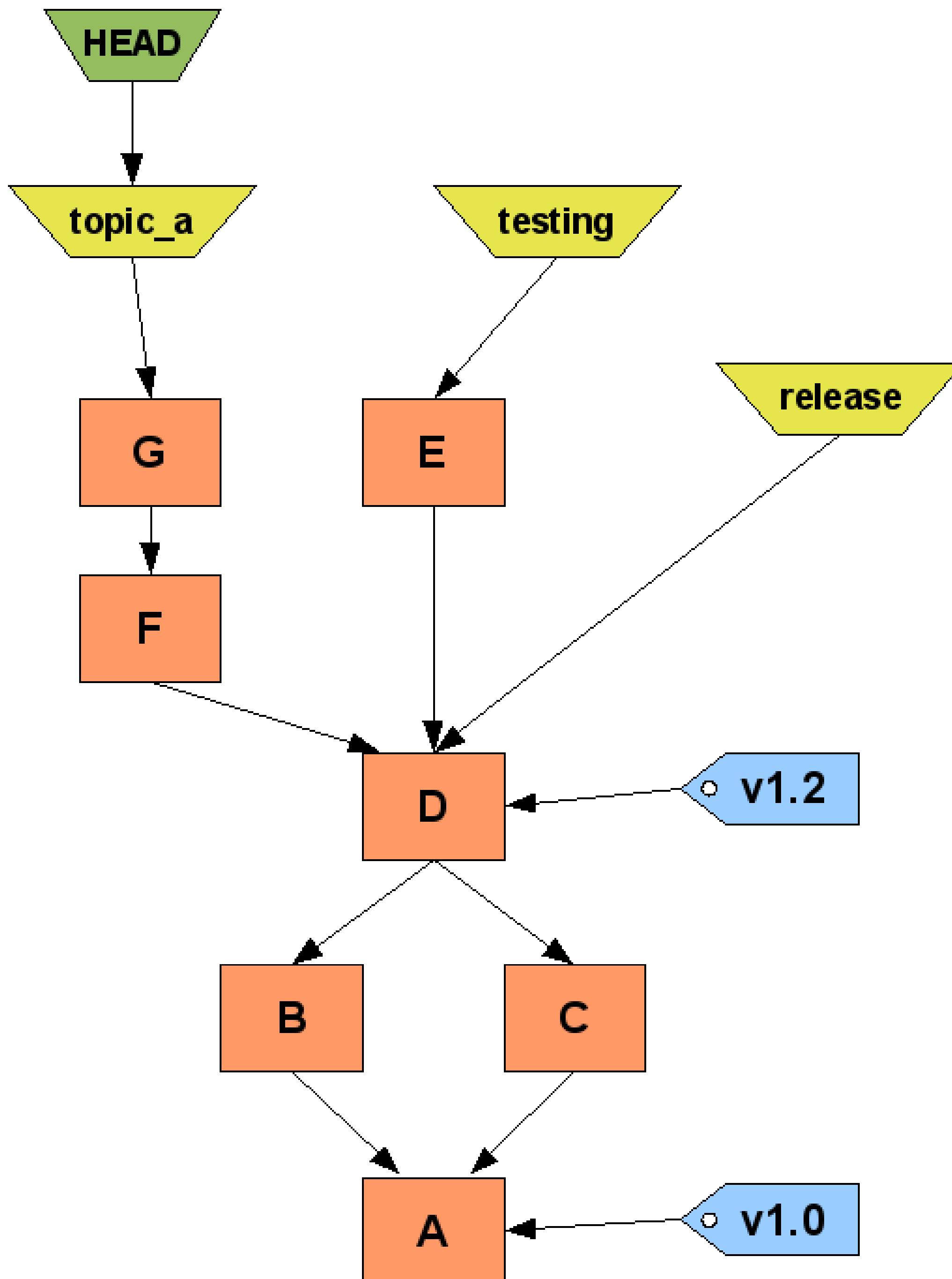

Figure 1: Repository history and data

Figure 2 represents the movement of data in a Git repository, including the related Git commands to accomplish that. The following section will describe the image in more detail.

First we have the *Repository* where the history or metadata of the files your are tracking are stored. Think of a repository as a database of changes. This is also where some configuration information for Git is stored. Next we have the *Index*, which is a staging area and contains information about what you want to commit in your next commit. Then finally we have a *Working Tree* which is the actual files and directories on disk that you can see. These are the files you will create and edit.

The process of moving the changes from the Working Tree to the Index is called *staging*. You will use commands like `add`, `remove`, `rename` etc. to accomplish this task. The next part is moving those staged changes from the Index into the Repository. This is called *committing* and you will use a command called `commit` to do this.

So far these were all commands to get information into the repository. Now we also have the reverse of that process, getting information out of the repository. You will use commands like `checkout`, `read-tree` and `reset` to update the *Index*. We also have commands that update the *Working Tree*, and use commands like `checkout`, `checkout-index` and `reset`.

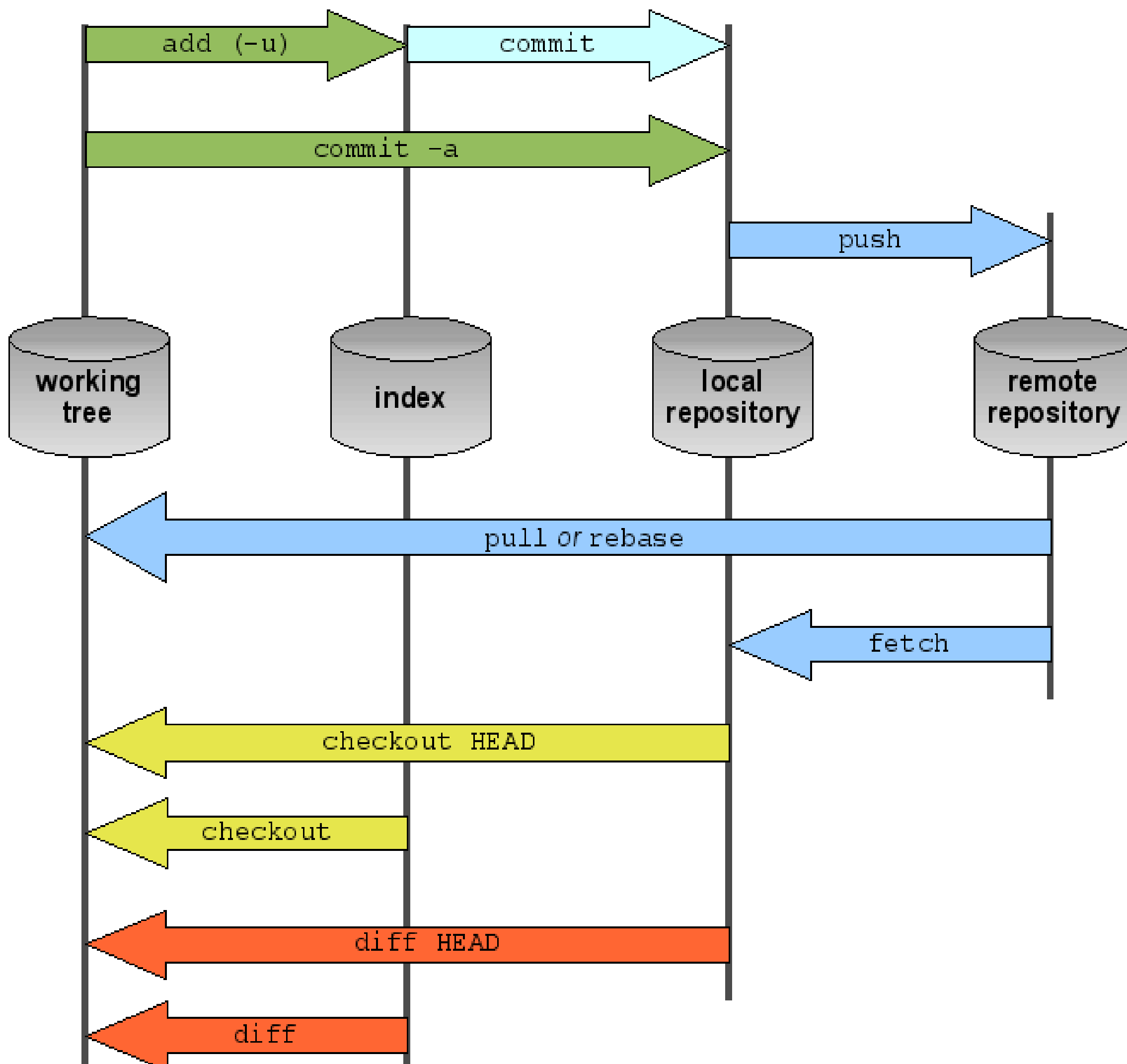

Figure 2: Git data transport

Another difference compared to SubVersion, is that unlike SubVersion which has a *.svn* directory in the top level directory and every sub-directory of your project, Git has only a single *.git* directory in the top level or root directory of your project. This directory contains the history, metadata and various git configuration files.

Most of the files inside the *.git* directory are not very important to us, but there are a few exceptions I will quickly explain.

```
.git/config
```

This is a per repository configuration file where you can set certain parameters and we will look at a few of those later.

```
.git/description
```

This file describes the project in whatever language - English, German etc... I believe this file only gets used by *gitweb* which is a program that allows you to view the git repositories via a web browser.

```
.git/info/exclude
```

This is a file you can edit and insert patterns of files you want Git to ignore, so Git does not put them in the repository. For example, if you are a Free Pascal developer, you would want Git to ignore *.ppu and *.o files.

Git Commands

All Git commands have the following consistent syntax:

```
$ git <global options> <command> <command options>
```

There are currently over 137 Git commands. This is quite overwhelming for users coming from systems like CVS or SubVersion. Luckily Git now has very good documentation for all those commands, but for most day-to-day work the majority of users will only use around 20 of those commands. To see a list of those 20 or so common commands you can execute the following.

```
$ git help
```

All Git commands will also answer to the `-h` parameter giving you a brief help output or syntax for that command. For more detailed help on commands you can access the manual pages. Since version 1.5 the help is really good. They give a lot of information and examples on how to use any of the available commands. You can access the manual pages in any of the following ways.

```
$ man git-<command>
$ git help <command>
$ git <command> --help
```

Earlier I mentioned the *.git/config* configuration file. This configuration file applies to a specific Git repository. Sometimes you would also like global configuration options which will be applied to all your Git repositories. The file used for storing such information can be found in the *$HOME/.gitconfig* file.

You can edit the information in that file via a text editor or via the `git config` command. For example, the minimum requirement for using Git, is to tell Git who you are.

Your name and email address. The reason Git needs that information is because that is what Git inserts into the commit history. You can set that information as follows:

```
$ git config --global user.name "Your Name"
$ git config --global user.email somename@somedomain.com
```

There are a few other options you can also set. One very handy option is to tell Git to use colour when it generates output.

```
$ git config --global color.ui auto
```

Another very nice option is to tell Git to use the pager `less`, but only if the output is more that what will fit into your screen. So if you have a 25 row terminal window and Git generates 24 rows of output, the pager will not be used. Any output over 25 lines, and Git will use the pager so you can scroll the output. Another handy option is to tell Git what editor to use when you need to write commit messages.

```
$ git config --global core.pager "less -FRSX"
$ git config --global core.editor mcedit
```

So now that we have the basic configuration out of the way, it is time we start working in a actual Git repository. So lets start by creating a Git repository. First we need to change into our project directory and execute the following command in the top level directory of that project

```
$ git init
```

This will initialize a Git repository and create the *.git* directory. At this point Git doesn't know about any of our project files. So for our first commit we need to stage some files which will be included in our first commit.

```
$ git add <filename>
$ git add .
```

The first example specifies one or more files to be added to the staging area also known as *Index*. Instead of specifying a specific file you can also specify wildcards like *.c to add all C source code files. The second method shown is to specify a . (dot) instead of a file name. This will tell Git to recursively add all files in our project directory that are not ignored by the exclude patterns setup in the *.git/info/exclude* file. As a matter of interest, there is another way to specify exclude patterns and that is via the *.gitignore* file which is in the top level directory of your project. The difference between the two are quite simple. All files in the *.git* directory are not tracked by Git, but the *.gitignore* file located in the top level directory can be tracked. If you wanted everybody working on the project to ignore the same set of files, it is recommended that you add those exclude patterns to the *.gitignore* file and tell Git to add that file to the repository. Any personal preferences you might have can be added to the *.git/info/exclude* file and will not affect anybody else's exclude patterns.

It is important to note that the *add* command has two uses. Firstly it can tell Git to add files into the repository that was not tracked before. The second usage is when you have

modified a few files, but you only want to commit a specific file's changes in the next commit.

To remove files from the repository you execute the following

```
$ git rm <filename>
```

You can even tell Git to rename files while it continues tracking its changes.

```
$ git mv <old filename> <new filename>
```

Now we can finally commit our staged changes by running the following:

```
$ git commit -m "some commit message"
```

We can also specify the `-a` parameter which will tell Git to commit all files that it tracked before and that contains changes, even though those files have not been explicitly staged first. This will not commit newly created files that Git was not tracking yet or files you deleted without telling Git about the deleted files. This is similar in a way to what SubVersion's `svn ci` command does.

If we want to inspect the repository we can use the `git status` command. This will show us all the staged, unstaged and untracked files. Figure 3 show some sample output of the *status* command.

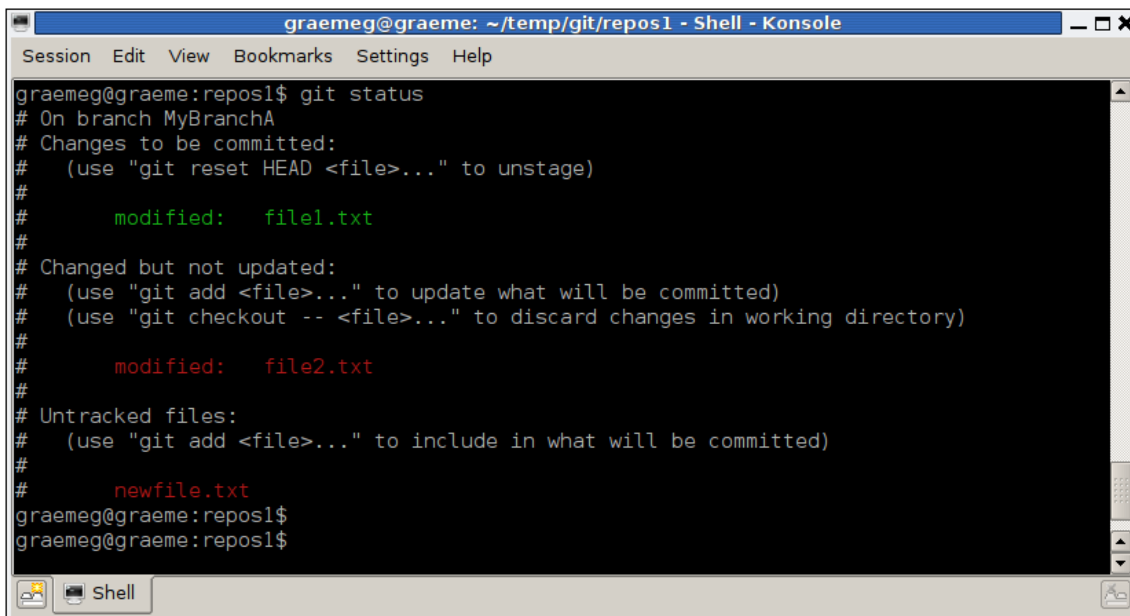

Figure 3: Sample 'git status' output.

You can see in Figure 3 that *file1.txt* is a tracked file and has been staged for the next commit. The second item is *file2.txt* which is also a tracked file, but has not been staged for the next commit. The last file in the screenshot, *newfile.txt*, is a newly created file that is not tracked by Git yet and so is not staged either. As you can see from the output, Git gives clear and user friendly messages to help the user by even giving sample com-

mands to use. If we had to do a commit using `git commit`, then only the one file, *file1.txt*'s changes would be committed. But if we executed `git commit -a`, then all changes to files *file1.txt* and *file2.txt* would be committed. Because *newfile.txt* is not tracked by Git yet, none of its changes would be added to the repository.

Another way of inspecting the repository is by using the *diff* command. The following two examples show the usage of the *diff* command and the differences in output when using different command options.

In the first example we used the `git diff` command by itself.

```
$ git diff
diff --git a/file2.txt b/file2.txt
index a7e2414..8a01ab9 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1 +1,2 @@
 The second file
+With some new changes.
\ No newline at end of file
```

This shows us the changes between the *Index* and the *Working Tree*. So in this case it shows us the changes in *file2.txt* which is a tracked file but its changes have not been staged yet. Remember, we are working with the files shown in Figure 3.

In the second example we used the `git diff --staged` command. This shows us the changes between *HEAD* and *Index*.

```
$ git diff --staged
diff --git a/file1.txt b/file1.txt
index e377738..6218780 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,4 +2,4 @@ Hello World!
 ------------

 Now what?
-
+Some new changes...
```

In this case Git shows us the changes in *file1.txt* which is a tracked file and its changes have already been staged. We can also execute `git diff HEAD` which will then show us all changes between *HEAD* and *Working Tree*. This will be a combination of the previous two commands. Another way of using the *diff* command is to ask for the changes between two commits. The manual pages will give you a lot more information and examples on this topic.

You will also have noticed that the output generated by the *diff* command is by default in the unified diff format which is a standard patch style format. Lines preceded by a minus sign are deleted lines. Lines preceded by a plus sign are changed or added lines. Git can also generate other diff formats for you.

I briefly showed some *diff* commands, but when you start using *diff* more often, you will need to start referencing various objects in the Git repository. For example, the difference between two commits or changes between two branches etc.. There are many ways

you can reference objects in Git. This is quite a powerful feature of Git and the syntax is very flexible. The following table lists some reference examples you can use.

What to reference?	Text used
full commit id (SHA1 hash) of a specific commit	87afbbf488d4397a2a6b294abe66f627bae86ee1
partial or short hash as long as in is unique	87afbbf
local branch by name	master
remote branch	origin/master
tag	v1.5
by a commit message. Git will search for "some text" and use the first commit id with that text.	":/some text"
current checkout	HEAD
commit before HEAD	HEAD^ or HEAD~1
4 commits before HEAD	HEAD^^^^ or HEAD~4
the HEAD of yesterday	HEAD@{yesterday}
HEAD at 1st February	HEAD@{1.February} or HEAD@{February.1}
yesterday on another branch	MyBranchA@{yesterday}
on master a few changes ago	master@{3}

Table 1: Various syntax to reference a commit object

Another way of looking at changes is to use the *show* command. If you run `git show` by itself, it will show you the changes of the last commit.

```
$ git show
commit 3e05aa2a9e897338c7821d61ef9065a320504a41
Author: Graeme Geldenhuys <graeme@mastermaths.co.za>
Date:   Tue Feb 24 13:13:04 2009 +0200

    Some minor changes

diff --git a/file1.txt b/file1.txt
index 3b18e51..e377738 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,5 @@
-hello world
+Hello World!
+------------
+
+Now what?
+
```

There is also a short form to show less information. With our small example above, it does not make much sense because we only modified one file in that commit and had a small amount of changes. Now if the previous commit affected a lot of files with many changes, the `git show` command output would have spanned many lines and the pager would have been required to view everything. Executing the `git show --stat` command shows only the summary information of a commit.

```
$ git show --stat
commit 3e05aa2a9e897338c7821d61ef9065a320504a41
Author: Graeme Geldenhuys <graeme@mastermaths.co.za>
Date:   Tue Feb 24 13:13:04 2009 +0200

    Some minor changes

 file1.txt |    6 +++++-
 1 files changed, 5 insertions(+), 1 deletions(-)
```

Here you can see that instead of showing you the patch information, it only tells you which files changed, the amount of changes per file and the number of insertions and number of deletions. This is a nice way of getting an overview of what a specific commit has done.

The *show* command also has a `--name-status` command option which generates output very similar to what SubVersion's svn client does.

```
$ git show --name-status
commit 1b71b73a33cdc432d5ed5dcf06e13e2936a9c9b4
Author: Graeme Geldenhuys <graemeg@gmail.com>
Date:   Fri Feb 27 23:06:37 2009 +0200

    various other changes.

M       file1.txt
D       file2.txt
A       newfile.txt
```

Here M is for modified, D is for deleted and A is for added. As shown below, the *show* command can also reference any other commit in the repository using the same object references as listed in Table 1 a bit earlier.

```
$ git show HEAD@{4}
```

Something you will do quite often is looking at the history log of the repository. Tracking changes of a file, or the last set amount of changes in the repository etc. Executing `git log` on its own outputs the commit log of the current branch, from newest to oldest. As the following default output shows, you get the commit id, author of the patch, date of the commit and the commit message.

```
$ git log
commit 1b71b73a33cdc432d5ed5dcf06e13e2936a9c9b4
Author: Graeme Geldenhuys <graemeg@gmail.com>
Date:   Fri Feb 27 23:06:37 2009 +0200

    various other changes.

commit 3e05aa2a9e897338c7821d61ef9065a320504a41
Author: Graeme Geldenhuys <graeme@mastermaths.co.za>
Date:   Tue Feb 24 13:13:04 2009 +0200

    Some minor changes

commit 3a297c8e8829d9f1702820e33d6691a4c8a3f4aa
Author: Graeme Geldenhuys <graeme@mastermaths.co.za>
Date:   Tue Feb 24 13:10:24 2009 +0200

    The second file
 :
```

As with the *diff* command, the *log* command can also handle various object references to filter the output more precisely. Table 1 lists various methods of references commit objects. Here are a few more examples using the log command.

```
$ git log <tag name>..<branch name>
```

Here we specified a range, which means Git will show all log entries from the one commit to the other commit. In this case, we specified a tag name and a branch name.

```
$ git log HEAD~5..
```

Here we left out the second part of the commit range which means Git defaults to the HEAD of the currently checked out branch. In the above example, we specified a range, but did not explicitly mention the ending range. Git will then default to HEAD. So this command will show all log entries from five commits back to the newest HEAD commit.

```
$ git log -5
```

This line will show the five most recent commits, from newest to oldest. The -5 limits the log to only five commits, but any number can be specified.

```
$ git log --since="Jan 1" --until="Feb 1"
```

Here it shows that we can even use a date range to list all the log entries from one date to another.

We are not limited to only displaying a linear commit log. We can also use a format to filter out certain commit logs.

```
$ git log --author=Graeme
```

This will run through the logs and only display log entries where the author's name

starts with "Graeme". Another way to filter the commit log is by grep'ing through the commit message.

```
$ git log --grep="*sometext"
```

We can also grep through the actual changes, not just the commit message. So if we know somebody made a change to a specific piece of code we can specify a section of that code and Git will output all commit logs where changes were made containing that code. You can also use the built-in regular expression engine in Git to help find the commits you are looking for - this is called *pickaxe*.

```
$ git log -S"some code change"
$ git log --pickaxe-regex -S"some.*code.*change"
```

This is one of the truly amazing features in Git. It really speeds up the process of finding specific commits out of thousands.

Obviously Git also supports a per file option. So you can list all commits that only relate to a specific file.

```
$ git log -- some/path/to/file
```

Just to give you a quick example of a more advanced log command. The following command generates a text based revision history graph based on all branches. This is one way to visualize the revision tree. As you can see in the command below, we asked Git to make the dates relative to the current date and we specified a custom formatting of the output. We even specified what colours to use in the terminal. The last thing we told Git to do, was to abbreviate the SHA1 id's so it only shows the first 7 characters of the full 40 character SHA1.

```
$ git log --all --graph --date=relative \
--pretty=format:"%Cred%h%Creset – %s %Cgreen(%cr)%Creset" \
--abbrev-commit
```

And here is some sample output of this command.

```
* 02346aa – A new file from a cloned repository. (22 minutes ago)
| * 87afbbf – New feature added (3 hours ago)
|/
| * 3e05aa2 – Some minor changes (3 hours ago)
| * 3a297c8 – The second file (3 hours ago)
|/
* e65ed56 – Added my first file to Git (4 hours ago)
```

The newest commit is at the top. It also shows us that we have 2 branches which divert from the *master* branch. For more details on this, you can reference the manual pages which explain all the options in more detail.

Speaking of branches... Branching and Merging are very important topics with Git. Unlike other source control systems like CVS or SubVersion where branching is a costly process and merging is very complicated, Git made huge improvements to make this as

effortless as possible. Under Git, branching is very lightweight to create and destroy. It is recommended to make branches for every feature you are working on and once that feature is complete and merged back into the master branch, you simply delete that feature branch.

The reason they spent so much time on improving branching and merging is because it happens so frequent in a distributed development environment. You could have pulled code from one repository. At the same time somebody else also pulled those changes, then that somebody else implemented a new feature and pushed it back to the original repository. After you completed your feature, you need to first pull all the new changes to your repository, which will now cause a merging of changes, then only can your changes be pushed to the original repository. Branching and Merging become common tasks when working under Git.

So how to we create a branch? Simply by running the following command.

```
$ git branch <branch name> <optional commit>
```

<branch name> is the name you want to give to the new branch. If you do not specify a specific commit, then the branch will be based on your current checked-out HEAD. The next logical thing would be to switch branches. Again, this is very easy to do, by simply telling git to checkout whatever branch you specified.

```
$ git checkout <branch name>
```

You can also do both creating and switching to that new branch in one command as follows.

```
$ git checkout -b <branch name> <optional commit>
```

Now lets say we created our new *MyNewFeatureA* branch and implemented this new feature. What we now want to do is merge that new feature back into the stable *master* branch. The way we do that is by calling the `git merge` command. The workflow would be as follows.

```
$git checkout -b MyNewFeatureA master
[...code...commit...code...commit...code...commit]
$git checkout master
$git merge MyNewFeatureA
```

The merge command also allows you to merge more that one branch at the same time and is called an *octopus* merge. That is a more advance topic I will not be covering here, but I just wanted to highlight that it is a possibility.

As with any other version control system, when you do merging you stand a chance of getting conflicts. Git will mark the conflicting lines in the specific files in a very similar way to what SubVersion does. Resolving such conflicts are a manual process as with all other version control systems. Again, the manual pages cover this in a lot more depth if you want to read up on it.

The last command I will cover in this article is cloning. This is probably also the first command you will be using when working with Git and other remote projects. The syn-

tax for cloning is as follows:

```
$ git clone <remote repos> <new directory>
```

What cloning a repository does is as follows: It first initializes a new empty repository on your local system. It will then replicate the remote repository and populate your local repository with all the information from the remote repository. By default, it will then checkout the *master* branch in your new local repository.

When cloning a repository you need to specify the repository's URL. The following table summarizes some of the Git URL's.

Protocol	Example
Local repositories	file:///home/graemeg/git/project.git/
	/home/graemeg/git/project.git/
HTTP protocol	http://git.mydomain.com/project.git/
Native Git protocol	git://git.mydomain.com/project.git/
SSH protocol	ssh://graemeg@mydomain.com/~git/project.git/
	graemeg@mydomain.com/~git/project.git/

Table 2: Examples of Git URLs

Local repositories are handy for experimental work on your own projects. As far as I know, using the HTTP is only read-only access - you cannot push changes via the HTTP protocol. The HTTP protocal is handy to publish some public projects though. The native Git protocol is a lot more optimized for network traffic and is the recommended protocol to use. Git via the SSH protocol is also popular due to obvious security features.

GUI tools

By default Git comes with two GUI tools, but there are probably more somewhere on the internet. These two tools fairly popular with Git users, but they are not a requirement to using Git. I am also fairly sure that as time goes by, more and more GUI tools will emerge. The first tool I am going to talk about is called *gitk,* which is a way to visualize the revision tree. You can start the program by running `gitk` from inside your repository.

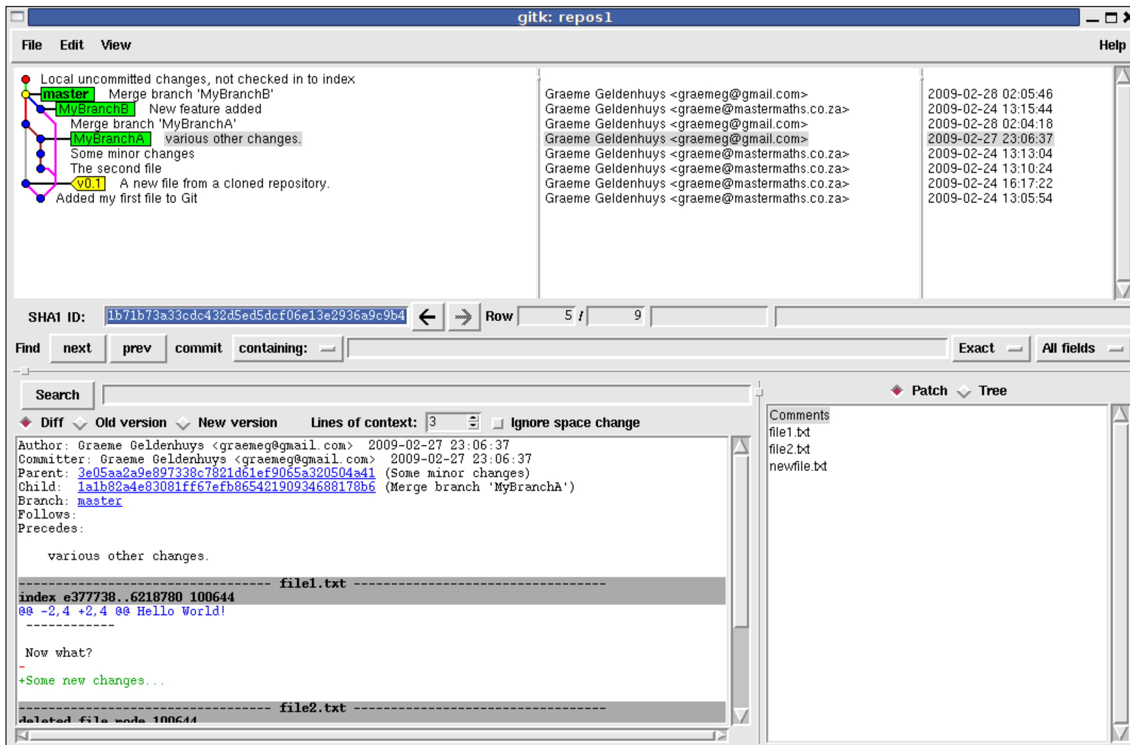

Figure 4: gitk showing our revision tree.

As you can see from the screenshot in Figure 4, we have the revision tree in the top left corner. The dots in the tree represent different states. The yellow dot is the currently checked out HEAD. The blue dots are previously committed commits. The red dot represents local changes that have not been staged yet. The green blocks represent the various branches with their names. The green box with the bold writing is the currently checked out branch which HEAD points to. The lines between the dots represent the parent commits. The yellow box at the bottom of the tree represents a *Tag* with the tag name - in this case it's named `v0.1`. To the right of the tree is the name of the commit author and the time stamp that that commit was made.

The bottom half of the window represents the commit being highlighted in the tree. On the left side we can see the commit details (author, committer, date, parent commit etc) and the commit message. Below that we see all the changes that were made in that commit to the various files.

In the View menu you can customize the view to only display a set amount of commits, or filter on a author etc... All the same things we covered earlier in this article, but this time from a GUI tool.

The second GUI tool is called *Git Gui*, which allows you to perform trivial tasks from a GUI. For example you can stage or unstage changes. Add or remove files, make commits etc. This is a handy tool for people that are more comfortable with GUI tools than the command line. You can start this tool by running `git gui` from inside your Git repository.

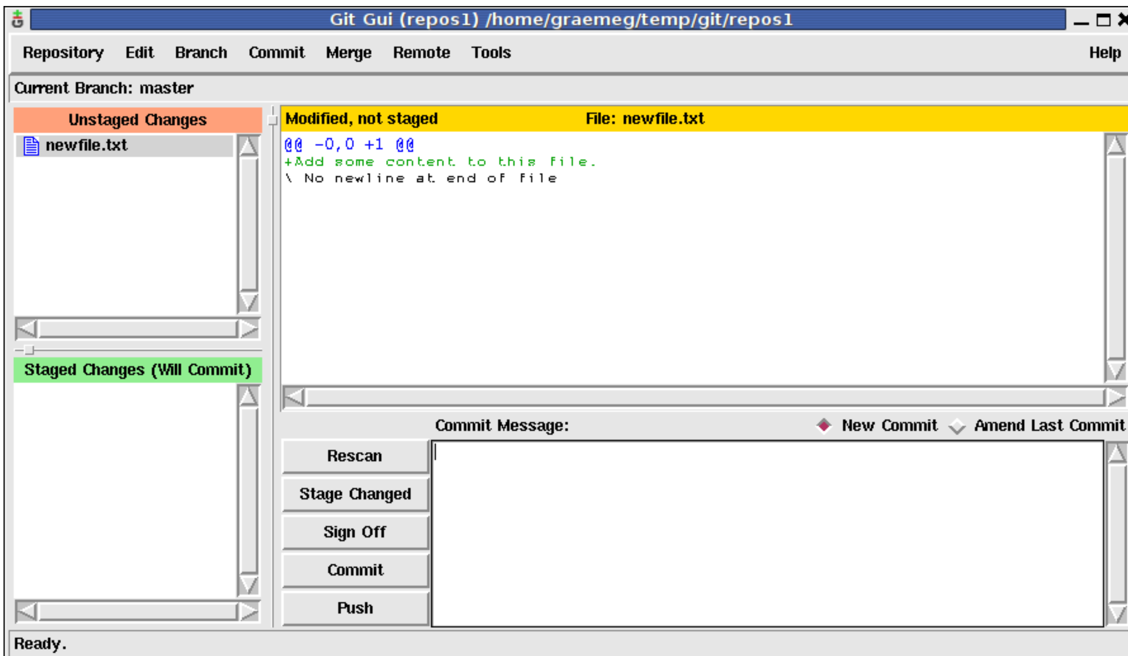

Figure 5: Git Gui showing one unstaged change.

Just below the menu bar it tells us in which branch we are. In this case we are in the *master* branch. At the left top in the window it shows us all the unstaged files in our *Working Tree*. Here it shows us that changes have been made to the tracked file called newfile.txt, but we haven't staged this change yet, so it will not be part of the next commit.

Below that window is the same thing, but this time it shows us all the staged changes. Changes that are in our *Index* and that will be in the next commit.

The top right window shows us the changes based on the selected file in the Unstaged or Staged windows. In this case it shows us the changes of the highlighted unstaged file called newfile.txt.

The bottom right windows is where you can type in your commit message and do some basic actions like Commit, Push etc...

One very nice feature of Git Gui is how easy it makes partial staged commits. For example, say you have a file with Object Pascal source code. Now lets say three changes have been made in different locations in that source file. One of those changes were just a temporary change with debug information and you did not actually want to commit that change. Git does not force you to stage and commit that whole file. For example, you can stage the whole file with all its changes, then use Git Gui to show you all the changes in the top right window. Then select the change you did not mean to commit. Right click and select "Unstage Hunk", and it will unstage only that change. So when you do a commit, only two out of the three changes in that file will be committed.

Final thoughts

I hope that you have learned something about the differences between client-server and distributed version control systems. I also hope that this article has made you more curious about Git and that it has convinced you to try it out for yourself.

This article covered only the basics of what Git has to offer, but it should have been enough to get you going with your own Git repository. There are many more options to all the commands I have shown you, and I urge you to read the manual pages to get to know the rest of the options and commands better. The Git website also has a lot of extra documentation and tutorials you can follow. I hope you found this introduction useful and enjoy learning and using Git as much as I did.