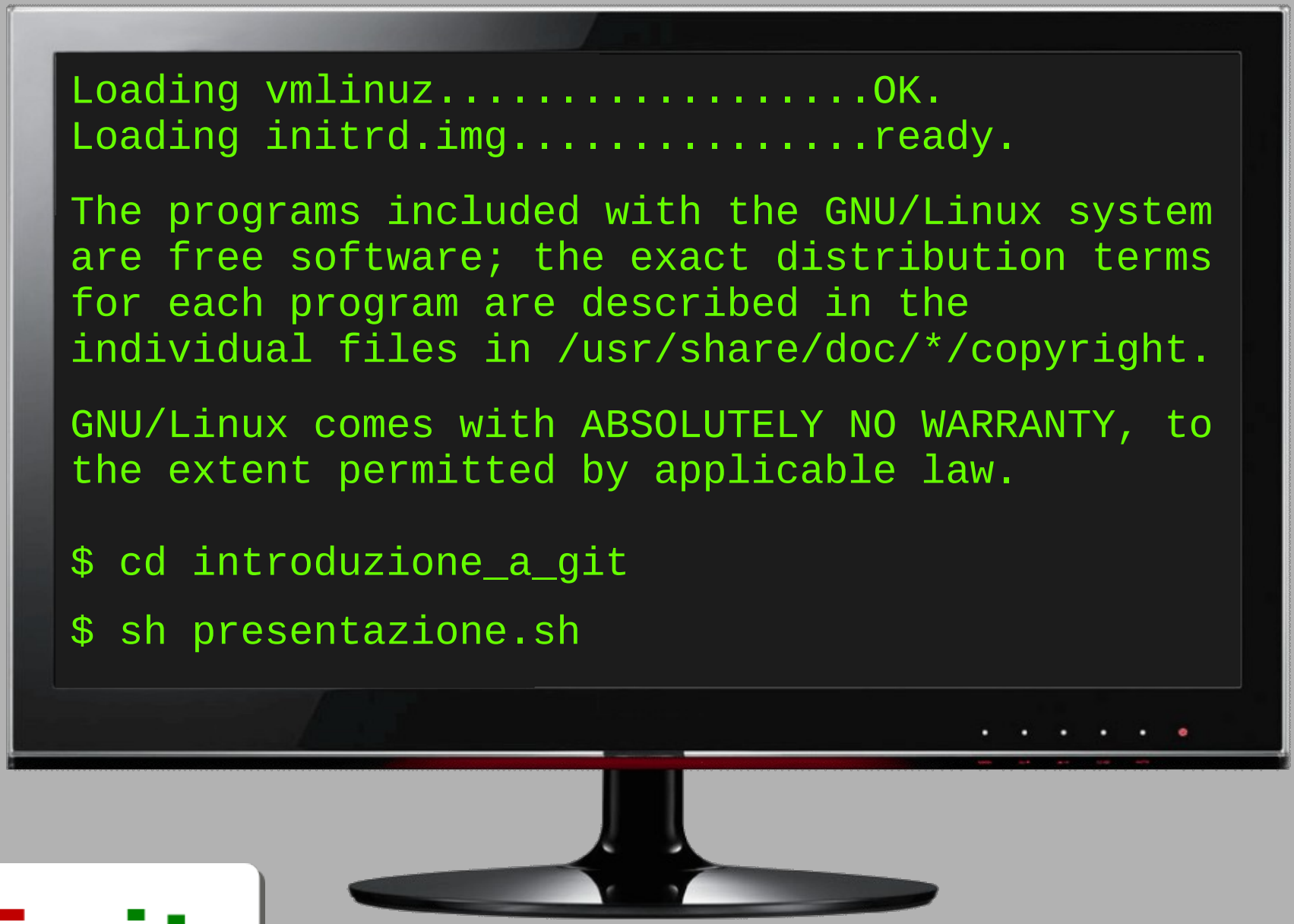
Introduzione a Git



Riccardo Macoratti <kimahri_san@zoho.com>

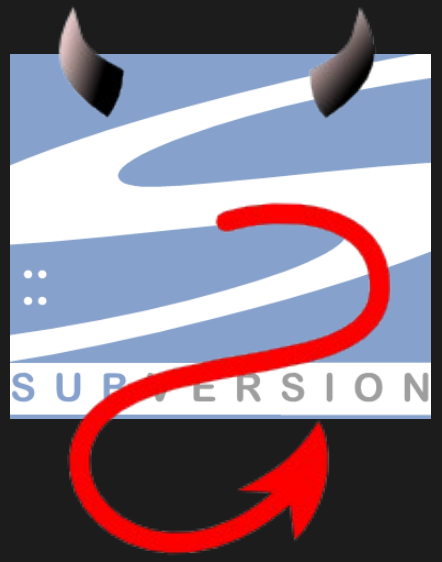
Che cos'è?

Git è un sistema software di controllo versione distribuito sviluppato da Linus Torvalds nel 2005.

Esistono altre applicazioni simili, anche open source:

Bazaar

Mercurial

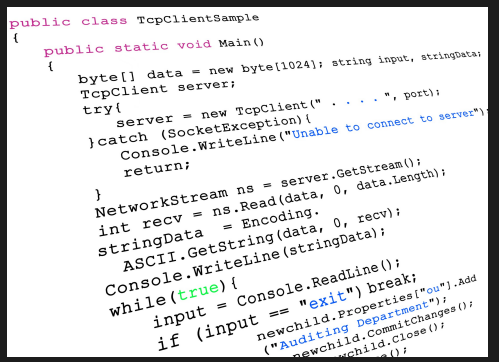


Subversion

Prima di Git, veniva usato un sistema di controllo versione centralizzato, *SVN* o *Subversion*. È stato preso come modello di cosa non fare nello sviluppo di Git.

Importanza di un CVS

Un software di controllo versione è importante per lo sviluppo e l'organizzazione del codice.
Permette una maggiore tracciabilità delle modifiche e dello stato di avanzamento del lavoro.



```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient(" . . . . ", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int recv = ns.Read(data, 0, data.Length);
        stringData  = Encoding.
            ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
                                    newchild.Properties("ou").Add
                                    ("Auditing Department");
                                    newchild.Commitchanges();
                                    newchild.Close();
```

Questa si propone come una guida pratica, per imparare in pochi minuti l'uso di Git.

Come iniziare

Git su può usare in due modi:

- In locale, all'interno di una directory del nostro sistema, con un'eventuale server HTTP tramite **gitweb** e/o **gitolite**;
- Affidandoci a un servizio esterno come **GitHub** oppure **BitBucket**, che ci mette a disposizione uno spazio online con un'interfaccia user friendly per poter creare e gestire i repository.

GitHub

BitBucket

Installazione

Su Debian/*buntu:

```
apt-get install git-core
```

Su Arch Linux:

```
pacman -S git
```

Su Fedora/CentOS/RHEL:

```
yum install git
```

Su Gentoo:

```
emerge --ask dev-vcs/git
```

Configurazione

Prima di poter inizare a usare Git è consigliato effettuare una semplice configurazione del nostro nome utente, per non effettuare modifiche anonime, e del nostro indirizzo email, per essere identificabili e contattabili immediatamente.

```
$ git config --global user.name “Kimahri_San”
$ git congig --global user.email “kimahri_san@zoho.com”
```

Progetto

Per illustrare l'uso di Git, costruiremo un semplice *hello world* in linguaggio C.

Successivamente lo implementeremo con altre funzionalità per simulare una sessione di lavoro realistica e mostrare i comandi utili per essere produttivi con questo tool.

Creazione del repo

Iniziamo col creare il repository che conterrà tutto il nostro codice e la *history* associata.

```
$ mkdir /home/$USER/repos
$ cd /home/$USER/repos/
$ mkdir hello_world
$ cd hello_world/
$ git init
Initialized empty Git repository in
/home/[...]/repos/hello_world/.git/
```

Aggiungiamo dei file

Ora aggiungiamo qualche file all'interno del repo.

```
$ touch hello_world.c
$ echo -e "#include <stdio.h>" >> hello_world.c
$ echo -e "int main(void) {" >> hello_world.c
$ echo -e "\tputs(\"Hello, world\!\");" >> hello_world.c
$ echo -e "\treturn 0;\n}" >> hello_world.c
$ cat hello_world.c
#include <stdio.h>
int main(void) {
   puts("Hello, world!");
   return 0;
}
```

Cos'è successo?

Per controllare cosa Git sa del vostro operato nel repo possiamo usare il comando **status**:

```
$ git status
On branch master

Initial commit

Untracked files:
   (use "git add <file>..." to include in what will be
   committed)

      hello_world.c

nothing added to commit but untracked files present
(use "git add" to track)
```

Staging area

Come sì è visto, Git sa che qualcosa è cambiato, ma le modifiche non sono ancora state tracciate. Per aggiungere il file alla **staging area** si esegue l'operazione di **add**.

```
$ git add hello_world.c
$ git status
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

      new file:   hello_world.c
```

Commit

Siamo pronti per “confermare” a Git i file creati e le modifiche effettuate, tramite l'operazione di **commit**.

```
$ git commit -m “Creato hello_world.c iniziale.”
[master (root-commit) f2251eb] Creato hello_world.c
iniziale.
   1 file changed, 9 insertions(+)
   create mode 100644 hello_world.c
$ git status
On branch master
nothing to commit, working directory clean
```

Lista dei commit

È sempre utile poter vedere una lista di tutti i commit effettuati all'interno di un repository.

```
$ git log
commit f2251eb719c4c1e40586a4b3e4459f70f5568ec9
Author: Riccardo Macoratti <kimahri_san@zoho.com>
Date:   Fri Jan 16 15:34:28 2015 +0100

    Creato hello_world.c iniziale.
```

Modifiche ai file

Proviamo a compilare il programma e modificare il file e committare i cambiamenti.

```
$ cc hello_world.c
$ ./a.out
Hello, world!
$ vi hello_world.c
$ cat hello_world.c
#include <stdio.h>
int main(void) {
   char name[256];
   printf("Inserisci il tuo nome: ");
   gets(name);
   printf("Hello, %s, good evening!\n", name);
   return 0;
}
```

Modifiche ai file (2)


```
$ git add .
$ git commit -m “Reso interattivo con stampa nome.”
[master 038229e] Reso interattivo con stampa nome
   2 files changed, 4 insertions(+), 5 deletions(-)
   create mode 100755 a.out
```

File indesiderati


```
$ ls
a.out  hello_world.c
```

Possiamo vedere che abbiamo aggiunto anche *a.out* al nostro repository. Come fare per eliminarlo?

```
$ git rm a.out
rm 'a.out'
$ git status
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

   deleted:    a.out
```

File indesiderati (2)


```
$ git commit -m “Rimosso residuo di compilazione.”
[master abff3d3] Rimosso residuo di compilazione.
   1 file changed, 0 insertions(+), 0 deletions(-)
   delete mode 100755 a.out
```

Per prevenire l'aggiunta di alcuni tipi di file bisogna creare il file **.gitignore**.

```
$ echo “*.out” > .gitignore # Ignora estensione .out
$ git add .
$ git commit -m “Aggiunto .gitignore per file *.out.”
[master a683c99] Aggiunto .gitignore per file *.out.
   1 file changed, 1 insertion(+)
   create mode 100644 .gitignore
```

Il commit

È possibile modificare il commento di un commit.

```
$ git commit --amend -m “Creato .gitignore per *.out.”
[master 43a0929] Creato .gitignore per *.out.
   Date: Fri Jan 16 18:56:49 2015 +0100
   1 file changed, 1 insertion(+)
   create mode 100644 .gitignore
```

Si può anche cancellare un commit, lasciandolo nella cronologia dei log.

```
$ git revert $COMMIT_HASH
```

Oppure eliminarlo senza lasciare traccia, mantenendo oppure eliminando i file modificati.

```
$ git reset --soft $COMMIT_HASH # Lascia le modifiche
$ git reset --hard $COMMIT_HASH # Elimina le modifiche
```

Lavorare in remoto

Quando si lavora in remoto ci sono altri passi da seguire in una tipica sessione di lavoro Git.

```
$ git add .
$ git commmit -m “Commento del commit”
```

Dopo i soliti comandi, va eseguito il **push**, cioè il trasferimento, dell'ultimo commit al repository remoto.

```
$ git push
```

Quando non si specificano altre opzioni, il push viene effettuato nella **branch** remota corrispondente a quella locale.

Equivalentemente, prima di una sessione di lavoro locale è bene aggiornare il repository con eventuali cambiamenti remoti.

```
$ git pull -u # Scarica e aggiorna il repository
```

Lavorare in remoto (2)

Per avere una copia esatta di un repository si effettua un'operazione di **clone**.

```
$ git clone git@gringo.linuxvar.it:notes.git

Cloning into 'notes'...
remote: Counting objects: 238, done.
remote: Compressing objects: 100% (92/92), done.
remote: Total 238 (delta 144), reused 238 (delta 144)
Receiving objects: 100% (238/238), 2.19 MiB | 17.00 KiB/s,
done.
Resolving deltas: 100% (144/144), done.
Checking connectivity... done.

$ ls

notes
```

È possibile clonare repository nostri o di altre persone, per poi poterci lavorare, ad esempio, su altri computer.

Branching

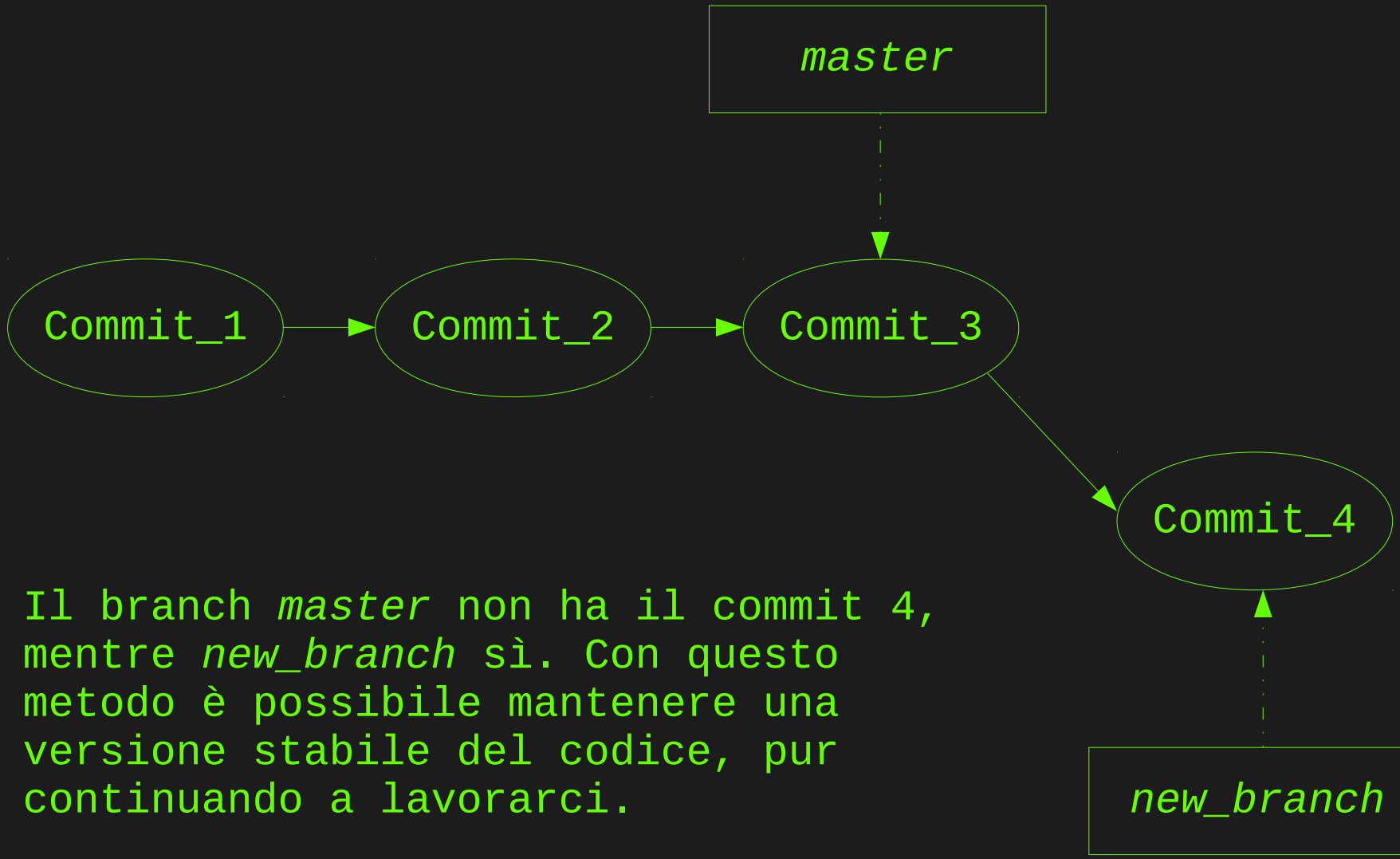
Come possono più sviluppatori lavorare sullo stesso codice, senza sovrascrivere le modifiche reciproche, oppure aggiungere funzionalità ad un progetto, senza minarne la stabilità? Utilizzando i **branch**.

I branch sono rami formati dal flusso dei commit, che si possono dividere, ed eventualmente unire, al flusso principale.

```
$ git init # Crea il branch master
$ git checkout -b new_branch # Crea e usa new_branch
$ echo “// Prova” > prova.h
$ git add .
$ git commit -m “Commit di prova”
$ git push
$ git checkout master # Torna alla branch master
Switched to branch 'master'
```

Branching (2)

La situazione che si è creata è questa:



Il branch *master* non ha il commit 4, mentre *new_branch* sì. Con questo metodo è possibile mantenere una versione stabile del codice, pur continuando a lavorarci.

Merging

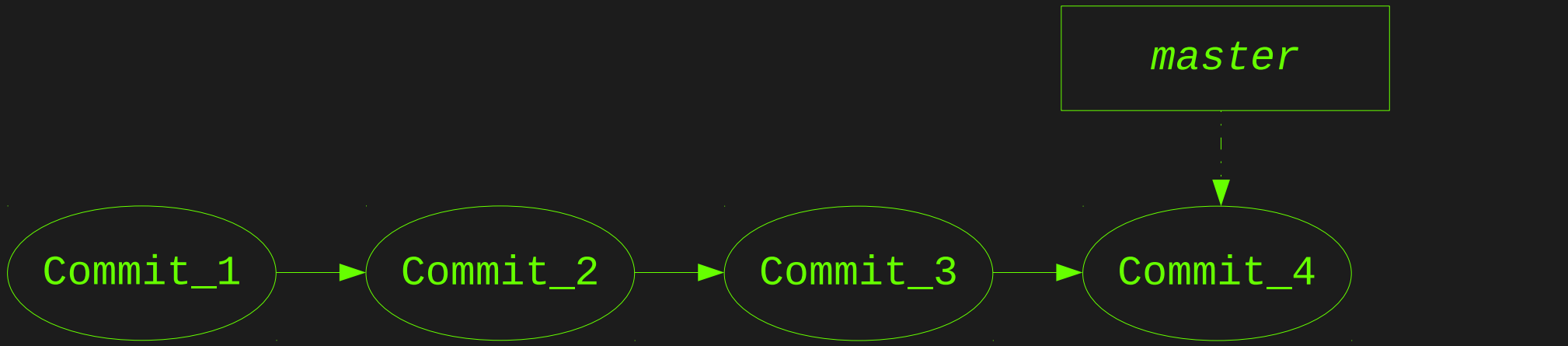
Ci sono quindi I due branch *master* e *new_branch*. Ad un certo punto *new_branch* sarà abbastanza stabile da poter essere intergrato nel flusso principale *master*. Procediamo così:

```
$ git checkout master

$ ls

hello_world.c

$ git merge new_branch

Updating 8282d58..7407cd9
Fast-forward
   prova.h | 0
   1 file changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 prova.h

$ ls

hello_world.c prova.h

$ git branch -d new_branch # Elimina new_branch
```

Merging (2)

Dopo il merge ci sarà una situazione di questo tipo:

Dopo l'eliminazione di *new_branch* si avrà questo stato:



Domande


```
1 #! /usr/bin/env bash
2
3 echo -n “Inserisci domanda: “
4 read DOMANDA
5
6 RISPOSTA=rispondi($DOMANDA)
7
8 echo “La risposta è: $RISPOSTA”
```