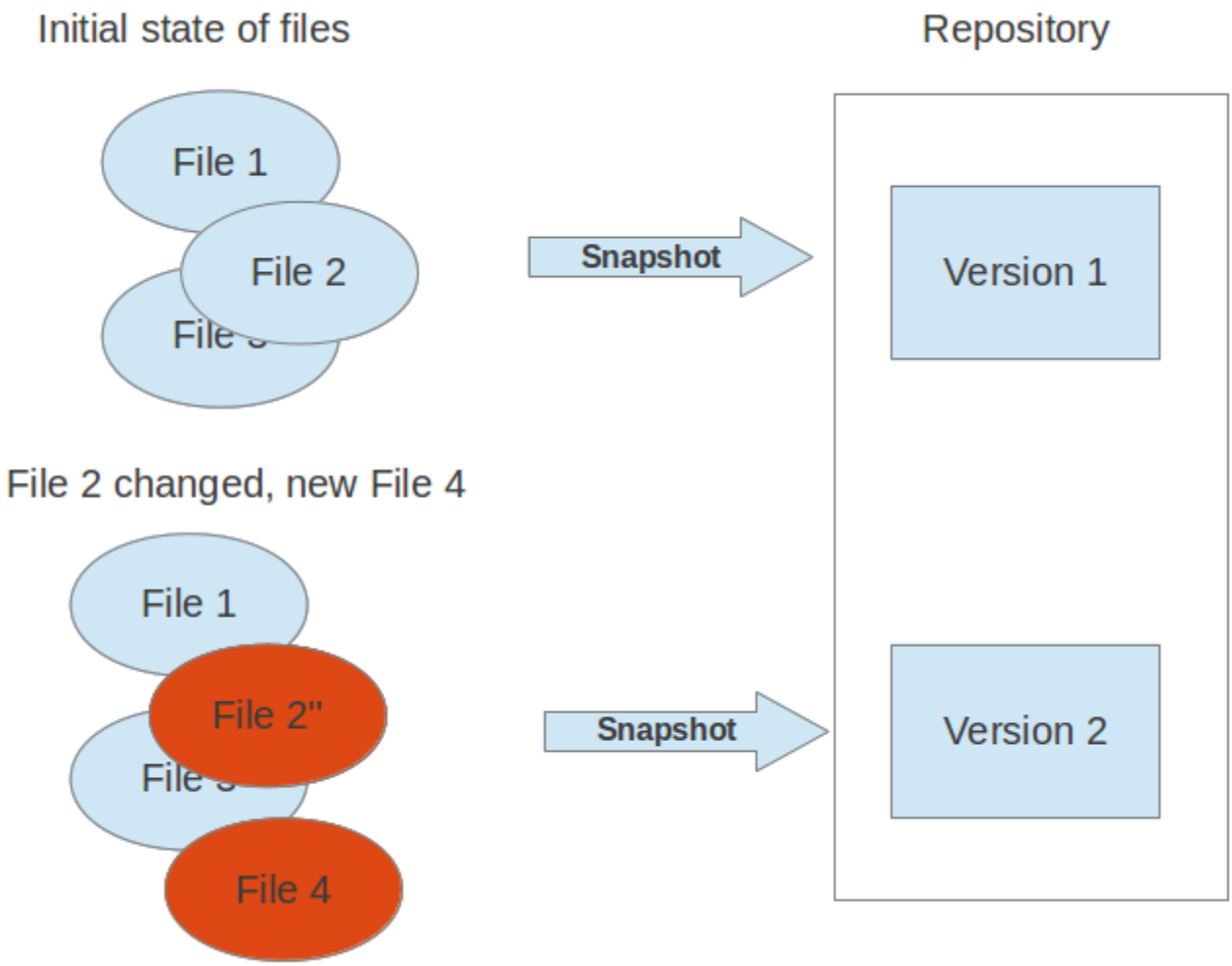
1. Git

1.1. What is a version control system?

A version control system allows you to track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of the files at a certain point in time. The collection of files is usually*source code* for a programming language but a typical version control system can put any type of file under version control.

The collection of files and their complete history are stored in a *repository*.

The process of creating different versions (snapshots) in the repository is depicted in the following graphic. Please note that this picture fits primarily to Git, other version control systems like CVS don't create snapshots but store file deltas.



These snapshots can be used to change your collection of files. You may, for example, revert the collection of files to a state from 2 days ago. Or you may switch between versions for experimental features.

1.2. What is a distributed version control system?

A distributed version control system does not necessarily have a central server which stores the data.

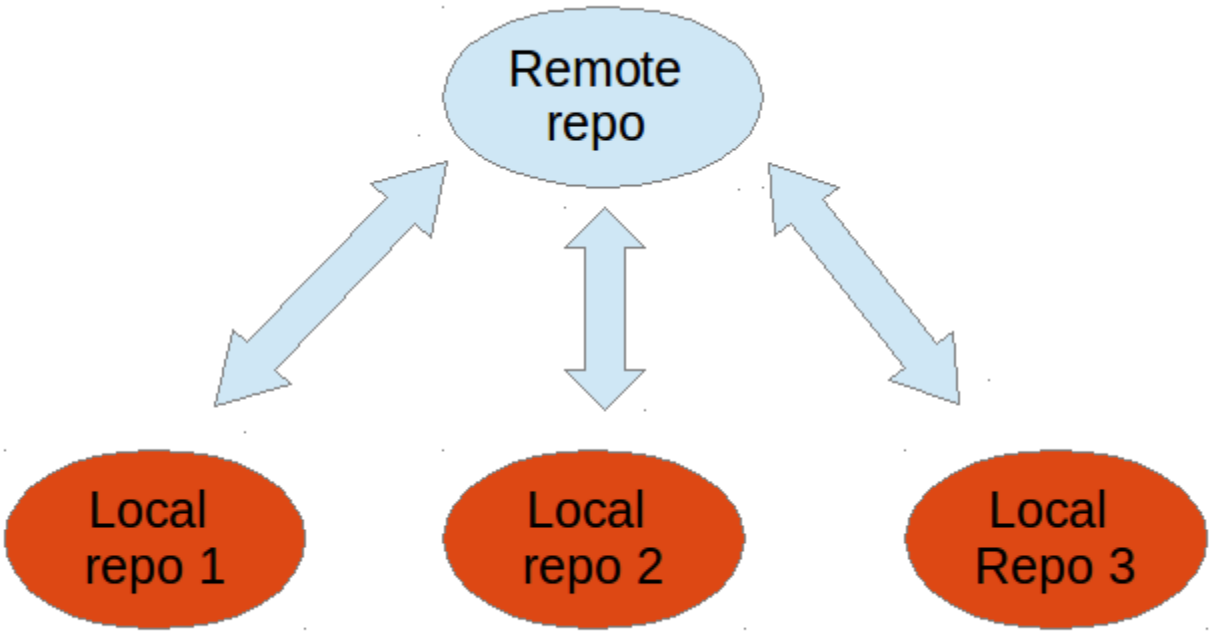
The user can copy an existing *repository*. This copying process is typically called *cloning* in a distributed version control system and the resulting repository can be referred to as *clone*.

Typically there is a central server for keeping a repository but each cloned repository is a full copy of this repository. The decision which of the copies is considered to be the central server

repository is pure convention and not tied to the capabilities of the distributed version control system itself.

Every *clone* contains the full history of the collection of files and a cloned repository has the same functionality as the original repository.

Every repository can exchange versions of the files with other repositories by transporting these changes. This is typically done via a repository running on a server which is, other than the local machine of a developer, always online.



1.3. What is Git?

Git is a distributed version control system.

Git originates from the Linux kernel development and is used by many popular Open Source projects, e.g. the Android or the Eclipse developer teams, as well as many commercial organizations.

The core of Git was originally written in the programming language *C*, but Git has also been re-implemented in other languages, e.g. Java, Ruby and Python.

1.4. Local repository and operations

After cloning or creating a repository the user has a complete copy of the repository. The user performs version control operations against this local repository, e.g. create new versions, revert changes, etc.

You can configure your repository to be a bare or a non-bare repositories.

- bare repositories are used on servers to share changes coming from different developers
- non-bare repositories allow you to create new changes through modification of files and to create new versions in the repository

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

1.5. Remote repositories

Git allows the user to synchronize the local repository with other (remote) repositories.

Users with sufficient authorization can *push* changes from their local repository to remote repositories. They can also *fetch* or *pull*changes from other repositories to their local Git repository.

1.6. Branching and merging

Git supports *branching* which means that you can work on different versions of your collection of files. A branch separates these different versions and allows the user to switch between these version to work on them.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch without affecting the state of your files in another branch.

Branches in Git are local to the respository. A branch created in a local repository, which was cloned from another repository, does not need to have a counterpart in the remote repository. Local branches can be compared with other local branches and with*remote tracking branches*. A *remote tracking branch* proxies the state of a branch in another remote repository.

Git supports that changes from different branches can be combined. This allows the developer, for example, to work independently on a branch called *production* for bugfixes and another branch called *feature_123* for implementing a new feature. The developer can use Git commands to combine the changes at a later point in time.

For example, the Linux kernel community used to share code corrections (patches) via mailing lists to combine changes coming from different developers. Git is a system which allows developers to automate such a process.

1.7. Working tree

The user works on a collection of files which may originate from a certain point in time of the repository. The user may also create new files or change and delete existing ones. The current collection of files is called the *working tree*.

A standard Git repository contains the *working tree* (single checkout of one version of the project) and the full history of the repository. You can work in this *working tree* by modifying content and committing the changes to the Git repository.

1.8. How to add changes to your Git repository

If you modify your *working tree*, e.g., by creating a new file or by changing an existing file, you need to perform two steps in Git to persist the changes in the Git repository. You first add selected files to the *staging area* and afterwards you commit the changes of the *staging area* to the Git repository.

1.9. Adding to the staging area

Note

The *staging area* term is currently preferred by the Git community over the old *index* term. Both terms mean the same thing.

You need to mark changes in the working tree to be relevant for Git. This process is called *staging* or *to add changes to the staging area*.

You add changes in the working tree to the *staging area* with the `git add` command. This command stores a snapshot of the specified files in the *staging area*.

The `git add` command allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

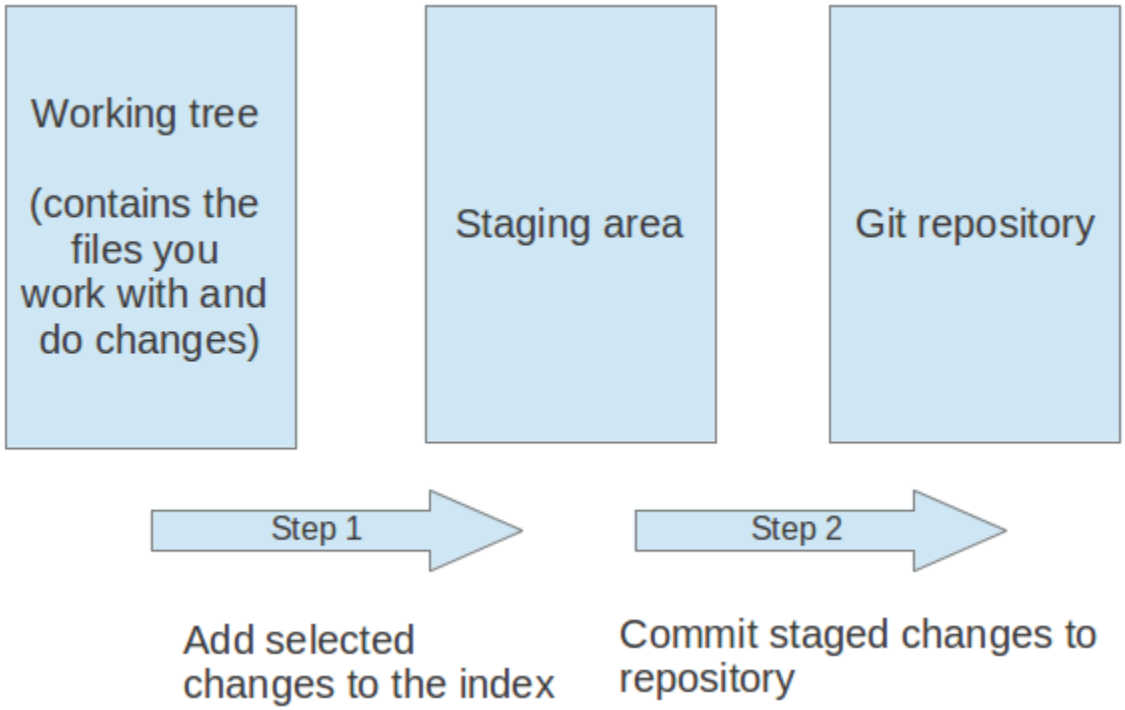
1.10. Committing to the repository

After adding the selected files to the *staging area*, you can *commit* these files to permanently add them to the Git repository.*Committing* creates a new persistent snapshot (called *commit* or *commit object*) of the staging area in the Git repository. A *commit object*, like all objects in Git, are immutable.

The *staging area* keeps track of the snapshots of the files until the staged changes are committed.

For committing the staged changes you use the `git commit` command.

This process is depicted in the following graphic.



1.11. Committing and commit objects

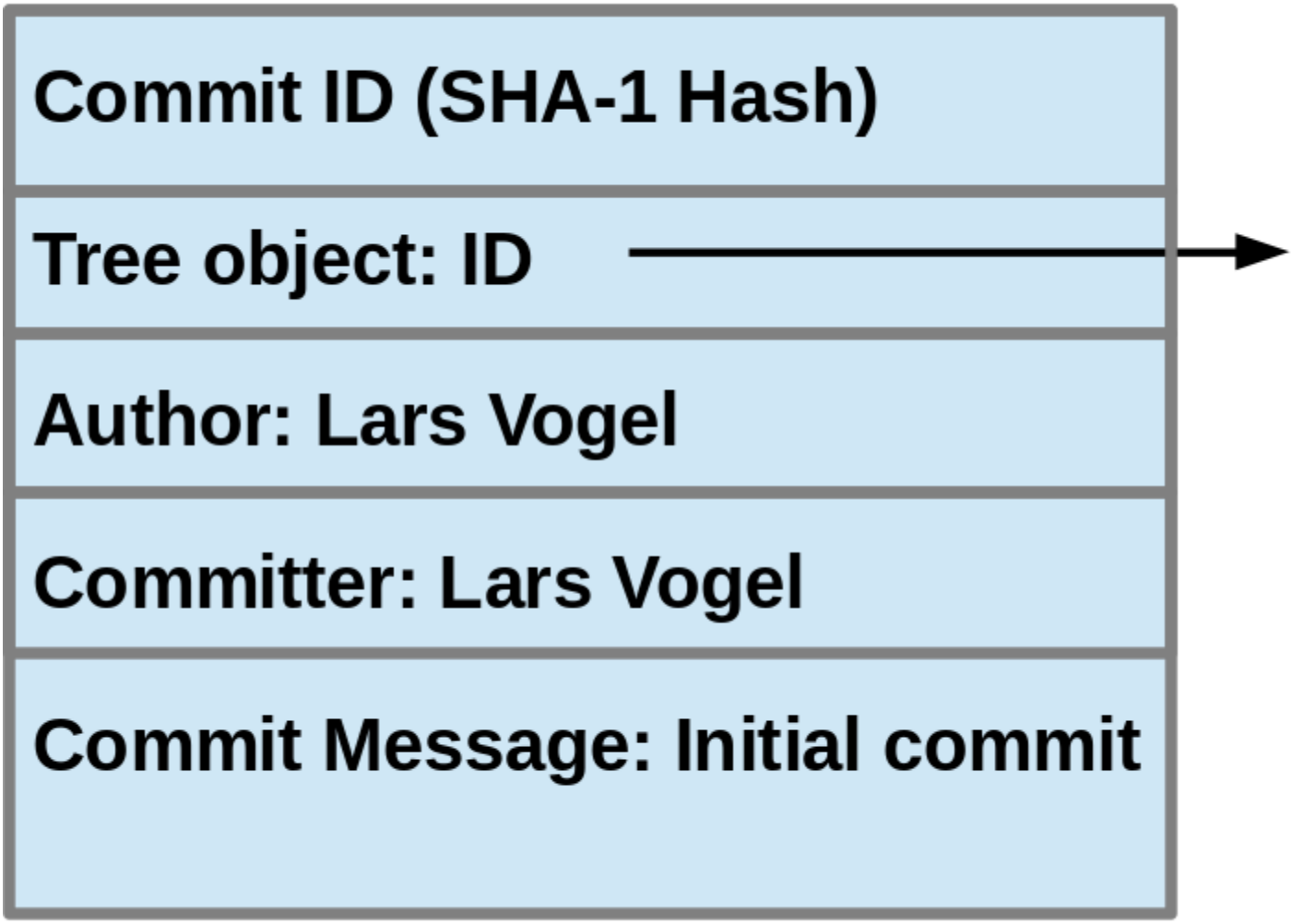
If you commit changes to your Git repository, you create a new *commit object* in the Git repository. This commit object is addressable via a *SHA-1 checksum*. This checksum is 40 bytes long and is a secure hash of the content of the files, the content of the directories, the complete history of up to the new commit, the committer and several other factors.

This means that Git is safe, you cannot manipulate a file in the Git repository without Git noticing that *SHA-1 checksum* does not fit anymore to the content.

The *commit object* points to the individual files in this commit via a *tree* object. The files are stored in the Git repository as *blob*objects and might be packed by Git for better performance and more compact storage. Blobs are addressed via their SHA-1 hash.

Packing involves storing changes as deltas, compression and storage of many objects in a single *pack file*. *Pack files* are accompanied by one or multiple index files which speedup access to individual objects stored in these packs.

A commit object is depicted in the following picture.



The above picture is simplified. Tree objects point to other tree objects and file blobs. Objects which didn't change between commits are reused by multiple commits.

2.0. File states in Git

A file in the working tree of a Git repository can have different states. These states are the following:

- untracked: the file is not tracked by the Git repository, this means it was neither staged, added to the staging area nor committed
- tracked: committed and not staged
- staged: staged to be included in the next commit

- dirty / modified: the file has changed but the change is not staged

3. Installation

4. Git Setup

4.1. Global configuration file

Git allows you to store global settings in the `.gitconfig` file located in the user home directory. Git stores the committer and author of a change in each commit. This and additional information can be stored in the global settings.

You setup these values with the `git config` command.

In each Git repository you can also configure the settings for this repository. Global configuration is done if you include the `--global` flag, otherwise your configuration is specific for the current Git repository.

You can also setup system wide configuration. Git stores theses values is in the `/etc/gitconfig` file, which contains the configuration for every user and repository on the system. To set this up, ensure you have sufficient rights, i.e. root rights, in your OS and use the `--system` option.

The following configures Git so that a certain user and email address is used, enable color coding and tell Git to ignore certain files.

4.2. User Configuration

Configure your user and email for Git via the following command.

```
# configure the user which will be used by git
# of course you should use your name
git config --global user.name "Example Surname"

# same for the email address
git config --global user.email "your.email@gmail.com"
```

4.3. Push configuration

The following command configure Git so that the `git push` command pushes only the active branch (in case it is connected to a remote branch, i.e.configured as remote tracking branches) to your Git remote repository. As of Git version 2.0 this is the default and therefore it is good practice to configure this behavior.

```
# set default so that only the current branch is pushed
git config --global push.default simple
# alternatively configure Git to push all matching branches
# git config --global push.default matching
```

You learn about the push command in **Section 13.2, "Push changes to another repository"**.

4.4. Query Git settings

To query your Git settings of the local repository, execute the following command:

```
git config --list
```

If you want to query the global settings you can use the following command.

```
git config --global --list
```

5. Getting started with Git

5.1. Target of this chapter

In this chapter you create a few files, create a local Git repository and commit your files into this repository. The comments (marked with #) before the commands explain the specific actions.

Open a command shell for the operations.

5.2. Create directory

The following commands create an empty directory which you will use as Git repository.

```
# switch to home
cd ~/
```

```
# create a directory and switch into it
mkdir ~/repo01
cd repo01

# create a new directory
mkdir datafiles
```

5.3. Create Git repository

The following explanation is based on a non-bare repository. See **Section 3, "Terminology"** for the difference between a bare repository and a non-bare repository with a *working tree*.

Every Git repository is stored in the `.git` folder of the directory in which the Git repository has been created. This directory contains the complete history of the repository.
The `.git/config` file contains the configuration for the repository.

The following command creates a Git repository in the current directory.

```
# initialize the Git repository
# for the current directory
git init
```

All files inside the repository folder excluding the `.git` folder are the *working tree* for a Git repository.

5.4. Create content

The following commands create some files with some content that will be placed under version control.

```
# switch to your new repository
cd ~/repo01

# create an empty file in a new directory
touch datafiles/data.txt
```

```
# create a few files with content
ls > test01
echo "bar" > test02
echo "foo" > test03
```

5.5. See the current status of your repository

The `git status` command shows the working tree status, i.e. which files have changed, which are staged and which are not part of the staging area. It also shows which files have merge conflicts and gives an indication what the user can do with these changes, e.g. add them to the staging area or remove them, etc.

Run it via the following command.

```
git status
```

5.6. Add files to the staging area

Before committing changes to a Git repository you need to mark those that should be committed. This is done by adding the new and changed files to the staging area. This creates a snapshot of the affected files.

Note

In case you change one of the files again before committing, you need to add it again to the staging area to commit the new changes.

```
# add all files to the index of the Git repository
git add .
```

Afterwards run the `git status` command again to see the current status.

5.7. Commit to Git repository

After adding the files to the Git staging area, you can commit them to the Git repository. This creates a new commit object with the staged changes in the Git repository and the HEAD reference points to the new commit. The `-m` parameter allows you to specify the commit

message. If you leave this parameter out, your default editor is started and you can enter the message in the editor.

```
# commit your file to the local repository
git commit -m "Initial commit"
```

6. Looking at the result

6.1. Results

The Git operations you performed have created a local Git repository in the `.git` folder and added all files to this repository via one commit. Run the `git log` command

```
# show the Git log for the change
git log
```

You see an output similar to the following.

```
commit e744d6b22afe12ce75cbd1b671b58d6703ab83f5
Author: Lars Vogel <Lars.Vogel@gmail.com>
Date:   Mon Feb 25 11:48:50 2013 +0100

    Initial commit
```