Linux For Embedded Systems

For Arabs

Cairo University
Computer Eng. Dept.
CMP445-Embedded Systems

Ahmed ElArabawy

Lecture 10:
Introduction to Git & GitHub (Part 1)

Introduction to Software Version Control (SVC)

Life before SVC

- We only have the latest snapshot of the code, we don't know when a certain feature was added, and who added it….
  - What if we don't want this feature anymore?
  - What if we want to maintain the code with and without this feature?
  - When was this code modification done, who did it? And what other modifications need to come with it ?
- We can not revert to the old version of the code, once we added some change, we can not bring back an old version
  - What if the new change caused a problem, and we want to use the older version ?
- Team collaboration is very complicated
  - What if multiple people want to modify the same file at the same time?
  - What if we want to work on different features at the same time?

Now…SVC is in Town

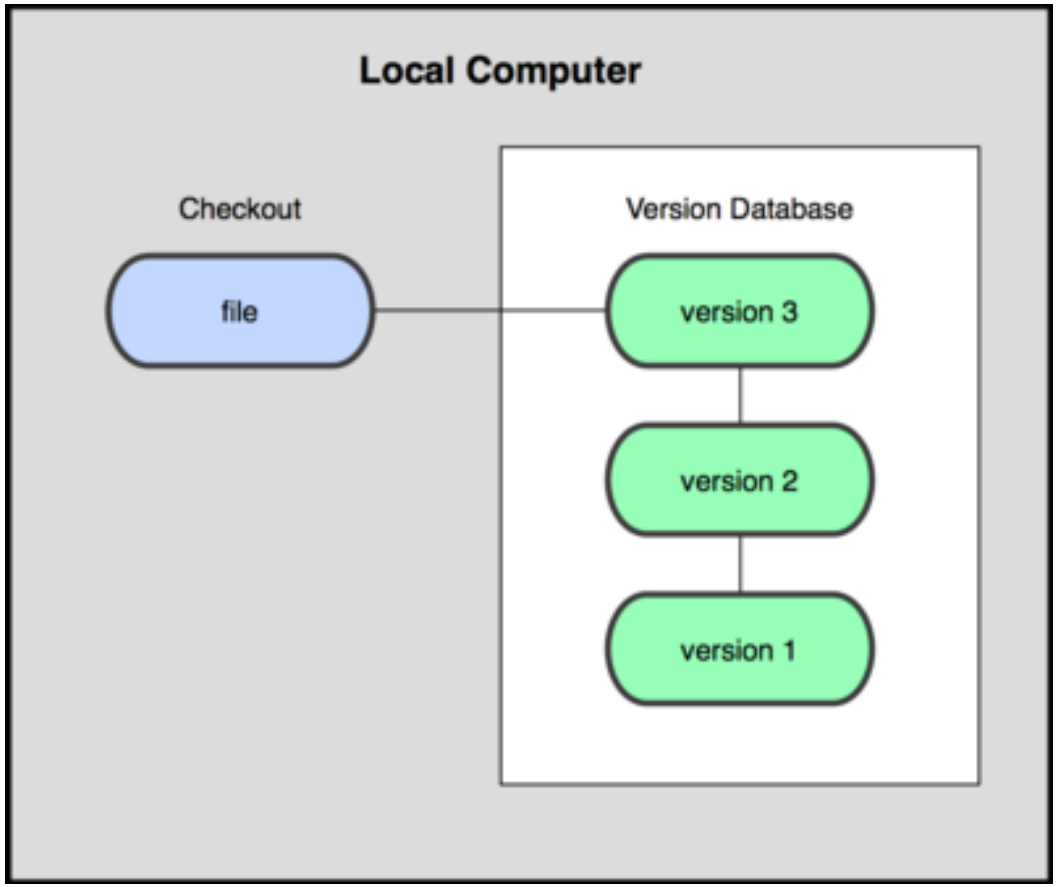
- Version Control provides,
  - A means to backup different snapshots of the code in case somebody introduced code that caused problems
  - Compare changes between different releases
  - Enables team collaboration
  - Enables working on multiple issues separately
  - Enables maintaining older releases and fixing issues, while working on new features in the new release, and later migrate the bug fixes
  - Protect the code no matter what happened from users, loss of data, corrupted drives, malicious handling,…etc

SVC Types

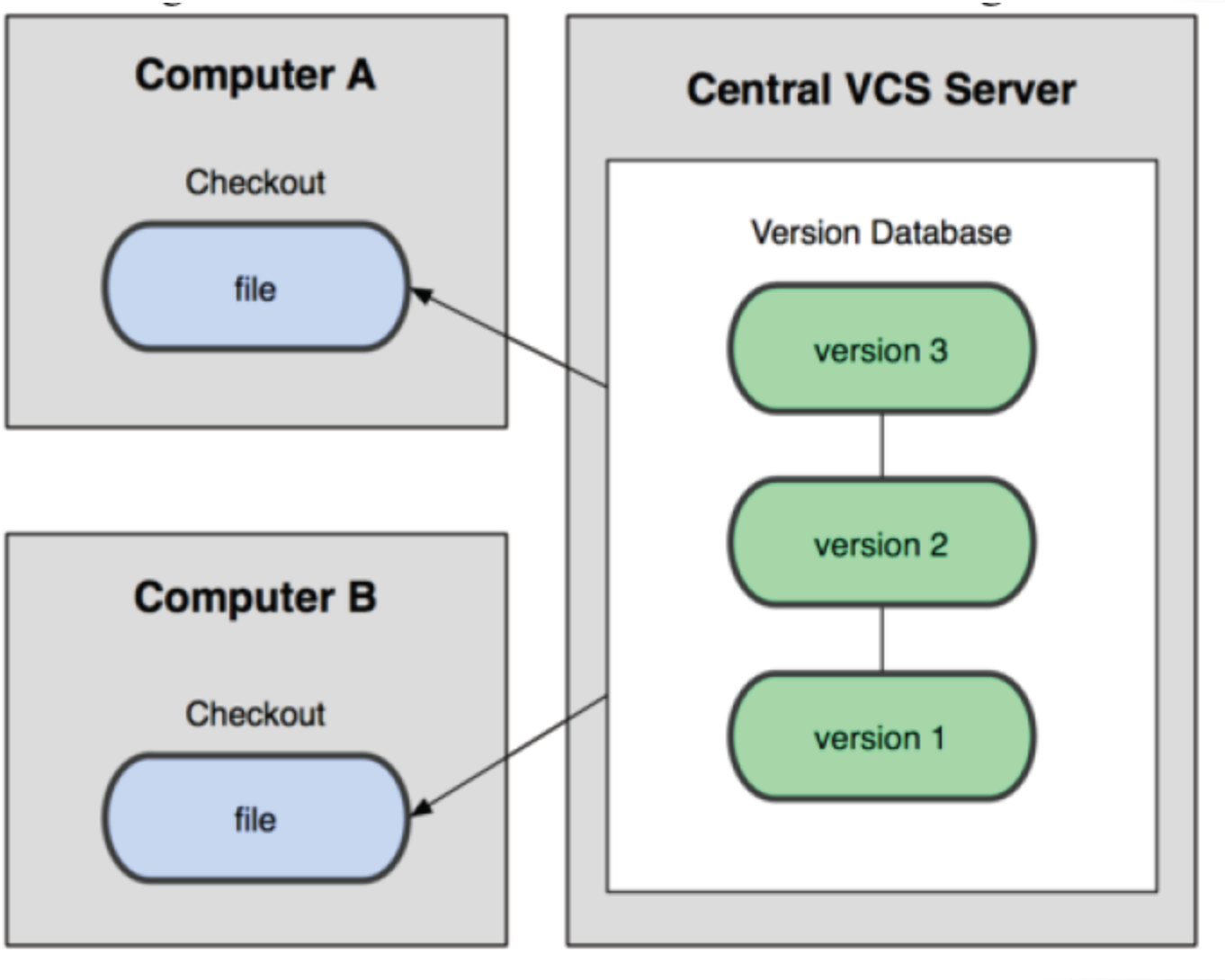
SVC Types: Local SVC

- For local use, manually copy old files in a backup folder
  - Error prone
  - Easy to corrupt data, and overwrite files
  - Not suitable for team collaboration
  - Takes big disk space
  - Only a limited number of snapshots available
- An automated tool to do local SVC is "***rcs***"
  - It keeps record of the patches between the different revisions for future reference
  - However, we still have the problems,
    - Does not facilitate team collaboration
    - Everything on one computer, a single point of failure

SVC Types: Local SVC



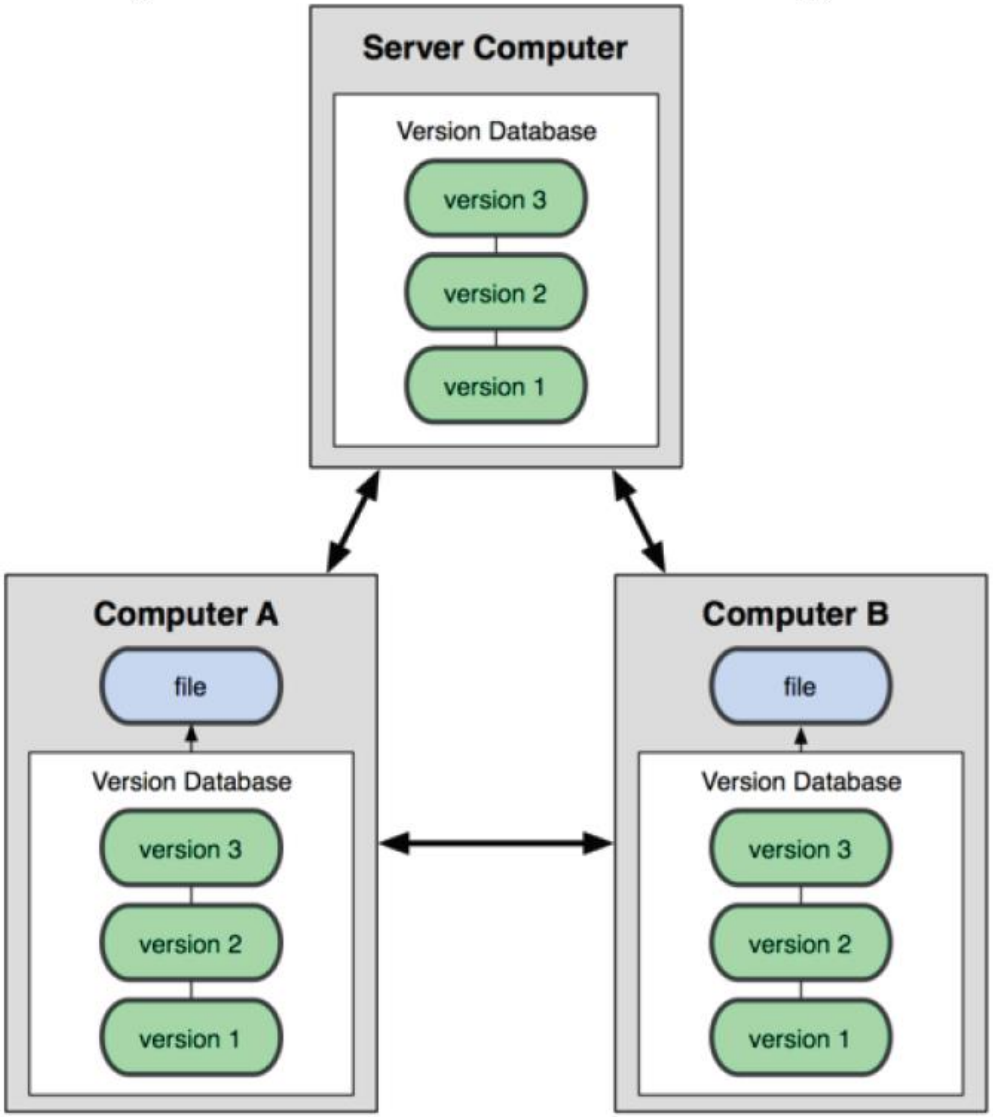
SVC Types: Centralized SVC



SVC Types: Centralized SVC

- The history info is stored in a central server (called the depot, or the repositories)
- User computers (clients) only checks out a snap-shot from the server
- This setup provides,
  - Protection from loss of data upon client data corruption
  - Team collaboration
- However, some limitations exist,
  - Slow, any file update operation require network access
  - Dependency on the connection to the server (can not work when offline)
  - The central server becomes a single point of failure
  - User can not do his own mini-projects without writing to the central drive
- Examples are,
  - ***CVS***
  - ***Subversion (SVN)***
  - ***Perforce***

SVC Types: Distributed SVC



SVC Types: Distributed SVC

- Each client have the full repo and not just a snapshot of the files
  - No single point of failure, the repo on the client can replace the one in the server (with minimum loss) in case of a failure in the server
  - Users can perform their work while offline
  - Faster response since a lot of the operations are done locally
  - Enable distributed (clustered/hierarchical) development which is useful in huge projects

- Examples are,
  - ***Git***
  - ***Mercurial***
  - ***Bazaar***

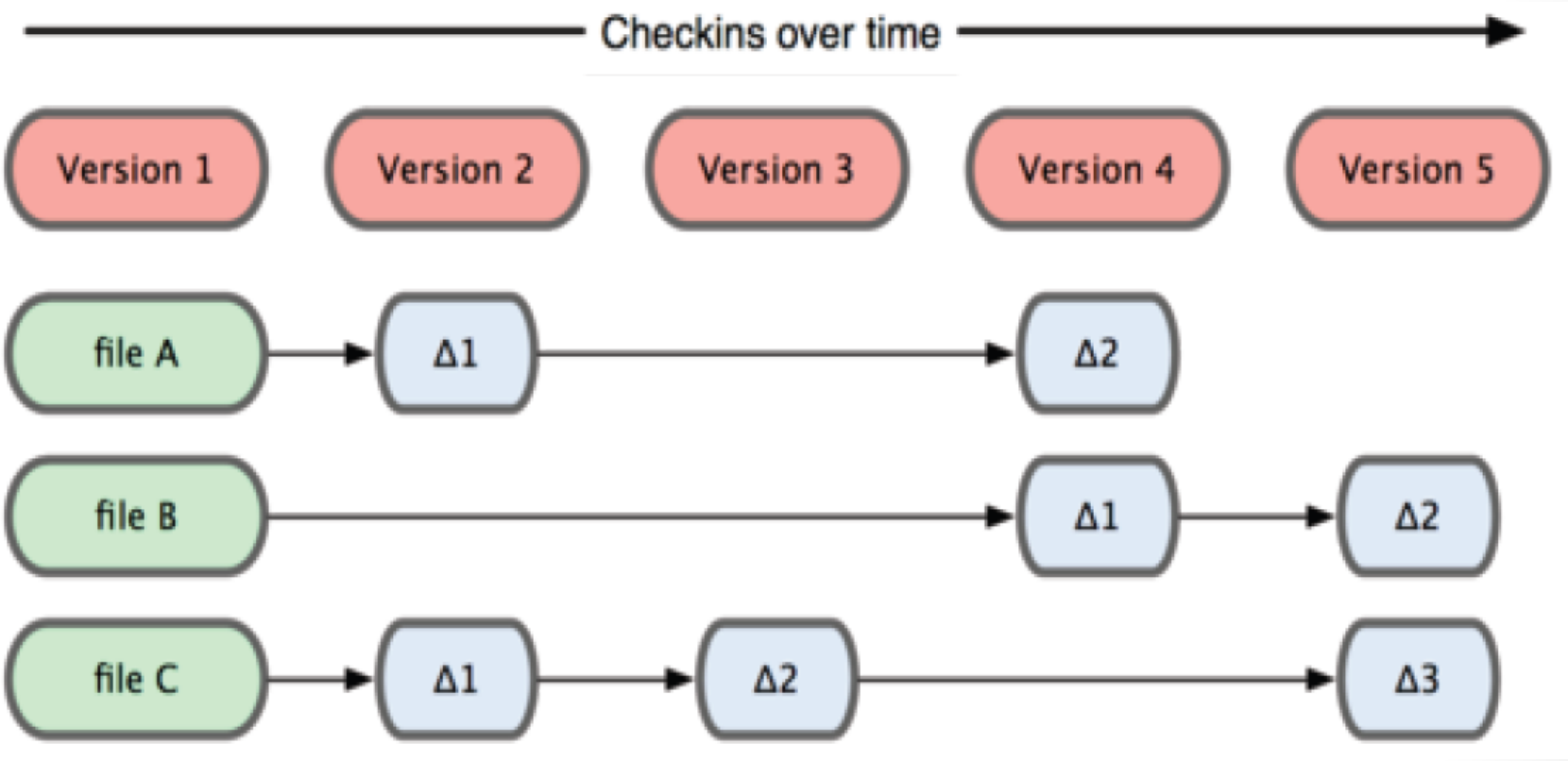
A few words about Git

History of Git

- Git is created by **Linus Torvalds**, the creator of Linux
- It was developed initially to manage the Linux development community
- Linux code has been managed,
  - 1991-2002: Using an archive of patches
  - 2002-2005: Using **BitKeeper**
  - 2005-Now: Using **Git**
- Target was,
  - Speed
  - Simple design
  - Fully distributed
  - Strong support for non-linear development (thousands of parallel branches)
  - Able to handle large projects like the Linux kernel efficiently

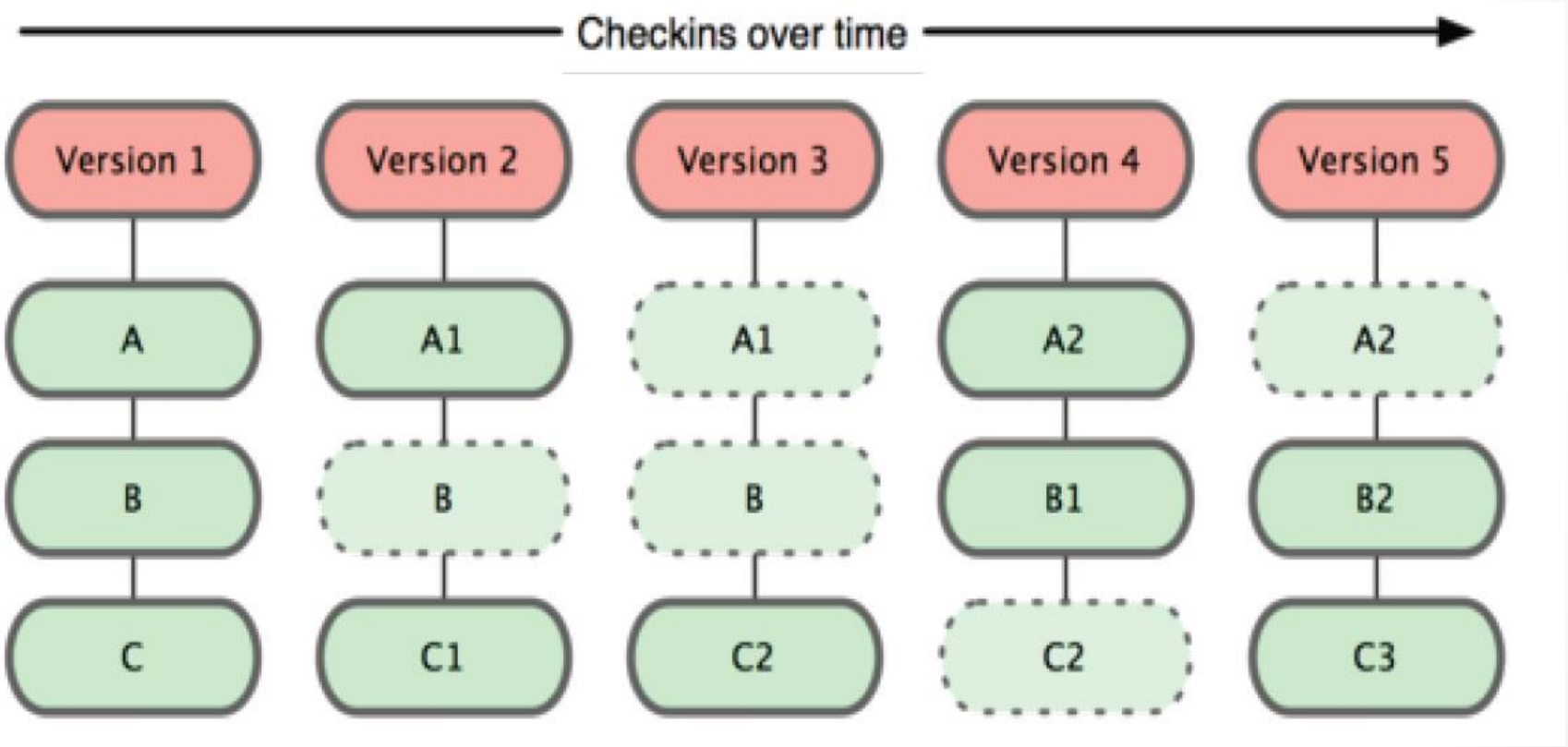
Git Storage: Snapshots, Not Differences

Typical Centralized SVC tools



Git Storage: Snapshots, Not Differences

Git Storage



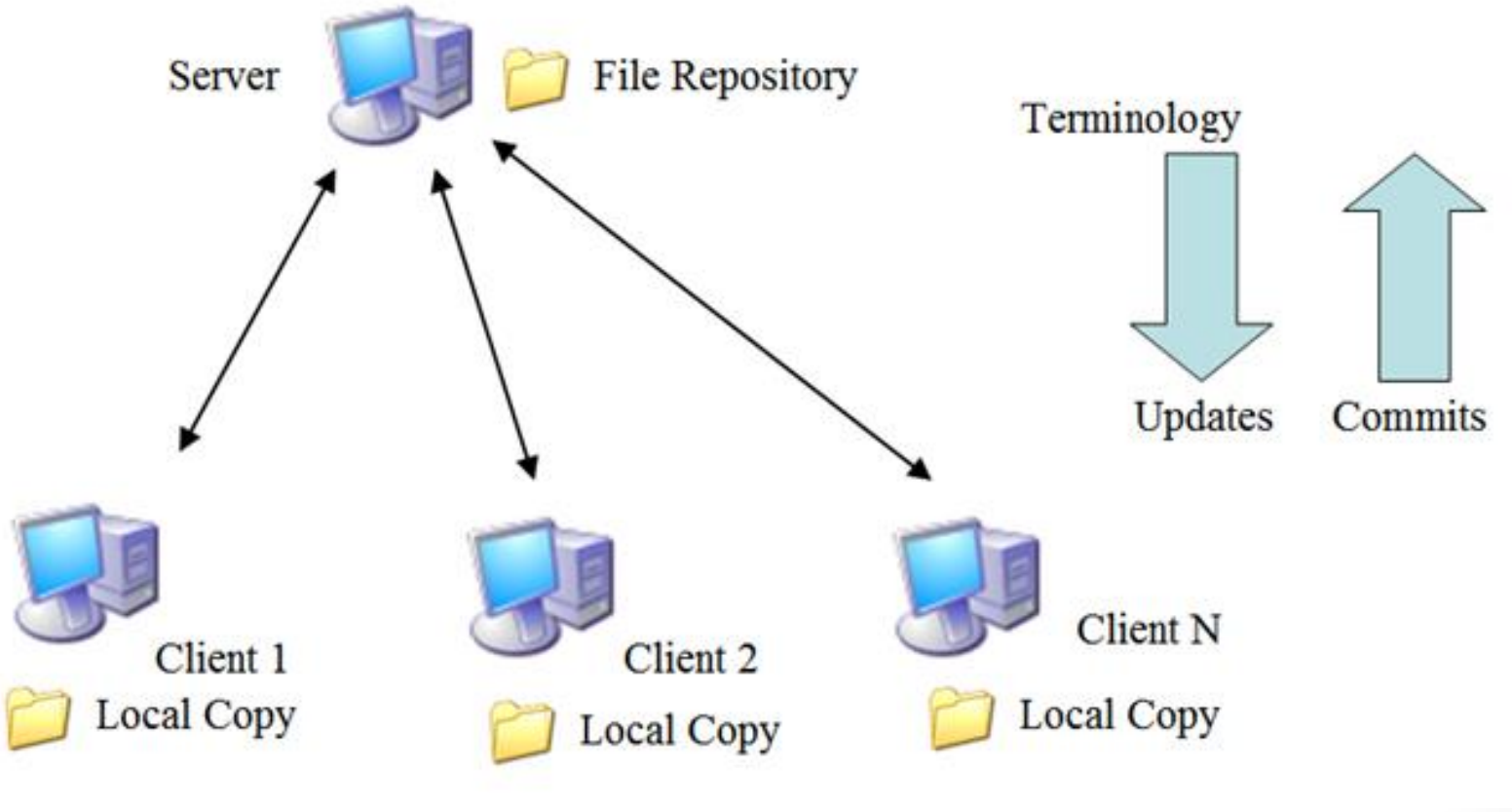
Git Repo

- A repo is a directory structure that holds all the project history and events
- The directory name is "***.git***"
- **Git** relies on having the full repo local to the user, so the user will need to copy it over from the server the first time (**cloning** the server)
- Then it performs all operations on the local repo (**commit**)
- When desired, the user can synchronize its repo with the server repo
  - User can **push** his own changes to the server repo
  - User can **pull** other users changes from the server repo
- The local **Git** repo is a complete, self contained, independent full fledged repo
- On top the of the repo (the **.git** directory), the user will have a snapshot of the code (***working copy***)
- User can check out from his local repo the version to work on. This does not need connection to the server

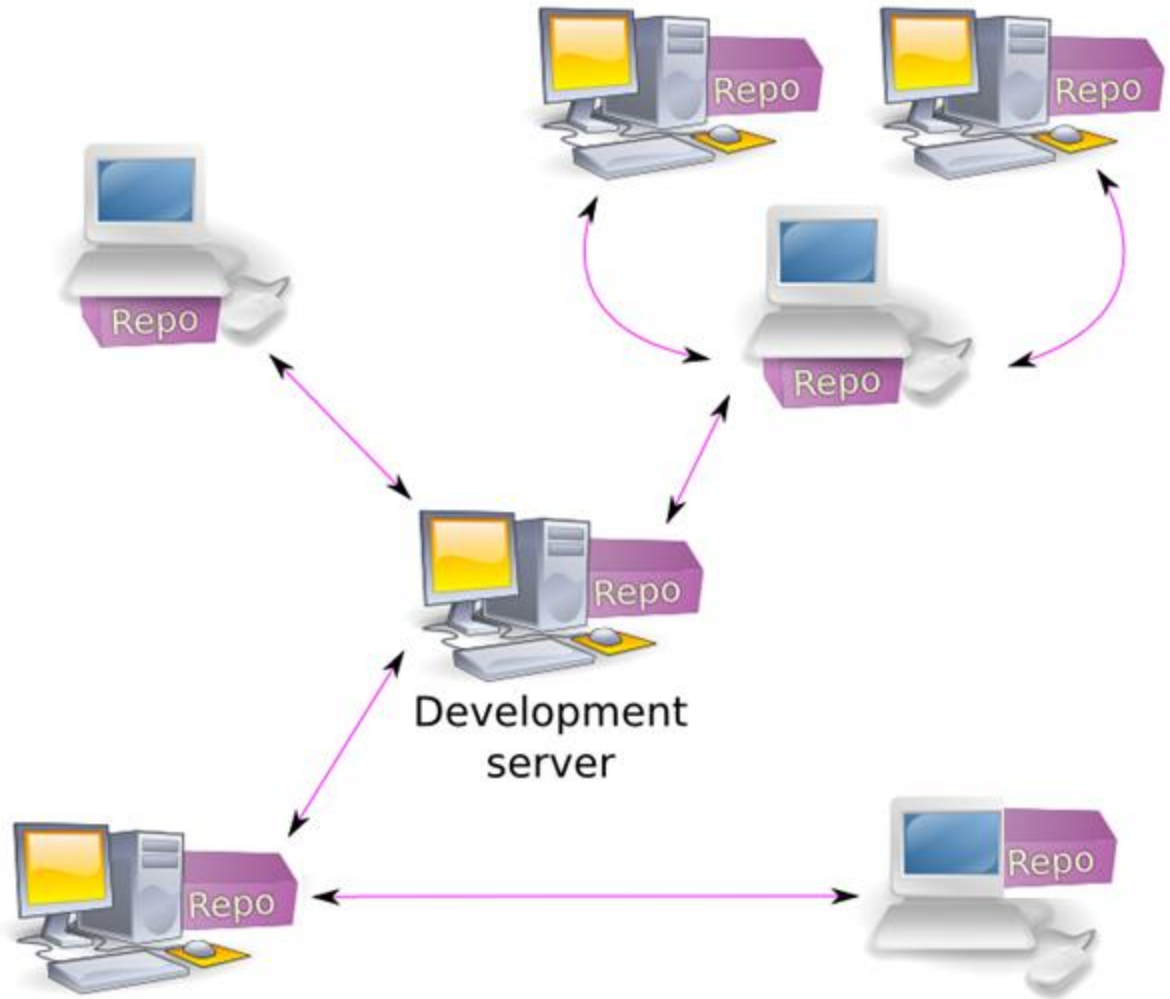
Most Operations are local

- Each user will have his own repository, and he will use **Git** tools to do all his work on this repo
- This is different from other tools like ***SVN*** where the repo is centralized in a server, and clients just own a code snapshot
- **Git** repo on the client is almost a full replica for the one in the server
- This makes **Git** very fast, and almost fully operational when working offline
- This also enables users to build independent teams that work together and synch their repos together without needing to connect to the server

Centralized SVC : SVN



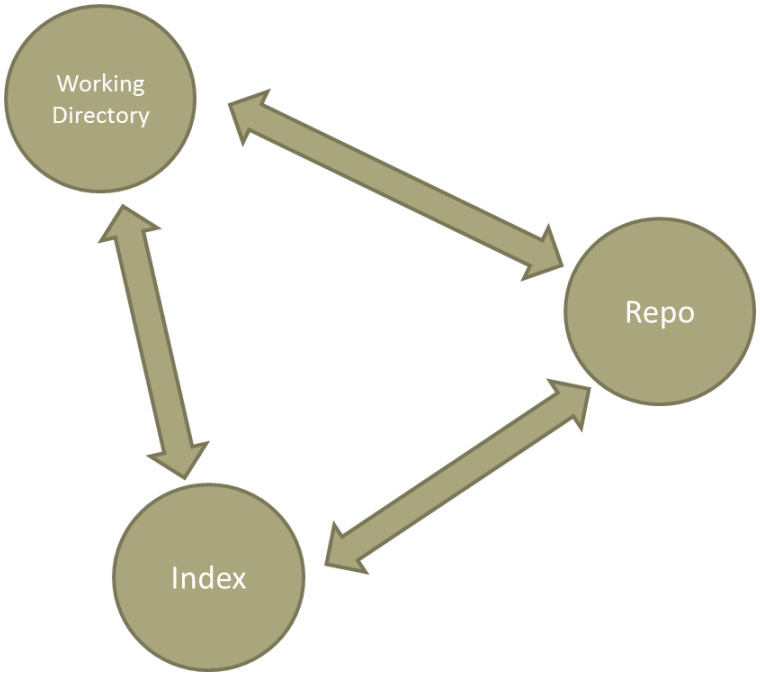
Distributed SVC: Git



Git File Integrity

- Each content or object stored in Git repo is goes through ***SHA1 hashing*** and the checksum is stored
- The checksum is 40 digits hex string, it is used as an id for the content/object in the git repo
- Accordingly, any data corruption is detected immediately (mismatch between the file contents and its SHA1 hash)

Repo, Working Directory & Index



- There are three areas in Git,
  - **Repo** (***.git*** directory):
    - Contains the history of all file modification
  - **Working directory** (Directory containing the ***.git***)
    - Contains the snap shot of files that we are working with
  - **Index** (also called **staging area**)
    - Contains the files that is "***staged***", i.e. prepared to be committed to the repo

Main Operations in Git

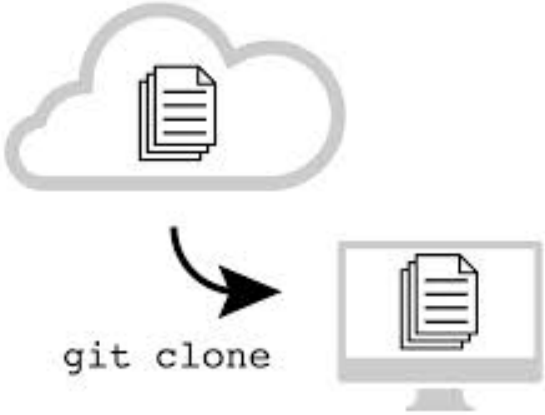
- The main operations in Git are,
  - Initialize a **Git** repo
  - Clone a remote **Git** repo
  - Checkout a code snapshot from **Git**
  - Stage a group of files
  - Commit a group of files
  - Tag a code snapshot
  - Pull changes from a remote machine
  - Push local changes to a remote machine
  - Branch
  - Merge two branches

Initializing a Git Repo

- This is the first operation to be done to create a **Git** repo locally
- Assuming you have a group of files that you want to start tracking with **Git**
- All you need to do is to go to the top directory of your code
- Then you need to issue the command,

  $ git init
- That creates an empty repo, which is a directory named ***.git*** in this directory
- This repo will carry the history of your project
- Any future commands in **Git** will make changes in this repo

Clone a Remote Repo



- Cloning a **Git** repo means replicating a copy of the full repo at the local machine

  $ git clone <remote project location>
- This means the local machine will have the full history of the project (the same as the remote machine)
- Normally, clone also checks out the latest from the repo to prepare the working directory

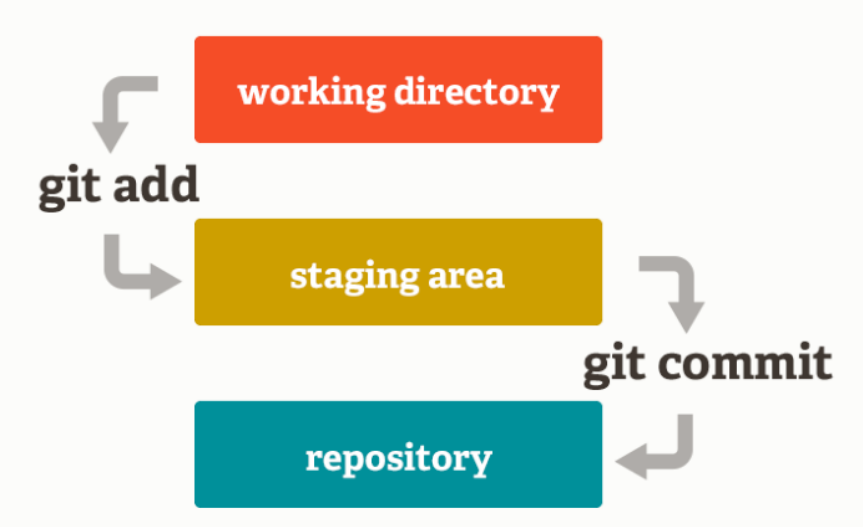
Checkout a Code Snapshot

- We can take a snapshot of the code from the repo into the working directory

  $ git checkout

- The snapshot can be the latest code or any point in the history of the code

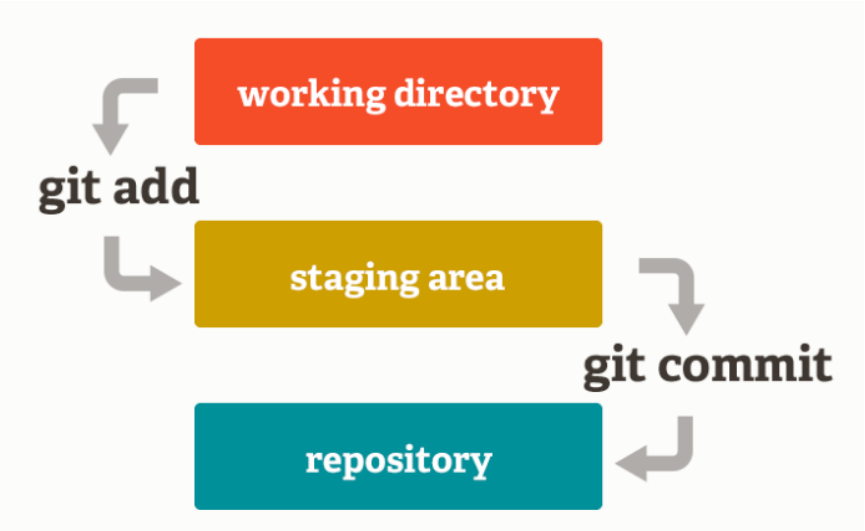
Staging Files



- Staging files means prepare these files to be submitted to the repo with the new modifications
- Files can be added to the staging area (index)
  - The first time (to start tracking the files through Git)
  - Subsequent times (to submit the new changes in the files to the repo)
- To stage a file

  $ git add <file>

Committing Files



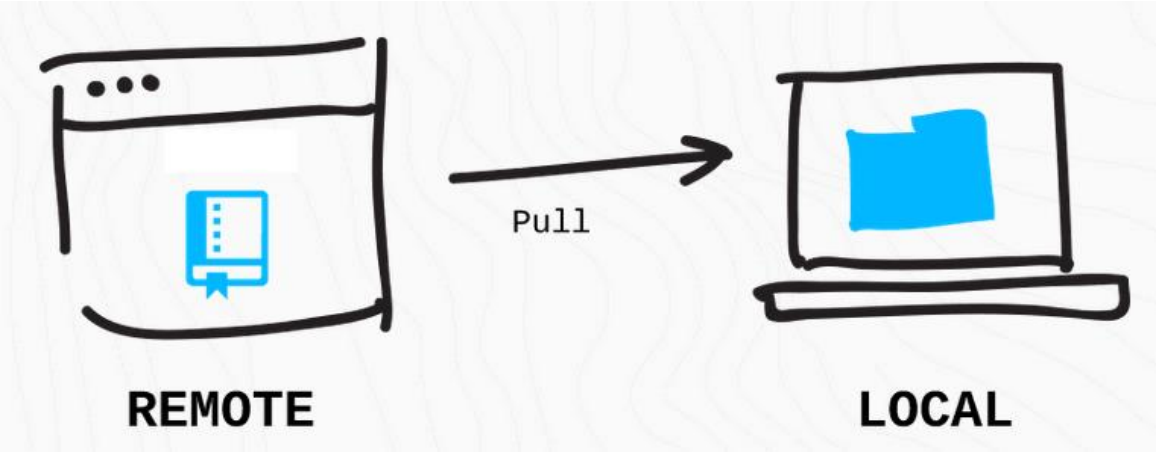
- Committing a file means submitting the new modifications in the file to be tracked by Git
- This means the new snapshot of the file is stored in the repo

$ git commit

Tagging a Code Snapshot

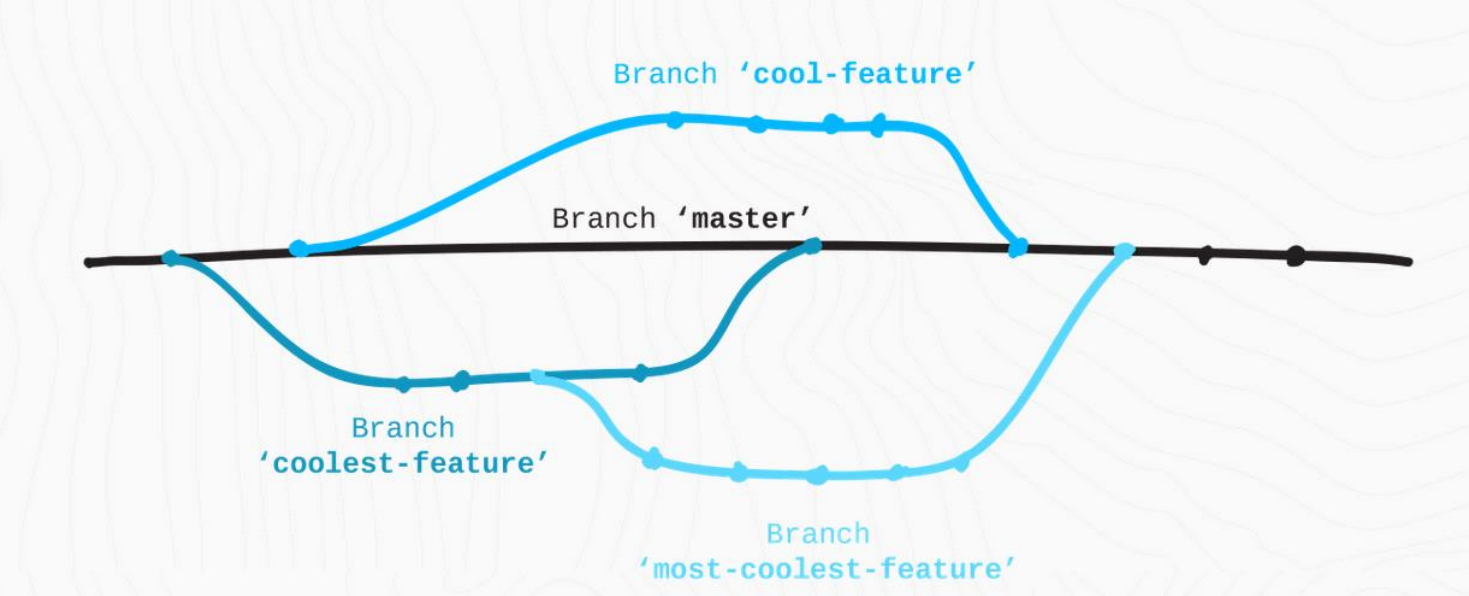
- Tagging a code snapshot means putting a label for this snapshot
- This enables us to deal with this snapshot in the future (such as checking it out, or comparing it to another code snapshot)

Pulling and Pushing



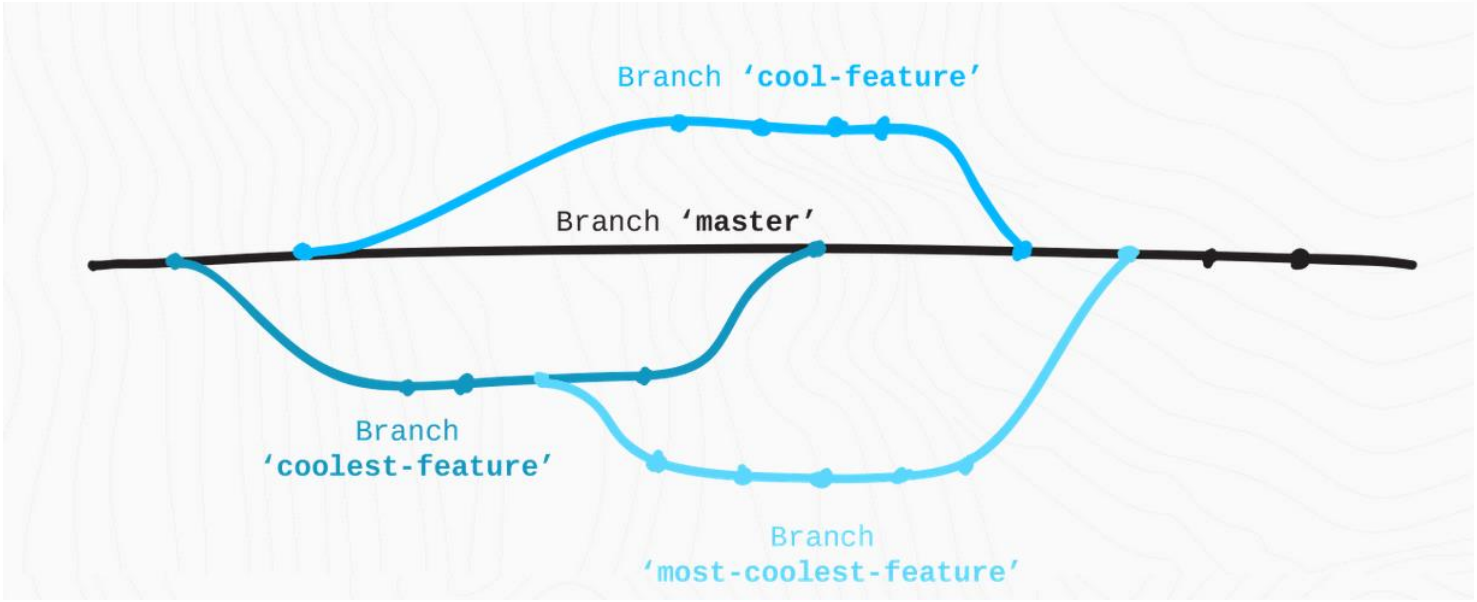
- Pulling means bringing updates in the remote machine repo to the local machine
- Pushing means sending the updates in the local machine repo to the remote machine

Branching



- Branching means maintaining the history of multiple copies of the code at the same time
- This is useful when working on multiple features at the same time
- Initially, Git provides a single branch named the ***master branch***
- When checking out code from the repo, you will need to specify which branch you want to check out

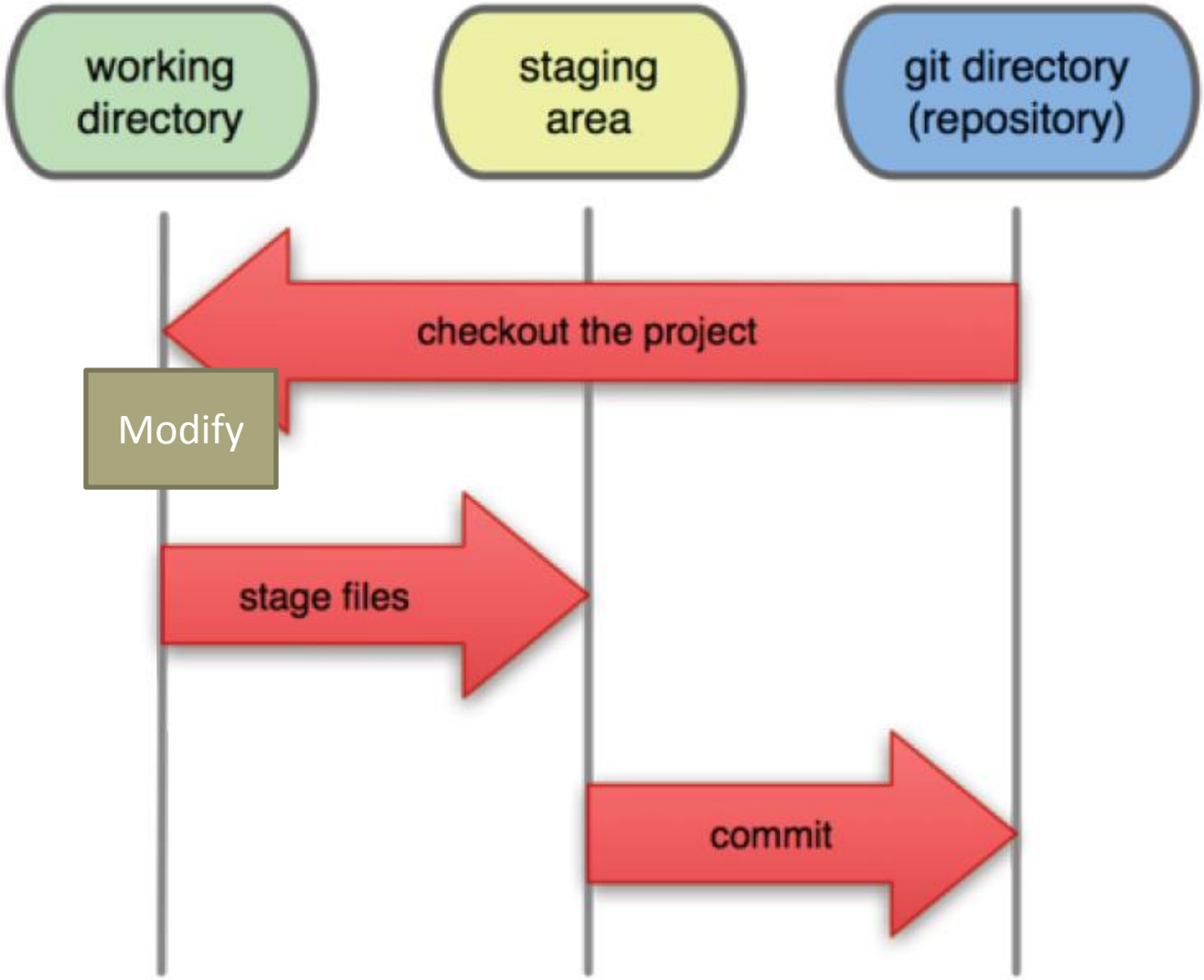
Merging Branches



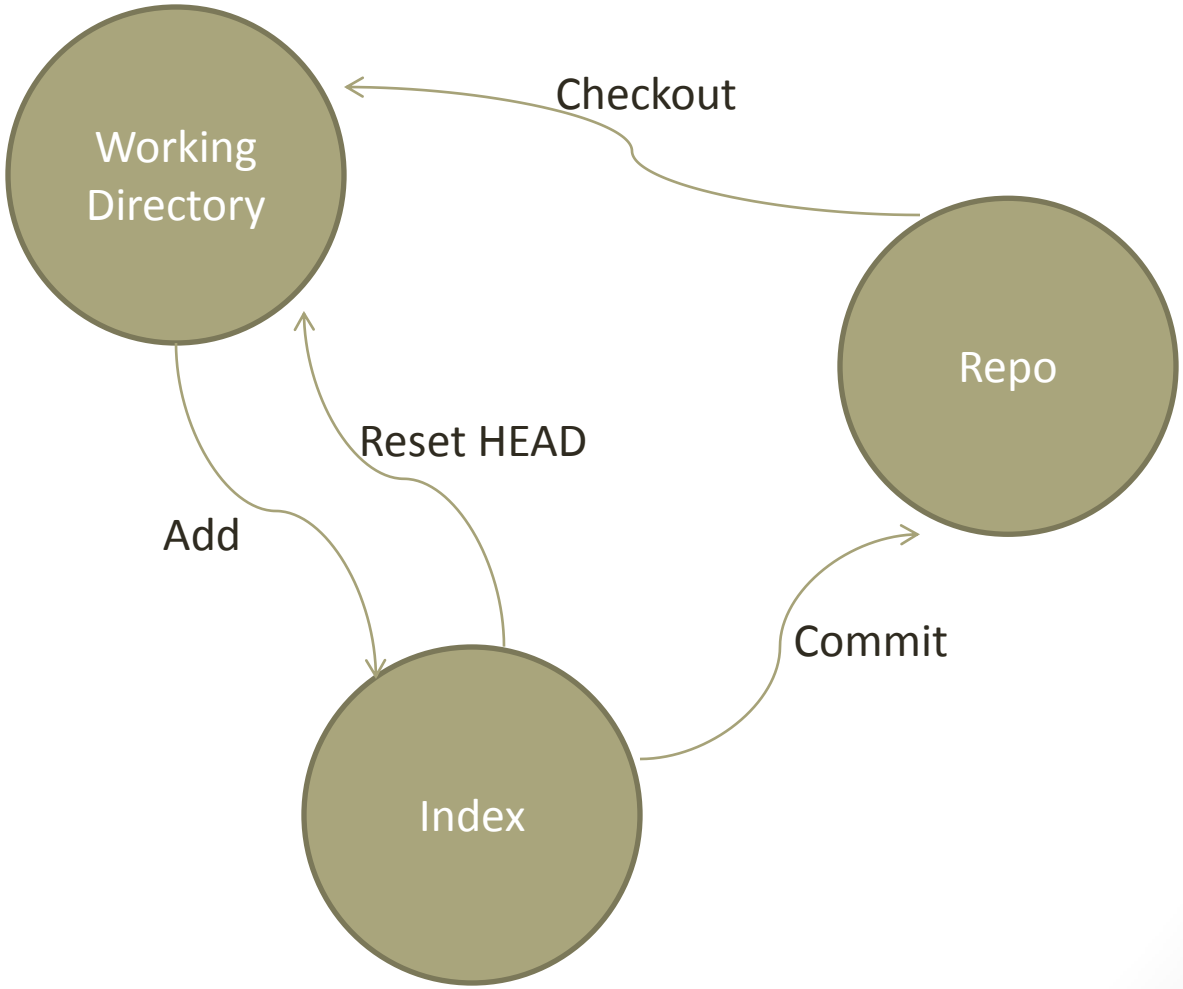
- Merging means joining 2 branches into a single branch
- This means merging the changes that was done in both branches in one code snapshot
- Merging can be simple when each branch modifies different part of the code
- But it can also be complicated and require manual intervention if the same parts of the code has been modified in the different branches

Files in Git

- When we create a new file in the working directory, it is called "***Untracked***" cause the repo does not know anything about it and Git is not tracking changes into it
- Once the file is committed once to the repo, now it is "***tracked***", which means Git will build a history for it in the repo
- If we modify a tracked file in the working directory, the file is called "***unstaged***" cause we did not prepare to commit the changes to the repo yet
- First step to commit the changes is to "***stage***" the file, i.e. copy it to the index (staging area). Now the file is called "***staged***"
- Now a staged file can be committed to the repo
- This means a file is :
  - **Untracked**: It was never committed to the repo, this file is not maintained by Git
  - **Unstaged**: A file that is tracked, but it has modifications and not yet staged (copied to the index)
  - **Staged**: It is a file that was copied to the staging area (index) in preparation to commit it to the repo (either for the first time, or because it contains modifications) but not yet committed
  - **Committed**: A file that is not modified from the version in the repo


working directory
staging area
git directory (repository)
checkout the project
Modify
stage files
commit

Working Directory/Index/Repo



Installing Git

Installation

- Git mainly runs on Linux (but runs on other OSs as well)
- To install Git,

  $ sudo apt-get install git-core
- Git Commands

  $ git <command> <arguments>
- To know what version is running

  $ git version
- To get help

  $git help

  $git help <command>

First Time Git Setup

- As we first use Git, we need to,
  - Identify user identity (name and email); this will be needed for commits in the future
  - Identify which editor to use
  - Identify which diff tool to use
- These configurations can be configured for,
  - Any user on the machine, by writing it in ***/etc/gitconfig***
  - Any project for a specific user on the machine; by writing it in ***~/.gitconfig***
  - For a specific project; by writing it in ***<project dir>/.git/config***

Defining the Author/Committer Identities

- Git stores for each commit the,
  - Author name/email
  - Committer name/email
- Git collects those contact info from (in order)
  - User can specify the author name/email in the commit command

    $git commit -a --author "Ahmed ElArabawy <aelarabawy@gmail.com>"
  - User can specify the environment variables,
    - ***GIT_AUTHOR_NAME***
    - ***GIT_AUTHOR_EMAIL***
    - ***GIT_COMMITTER_NAME***
    - ***GIT_COMMITTER_EMAIL***
  - User can configure the author/committer info on the machine/user/project using "***git config***" command
  - Git will collect the info from the ***EMAIL*** environment variable
  - Git will collect the info from other places by querying the kernel

Configuring User Identities (git config Command)

- The file can be configured manually or via the commands

  $git config --global user.name "Ahmed ElArabawy"

  $git config --global user.email "aelarabawy@gmail.com"

- The use of ***--global*** makes the configuration in ***~/.gitconfig***
- If ***--global*** is replaced by ***--system***, configuration will apply on the whole machine, and stored in ***/etc/gitconfig***
- If ***--global*** is not used, the configuration applies only to the project in hand, and it is written in ***./.git/config***

Configuring the Editor/Diff tool

- By default, **Git** will be using the default editor (***$EDITOR***)
  - This can be customized by,

    $ git config --global core.editor vim
- The same applies for the diff tool

  $ git config --global merge.tool kdiff3

  Note that the following diff tools are acceptable:

  kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, opendiff
- Note that ***"--global"*** can be replaced by ***"--system"*** or completely removed as discussed earlier

Checking the saved Settings

- This can be done via browsing the configuration file directly
- Another way is to check the applicable settings on a specific project,

  $ git config --list

  - Git will display all settings in all the files, and hence the same key may show up multiple times, git will be using the last value
- To check for a specific key

  $ git config <key>

  $git config user.name

Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com