Linux For Embedded Systems

For Arabs

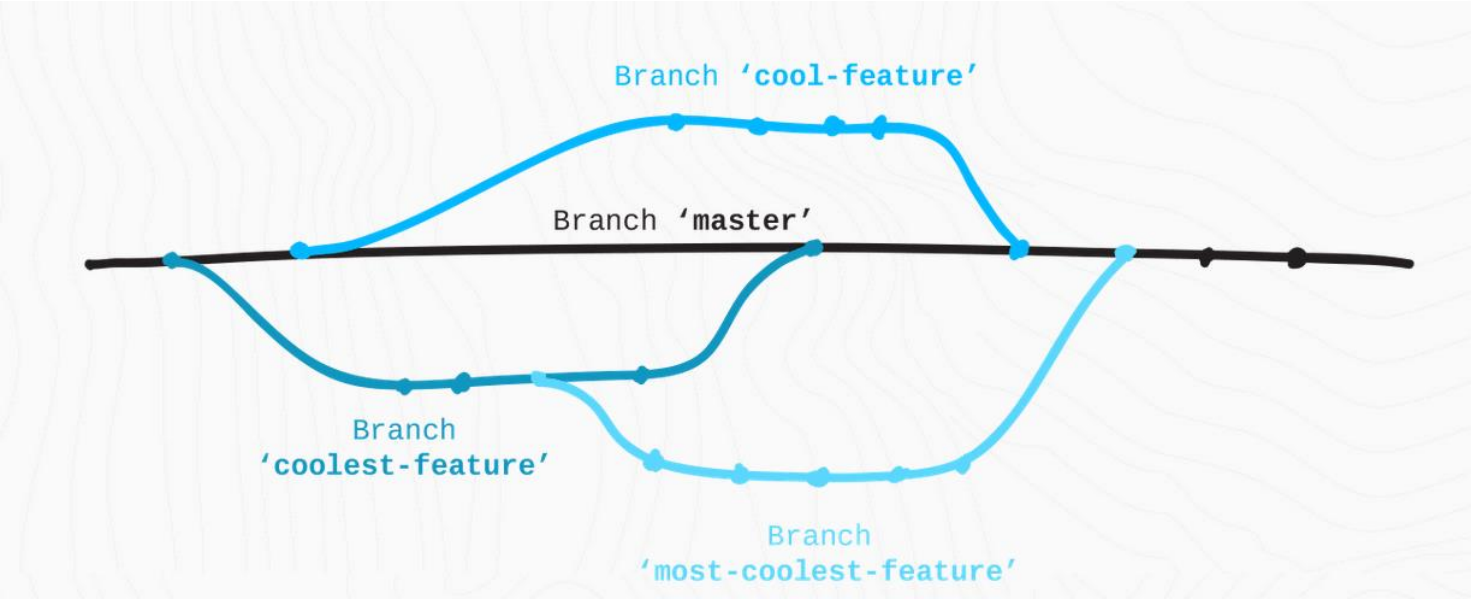
Cairo University
Computer Eng. Dept.
CMP445-Embedded Systems

Ahmed ElArabawy

Lecture 12:
Introduction to Git & GitHub (Part 3)

Working with Branches

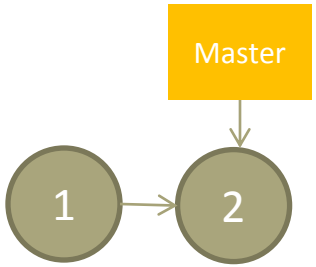
Branches



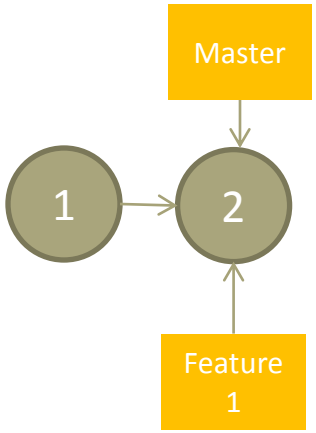
- Branches are used for:
  - Building different features independently
  - Separating between stable code and the code under development
  - Separating between Code releases
- Branches can be created or merged

Branches in Git

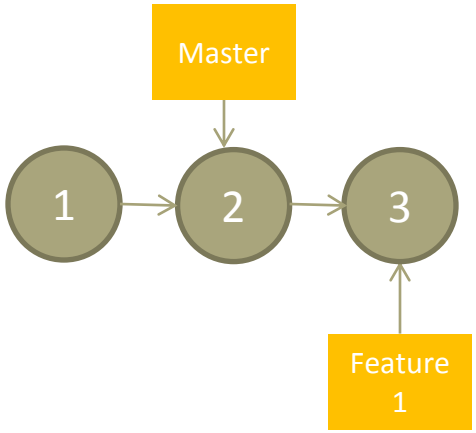
Branches in Git



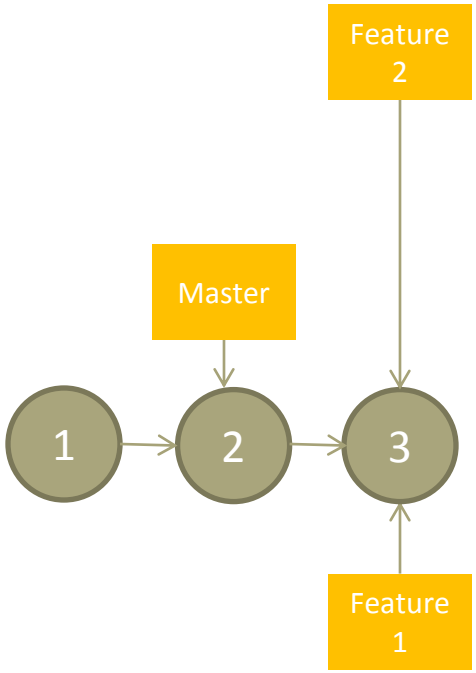
Branches in Git



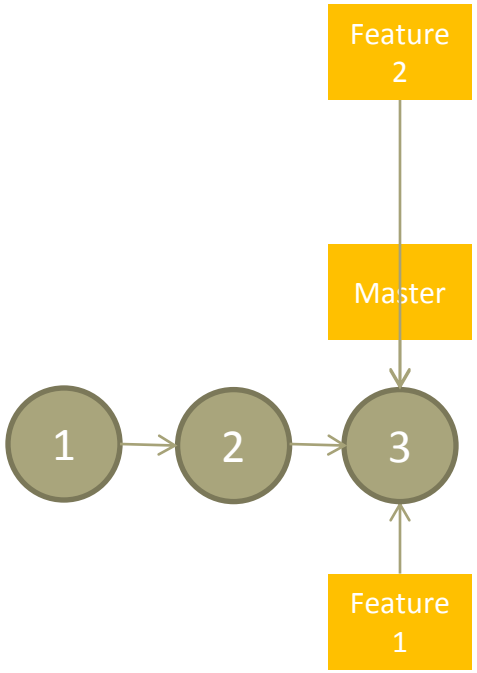
Branches in Git



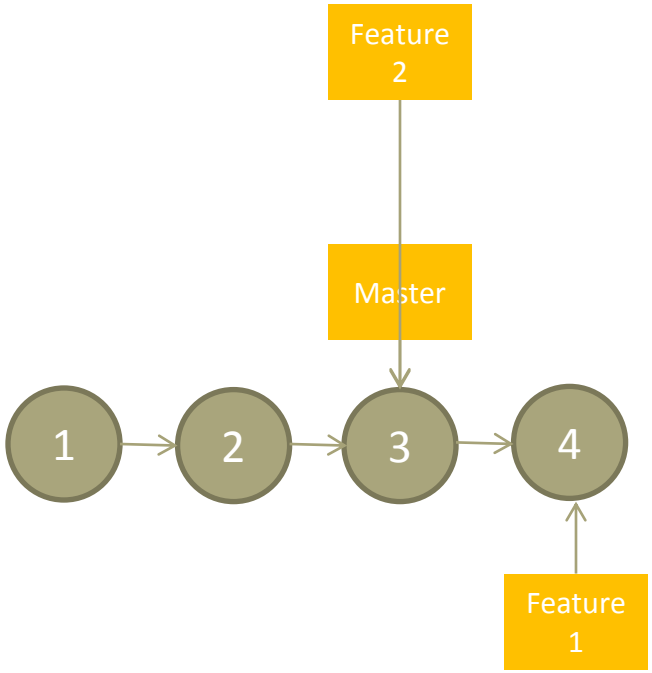
Branches in Git



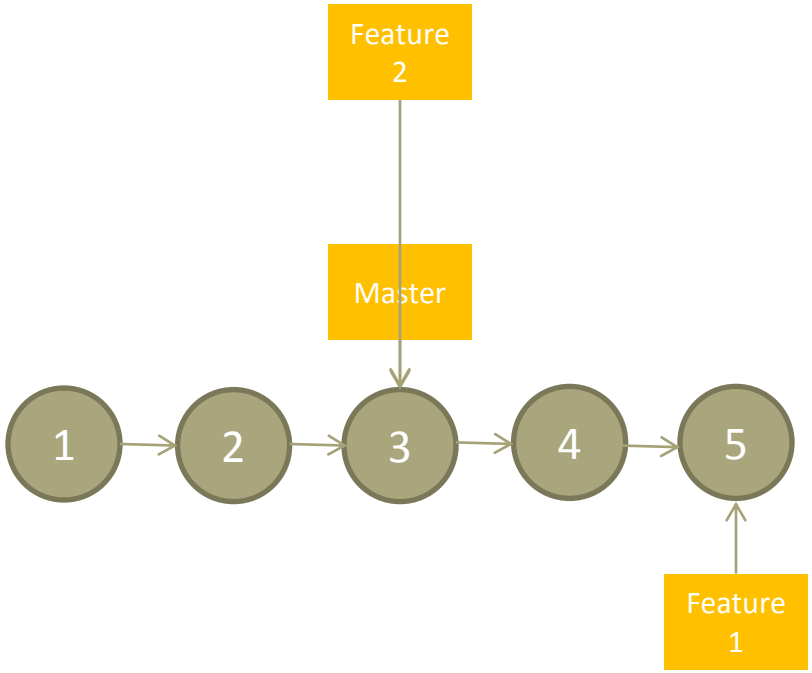
Branches in Git



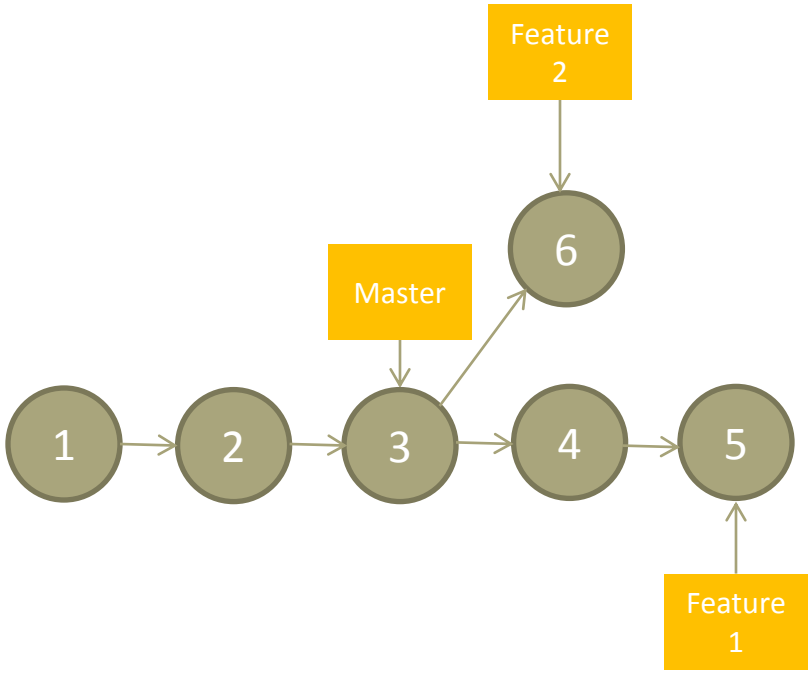
Branches in Git



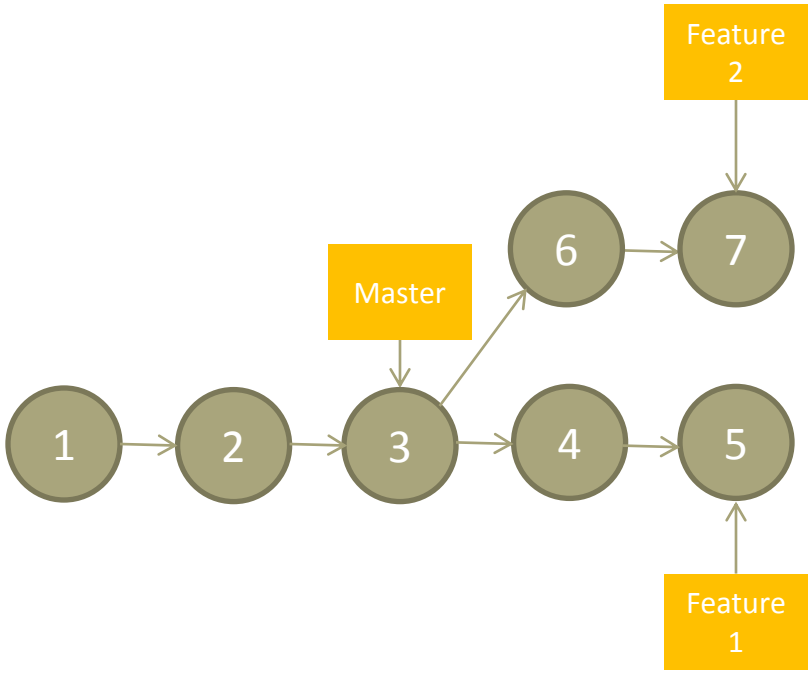
Branches in Git



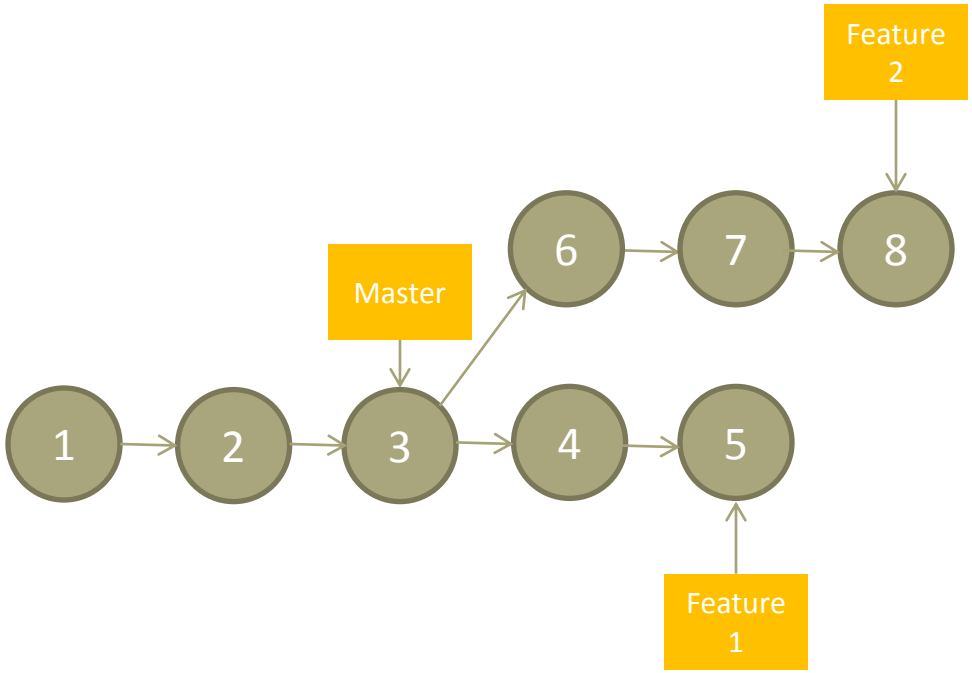
Branches in Git



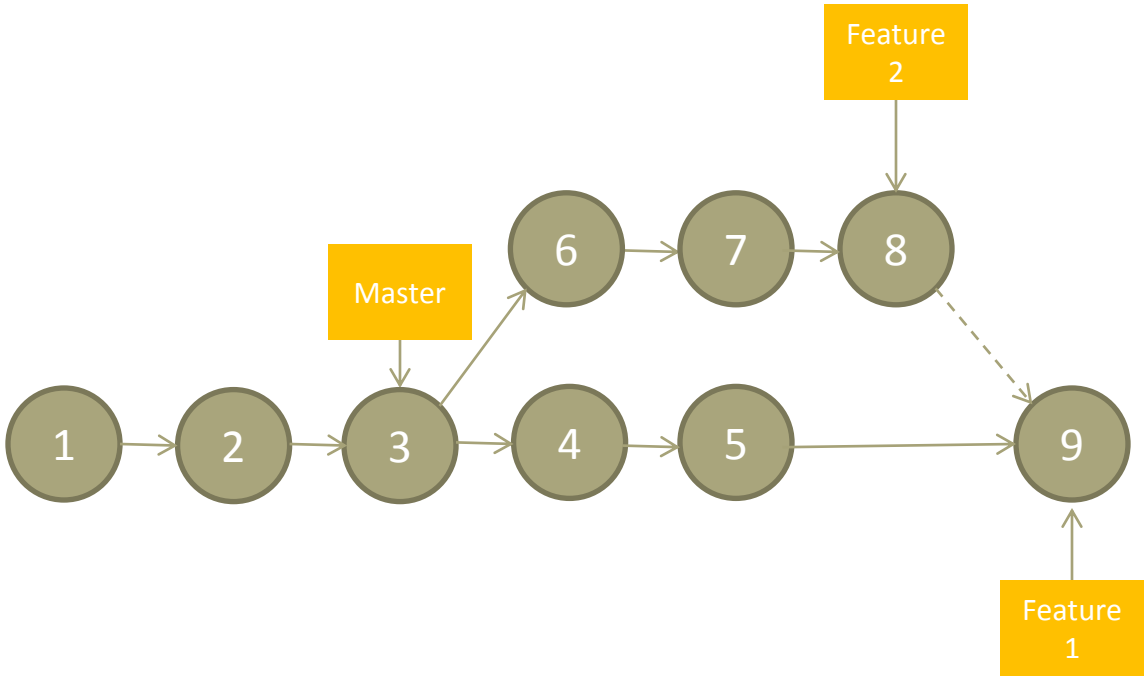
Branches in Git



Branches in Git



Branches in Git



Branches in Git

- Branches in **Git** are some sort of labels that point to a Commit
- We can have multiple branches that point to the same commit
- When creating a branch on a certain commit, this means we create a label that points to this commit
- As more commits are created the branch label moves to the ***Tip*** of the branch (the latest commit for the branch)
- If one branch is just ahead of another branch with some commits, then merging the two branches is called ***fast forwarding***
- After the merge, both branches labels point to the same commit
- If however 2 branches have different commits, then merging is not a simple fast forwarding, and it involves comparing the differences and taking proper changes from each branch
- If modifications are in the same files and same parts of the code, then merging will result in conflicts that will need to be resolved manually

Creating Branches

- The initial repo comes initially with the "***master***" branch
- To create another branch

  $ git branch <new branch name>

  $ git checkout -b <new branch name> (this creates the branch and switches to it)

  $ git branch <new branch name> <start point> (create a branch at a certain commit)
- To list all branches, and know the current branch

  $ git branch
- To switch between branches

  $ git checkout <branch name>
- To delete a branch

  $ git branch -d <branch name> (checks on un-merged commits)

  $ git branch -D <branch name> (deletes the branch even if some commits are unmerged

  Note that you can not delete a branch while currently on it

Comparing Branches

- To compare tips of two branches

  $ git diff master..new-branch

- To compare between the tip of a branch and the point it diverged from another one

  $ git diff master…new-branch

- To compare work directory with a branch

  $ git diff <branch name>

- To limit the comparison to a file, or a folder

  $ git diff <branch name> -- <file name or folder name>

- To limit the diff to just the statistics (file count)

  $ git diff --stat <whatever we are comparing>

Merging Branches

- After making changes and committing it to a branch, and need to merge it in another branch
  - Switch to the merge-into branch

    $ git checkout <branch 1>
  - Merge the modified branch into this branch

    $ git merge <branch 2>
- Outcome of merge can be one of three,
  - If the **branch_1** had no commits and **branch_2** is simply ahead of **branch_1** with some commits, the merge happens automatically, and no merge commit is performed, the HEAD of **branch_1** simply moves forward, to coincide with the HEAD of **branch_2**, this is called "***fast forward***"
  - If both branches have commits, and no conflicts exists, the **HEAD** is fast forwarded, followed by a merge commit to mark the merge
  - If both branches have commits and a conflict exists, a message will show and the conflicts are marked in the files in the working directory. Will need to make corrections, and then do a regular commit using for example,

    $ git commit –a
  - The commit, will show as a merge between two commits
  - Note that if a merge has a conflict, the index is also put in a state that will make any trial to do a commit fail, with a warning listing the files that needs to be merged. The same list will also show when showing status

Undoing a Merge

- If the merge in the middle or resolving conflicts, and things start to get messy, and we want to undo the merge, do

  $ git reset --hard HEAD

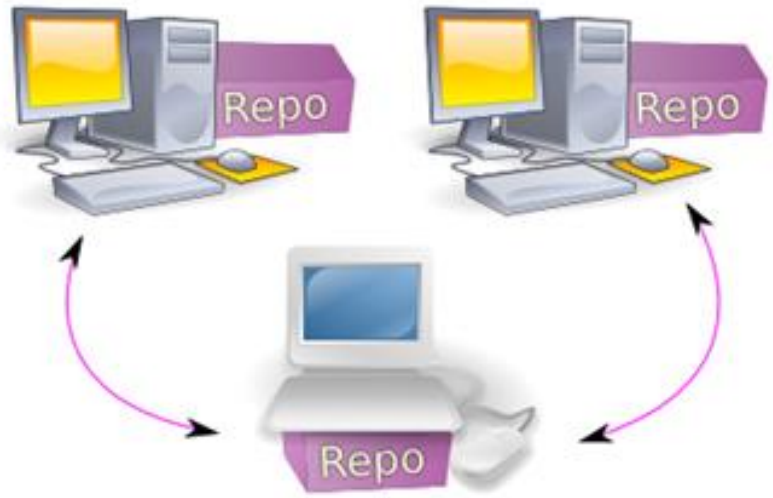
- If merge is complete, and committed, and we want to undo it (remove the merge commit),

  $ git reset --hard ORIG_HEAD

  Note: Don’t use this command if this commit is used in another merge

Working With Remotes

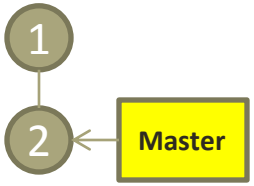
What is a Remote



- The repo running on a local machine may be tracking other repos for the same code running on different machines
- At any point in time, the user can synchronize its local repo with remotes
- This means pulling changes done in remote repos, or pushing its changes into them
- Keep in mind that this is a synchronization between repos and does not involve the working directory

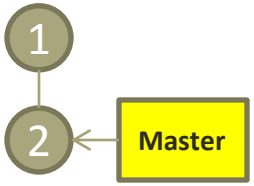
Remotes in Git

- Git manages the remote repos by creating a branch for each tracked remote repos in the local repo to map the repo in the remote machine
- This means that when you clone a project from a remote repo, your local repo will start with two branches
  - Local **Master** Branch (this is the branch that will contain all of your local commits)
  - **Origin** Branch (this is the branch that will track updates in the remote machine that the project was cloned from)
- User can add more remotes (link to remote repos)
  - This will internally create a branch for each remote repo
- Note that the remote branches are not automatically synchronized with the remote machine. User will have to execute a command for this to happen

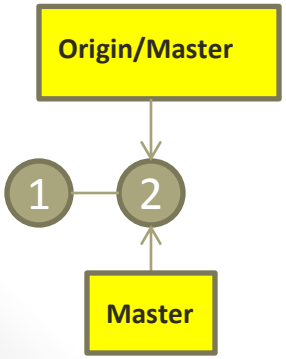

1
2
Master

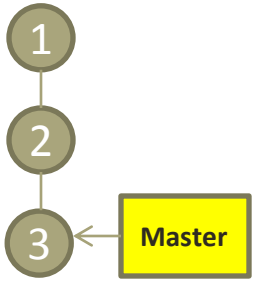

Repo


Repo

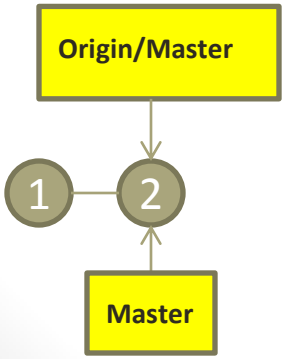


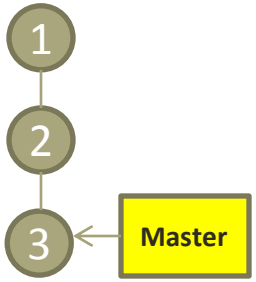
Origin



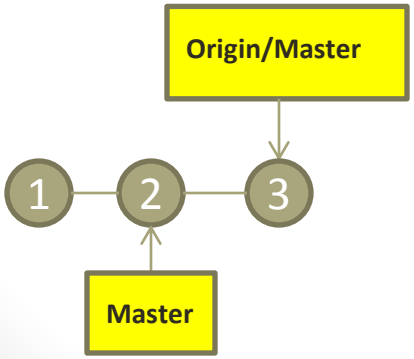


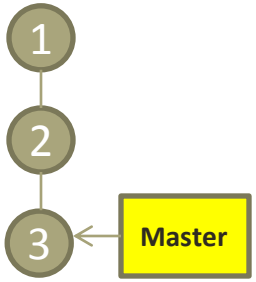
Origin



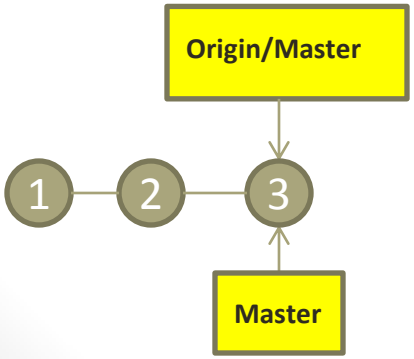


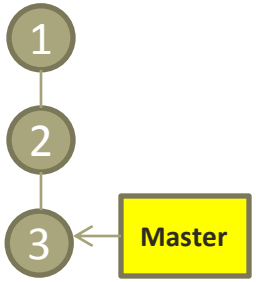
Origin



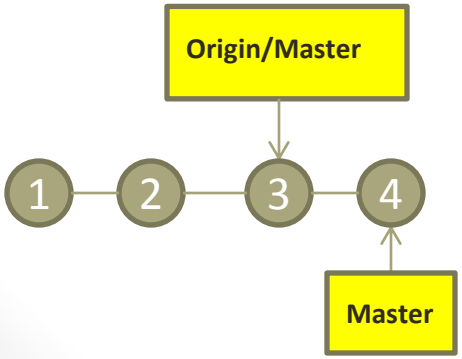


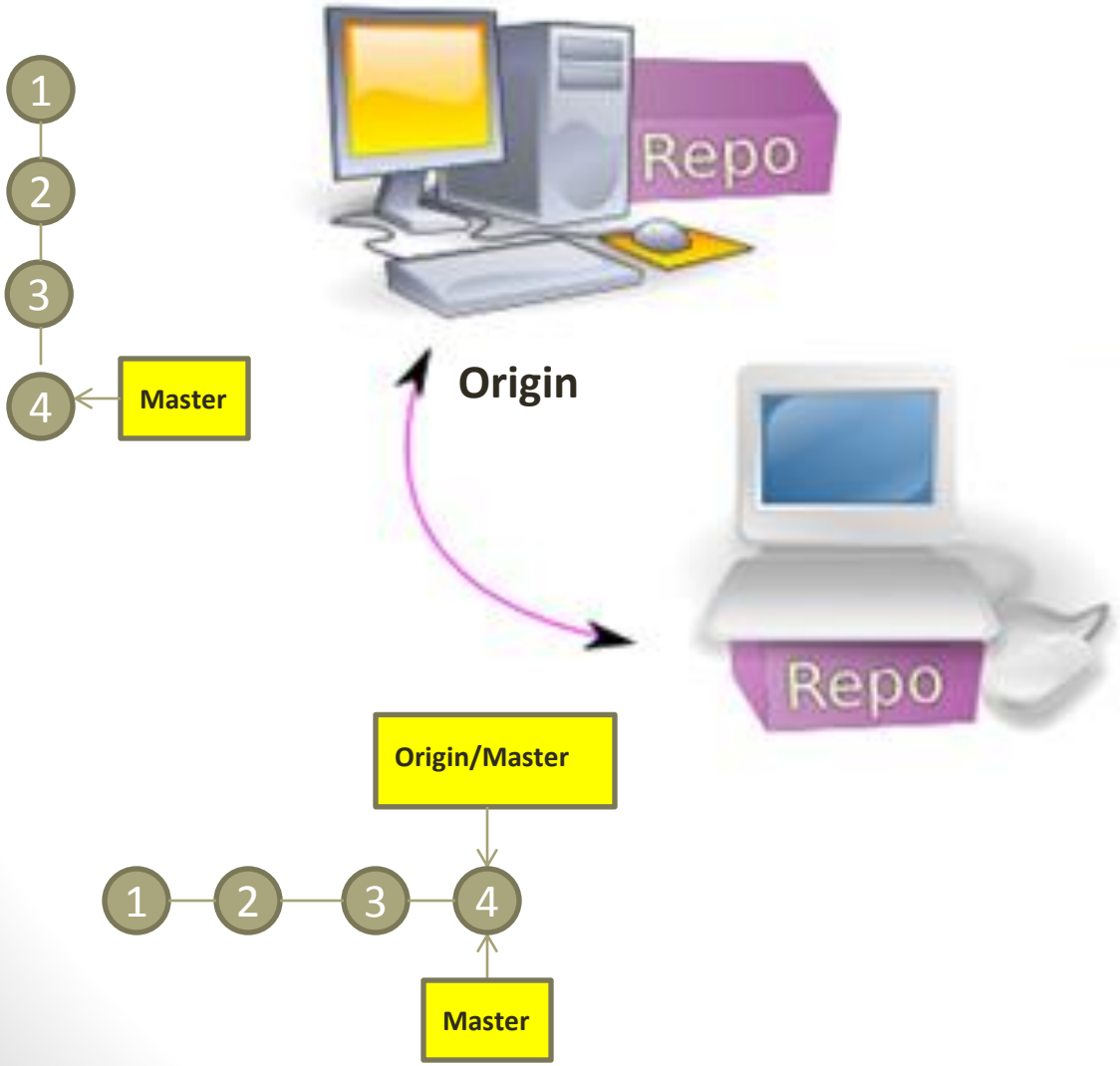
Origin

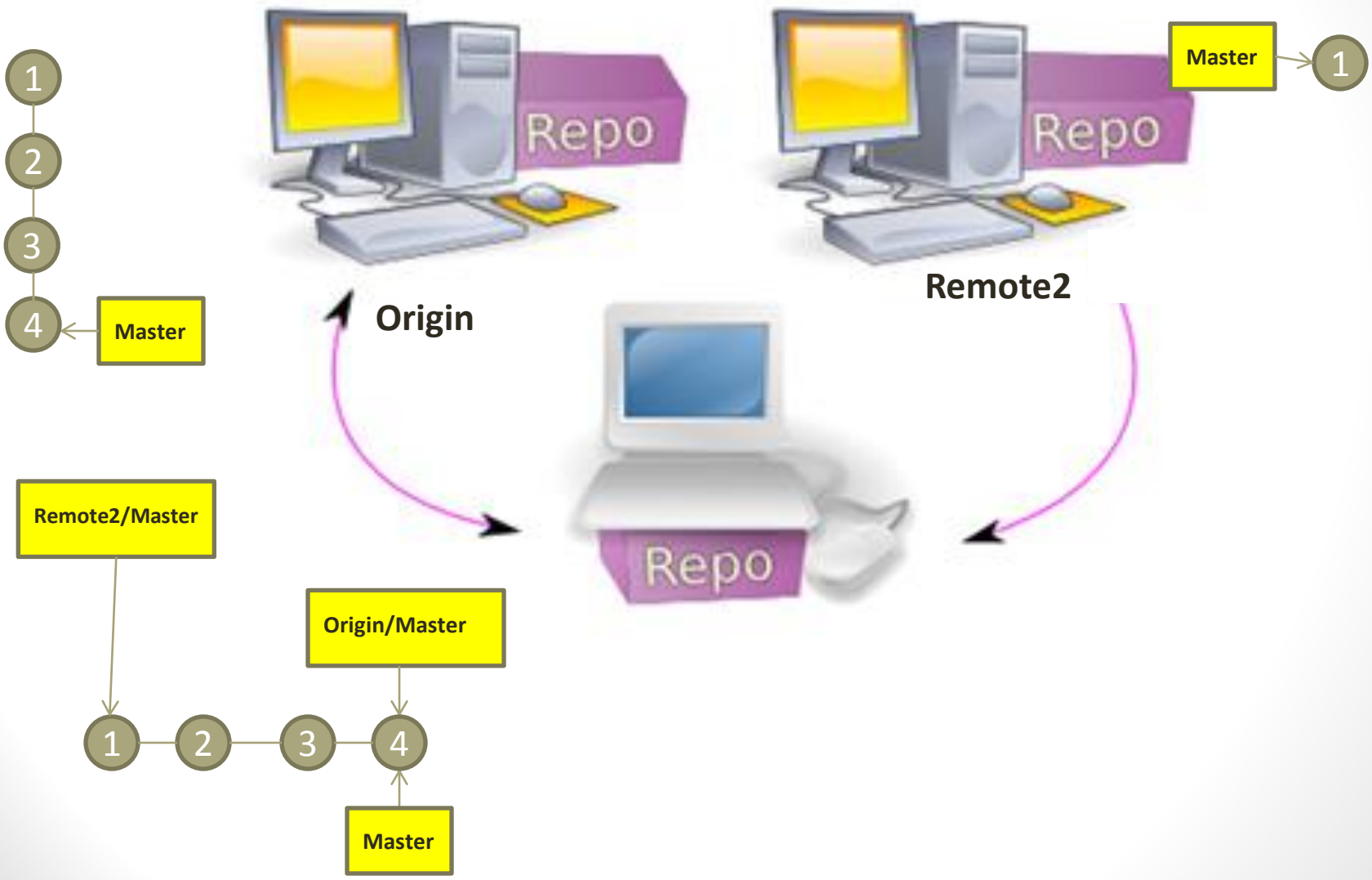


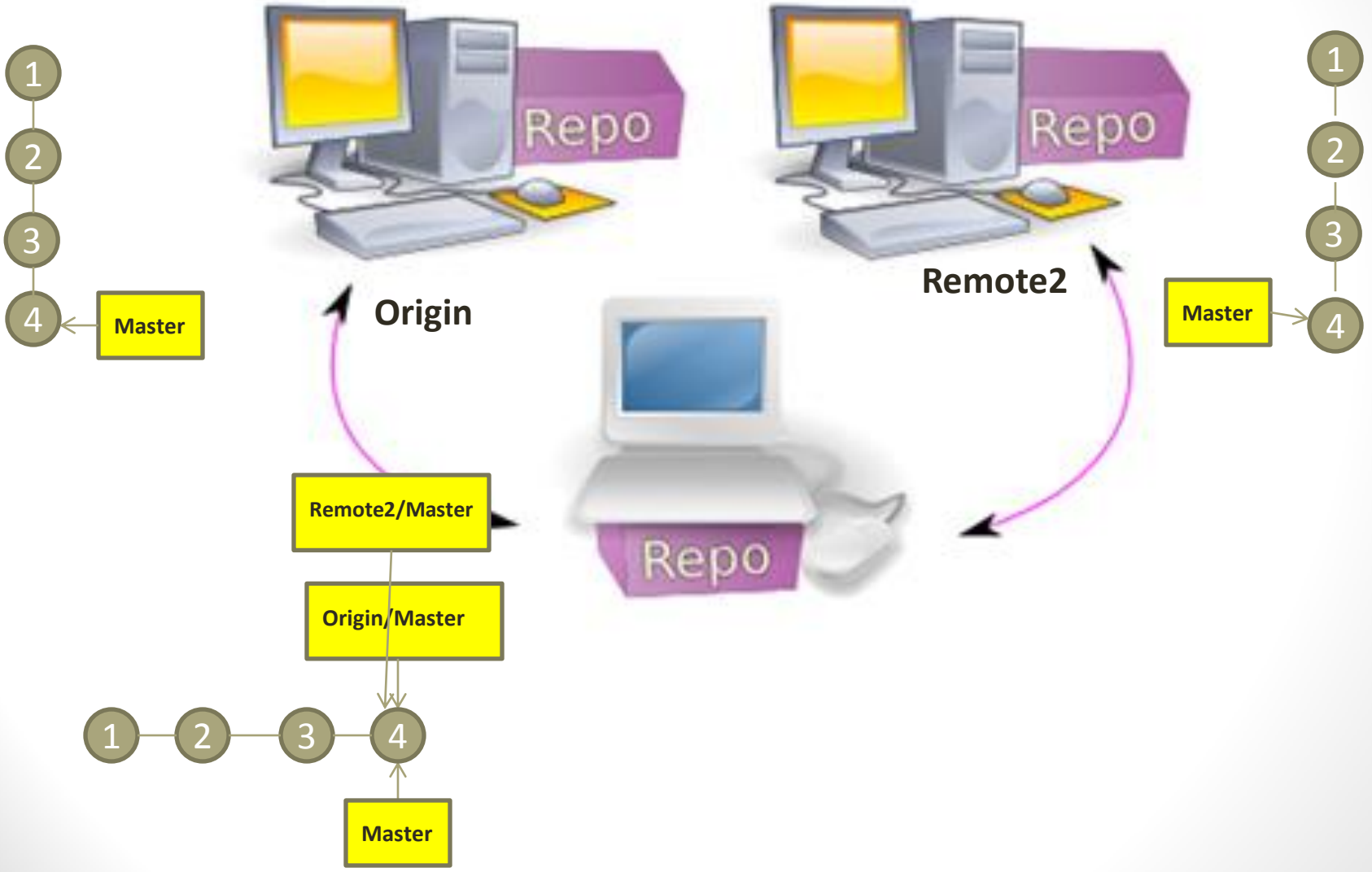


Origin




1
2
3
4
Master
Repo
Origin
Repo
Origin/Master
1
2
3
4
Master


1
2
3
4
Master
Repo
Origin
Repo
Master
1
Remote2
Repo
Remote2/Master
Origin/Master
1
2
3
4
Master


1
2
3
4
Master
Repo
Origin
Repo
Remote2
1
2
3
4
Master
Repo
Remote2/Master
Origin/Master
1
2
3
4
Master

Remote Repos

- Remote repos are the repos for the same project running on different hosts
- Most common remote is the "***origin***" which is the repo that was cloned by the user, origin is automatically tracked by the master branch of the repo
- User can alias the remote repos, to facilitate **fetch**, **pull**, **push** changes from/to them
- Managing remotes include,
  - List remotes
  - Adding a remote
  - Removing a remote
  - Tracking a remote
  - Fetch a remote

Managing Remotes

- To have a list of remotes,

  $ git remote

  $ git remote -v (prints the URL for each remote as well)

- To add a remote,

  $ git remote add <alias name> <url>

  $ git remote add team1 git://github.com/team1/code.git

- To rename a remote,

  $ git remote rename <old name> <new name>

- To delete a remote,

  $ git remote rm <remote name>

Fetching from a Remote

- To fetch changes of the remote,

  $ git fetch origin
- Note that if no remote name is mentioned, then the tracked remote is assumed
- When cloning a project, the master branch is automatically set to track “origin”

  $ git fetch
- When just specifying the remote repo, all branches in the remote are fetched, if we want a specific branch, then it needs to be specified

  $ git fetch origin master
- Fetching means copying all objects, and commits from the remote repo, and pointing to it via a fixed reference
- Fetching does not affect the working directory, or the current branch. It only updates the remote branch with its commits
- To merge the remote after fetching it,

  $ git pull <remote>

Sharing Changes

Pulling changes from the original repo

- To pull a change in the original repo into the current branch of the cloned repo (and into the working directory as well)

  $git pull

- You can specify source, and branch from which to pull

  $ git pull <url> <branch name>

- You can set a URL as a remote, to use it in later pulls as an alias

  $ git remote add <name of remote> <url>

- All of this result in automatic pull without the chance to review the changes
- Pull will fail if it does not result in fast forward, in this case, we will need **fetch/merge**

Review Before Pulling

- Another option, to do it in multiple steps
  - First get the changes, in a remote,

    $ git fetch <url> <branch name>

    $ git fetch (no need to mention url if it is the origin)

    $ git fetch <remote>
  - Compare the two branches,

    $ git diff master..<remote>/<remote branch name>

    $ git diff master..orig/master
  - Do the merge

    $ git merge <remote>/<remote branch name>

Push Changes

- Sometimes, there is a public server where everybody push their changes to

  $ git push <url> <remote branch name>

- We can have an alias for the url as a remote (same as for pull)
- Pushing changes may fail if it does not result in a simple fast forwarding, in this case, the user will need to pull (or ***fetch + merge***) first, then push his changes

Working With Tags

Lightweight Tags

- Lightweight tags are just a reference to a commit
- A simple reference to a commit (with a more human readable name)
- It is like an alias, or a branch that never moves pointing to this commit
- It does not contain a tag object, so no signature or tag comment or description
- To create a lightweight tag

  $ git tag <tag name> <commit id>
- Then the tag can be used anywhere the commit id is needed

Annotated Tags

- An annotated tag containa a tag object with all associated info (description, tagger, date, ..)
- Can point to a commit (most often) or a tree
- The tag points to the tag object, which in turn points to the tagged object
- Check-summed and has an SHA-1 name
- Can be GPG (GNU Privacy Guard) signed
- To create a tag object,

  $ git tag -a <tag name> <object Id>
- In this case, a tag object is created, and the tag points to it instead of the commit itself
- Although tagging a commit is the most common, but any object can be tagged this way
- Tags created this way, are not signed (signed tags are out of our scope now)

Managing tags

- To list the existing tags

  $ git tag

  $ git tag -l 'v1.3.*' (lists only tags with the pattern v1.3.*)

- To create a light-weight tag

  $ git tag <tag name> <commit id>

  $ git tag <tag name> (to tag the HEAD)

- To create an annotated tag, just use the '-a' option

  $ git tag -a <tag name> <commit id> (an editor will open to take the tag description)

  $ git tag -a <tag name> -m <tag description message>

- To show an annotated tag details

  $ git tag show <tag name>

Sharing Tags

- When pushing changes to a server, tags are not pushed along with the contents, and commits
- This is smart, since a lot of these tags are for internal reference
- To push a specific tag with the content (of public nature)

  $ git push <remote> <tag name(s)>
- To push all available tags with the content

  $ git push <remote> --tags

GitHub

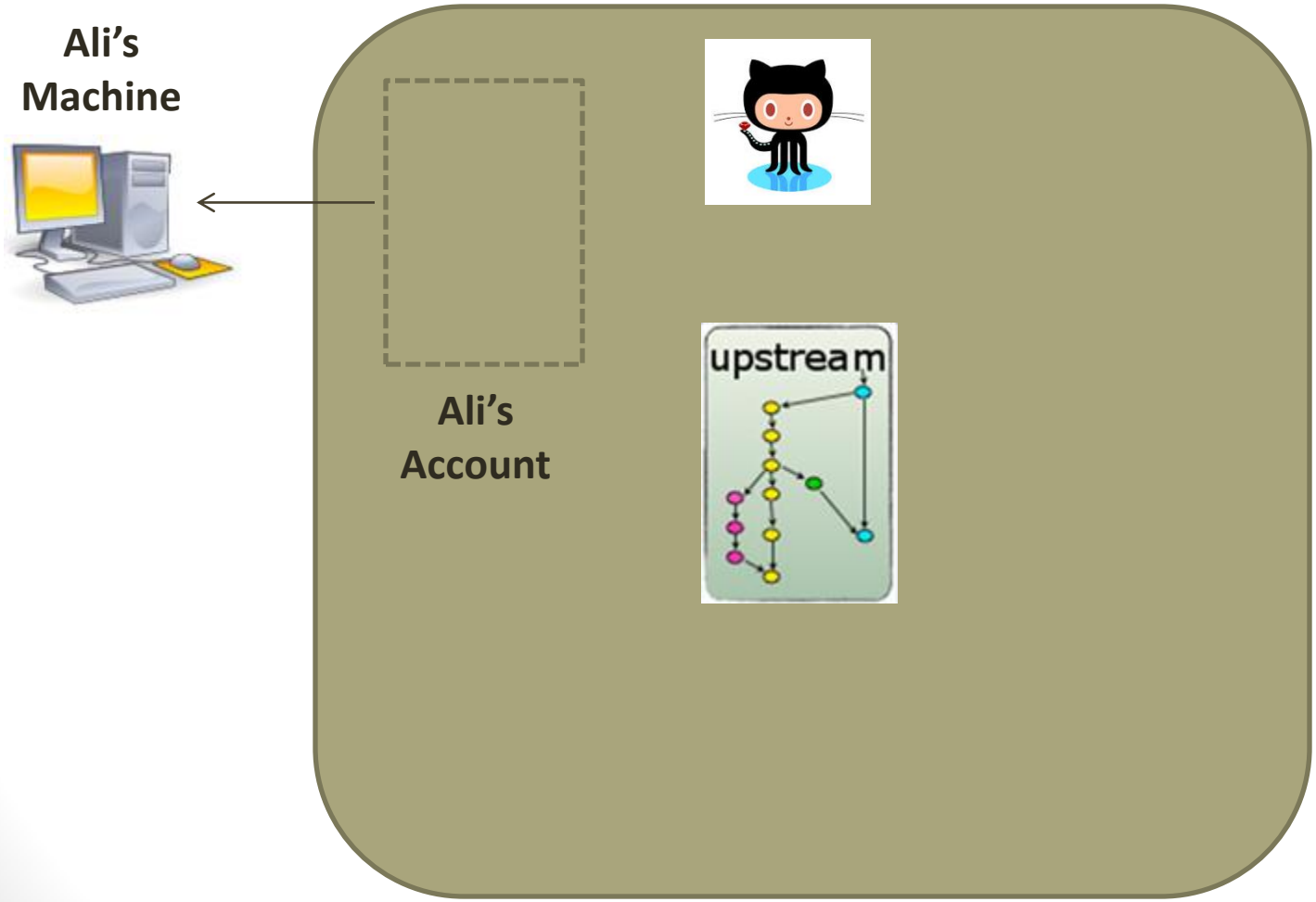
What is GitHub

- **GitHub** is a web-based Git repository hosting service
- It enables its users to have both public and private repositories
- Hosting public repositories is free (unlimited count) while the private repos require a paid account
- This enable users to :
  - Host their own repos so they can access them from different machines
  - Share a repo within a team for collaborated work
  - Fork other public repos to their account, and work on the forked copy
  - Changes made to repos that was forked can be proposed to be included in the original repo
- There are similar tools such as ***GitLab***, ***Gitorious***, …

Development Models

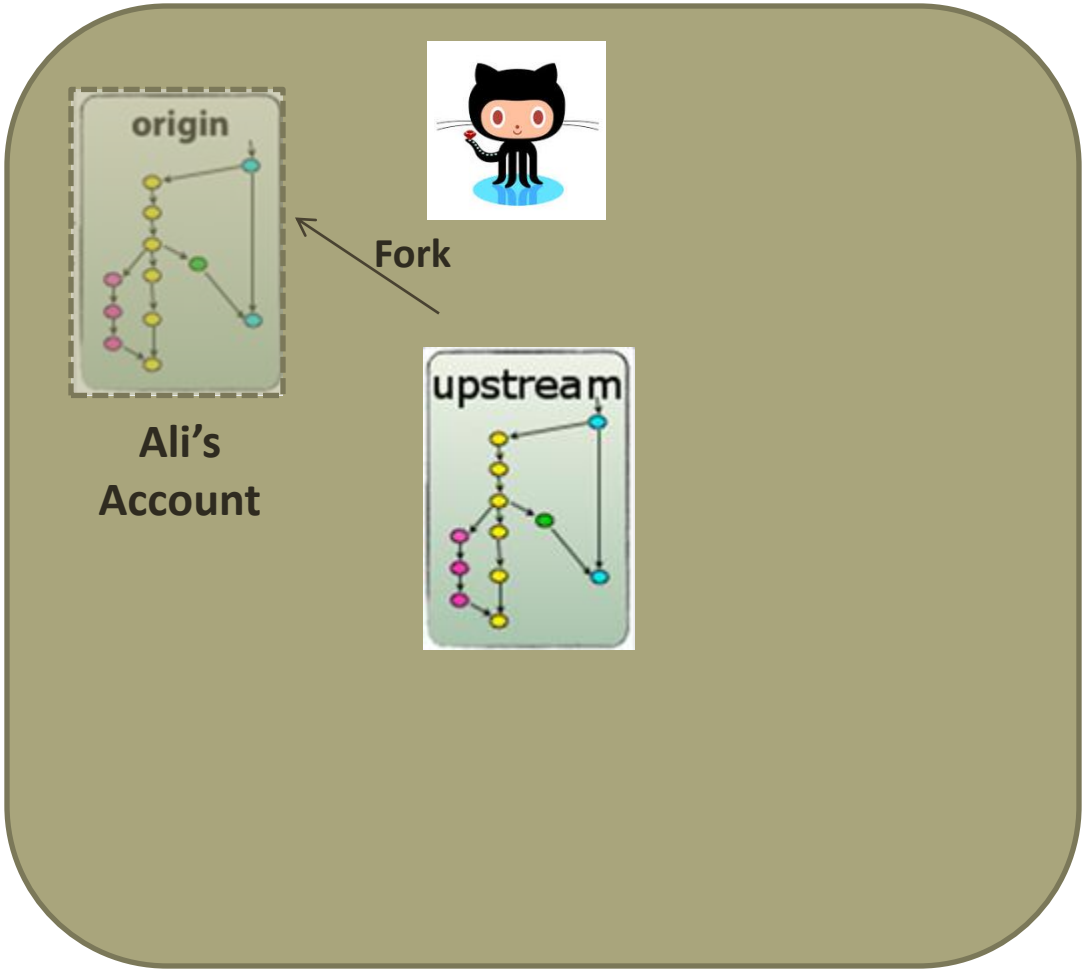
- There are two models of development with GitHub,
  - **Fork and pull**
    - Suitable for open source projects
    - Minimum friction
    - The user will fork the project repo, and push its commits to the forked repo
    - When there is room for contribution, the user will need to issue a ***pull request***
  - **Shared Repo**
    - More suitable to closed and private projects
    - Also useful for project shared with small group of developers
    - All share the same repo and push their contributions to it

Forking a Project

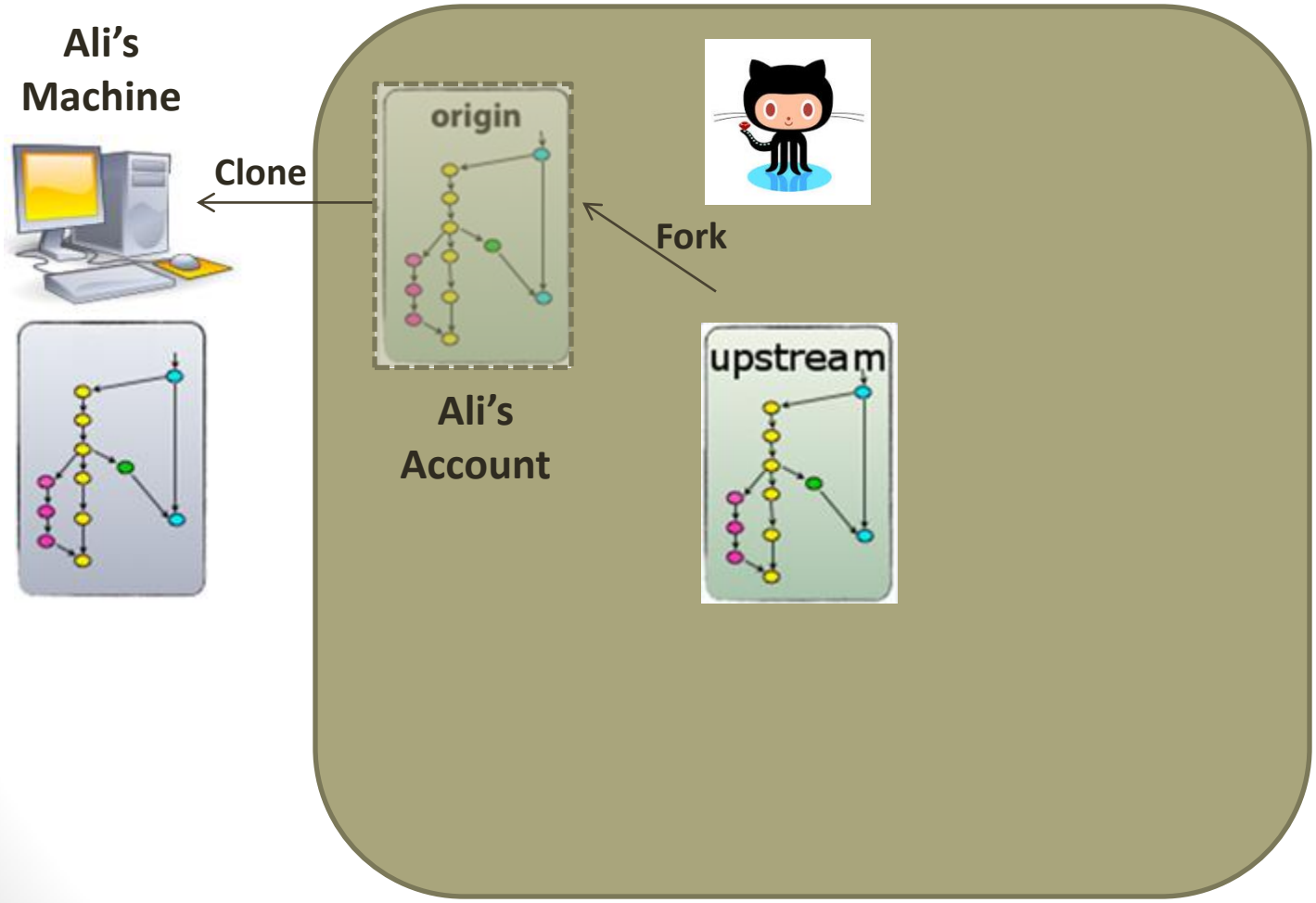


Forking a Project

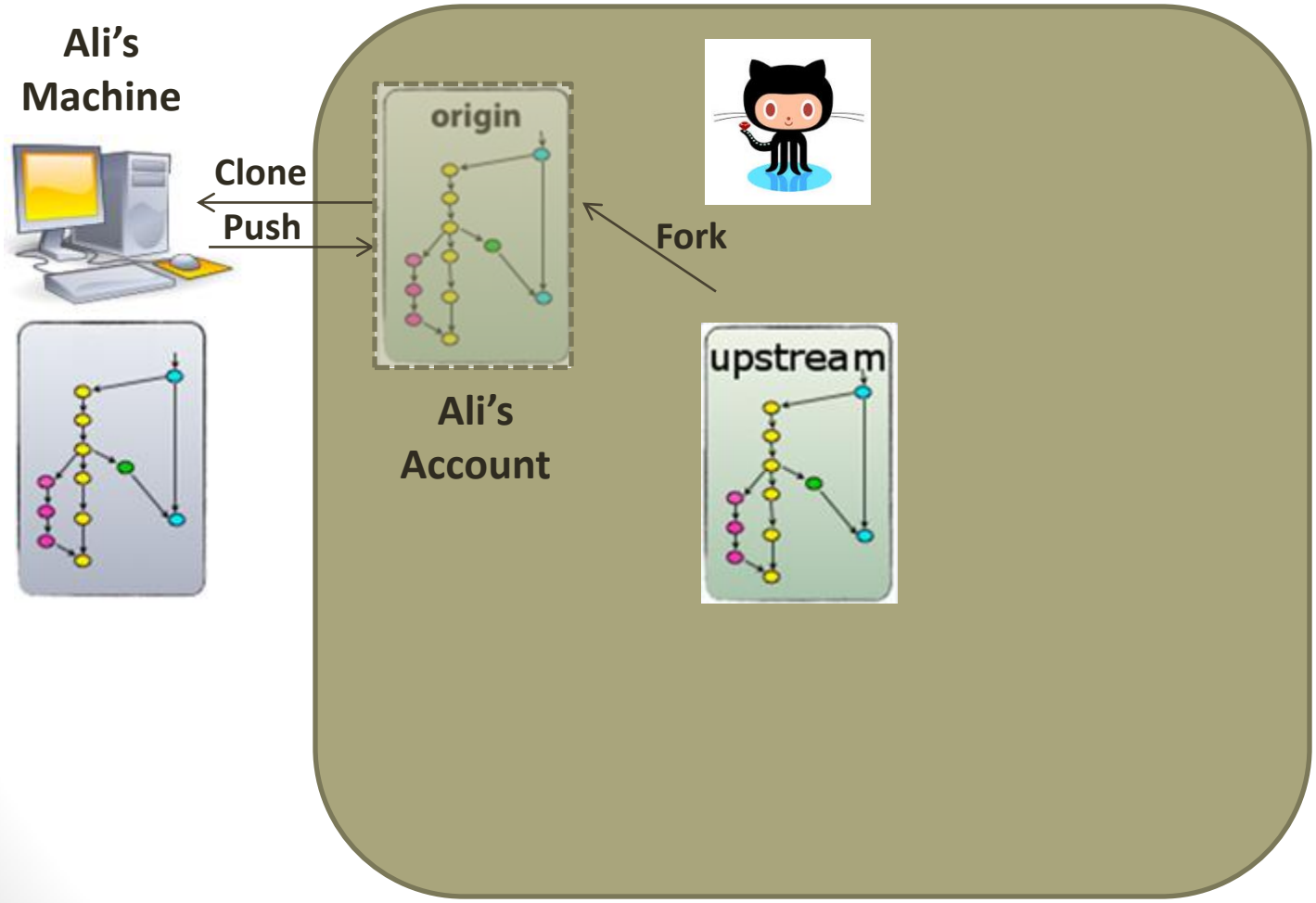
Ali's Machine



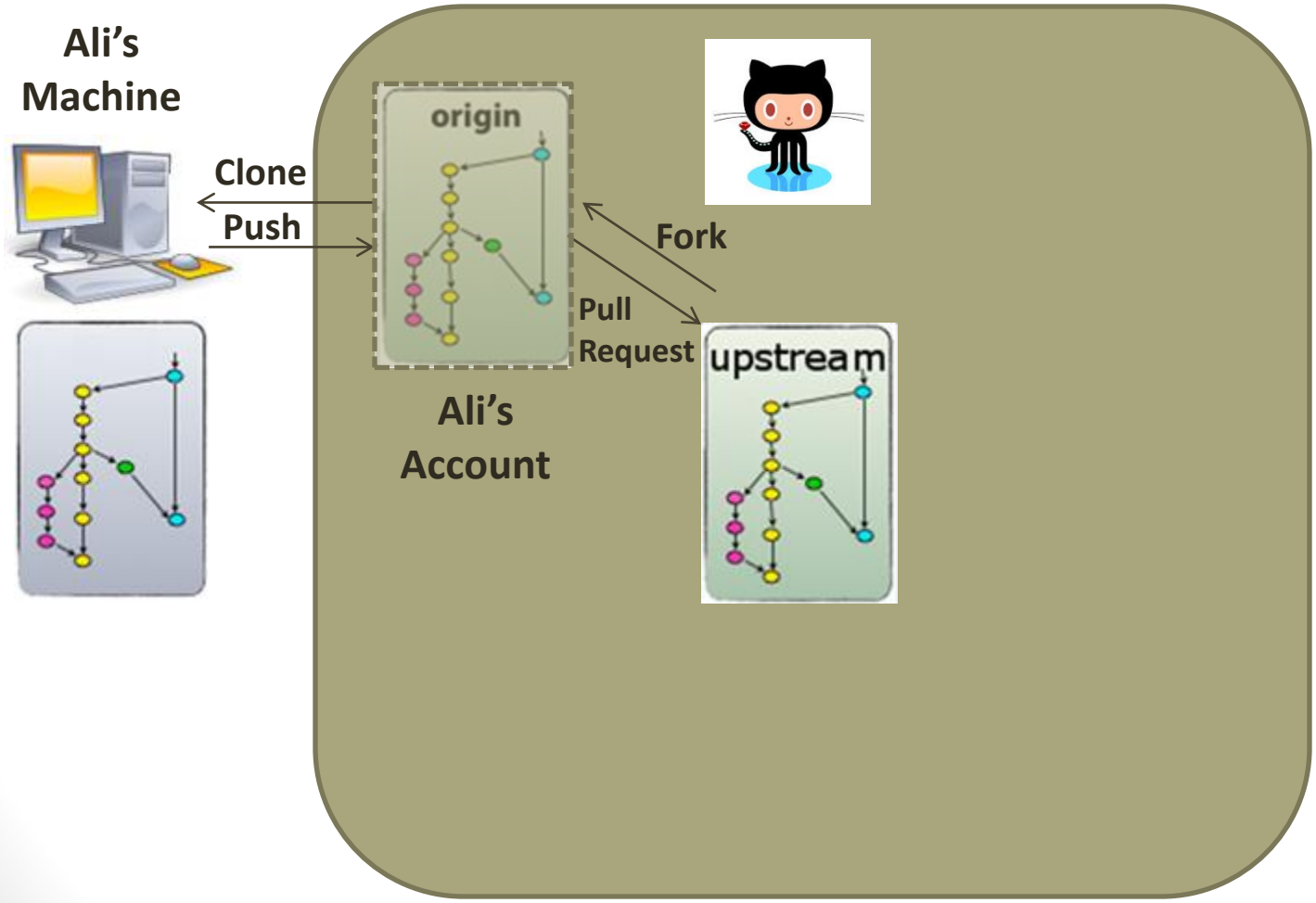
Forking a Project



Forking a Project



Forking a Project



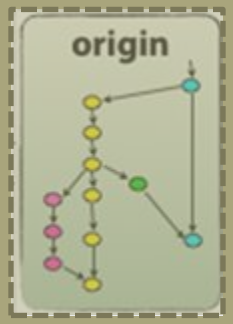
Sharing a Project

Ali's Machine

Project Account

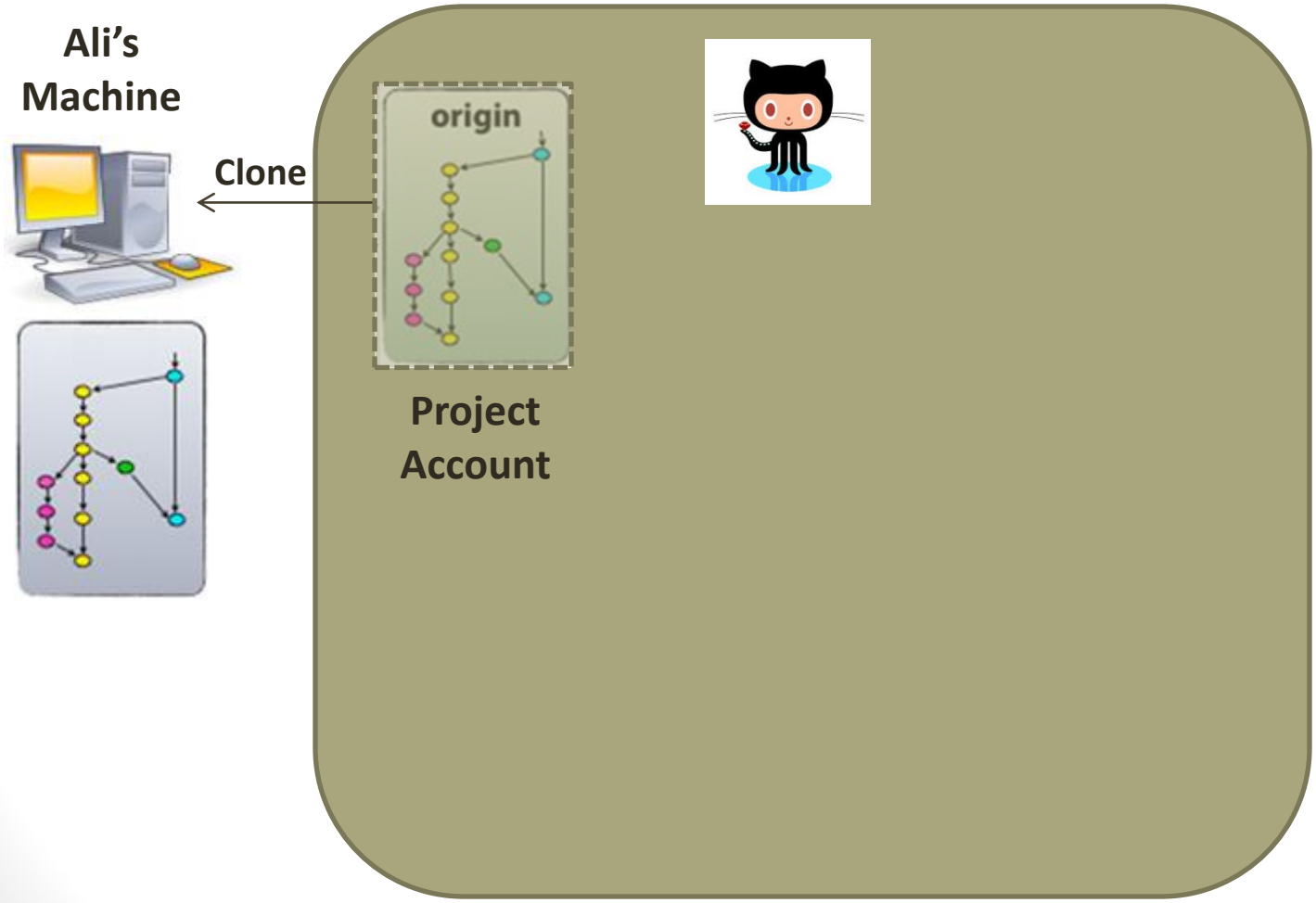
Sharing a Project

Ali's Machine

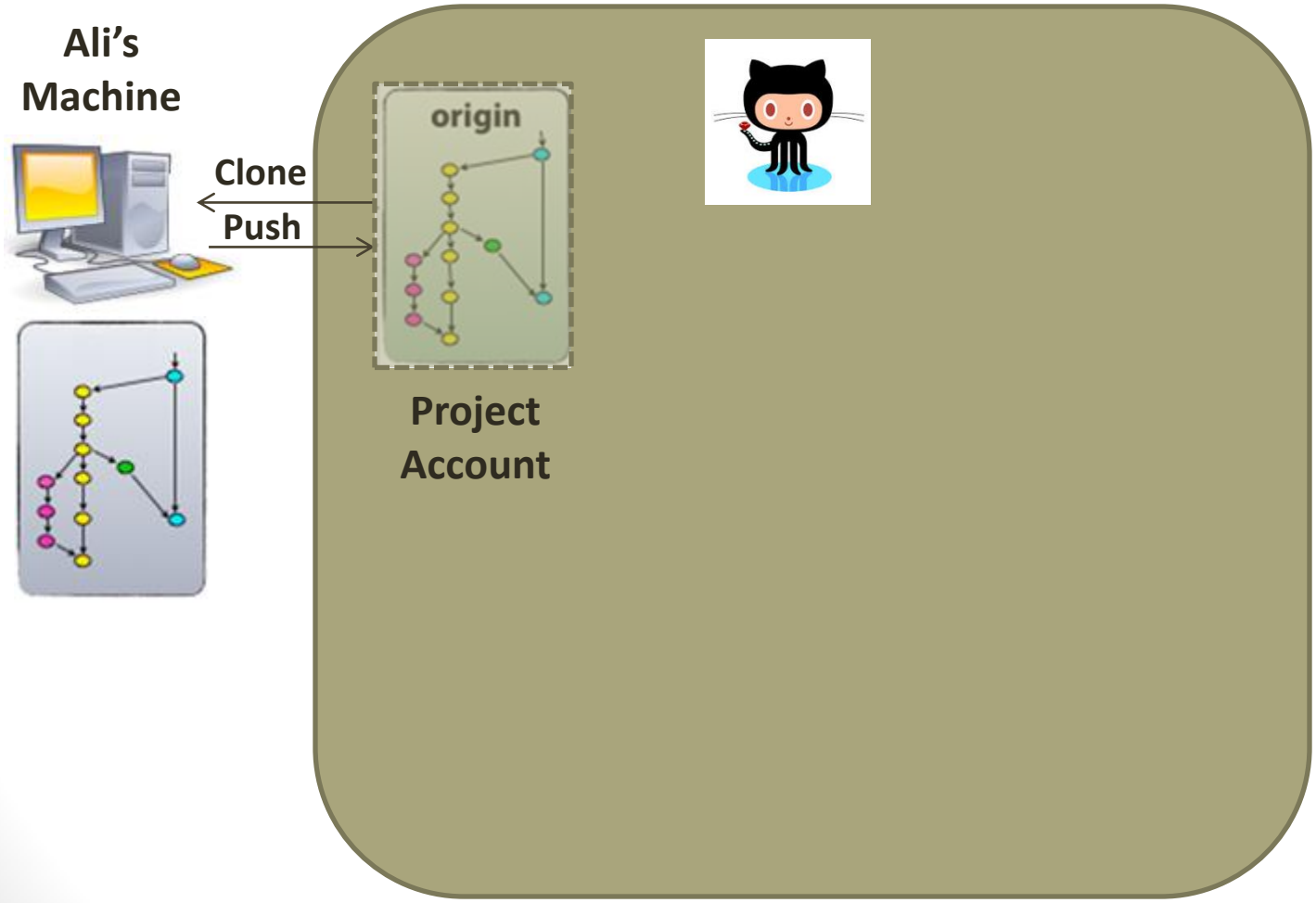


Project Account

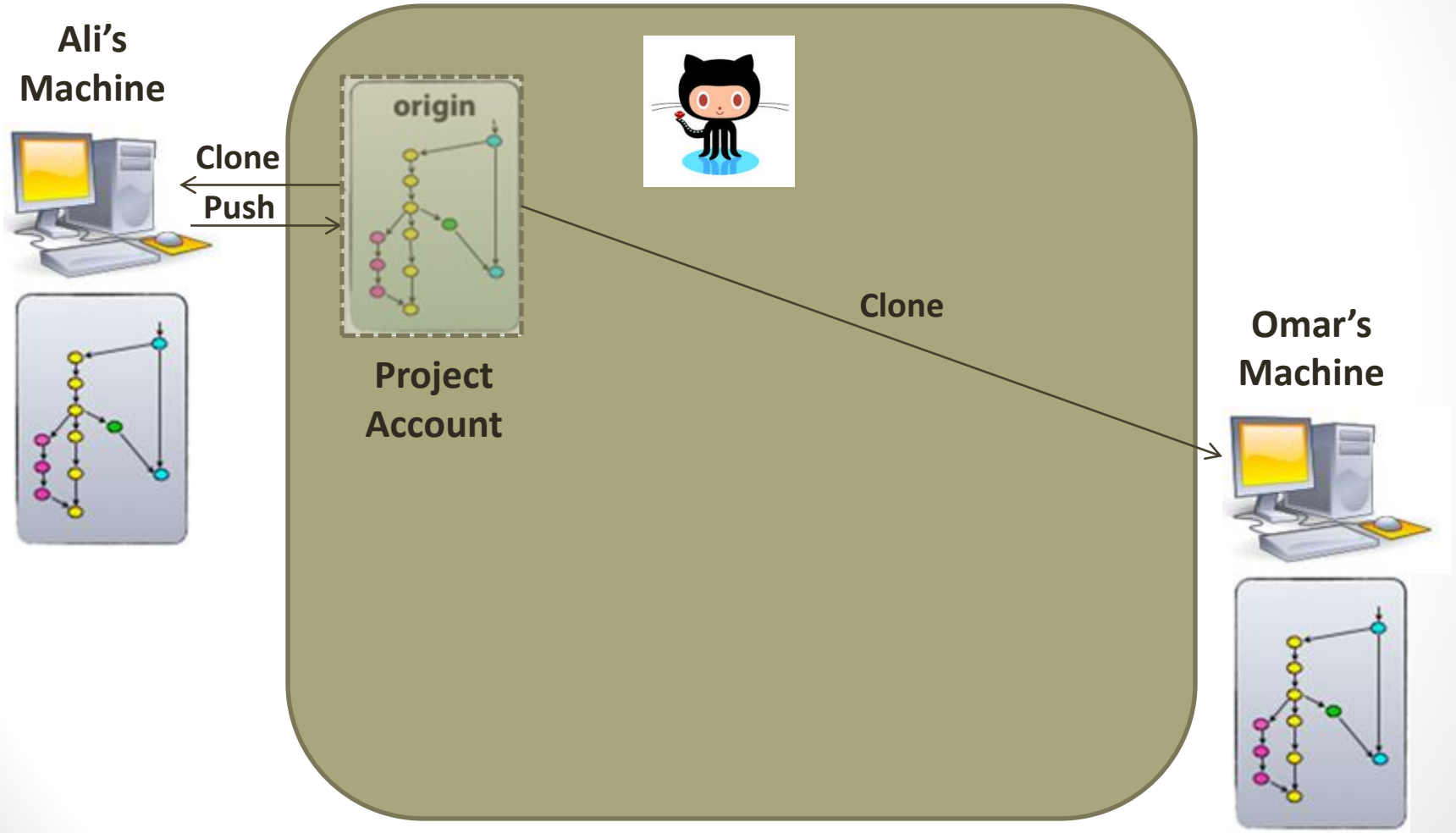
Sharing a Project



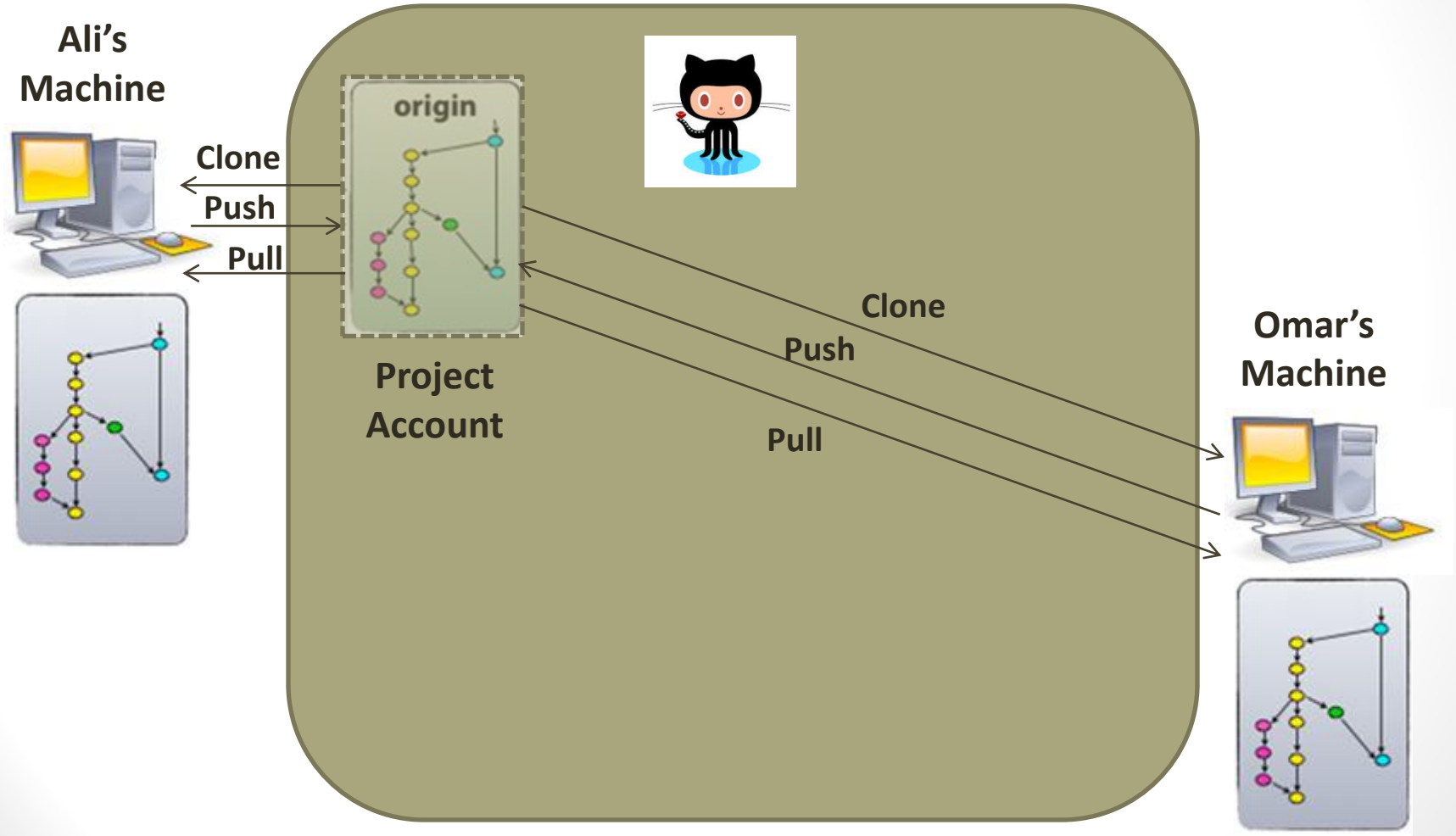
Sharing a Project



Sharing a Project

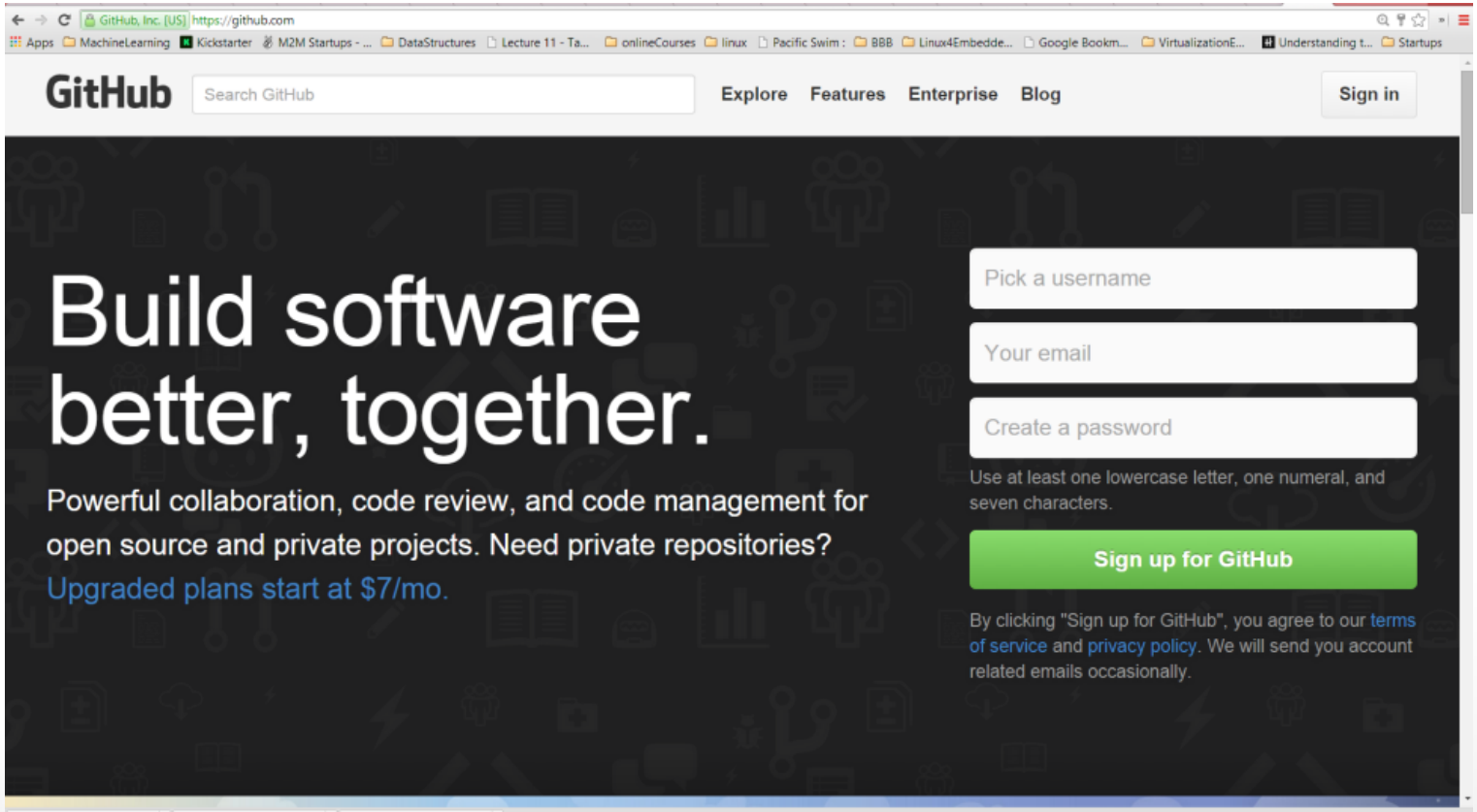


Sharing a Project



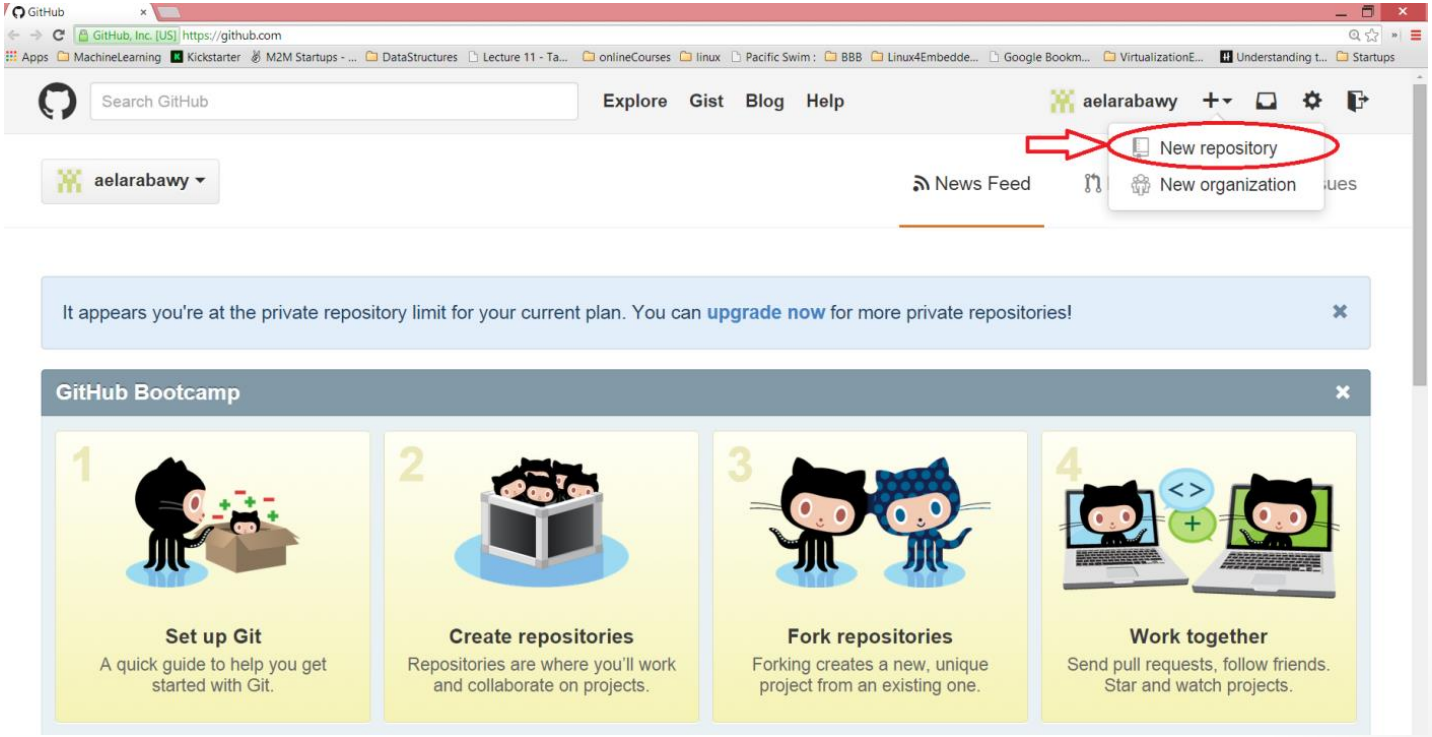
Preparation Steps GitHub Account

- The following steps need to be taken:
  - Create a GitHub account



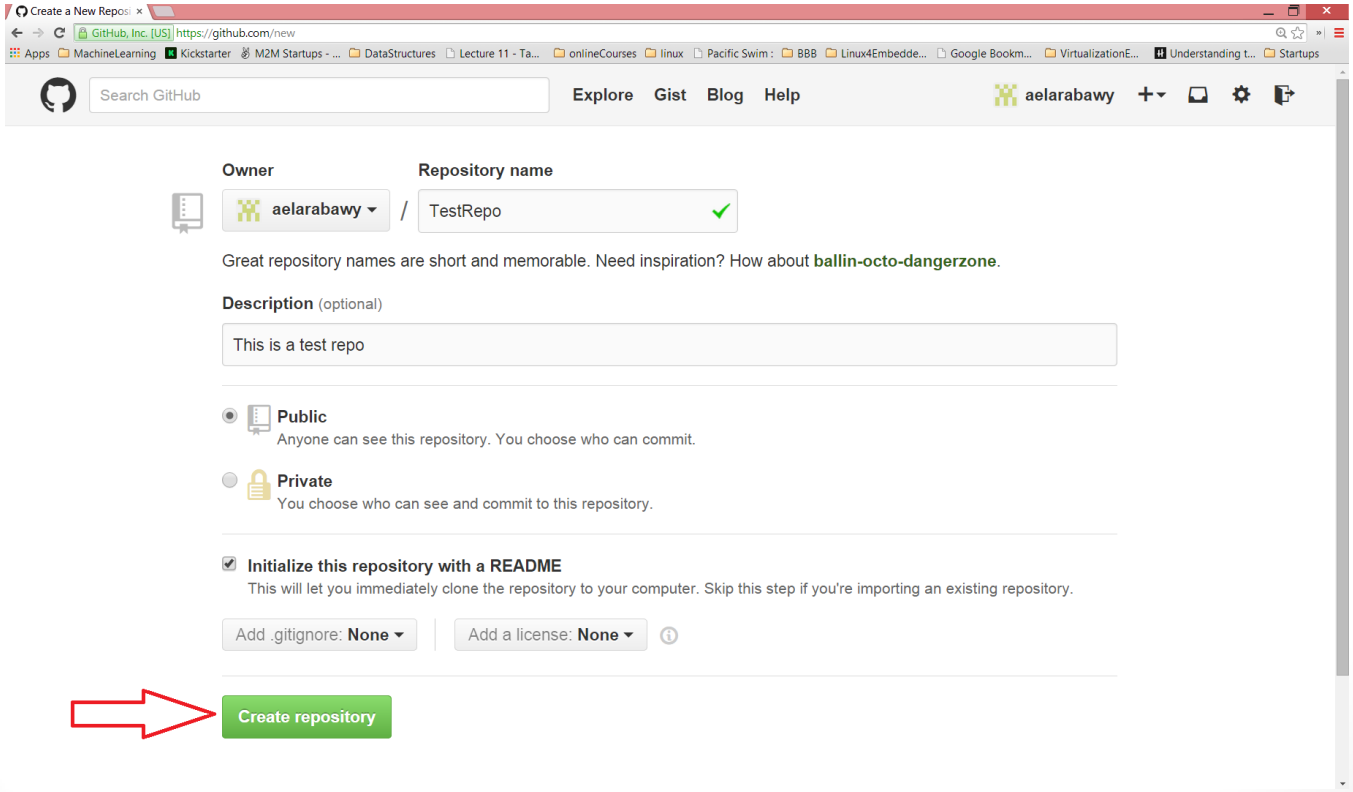
Preparation Steps
GitHub Account

- The following steps need to be taken:
  - Create a GitHub account
  - Sign in into your account
  - Create a Repo



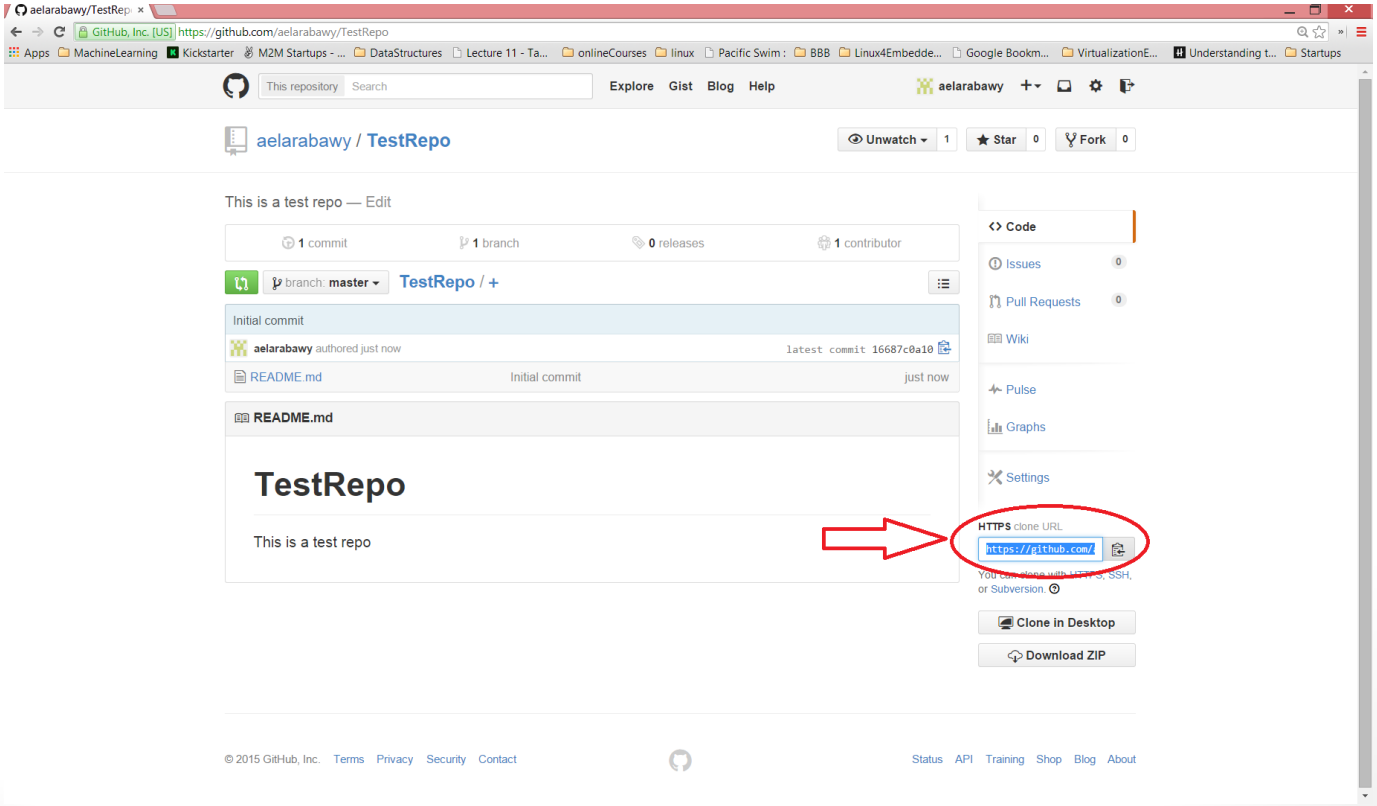
Preparation Steps
GitHub Account

- The following steps need to be taken:
  - Create a GitHub account
  - Sign in into your account
  - Create a Repo



Preparation Steps
GitHub Account

- The following steps need to be taken:
  - Create a GitHub account
  - Sign in into your account
  - Create a Repo
  - Get the repo URL



Preparation Steps
Setup Git On Local Machine

- Install Git

  $ sudo apt-get install git-core

  S sude apt-get install gitk

- Configure the user name & email (use same email as used in github)

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

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

- Save your username/password for logging into remote servers (when using https cloned repos)

  $ git config --global credential.helper cache

  $ git config --global credential.helper 'cache --timeout=3600'

Preparation Steps
Create the SW Project

- Move to the parent directory of your project

  $ cd <some directory>
- Clone the empty repo from GitHub

  $ git clone <repo URL>
- Now, you can add your files to your working directory
- Commit the files to your local repo

  $ git add <files>

  $ git commit
- Once you are ready, you can push your commits to the remote repo on GitHub

  $ git push origin master
  - Note that you may be required to enter your login credentials at this step

Preparation Steps
Clone the Project

- These steps are applicable for:
  - Same user who wants to have the project on another machine
  - Another user who has access to login credentials of the GitHub account (Enterprise account)
- Steps:
  - Move to the parent directory of the SW Project

    $ cd <some directory>
  - Clone the repo

    $ git clone <project URL>
  - Now you can work with the project, modify files or add new files
  - Commit your changes

    $ git add <files to stage>

    $ git commit
  - When needed you can synchronize with the remote repo

    $ git pull origin master

    $ git push origin master

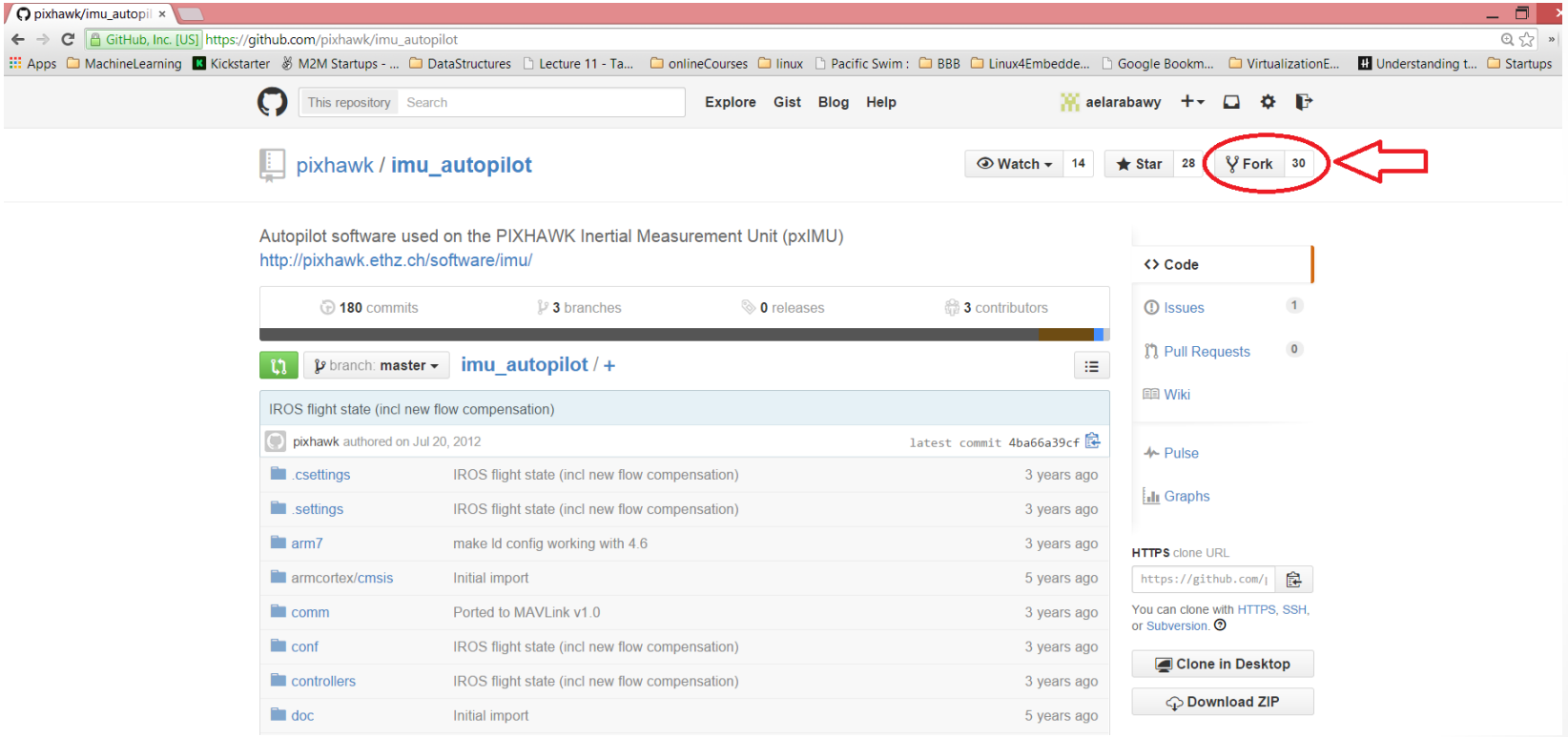
Preparation Steps
Fork the Project

- These steps are applicable for a different user who does not have access to the GitHub account which contain the repo
- Hence this user will need to fork the repo to his account, then work with the forked version
- Steps:
  - Login to user GitHub account
  - Search for the desired repo
  - Select to Fork this repo, now the user will have a copy of the repo on his account
  - On the local machine, repeat steps for (clone/add/commit/pull/push)
  - You can synch with the original repo using pull requests
  - Note that you can synch with the original project directly using Git by setting up a remote and synch with that remote using Git Commands

    $ git remote add upstream <url of the original repo>

    $ git pull upstream

Preparation Steps
Fork the Project



Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com