Introduction to Version Control (part 2)

ARCHER Virtual Tutorial

EPSRC

NERC SCIENCE OF THE ENVIRONMENT

Reusing this material

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Outline

Part 1: common features of version control systems

Part 2: different models of version control, workflows:

- Centralised (client-server) version control (CVS, SVN)
- Distributed (peer-2-peer) version control (Git, Mercurial)
- Workflows
- Hosting & additional features: issue tracking, pull requests
- Choosing a version control system

Centralised version control

Examples of centralised (also known as client-server) version control systems:

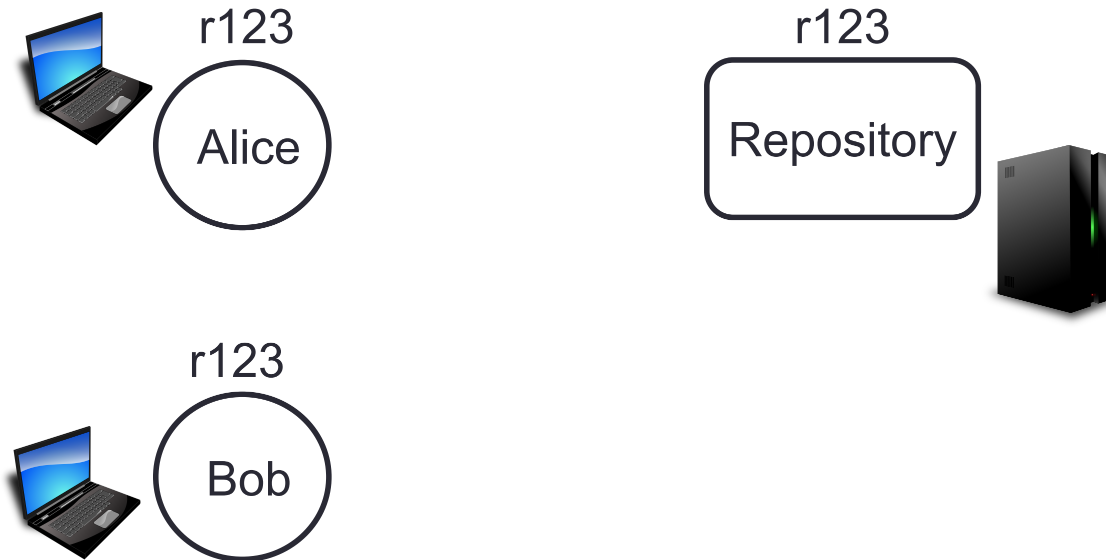
- CVS
- Subversion (SVN)
- Perforce

Centralised version control

Repository located on a central server

Centralised version control

Each user has a working copy of the repository



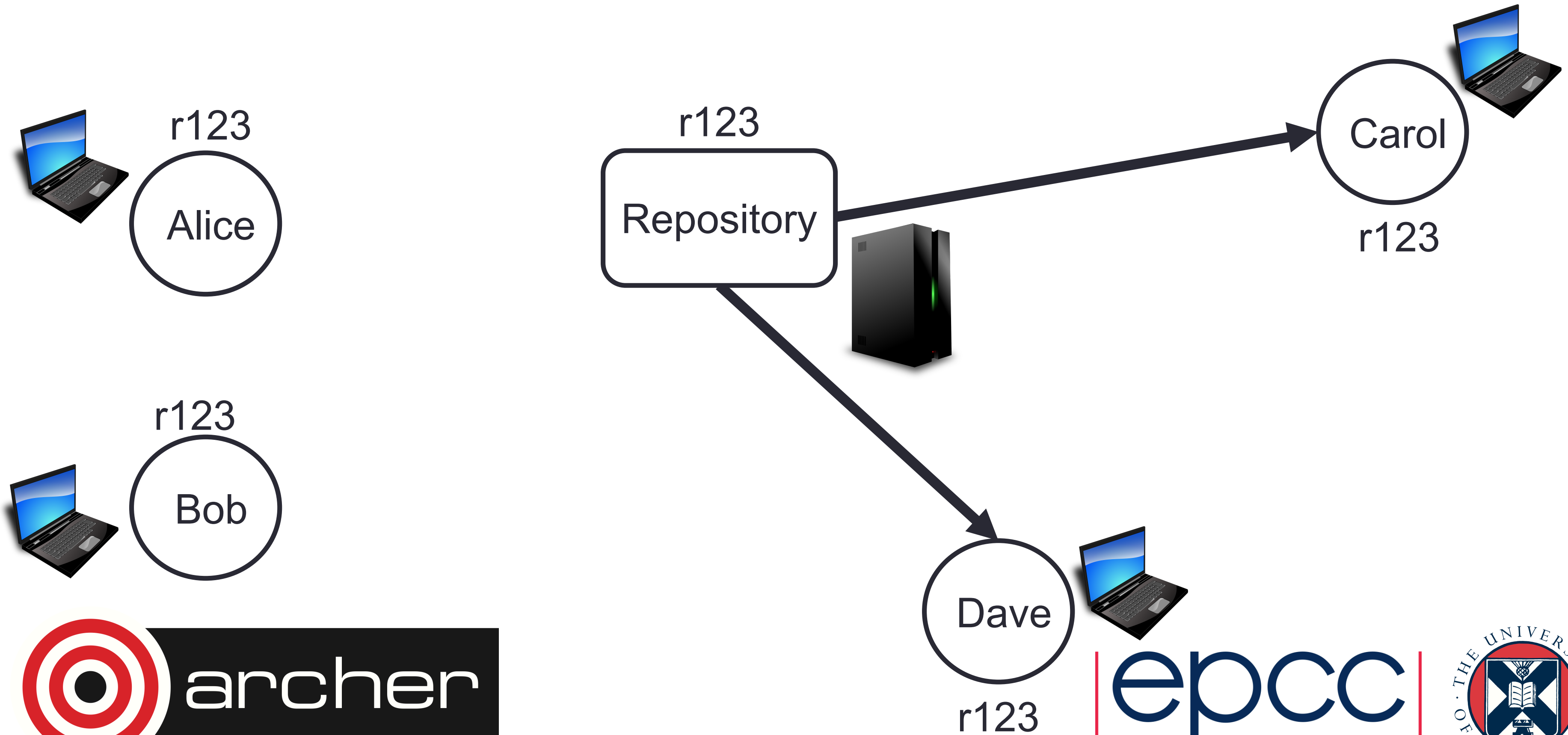
archer

epcc

THE UNIVERSITY OF EDINBURGH

Centralised version control

New users check out a fresh working copy



Centralised version control

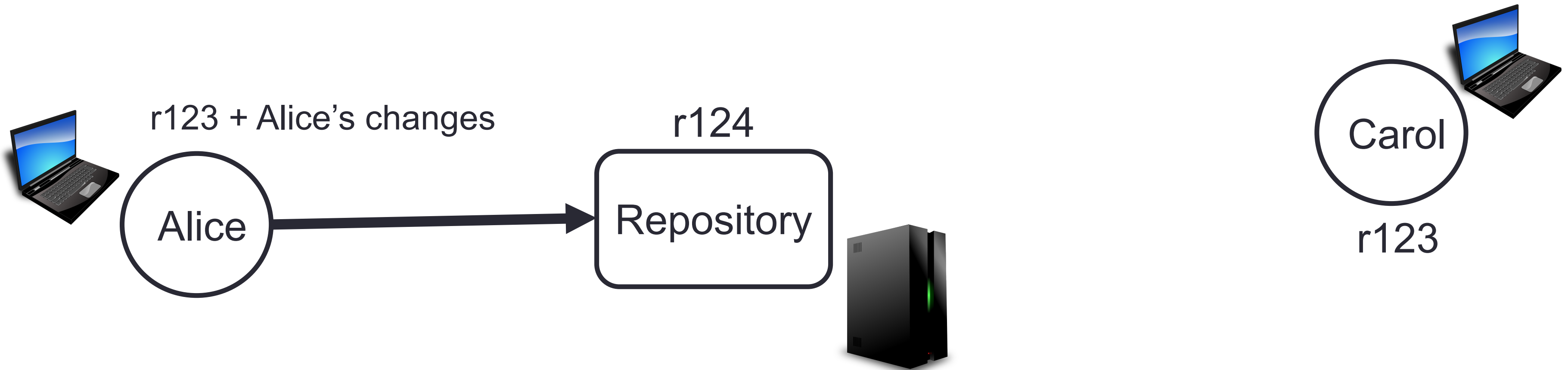
Some users make changes to their working copy

r123 + Bob's changes

Bob

Centralised version control

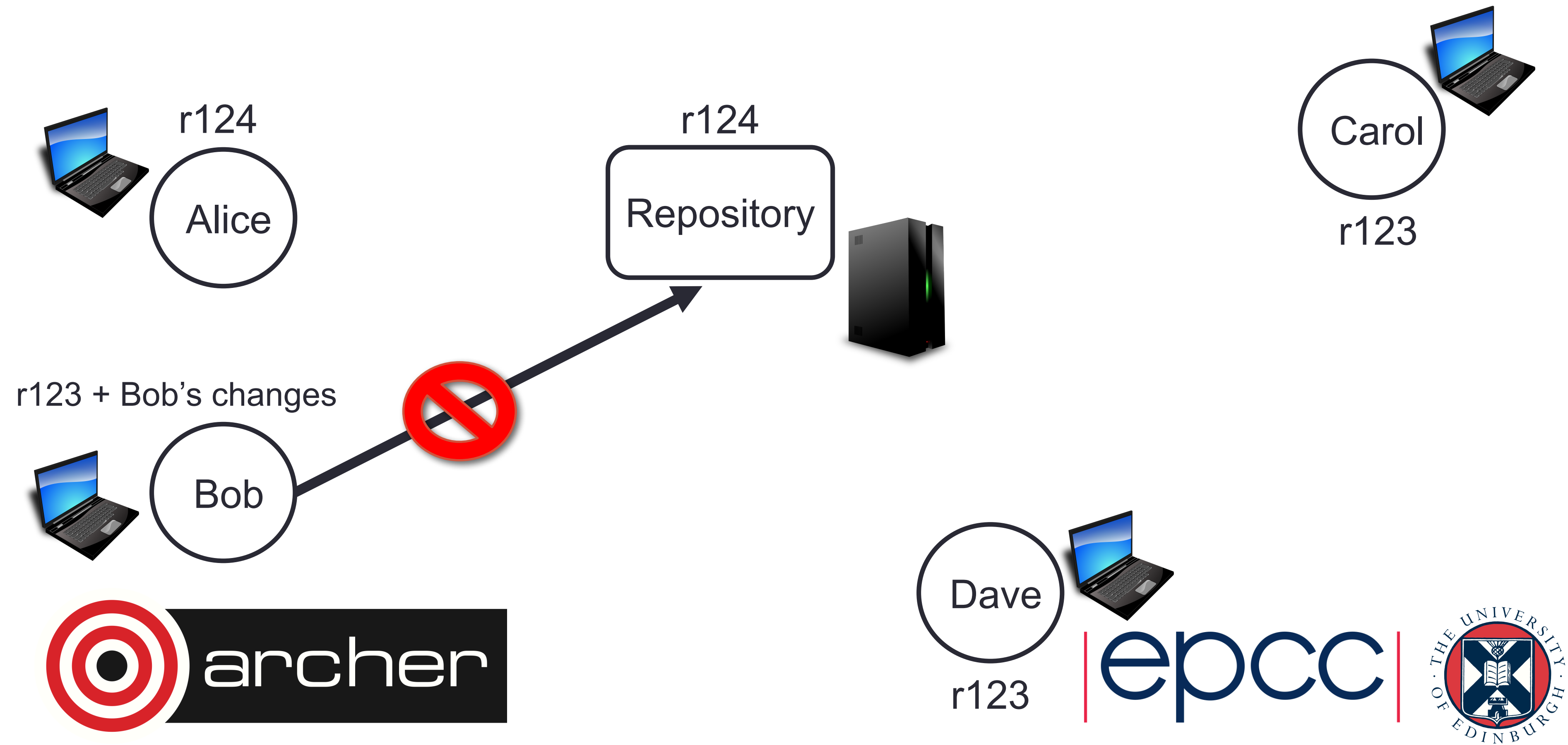
Users commit their changes to the repository



Centralised version control

Users commit changes to the repository

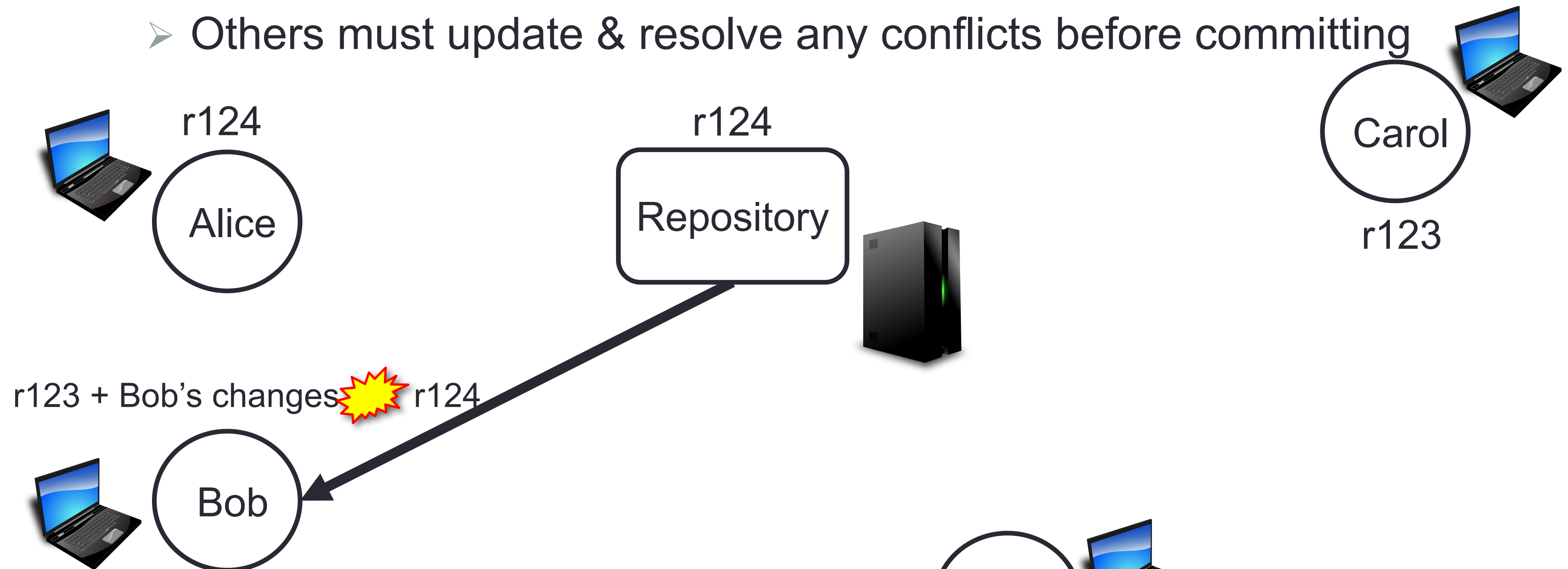
- First one to commit “wins”



Centralised version control

Users commit changes to the repository

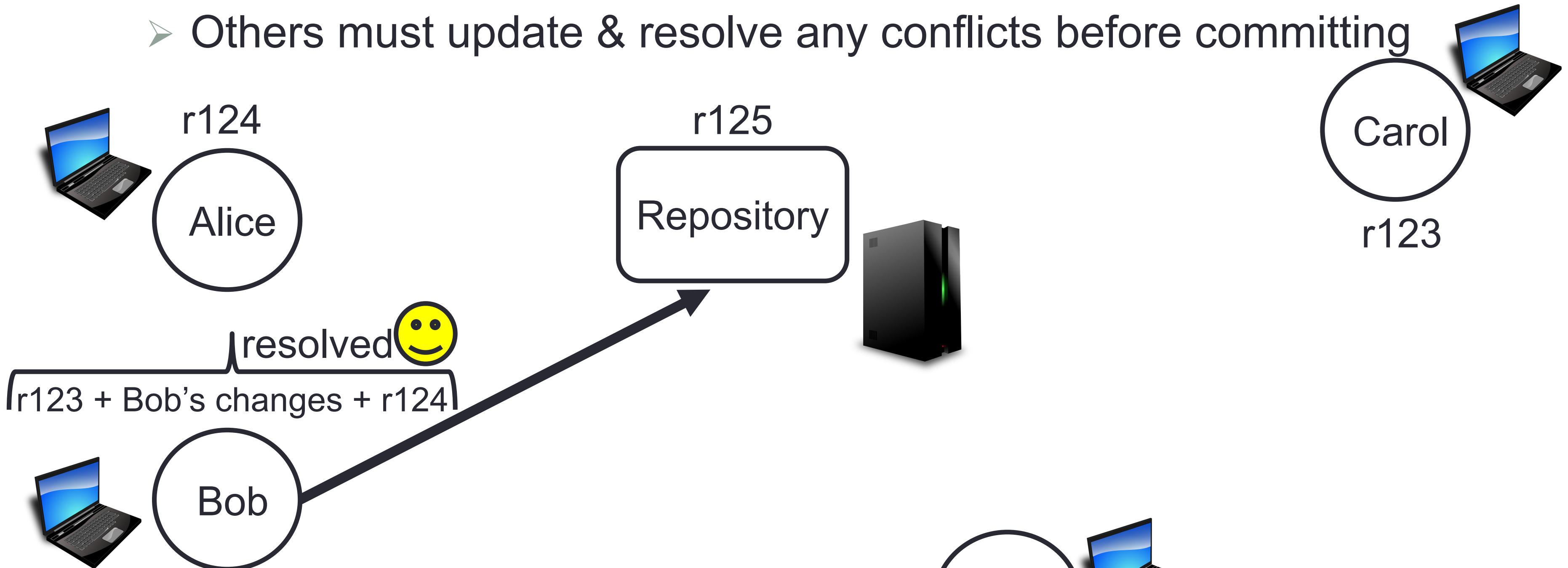
- First one to commit “wins”
- Others must update & resolve any conflicts before committing



Centralised version control

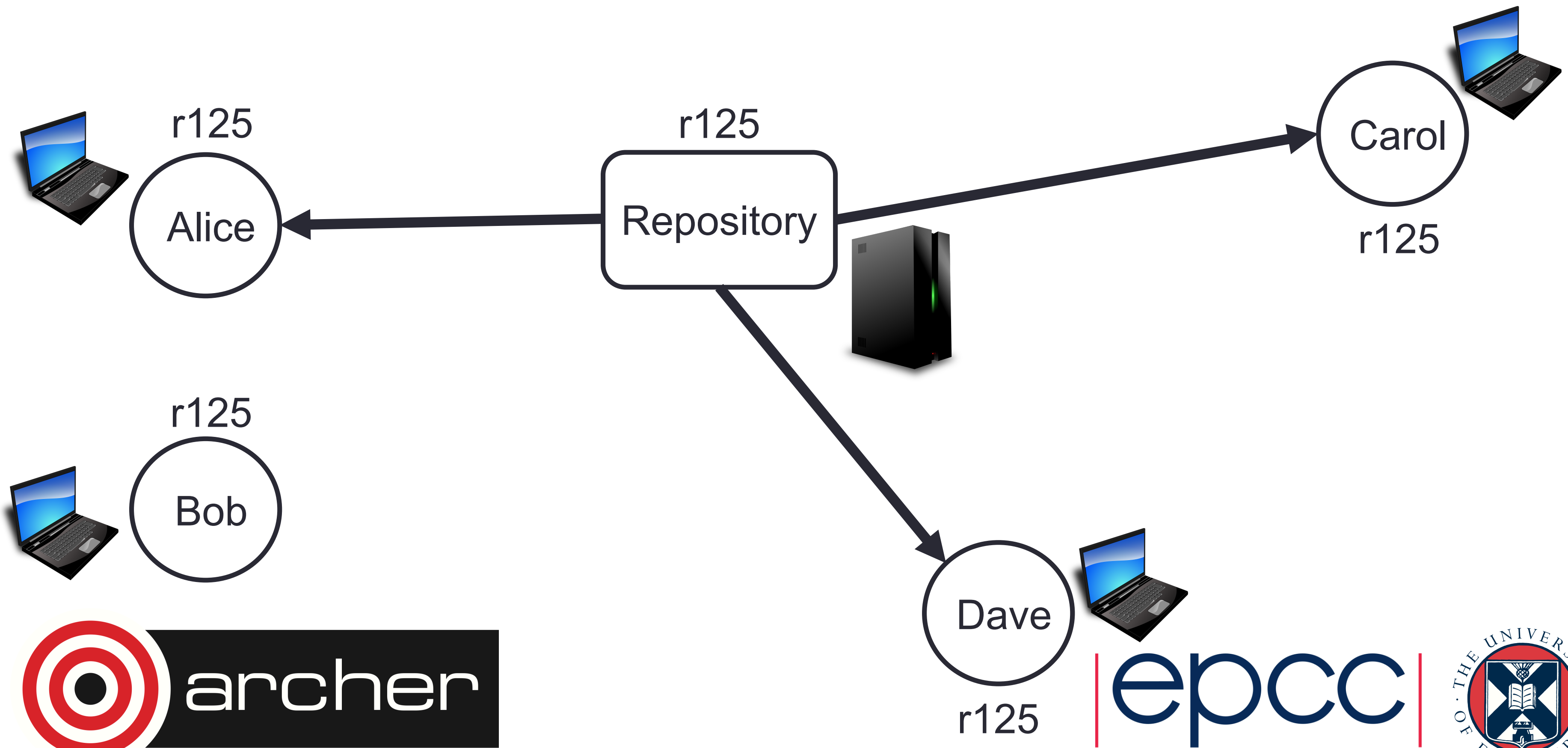
Users commit changes to the repository

- First one to commit “wins”
- Others must update & resolve any conflicts before committing



Centralised version control

Users periodically synchronise by updating their working copies with the canonical content in the central repository



Centralised version control

- Enforces
  - centralised workflow
  - linear “global progress” view (incrementing revision numbers)
- Need to be online (able to connect to machine hosting central repository) to commit any changes
- Past versions of files not stored locally, need to be online
  - To check out any past committed versions of files
  - To check the revision history (CVS)
- All commits visible by all users of a repository
  - Can discourage committing, experimenting
  - Can discourage creating many branches

Centralised version control

- Communications with server cost time
- Server is single point of failure, requires configuration & maintenance:
  - Downtime can affect many users
  - Backups
  - Security

Distributed version control

- Examples of distributed (also known as peer-2-peer) version control systems:
  - Git
  - Mercurial

Distributed version control

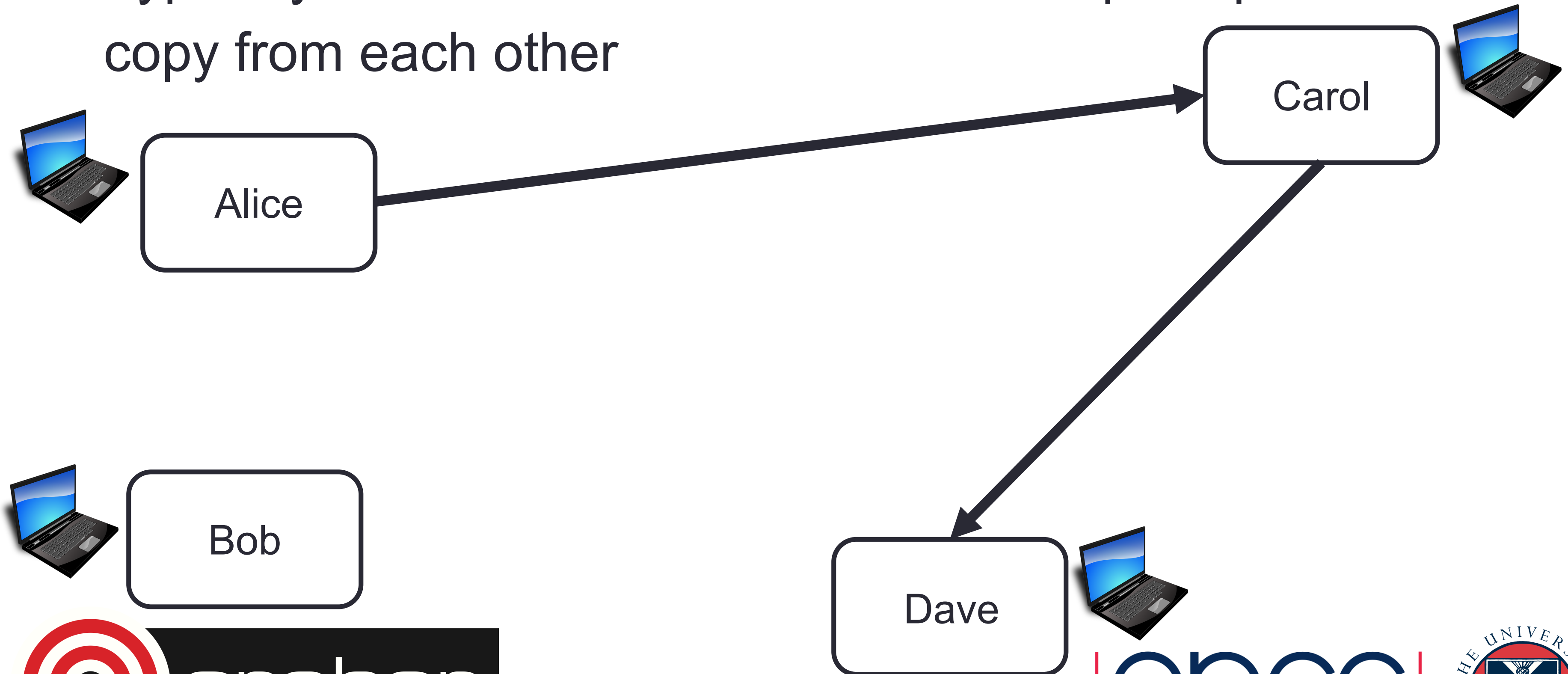
Each user has their own repository copy stored locally

Central server is optional (in practice often useful)

Distributed version control

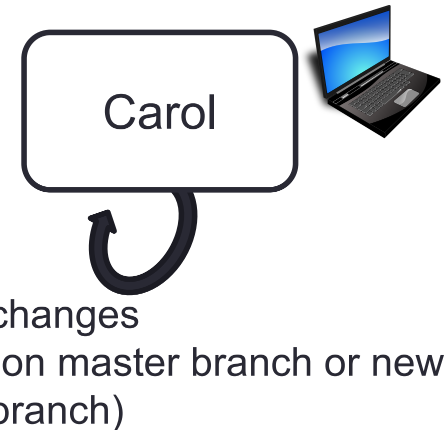
New users clone, i.e. copy, an existing repository

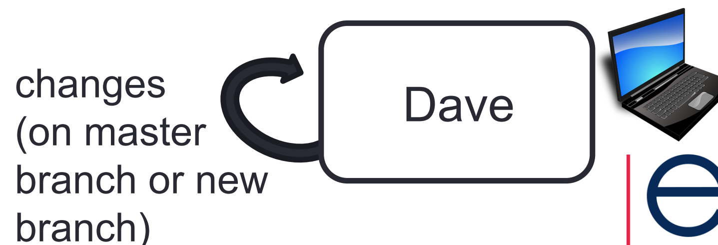
- Typically from a central server but can in principle copy from each other



Distributed version control

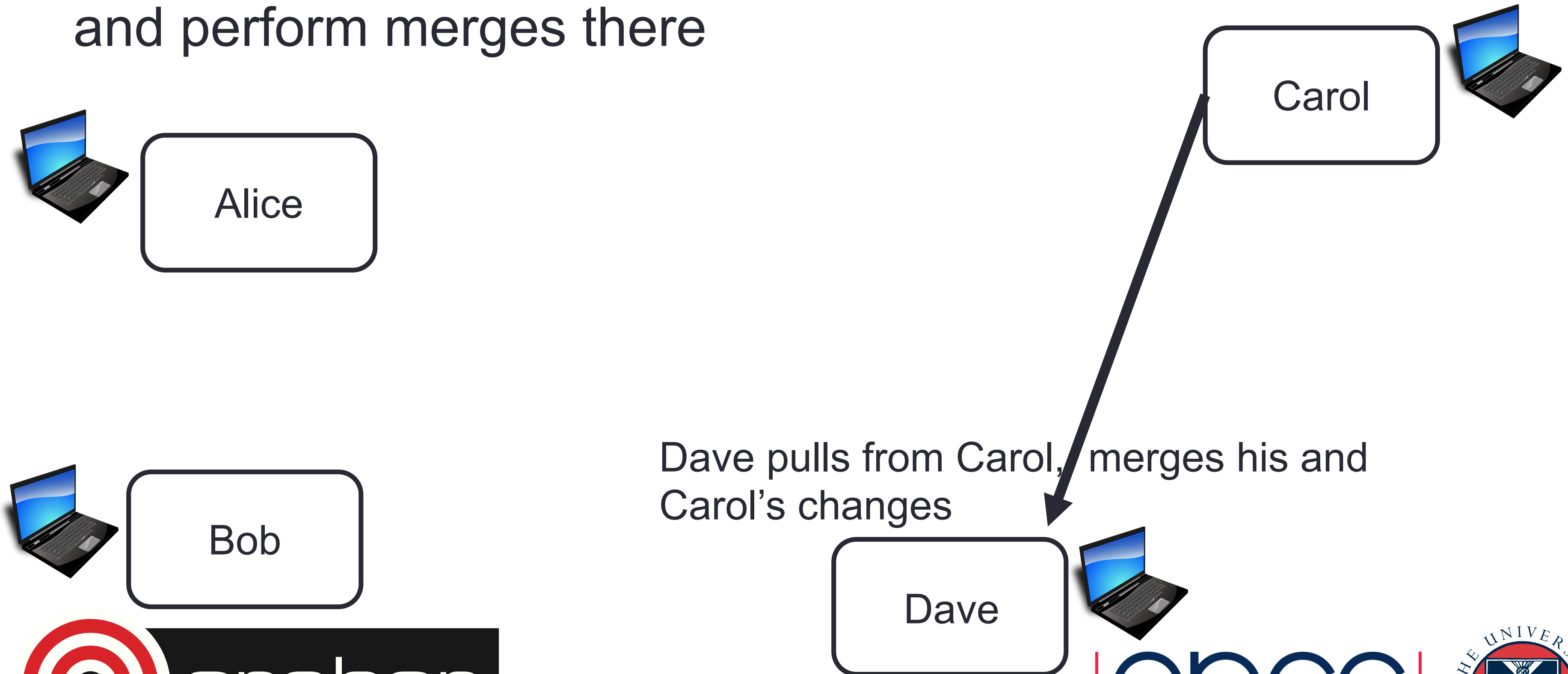
Users make changes in their working copy and commit this to their local repository ➔ repositories diverge





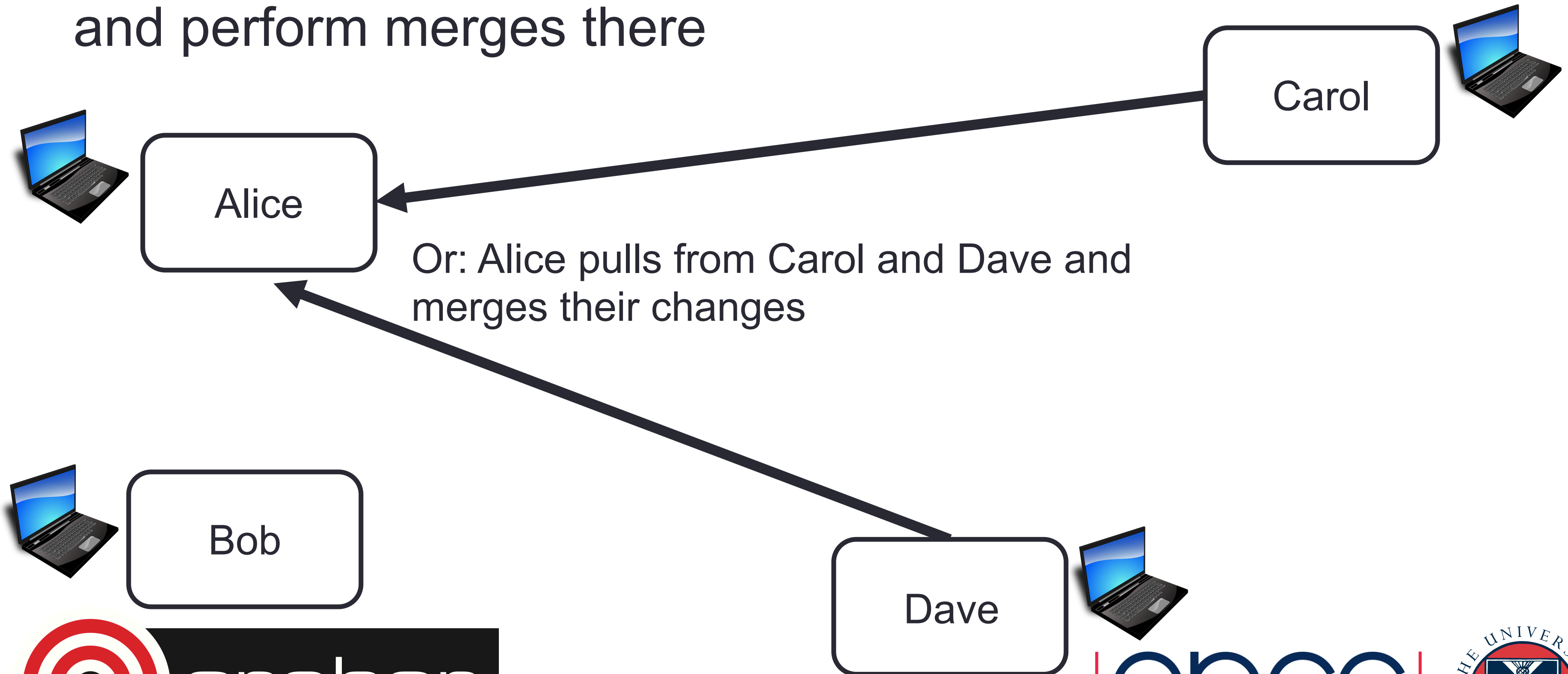
Distributed version control

To combine content from different repositories someone has to fetch other people's changes into their working copy and perform merges there



Distributed version control

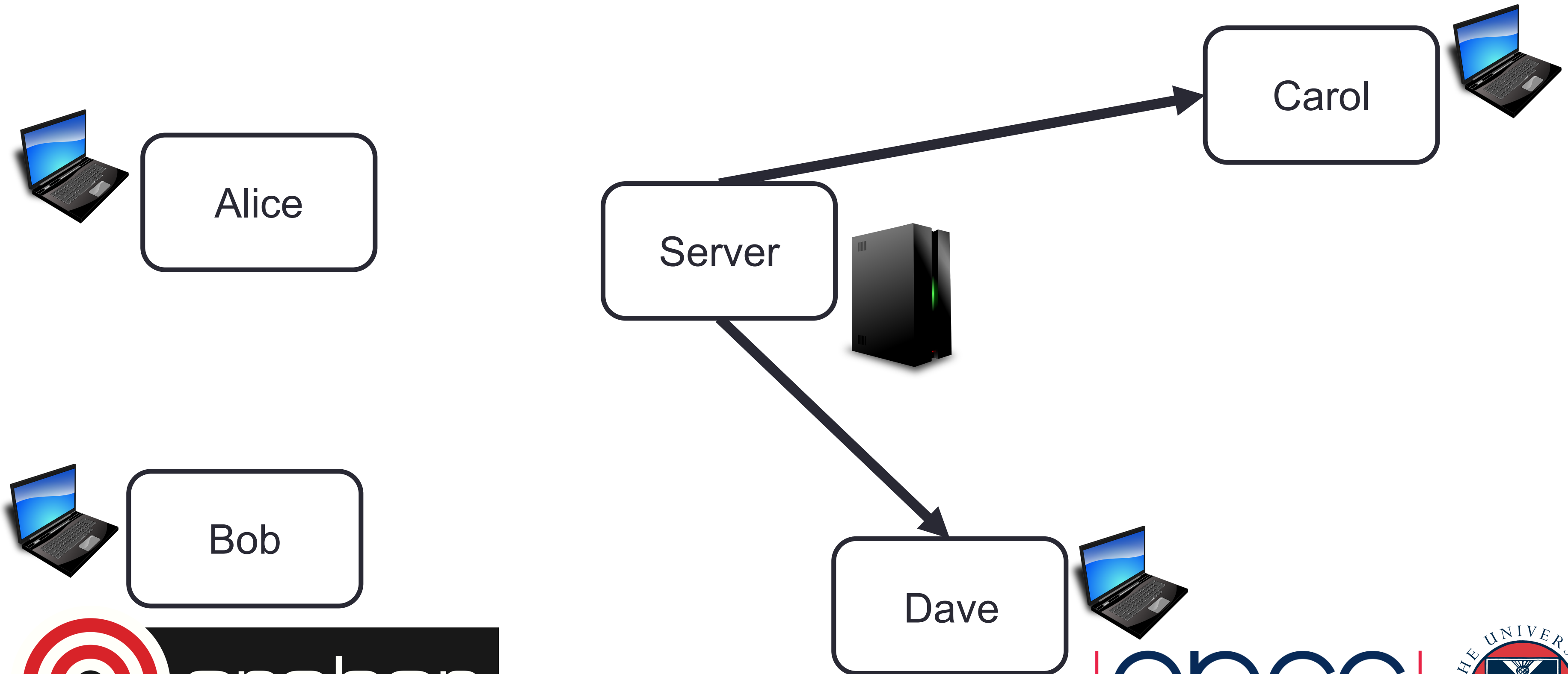
To combine content from different repositories someone has to fetch other people's changes into their working copy and perform merges there



Distributed version control

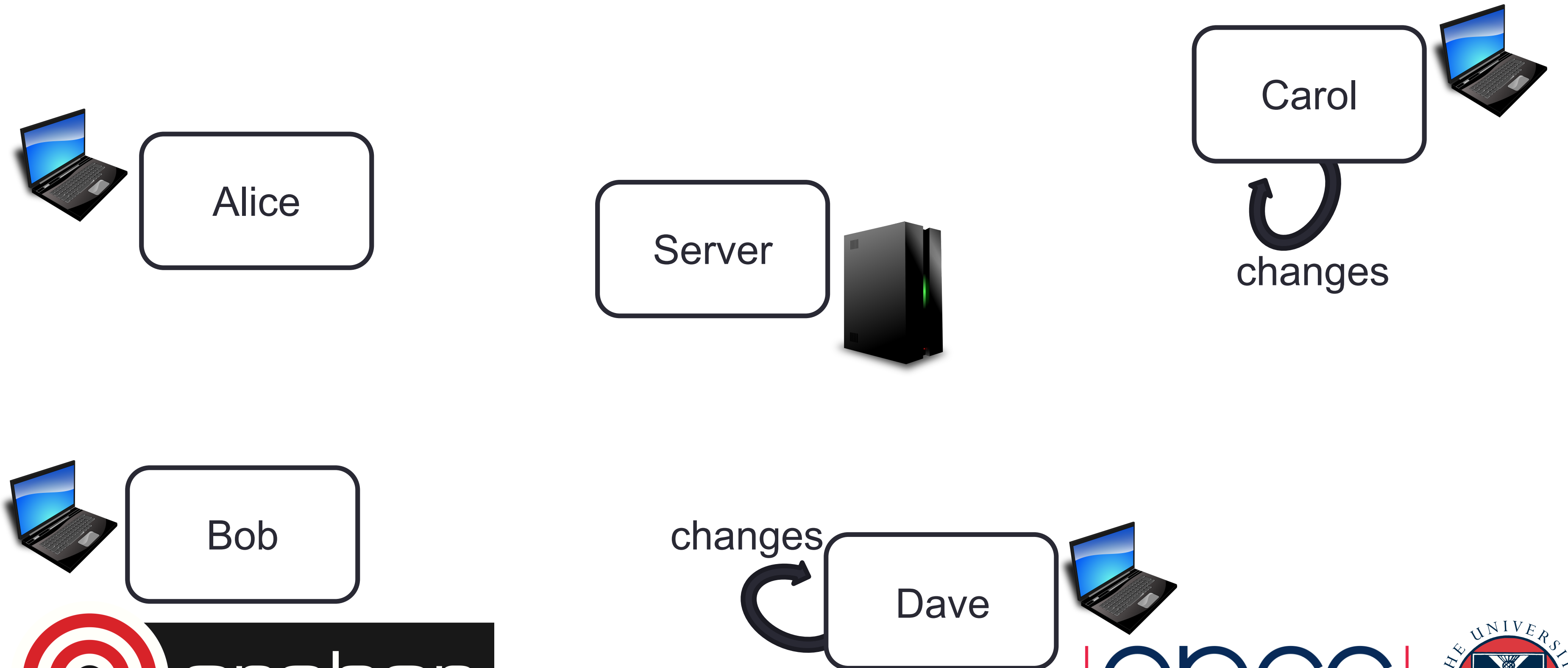
Often use a central server for convenience:

- Clone



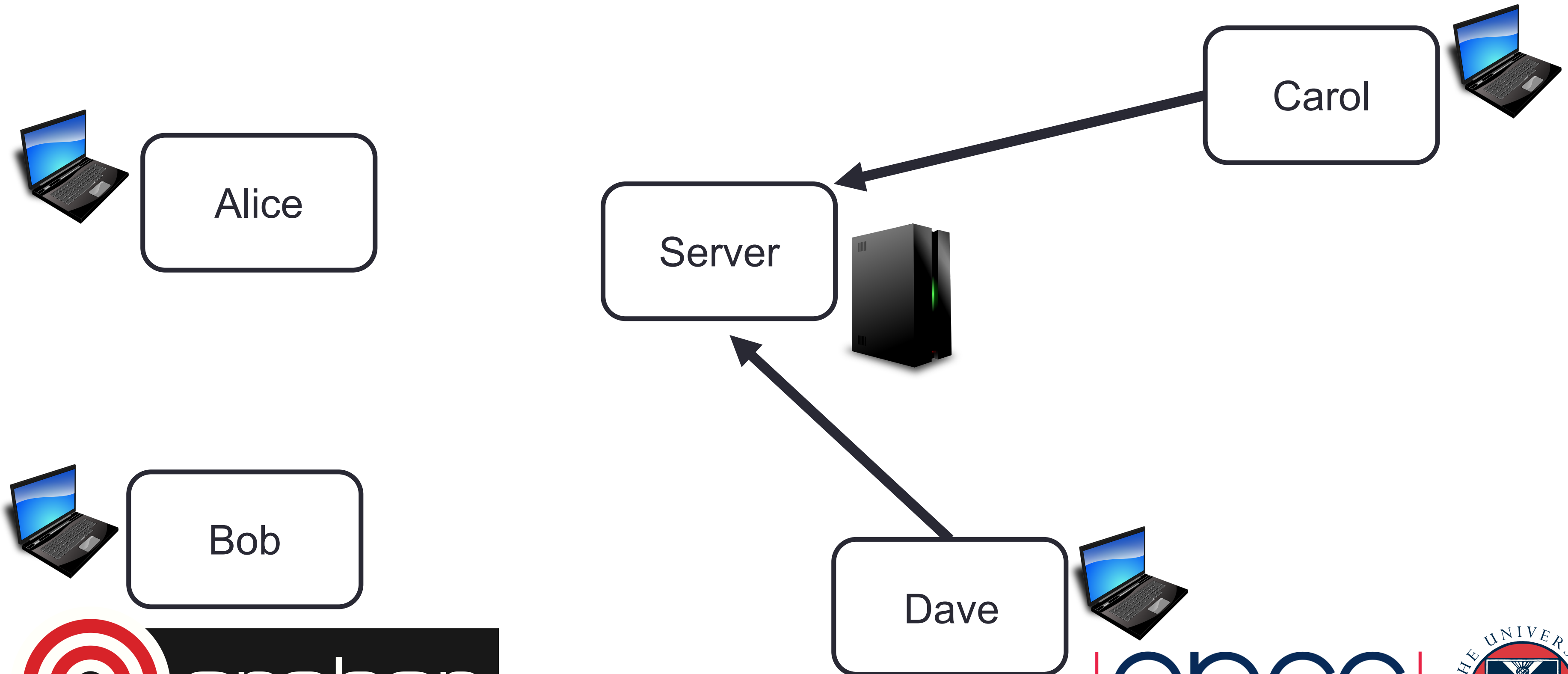
Distributed version control

- Commit local changes



Distributed version control

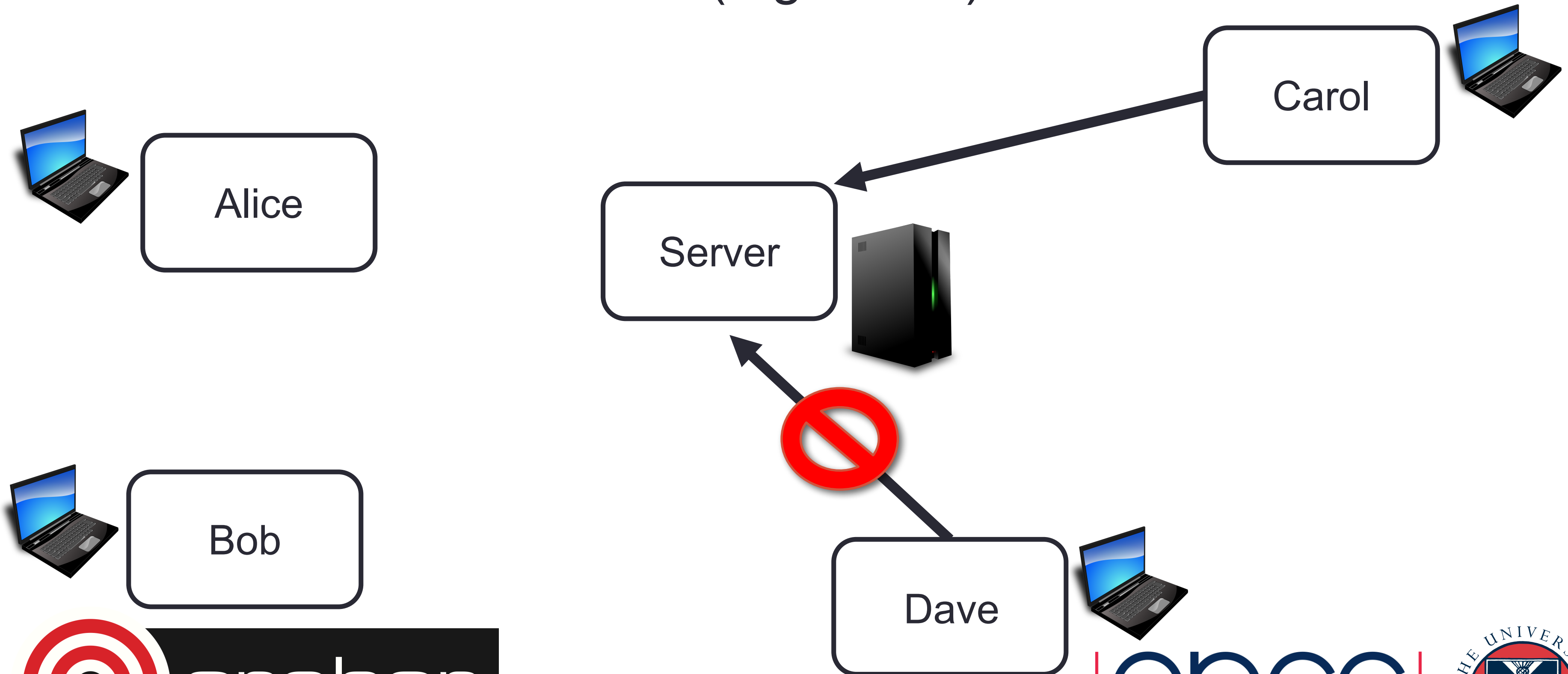
- Push changes to server repository



Distributed version control

If changes were made to master branch:

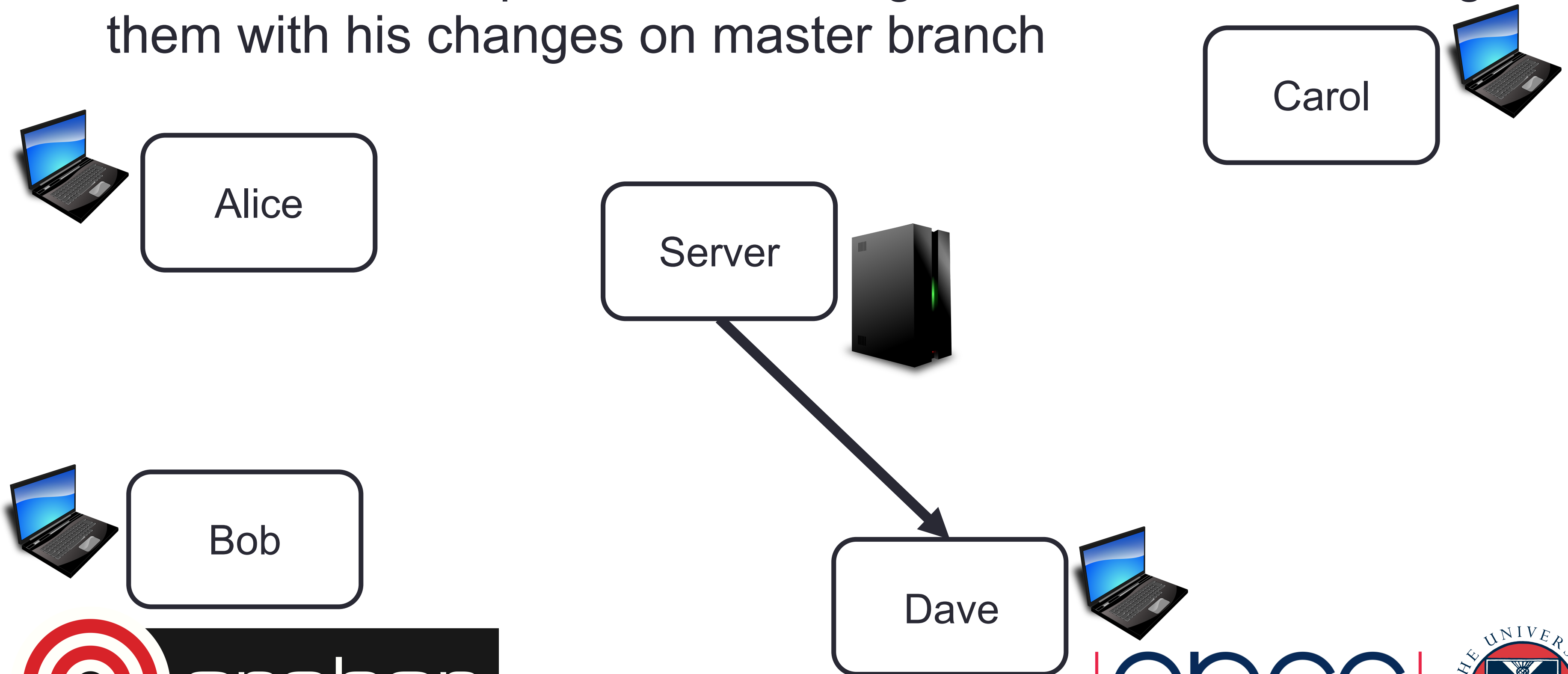
- First to commit to server (e.g. Carol) “wins”



Distributed version control

If changes were made to master branch:

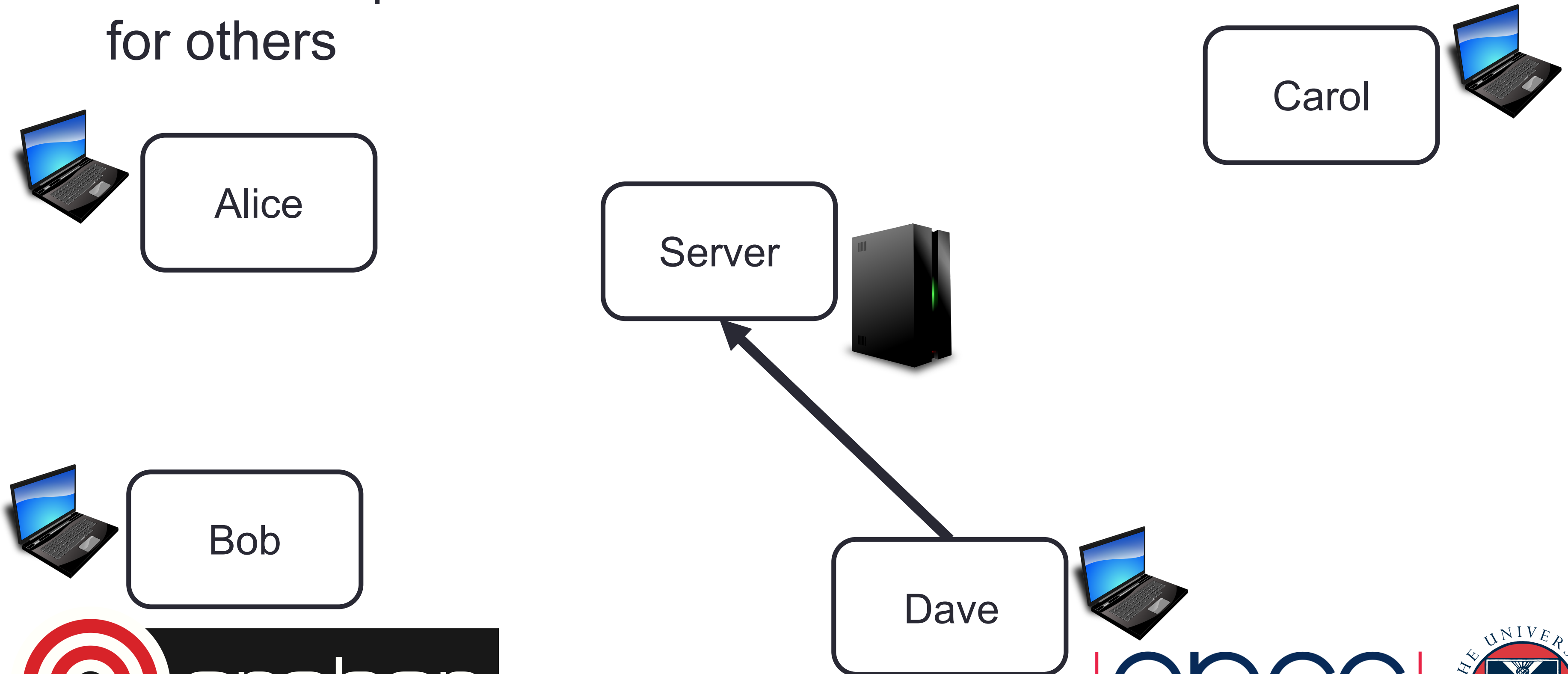
- Dave has to first pull Carol's changes from server and merge them with his changes on master branch



Distributed version control

If changes were made to master branch:

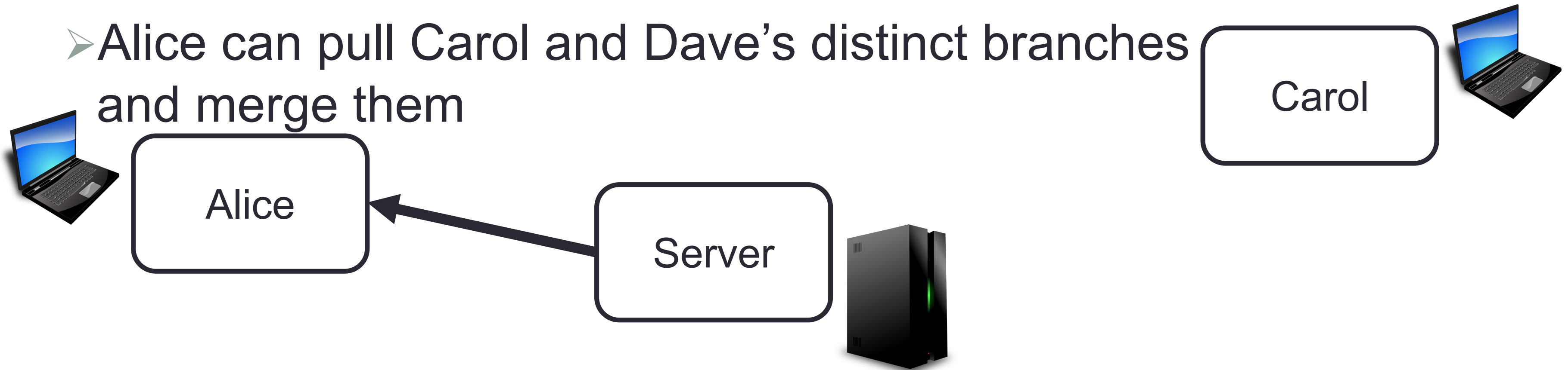
- Dave then pushes result back to master branch on server for others



Distributed version control

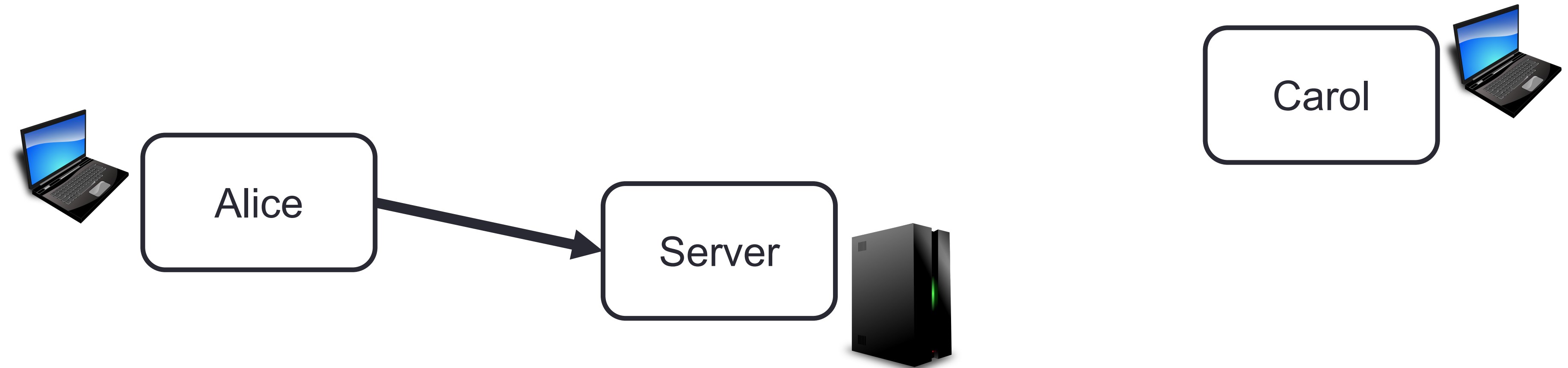
If changes were made to new branches:

- Carol and Dave push to different branches on server
- Alice can pull Carol and Dave's distinct branches and merge them



Distributed version control

- Alice then pushes result back to a single branch on server (can be master branch or another branch) for others



Distributed version control

- Don’t need to be online to commit changes
- Changes can be committed privately
  - Encourages committing early on
  - Encourages branching to commit e.g. experimental code
- Full revision history (log & past versions) available locally
- Can adopt workflows other than centralised for combining content from contributors
- Many common operations are faster because no communication with server needed

Distributed version control

How does the version control system know how to merge content from different repositories?

How does it determine how far back to go in the revision history of two branches being merged until it finds a common ancestor?

No single canonical repository so no global revision number (r###) that can be used to judge when commits diverge

Distributed version control

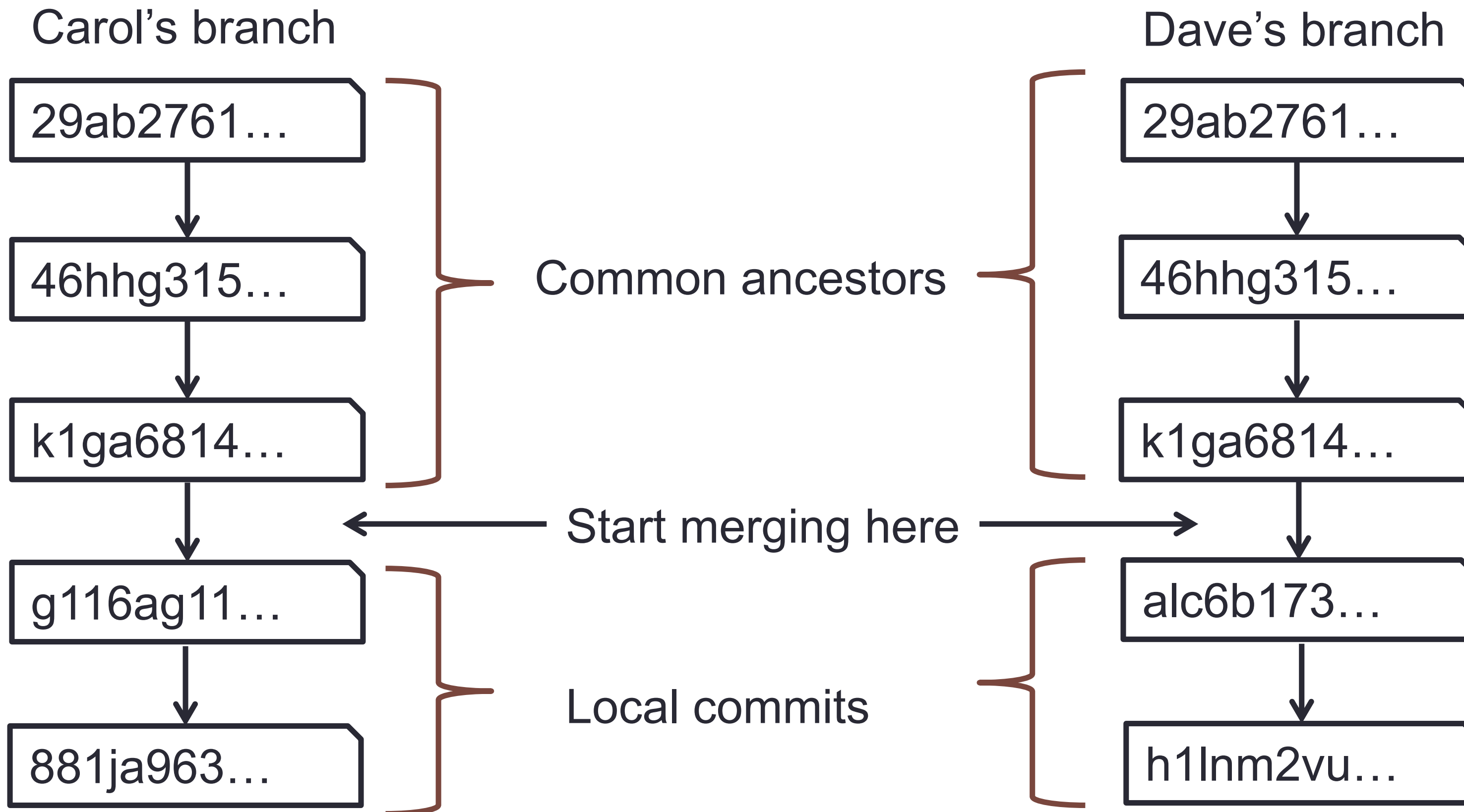
Solution:
Compute an ID uniquely identifying each commit

Even better:
Compute an ID uniquely identifying each commit and its preceding revision history

Git and Mercurial accomplish this using a hash function (SHA-1) that generates a 40-digit hexadecimal number

If two commits from different repos have the same ID they have identical revision histories and hence are common ancestors

Distributed version control



archer

epcc

THE UNIVERSITY OF EDINBURGH

Distributed version control

- No need to set up and maintain a server
- No single point of failure
- As many backups as repository clones
- Hash IDs allow exact verification of integrity of data
- Branches play a very important role
  - Used to communicate between repositories
  - Mercurial and Git have very efficient implementations of branching – branching is cheap, and merging is clever

Hosting & additional features

Distributed version control systems became very popular over the past ~8 years (Git born 2005)

A number of websites (GitHub, Bitbucket, …) have helped fuel this trend and exploit the potential of distributed version control.

GitHub etc. offer repository hosting and management and additional features that facilitate collaborative software development

Hosting & additional features

Aditional features:

- Wiki to track and discuss bugs, feature requests etc. tightly integrated with version control workflow
- “Pull request” mechanism allowing developers to clone (fork) a repository, make changes, then suggest to the original owner that these changes are integrated into the parent repository

Site-installable web-based repository management frameworks (e.g GitLab) offer similar features.

Demonstration using Git & Github

- Going to find a code repository on Github
- "Fork" a copy on Github that we will be able to write to
- Check out (clone) a local copy
- Make changes to a file
- Commit these changes
- Create a new file, add this to the repository
- Push these changes to the remote repository on Github

Distributed workflows

Integration manager workflow:

blessed repository

developer public

developer public

integration manager

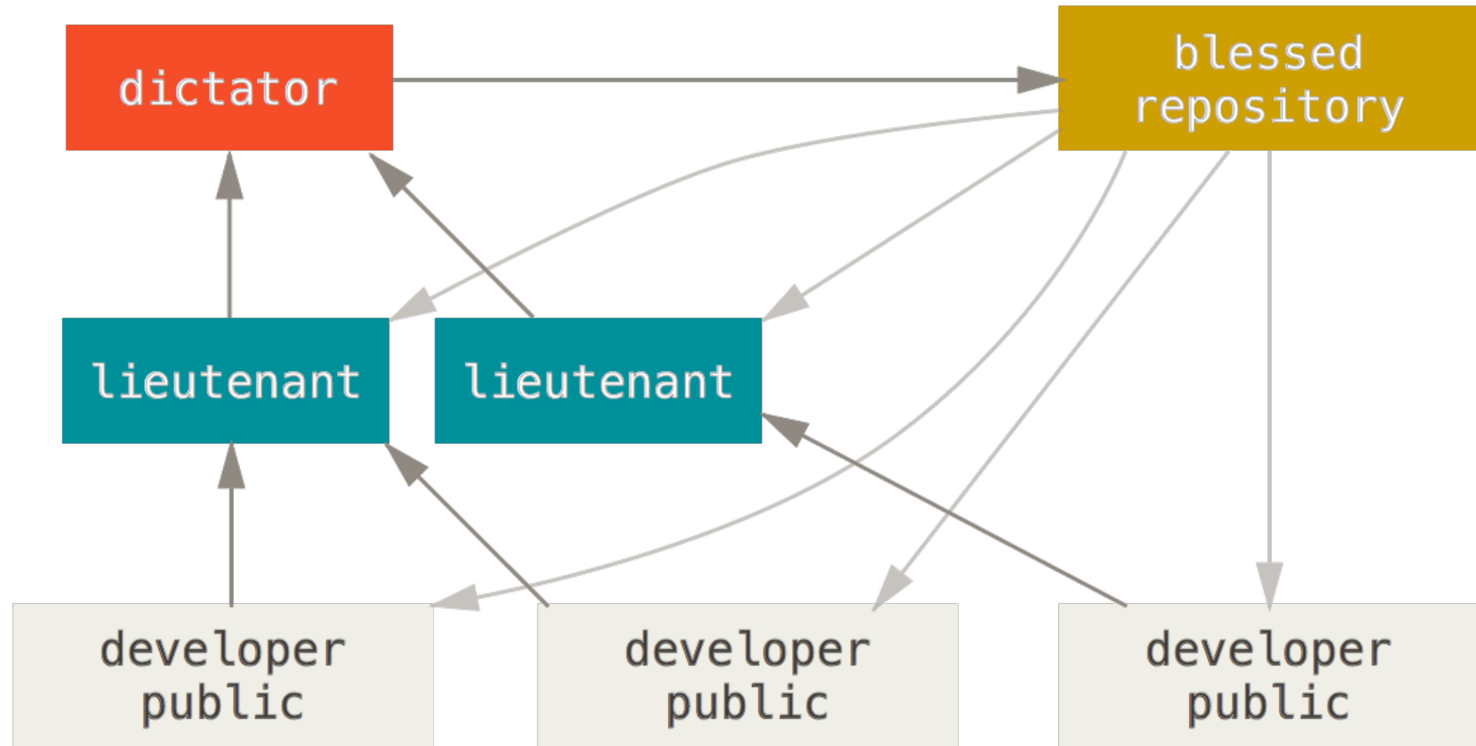
developer private

developer private

Reproduced under CC Attribution Non-Commercial Share Alike 3.0 license
see http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

Distributed workflows

“Dictator and lieutenants” workflow:



Reproduced under CC Attribution Non-Commercial Share Alike 3.0 license
see http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

Which version control system should I use?

- If you are joining an existing project: whatever is already being used! (unless there are big problems)
- Whatever your most important collaborators are used to
- Experiment!
- Git or Mercurial will allow you to immediately start committing privately and are fast and powerful
- Especially Git offers powerful options
  - But easier to get lost than Mercurial when starting out

References – further reading

- http://git-scm.com/book
- http://svnbook.red-bean.com/en/1.7/index.html
- http://www.github.com