an introduction to git

Ewen Cheslack-Postava

Outline

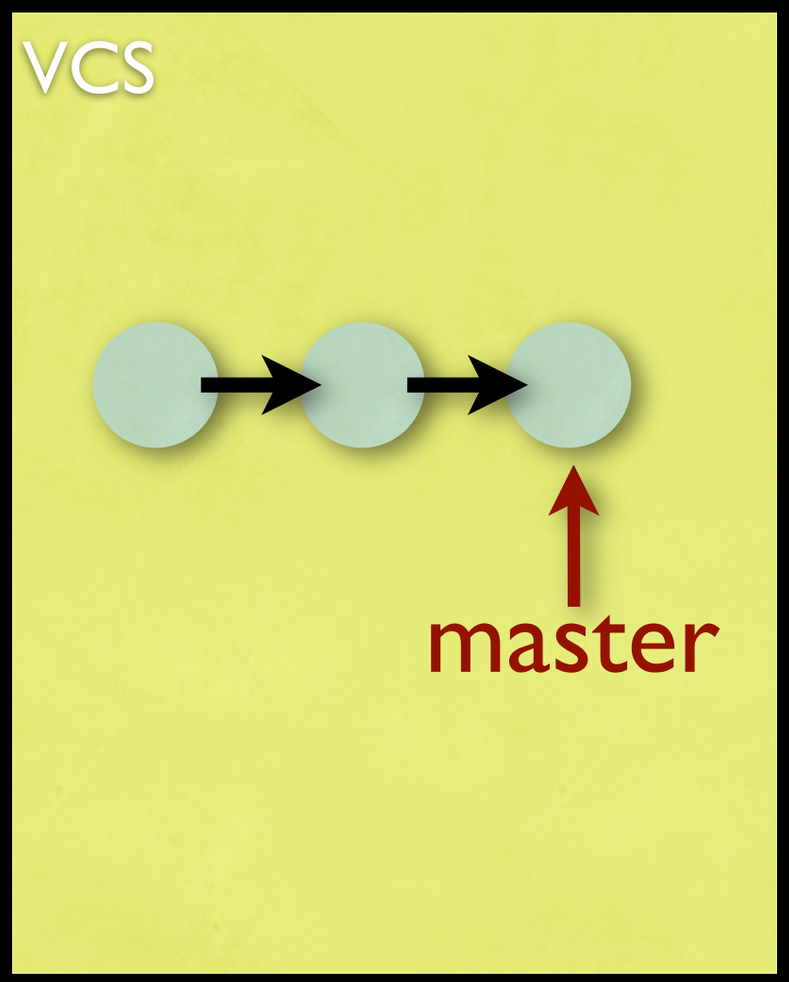
- Review of Version Control
- Distributed Version Control
- Repository Model
- Working Locally
- Workflows
- Additional Resources

Version Control Basics

- Developer chooses checkpoints
- VCS provides atomic commits
- And method to exchange them
- Handles conflicts
  - Other developer committed to same file before you

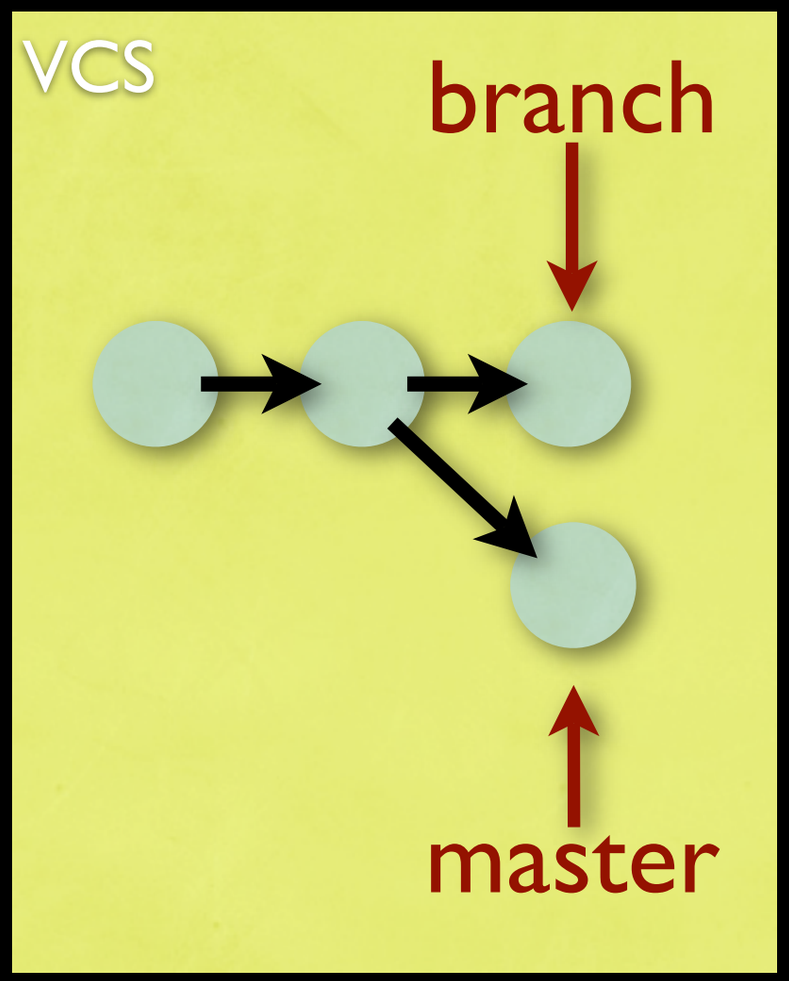
Version Control Basics

- Simple, linear model



Version Control Basics

- Branching model
  - Parallel development
  - Bug fixes vs. next release
  - Merges reconnect branch and master



Version Control Basics

- Useful even for single developer!
- Attach messages to commits to indicate what was done and why
- Manage work on multiple features using branches
- Binary search to find bug-causing commit

Centralized VCS

- Repository is centralized
- Most operations interact with server
- Examples: CVS, SVN
- In practice:
  - Branching is a copy
  - Merging isn't handled well
  - These are getting better (see newest svn)

How does git differ?

- ... or Arch?
- ... or Monotone?
- ... or Darcs?
- ... or Hg?
- ... or Bazaar?

So how does git differ from the “standard” centralized model of revision control?

Actually, git isn’t particularly special -- we could ask the same of Arch, Monotone, etc. All of these are **distributed revision control systems** or DVCS. While they are different, they all share common core concepts.

How do DVCS differ?

How do DVCS differ?

- No checkouts, only repositories

How do DVCS differ?

- No checkouts, only repositories
- Avoid network except to sync

How do DVCS differ?

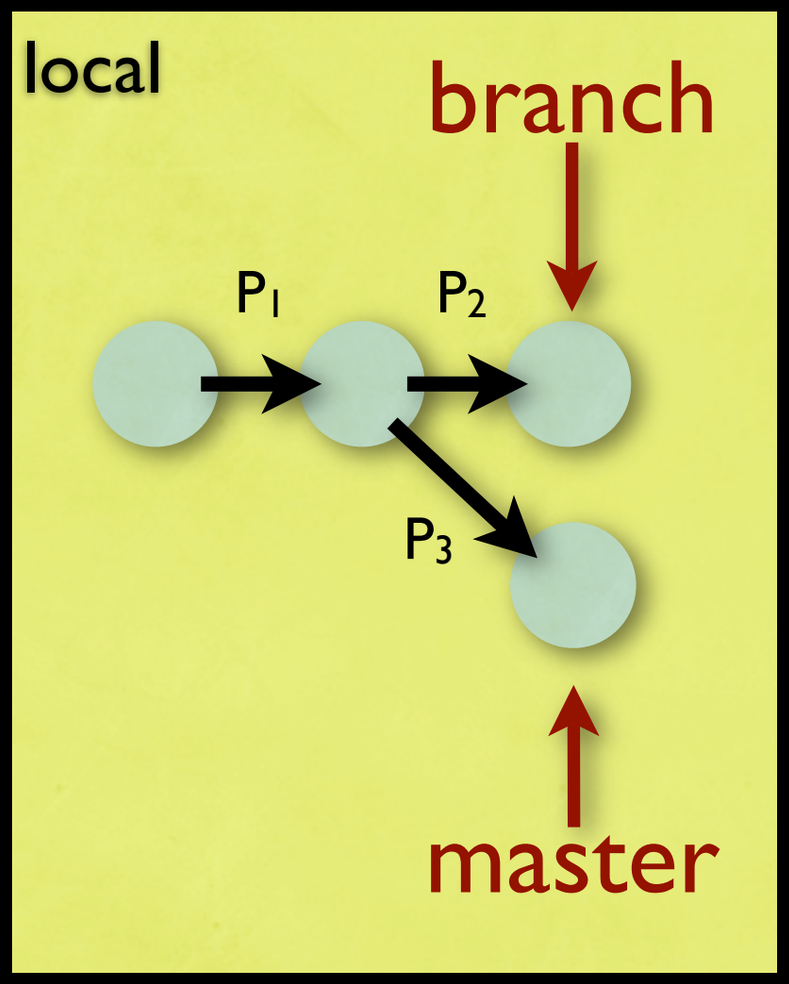
- No checkouts, only repositories
- Avoid network except to sync
- Encourage branching - cheap, local

How do DVCS differ?

- No checkouts, only repositories
- Avoid network except to sync
- Encourage branching - cheap, local
- Better merge (usually)

Repository Model

- Graph of commits - hash of content
- Edges are patches
- Only branches, no “trunk”
  - Default master branch, not special
- GC using branch HEADs and tags (a.k.a. refs)

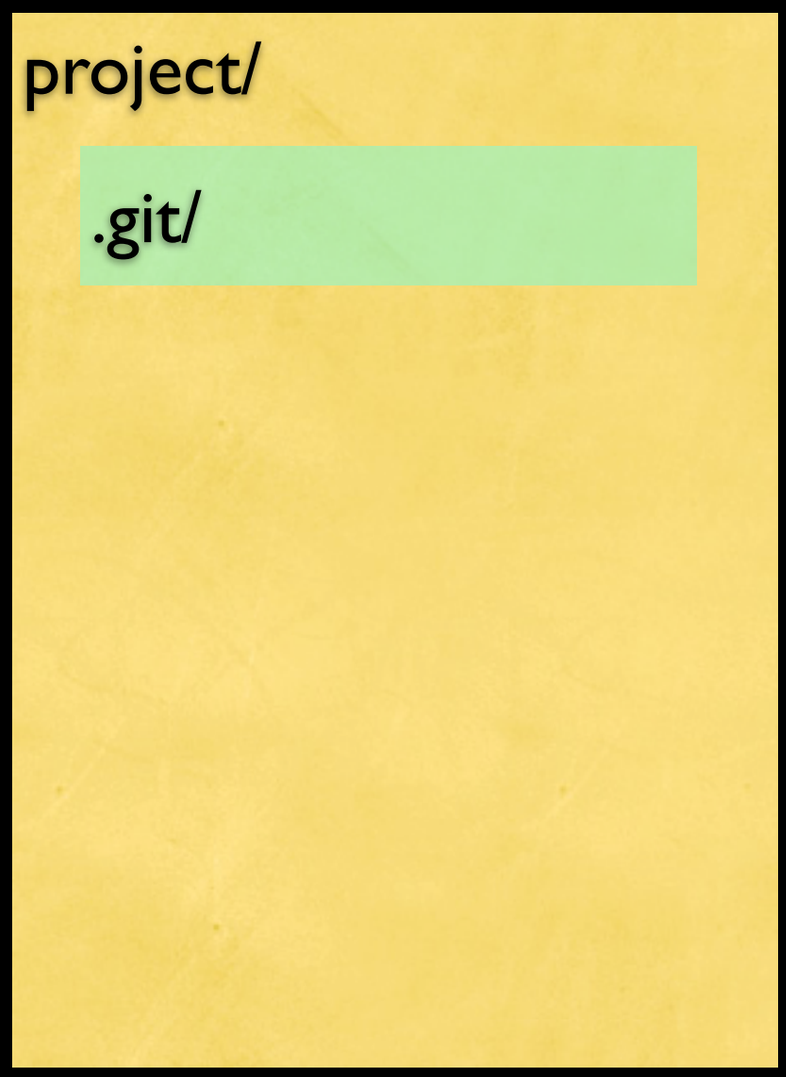


Local - Cloning

The basic workflow with DVCS is both similar and different from normal VCS. Instead of checking out a current revision, you clone the entire repository. Every revision ever checked in, and all branches and tags, will be copied locally to your machine (in the .git directory). A “checkout” now is just getting a certain copy into the working directory so you can edit it, but it always comes from local files.

Local - Cloning

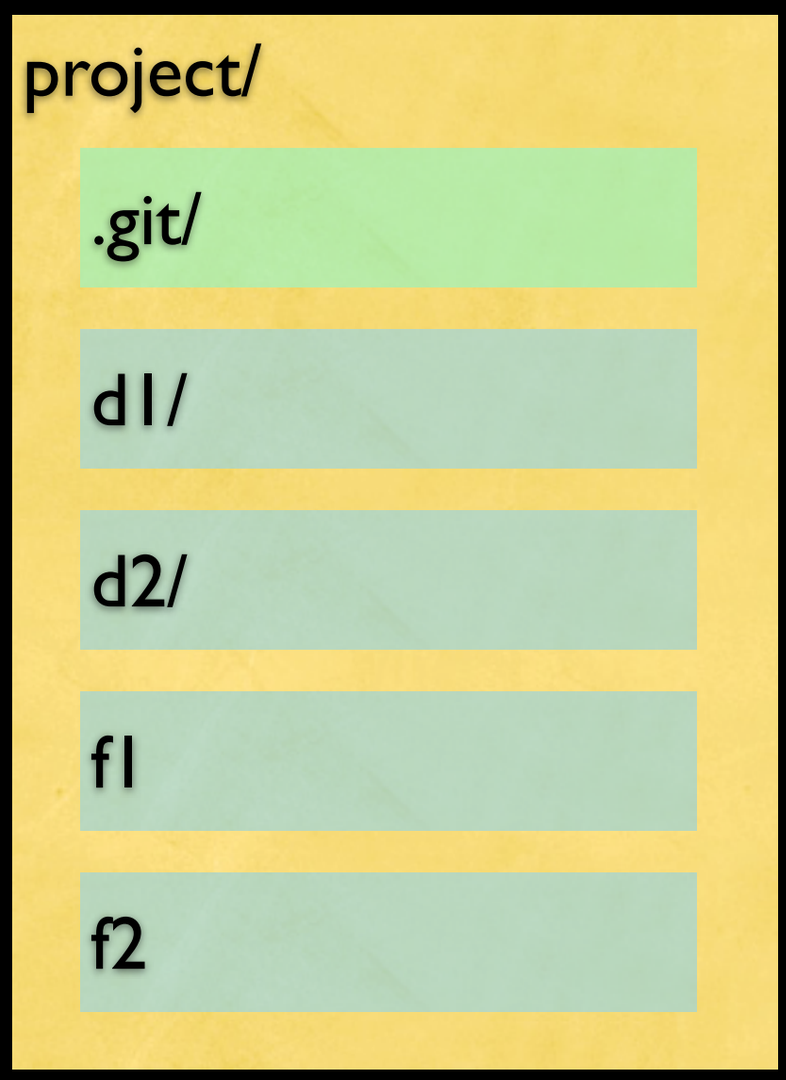
- Don't **checkout**, instead **clone**
  - `git clone git://user@server/repo.git [dest]`
- Complete copy of repository



The basic workflow with DVCS is both similar and different from normal VCS. Instead of checking out a current revision, you clone the entire repository. Every revision ever checked in, and all branches and tags, will be copied locally to your machine (in the .git directory). A "checkout" now is just getting a certain copy into the working directory so you can edit it, but it always comes from local files.

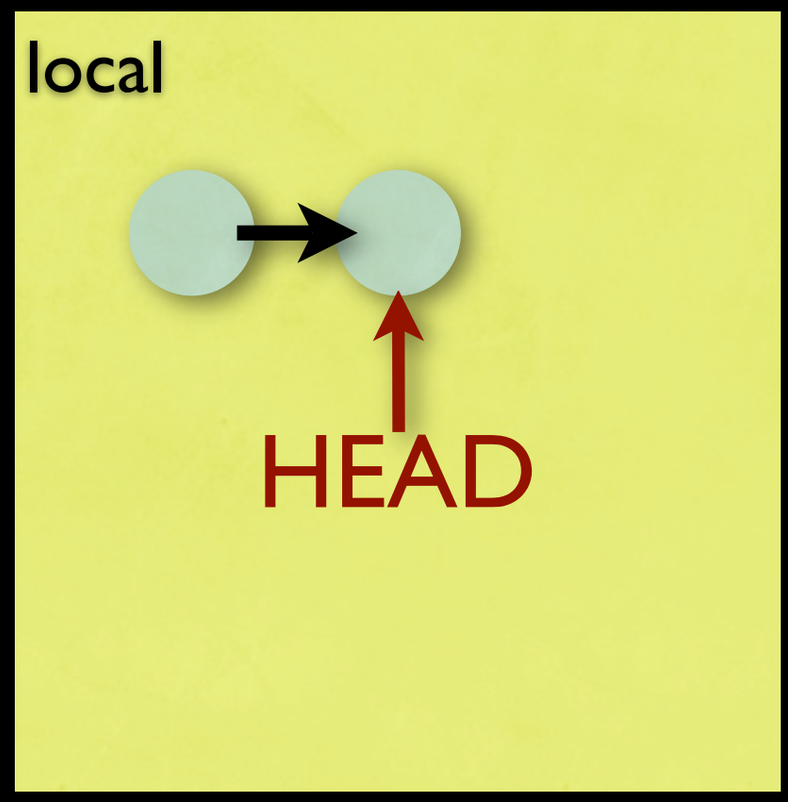
Local - Cloning

- Don't **checkout**, instead **clone**
  - `git clone git://user@server/repo.git [dest]`
- Complete copy of repository
- Checkout gets a working version
  - `git checkout master`



The basic workflow with DVCS is both similar and different from normal VCS. Instead of checking out a current revision, you clone the entire repository. Every revision ever checked in, and all branches and tags, will be copied locally to your machine (in the .git directory). A "checkout" now is just getting a certain copy into the working directory so you can edit it, but it always comes from local files.

Local - Commits



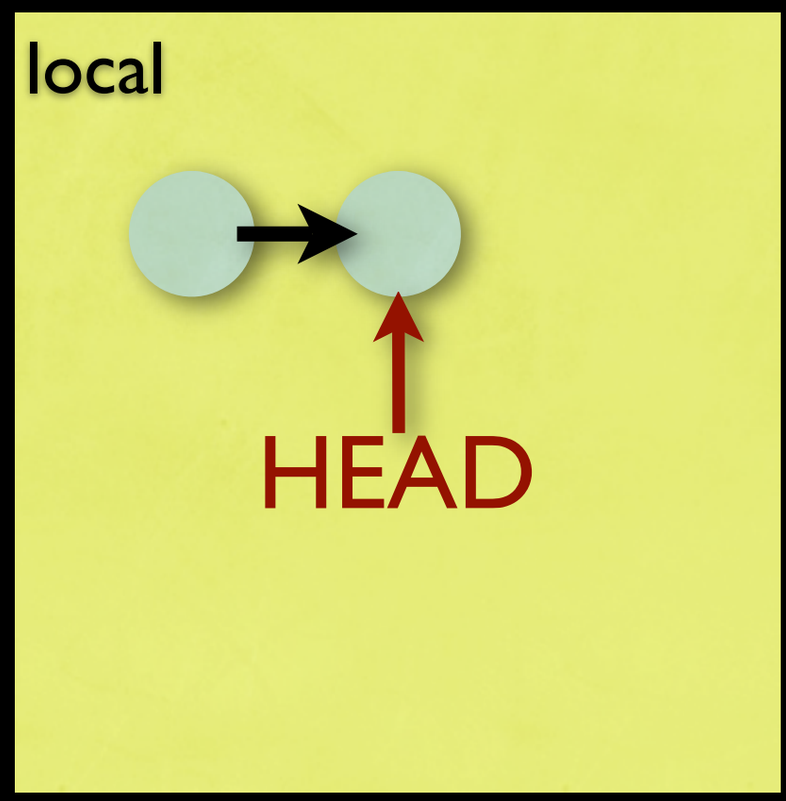
In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There's even a mode where you can select parts of patches to files to add called "interactive add."

Adding files puts those changes in the "staging area." The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won't lose them by editing files.

Finally, you can commit. I've specified a message on the command line, and git will always prompt you for a commit message. If you don't put a descriptive commit message, your developer friends won't like you.

Local - Commits

- Commits are local



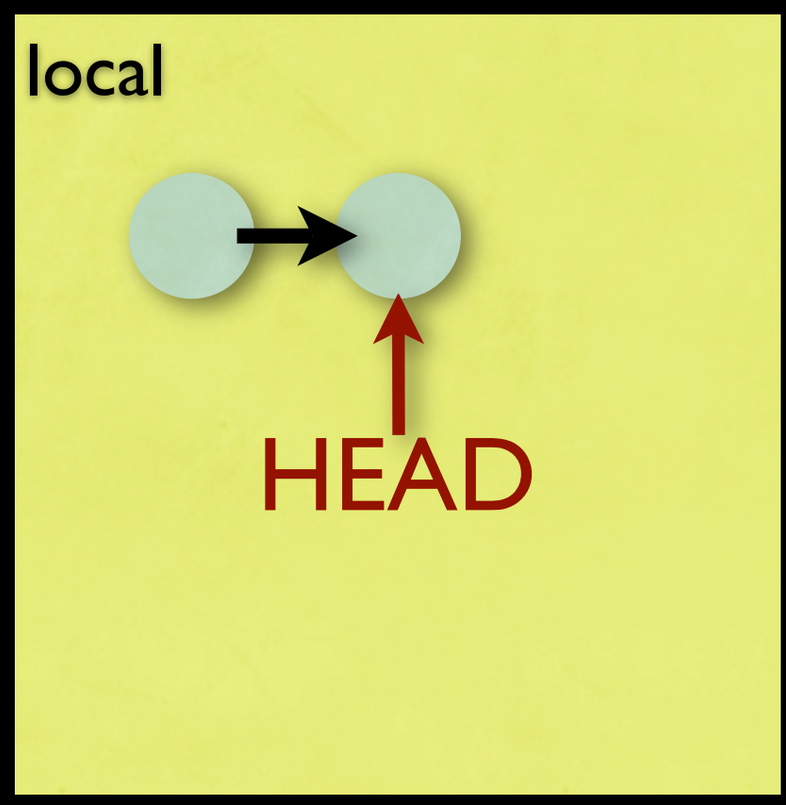
In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There's even a mode where you can select parts of patches to files to add called "interactive add."

Adding files puts those changes in the "staging area." The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won't lose them by editing files.

Finally, you can commit. I've specified a message on the command line, and git will always prompt you for a commit message. If you don't put a descriptive commit message, your developer friends won't like you.

Local - Commits

- Commits are local
- Add files or changes
  - `git add file1 file2`
- Staging area



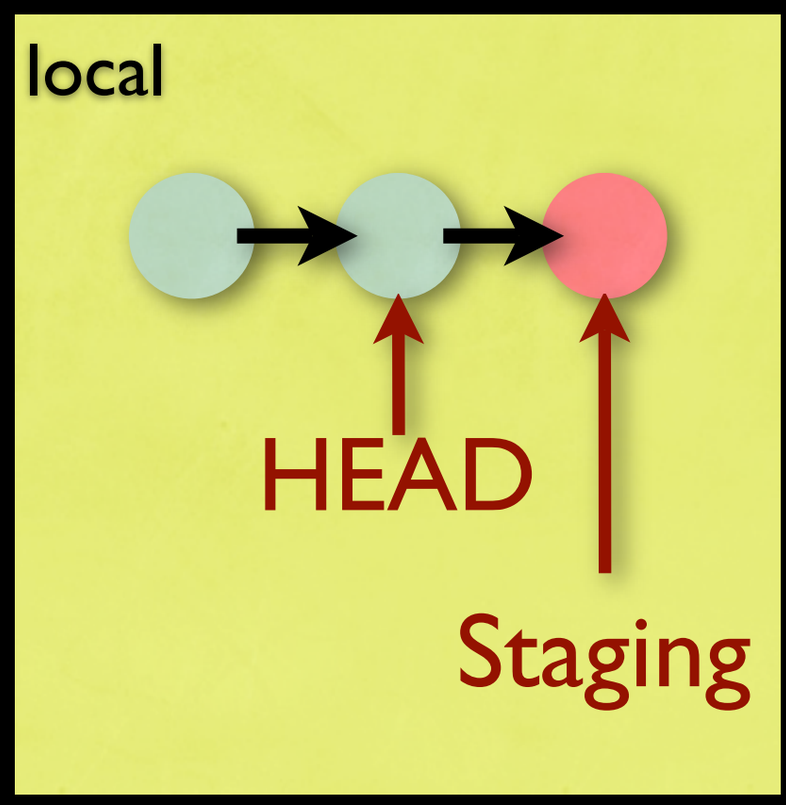
In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There's even a mode where you can select parts of patches to files to add called "interactive add."

Adding files puts those changes in the "staging area." The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won't lose them by editing files.

Finally, you can commit. I've specified a message on the command line, and git will always prompt you for a commit message. If you don't put a descriptive commit message, your developer friends won't like you.

Local - Commits

- Commits are local
- Add files or changes
  - `git add file1 file2`
- Staging area



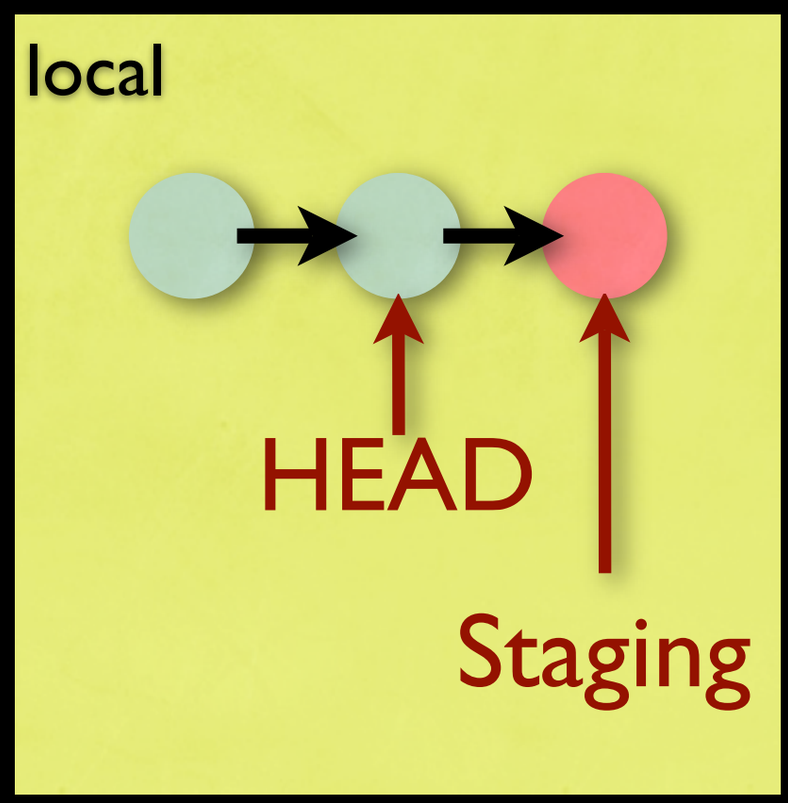
In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There's even a mode where you can select parts of patches to files to add called "interactive add."

Adding files puts those changes in the "staging area." The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won't lose them by editing files.

Finally, you can commit. I've specified a message on the command line, and git will always prompt you for a commit message. If you don't put a descriptive commit message, your developer friends won't like you.

Local - Commits

- Commits are local
- Add files or changes
  - `git add file1 file2`
- Staging area
- Commit
  - `git commit -m “Useful commit message.”`



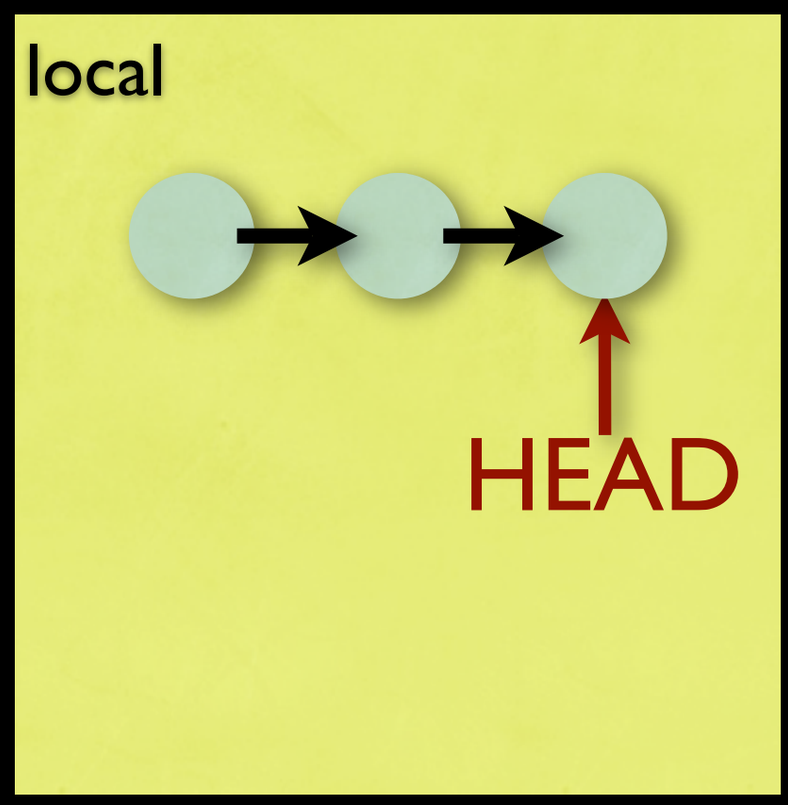
In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There’s even a mode where you can select parts of patches to files to add called “interactive add.”

Adding files puts those changes in the “staging area.” The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won’t lose them by editing files.

Finally, you can commit. I’ve specified a message on the command line, and git will always prompt you for a commit message. If you don’t put a descriptive commit message, your developer friends won’t like you.

Local - Commits

- Commits are local
- Add files or changes
  - `git add file1 file2`
- Staging area
- Commit
  - `git commit -m “Useful commit message.”`



In git, and other DVCS, commits are local. This means that they only affect your local repository. Git is a bit odd in that it requires you to add and then commit files. This gives you finer granularity when selecting what to commit. There’s even a mode where you can select parts of patches to files to add called “interactive add.”

Adding files puts those changes in the “staging area.” The staging area is used in a number of commands as temporary storage before completing an operation. You can think of it like an in-progress commit. While the changes are there, they are trivial to back out, but you also won’t lose them by editing files.

Finally, you can commit. I’ve specified a message on the command line, and git will always prompt you for a commit message. If you don’t put a descriptive commit message, your developer friends won’t like you.

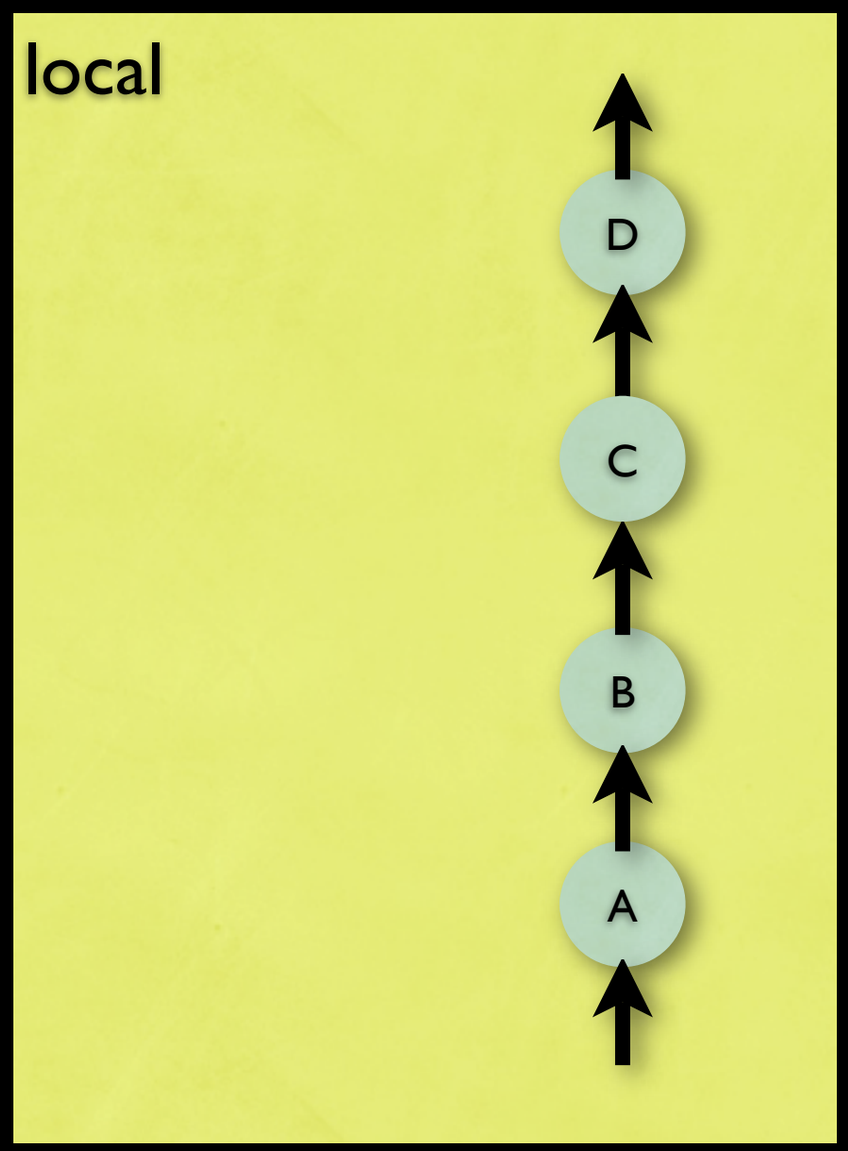
Local - Blame

- Blame - who committed this obviously broken code? Line-by-line committer
  - `git blame file`

```
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 144)    // 1. If the source is the space, somebody is messin
0facfde9 (Daniel Reiter Horn    2009-12-15 17:26:46 -0800 145)    bool space_source = (front.obj_msg->source_object()
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 146)    if (space_source) {
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 147)        SILOG(cbr,error,"Got message from object host cl
0facfde9 (Daniel Reiter Horn    2009-12-15 17:26:46 -0800 148)        delete front.obj_msg;
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 149)        return;
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 150)    }
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 151)
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 152)    // 2. For connection bootstrapping purposes we need
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 153)    // Note that we need to check this before the connec
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 154)    // be connected yet.  We dispatch directly from here
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 155)    // connection to be passed along as well.
0facfde9 (Daniel Reiter Horn    2009-12-15 17:26:46 -0800 156)    bool space_dest = (front.obj_msg->dest_object() == U
0facfde9 (Daniel Reiter Horn    2009-12-15 17:26:46 -0800 157)    bool session_msg = (front.obj_msg->dest_port() == OB
78b38d90 (behram mistree        2009-09-14 15:27:42 -0700 158)    if (space_dest && session_msg)
78b38d90 (behram mistree        2009-09-14 15:27:42 -0700 159)    {
0facfde9 (Daniel Reiter Horn    2009-12-15 17:26:46 -0800 160)        handleSessionMessage(front.conn_id, front.obj_ms
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 161)        return;
3aa995bb (Ewen Cheslack-Postava 2009-09-04 15:16:11 -0700 162)    }
5abdef87 (Ewen Cheslack-Postava 2009-09-10 13:25:48 -0700 163)
```


Two handy commands you can now use locally are blame and bisect. Blame tells you who committed each line.

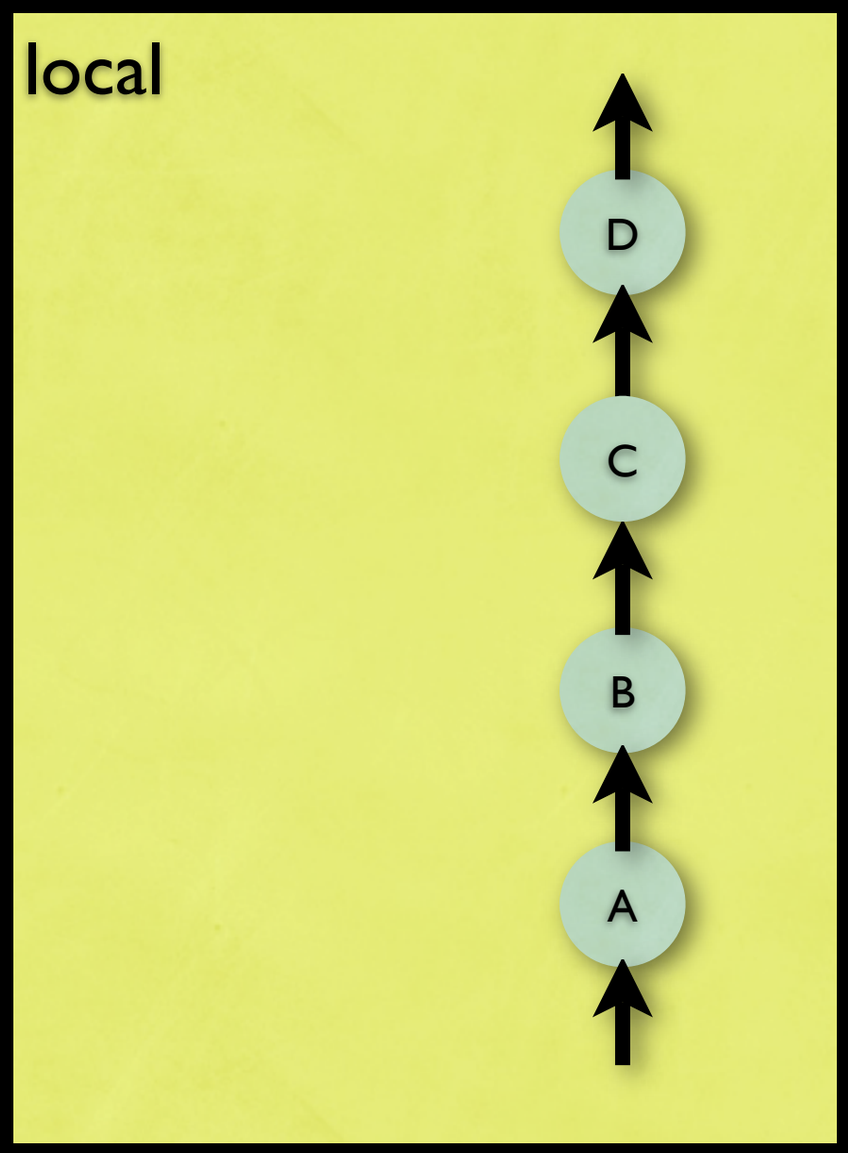
Local - Bisect



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

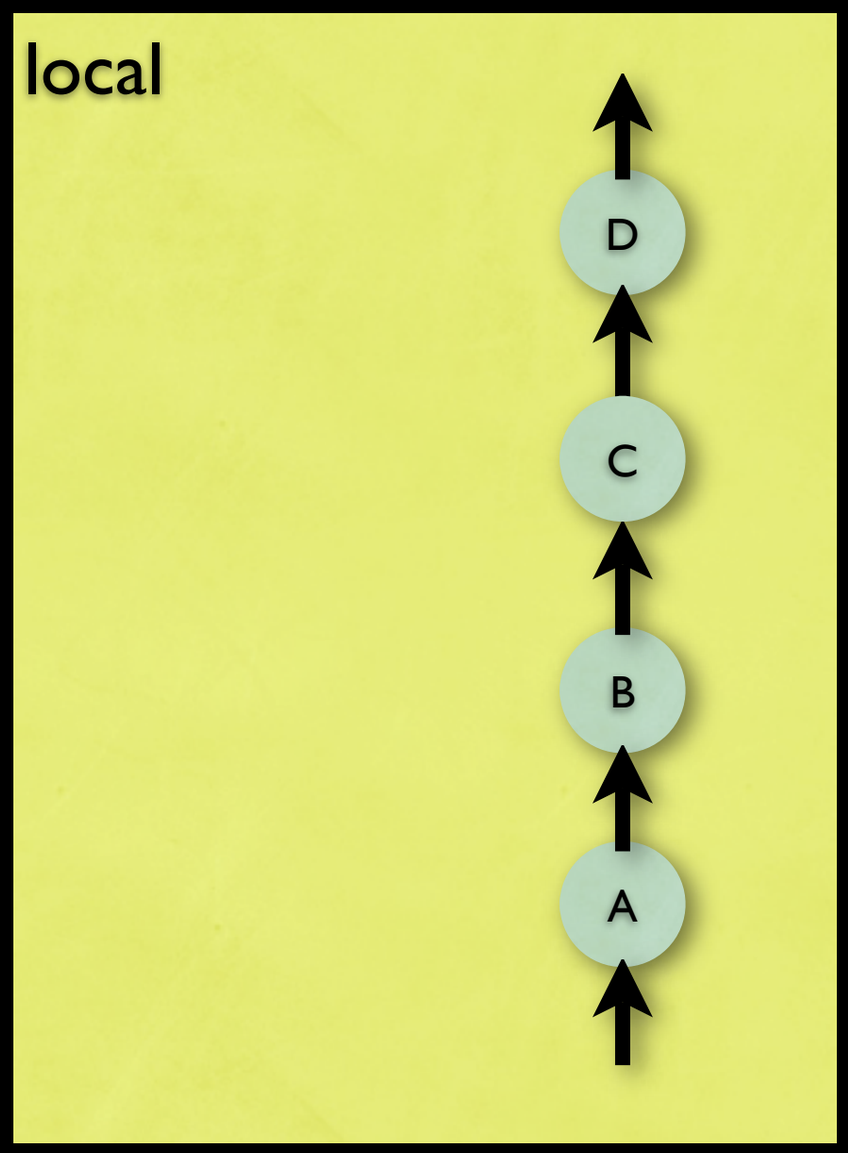
- Bisect - which commit caused the error



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

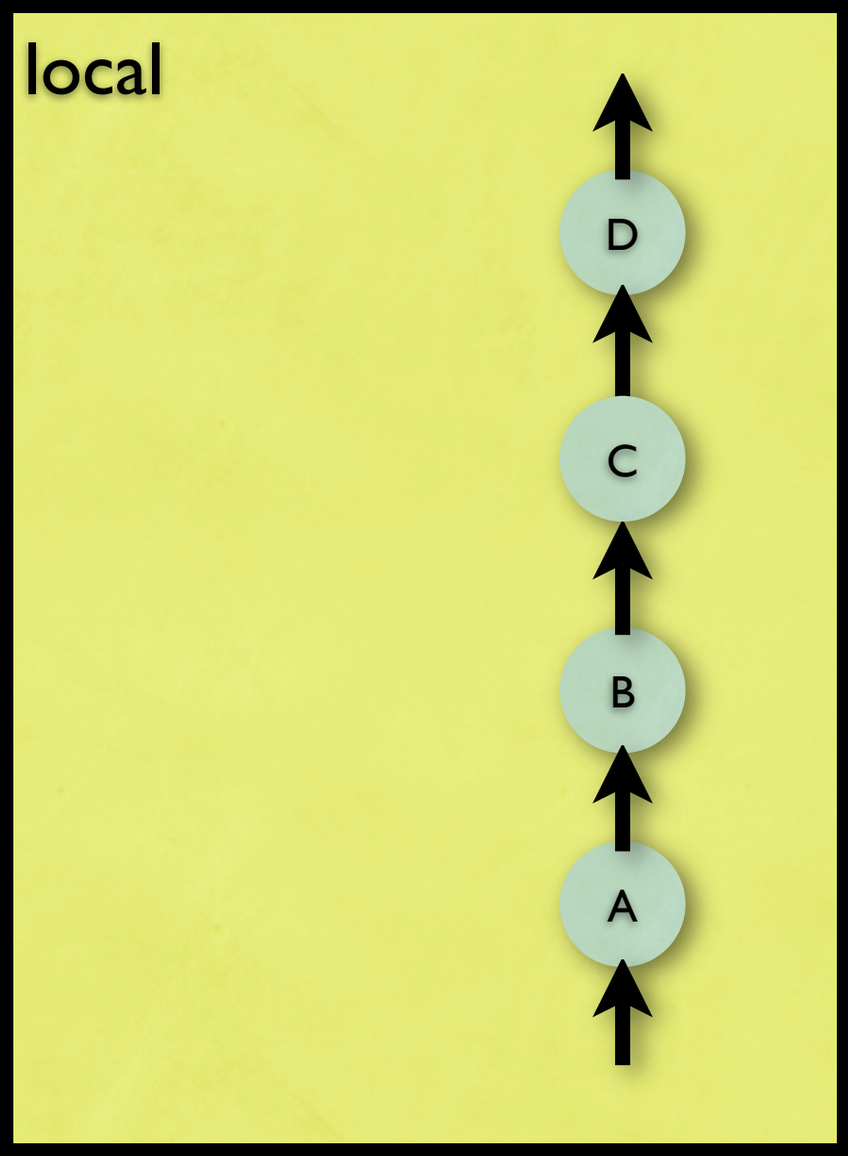
- Bisect - which commit caused the error
- `git bisect start`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

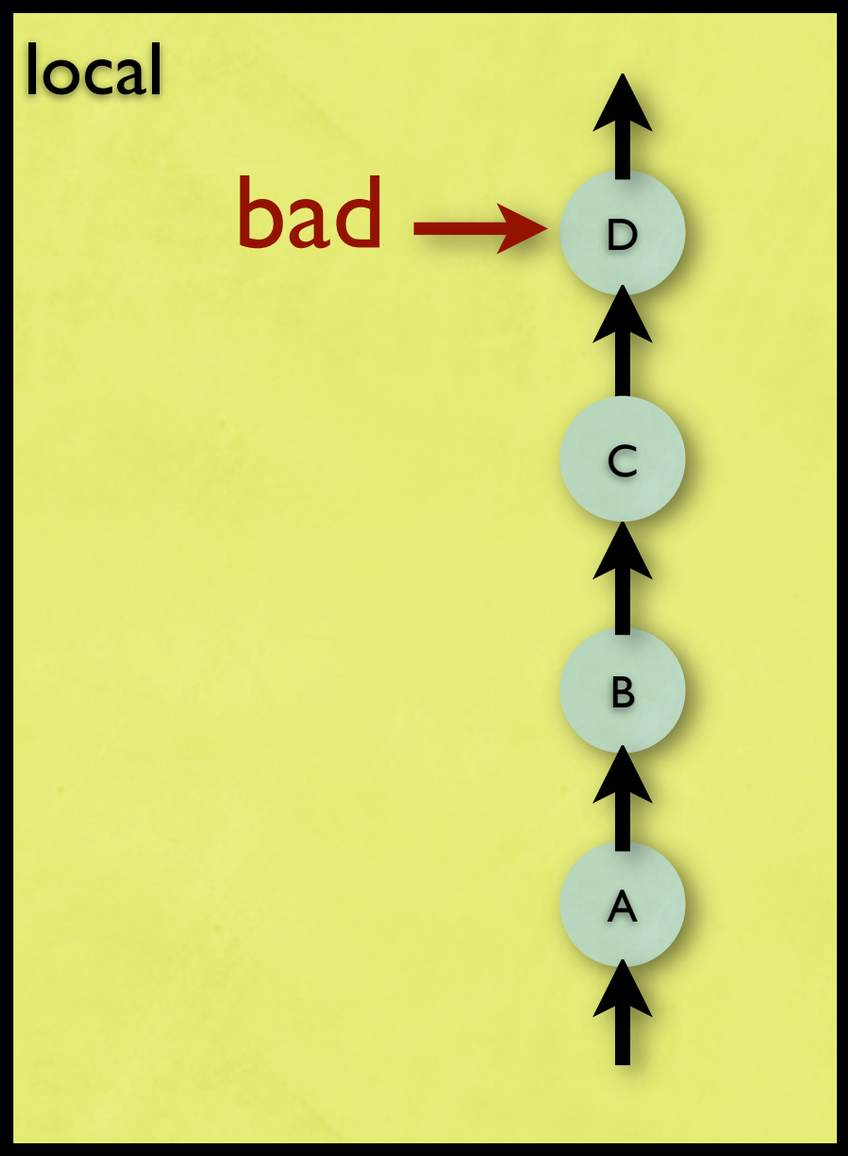
- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

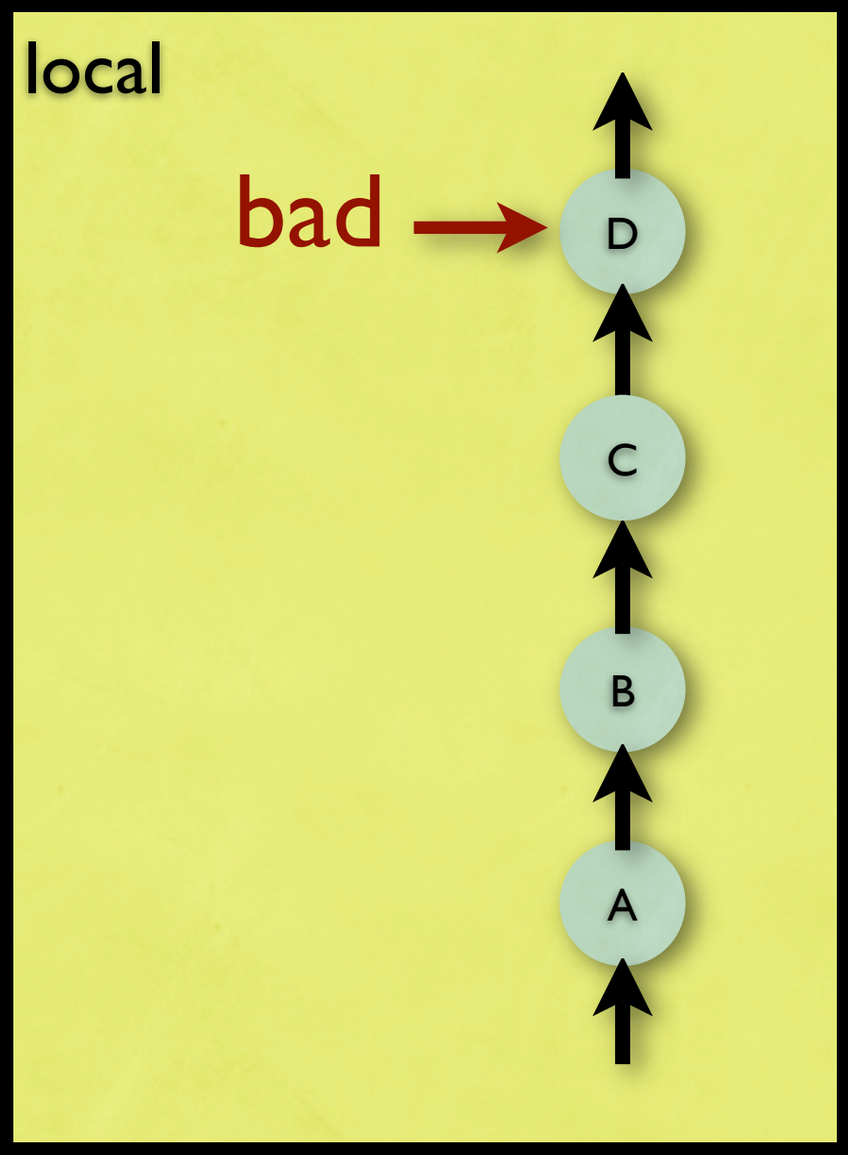
- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

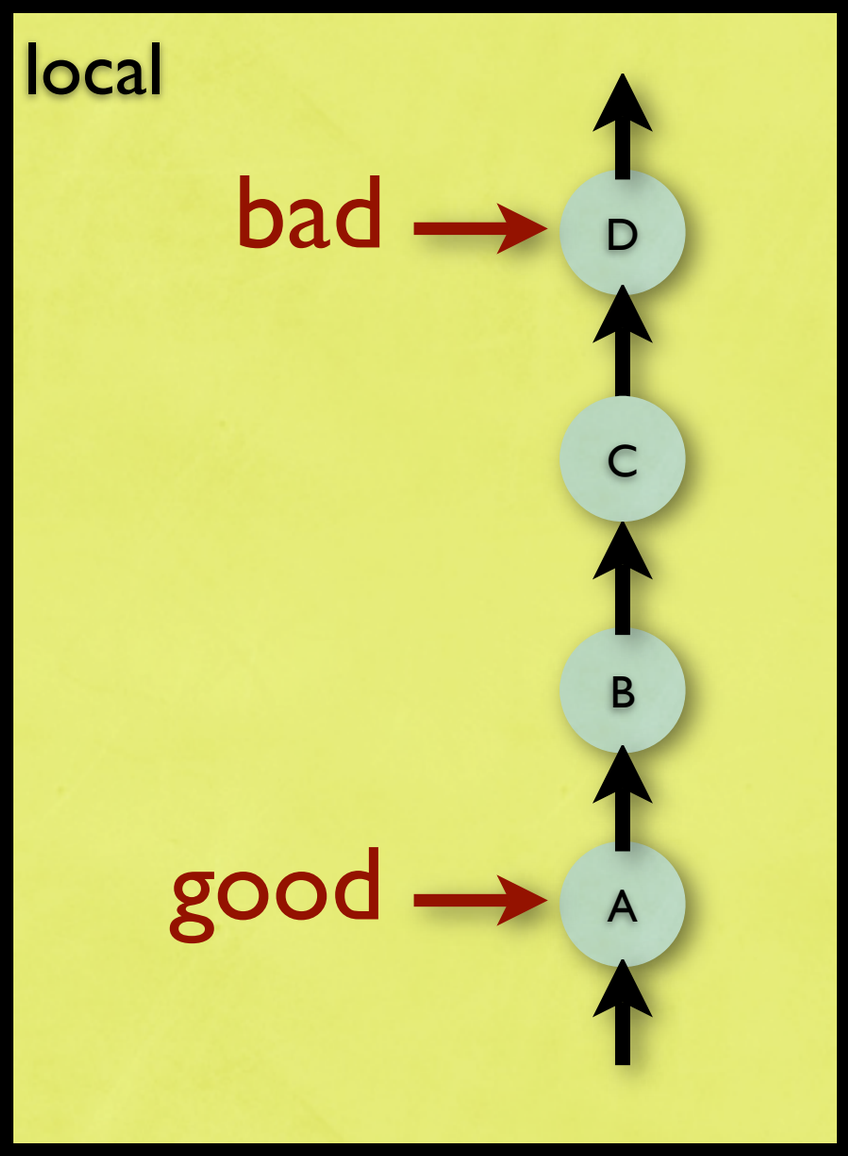
- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`
  - `git bisect good [rev]`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

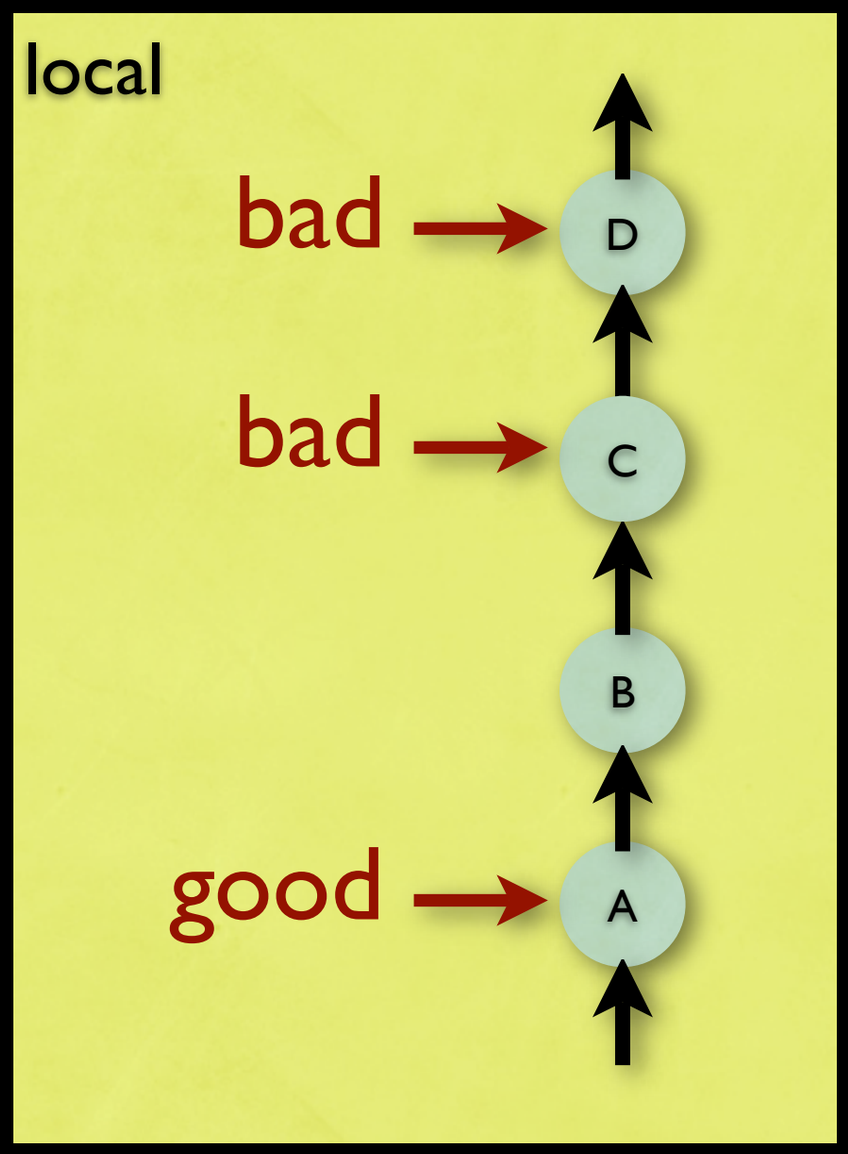
- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`
  - `git bisect good [rev]`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Bisect

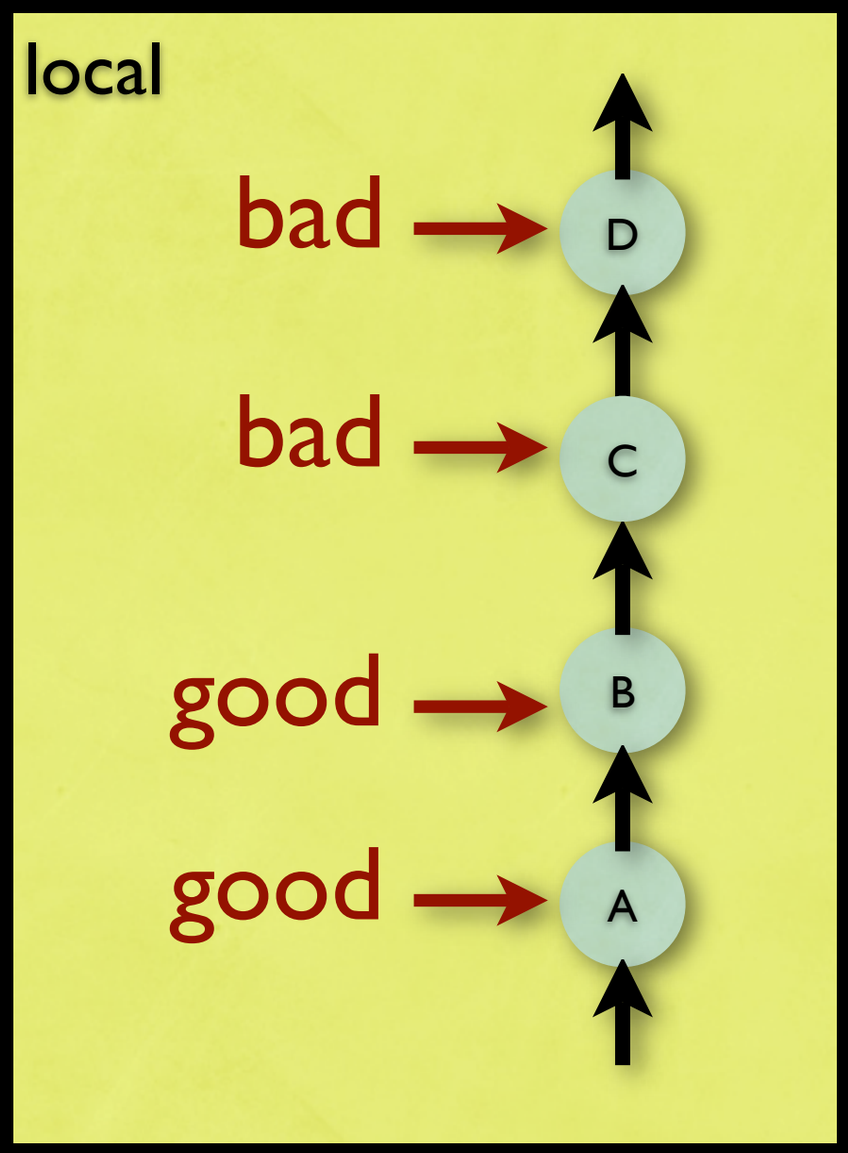
- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`
  - `git bisect good [rev]`



Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

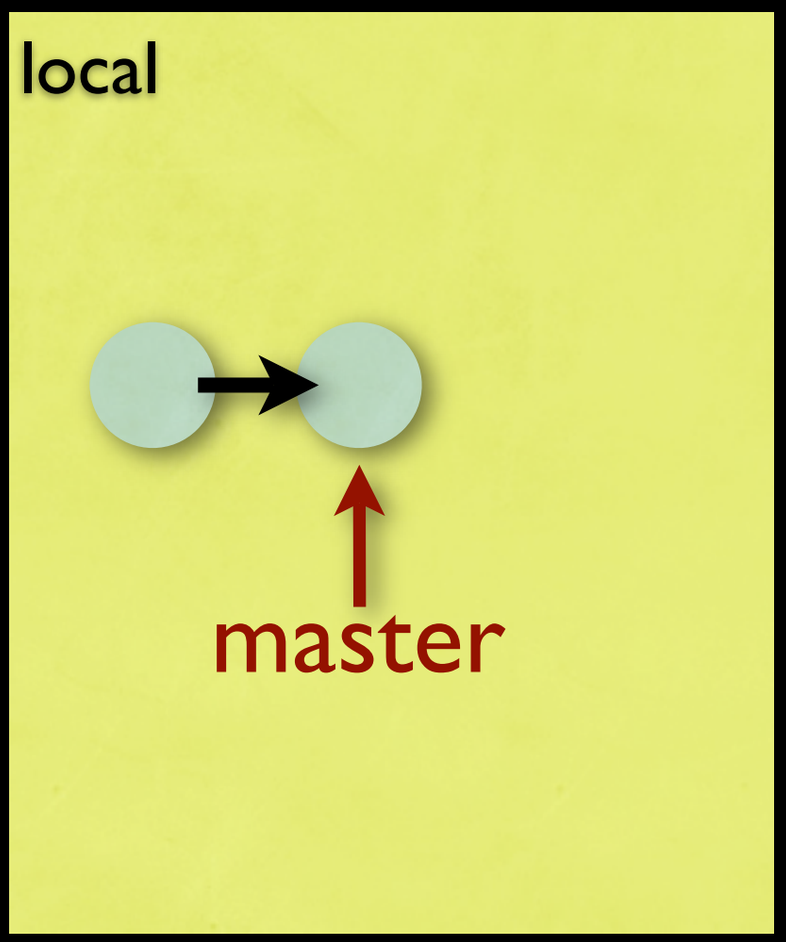
Local - Bisect

- Bisect - which commit caused the error
  - `git bisect start`
  - `git bisect bad [rev]`
  - `git bisect good [rev]`



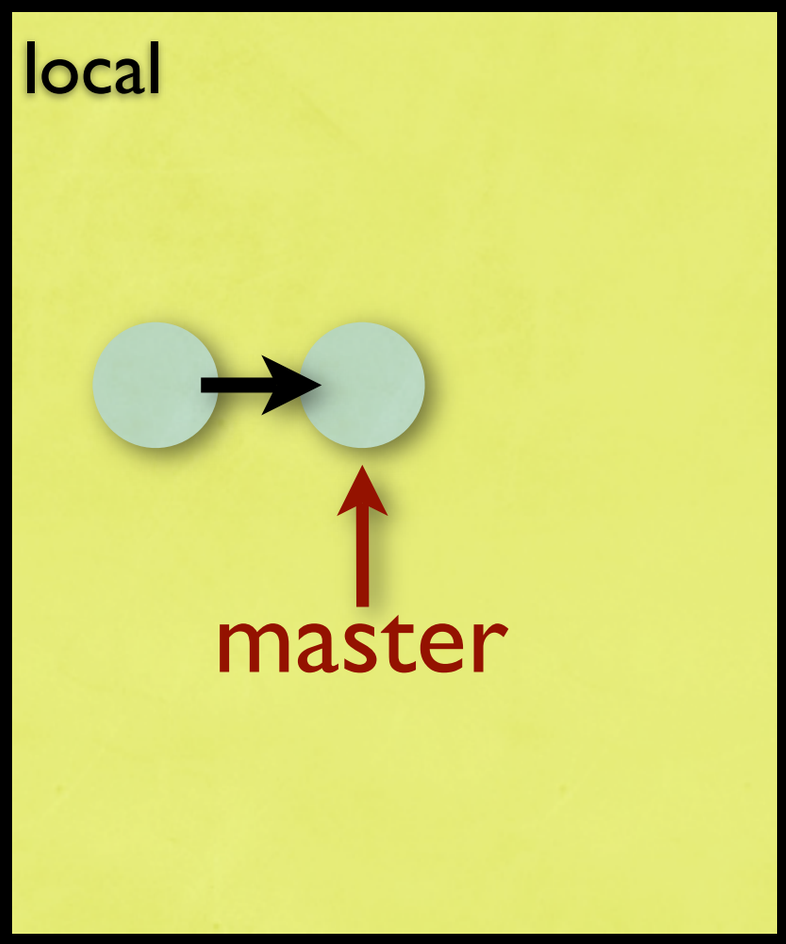
Bisect helps you determine which commit broke some functionality to help you track the cause of the bug. Neither of these are unique to git or DVCS, but they are easier when you have everything available locally.

Local - Branches



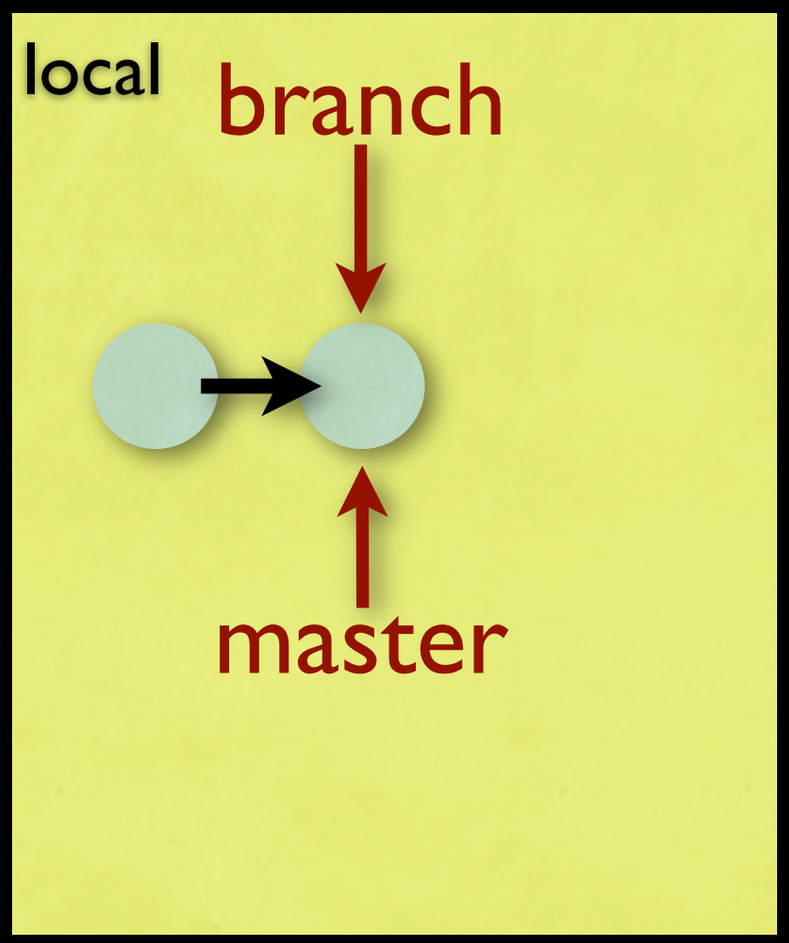
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`



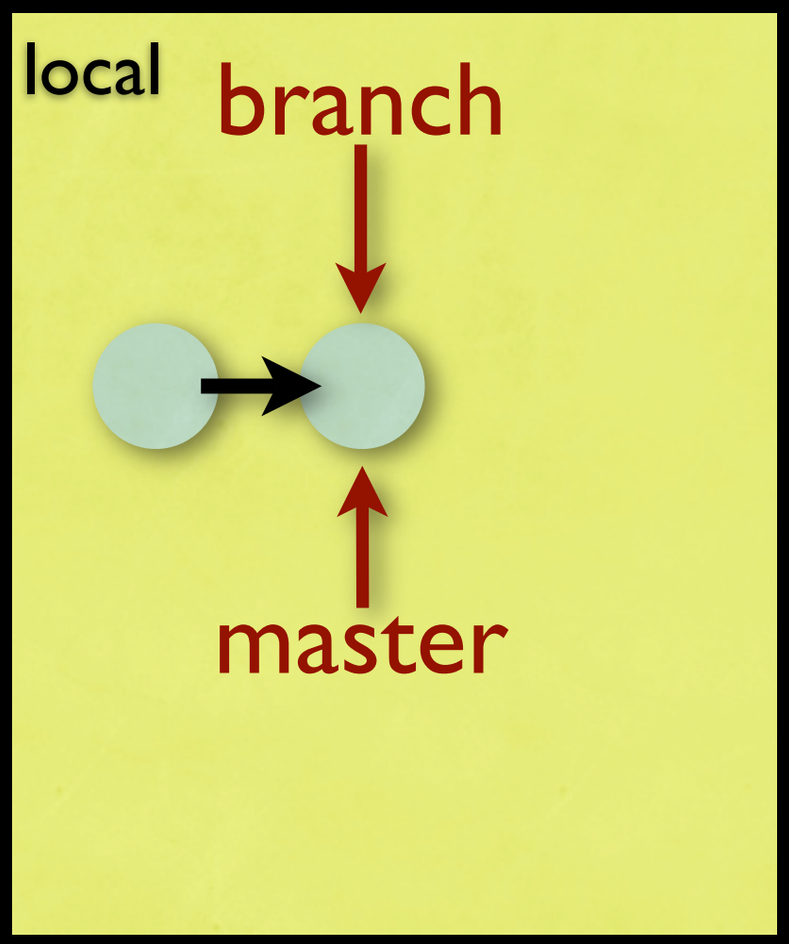
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`



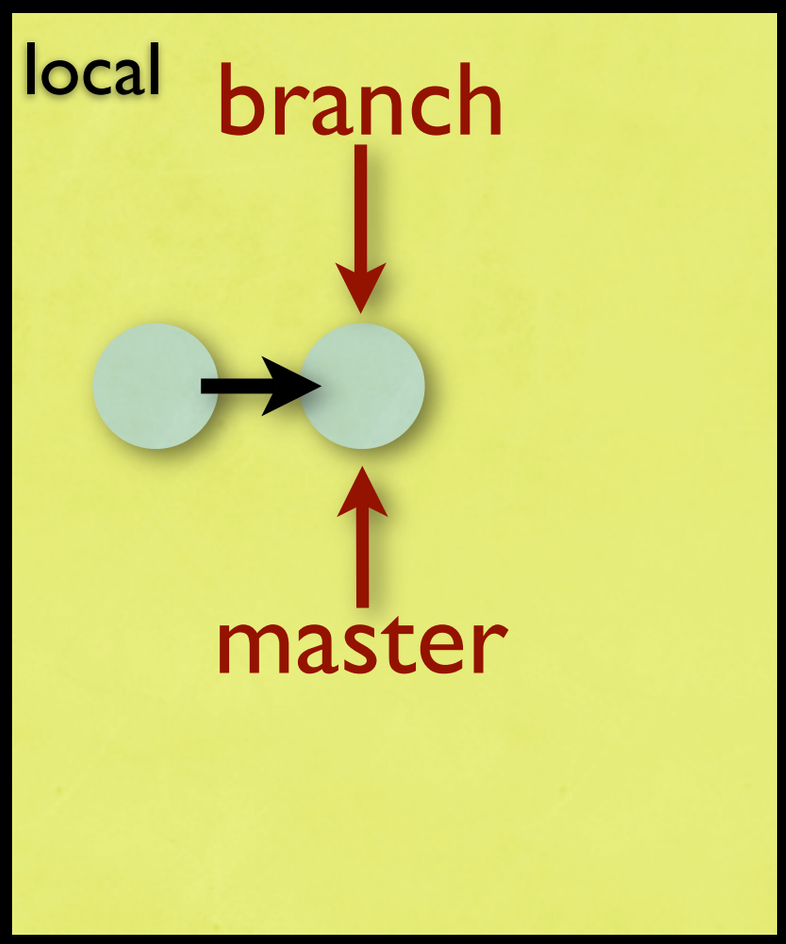
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`
- Work on a branch
  - `git checkout bname`



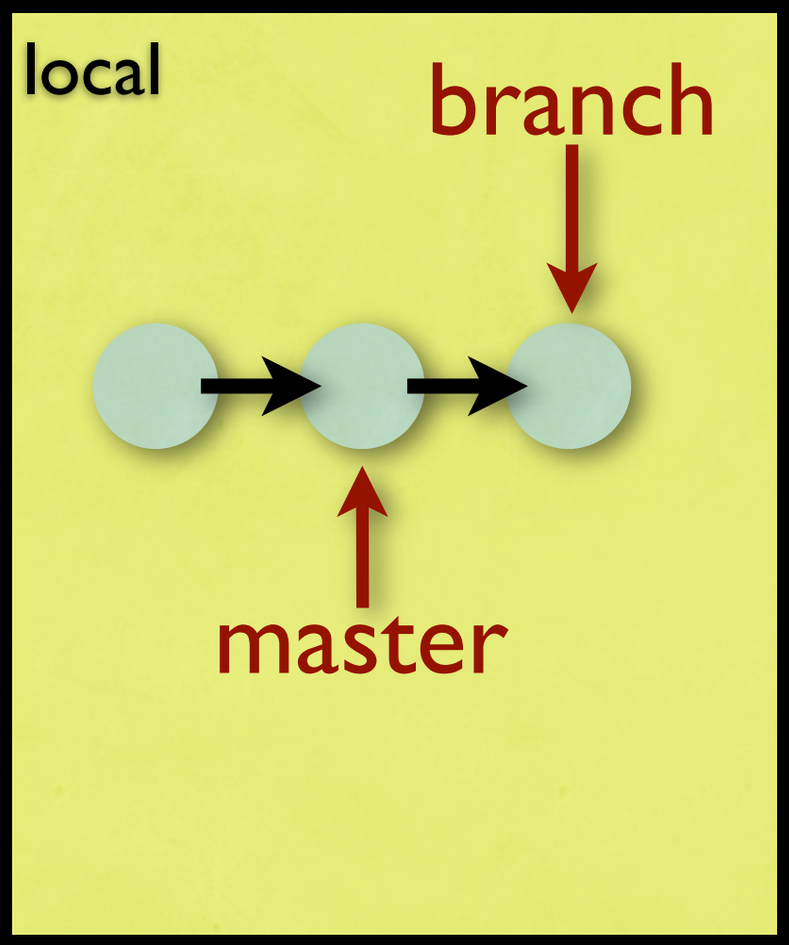
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`
- Work on a branch
  - `git checkout bname`
- Commit
  - `git commit -a -m “Feature.”`



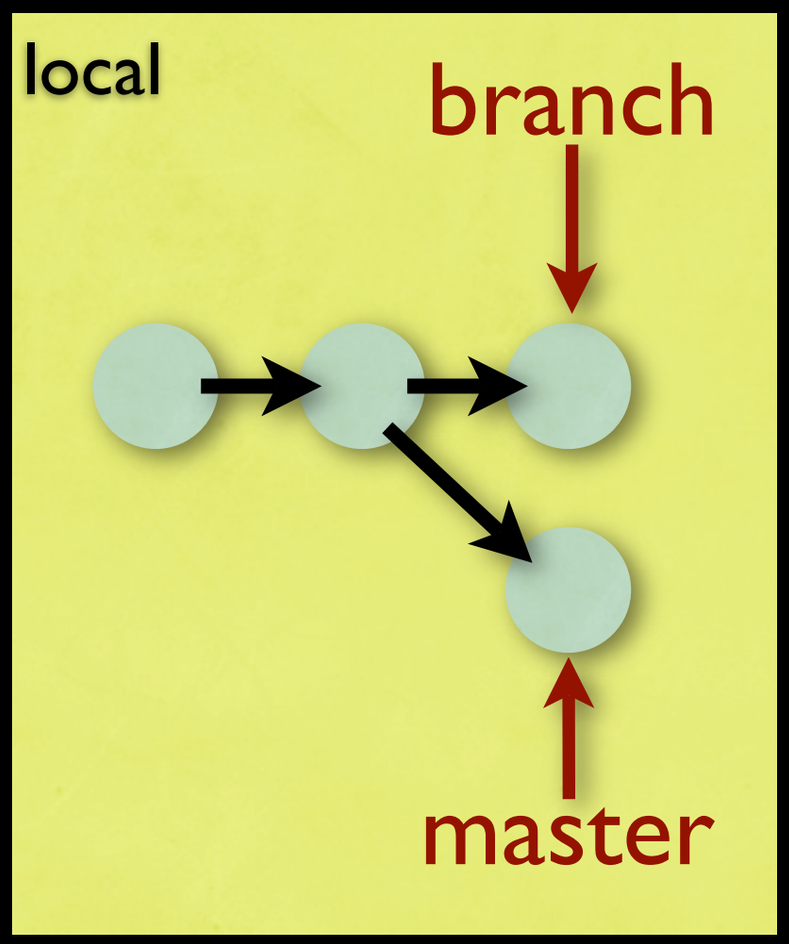
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`
- Work on a branch
  - `git checkout bname`
- Commit
  - `git commit -a -m “Feature.”`



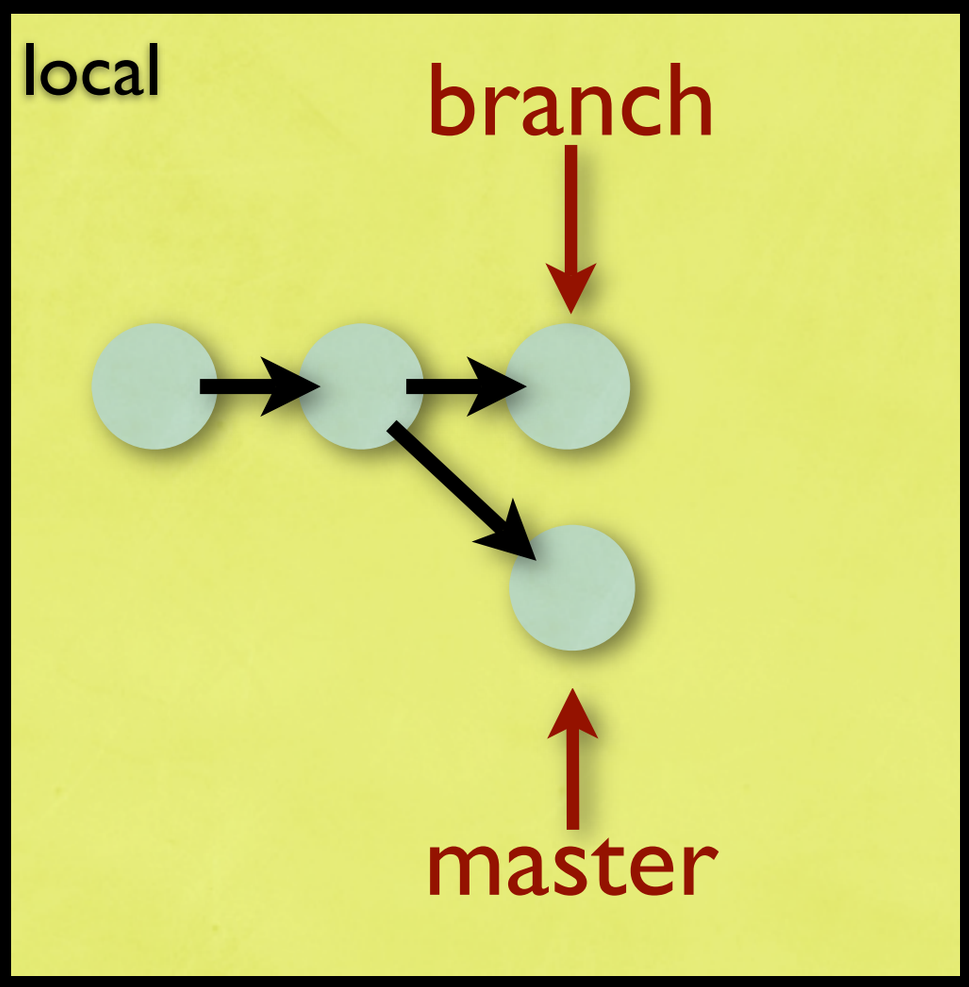
Local - Branches

- Full repository - create branches locally
  - `git branch bname [start]`
- Work on a branch
  - `git checkout bname`
- Commit
  - `git commit -a -m “Feature.”`



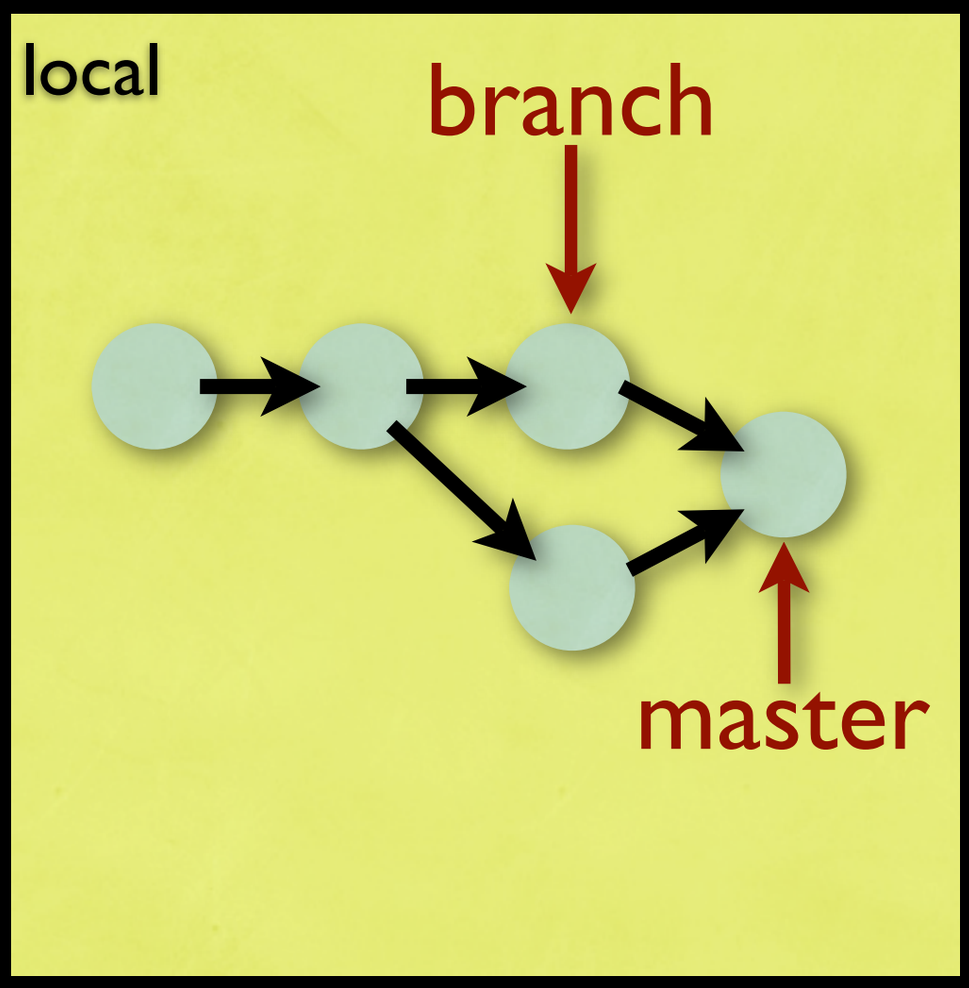
Local - Merging

- Merge branch into master
  - `git checkout master`
  - `git merge branch`
- Intelligent - knows branch point, works with patch graph
- Merge failure prompts to fix conflicts



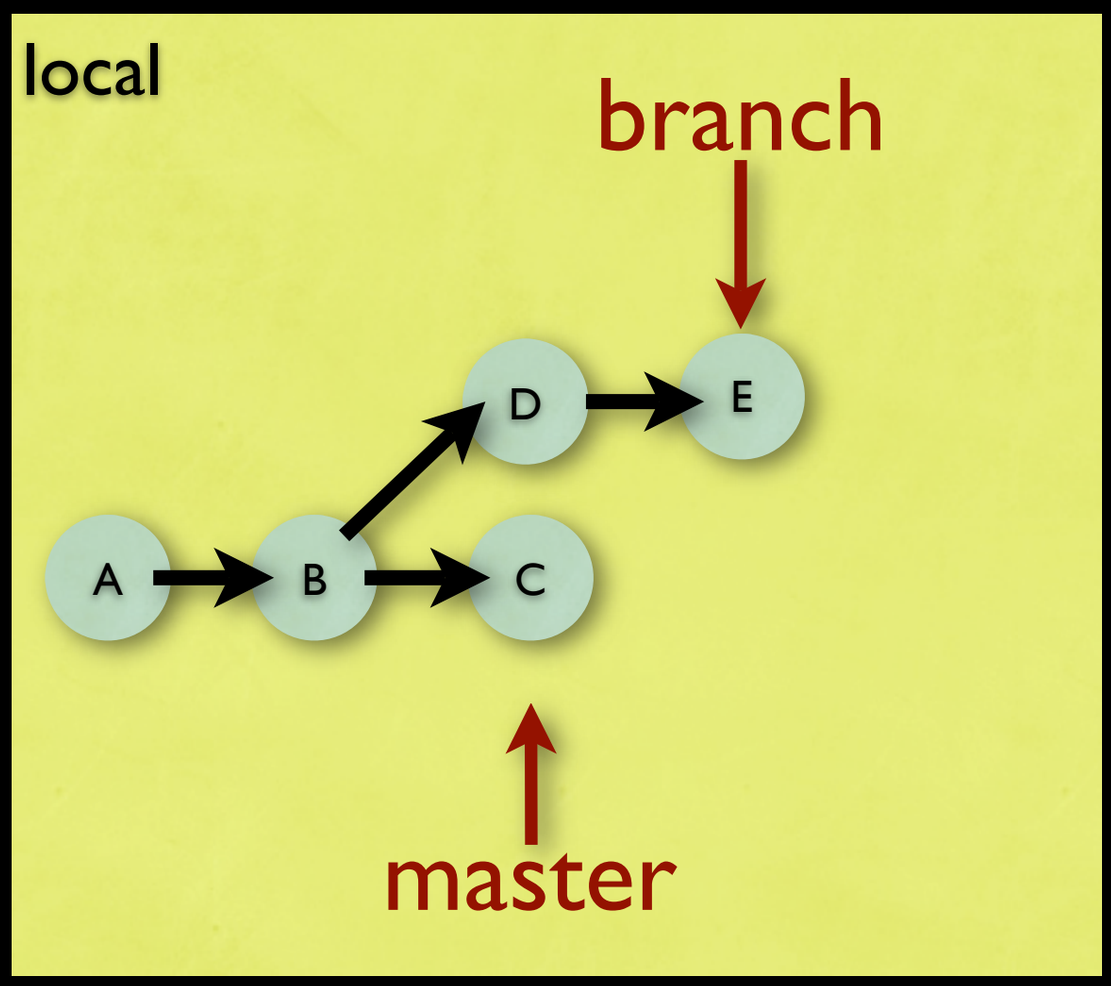
Local - Merging

- Merge branch into master
  - `git checkout master`
  - `git merge branch`
- Intelligent - knows branch point, works with patch graph
- Merge failure prompts to fix conflicts



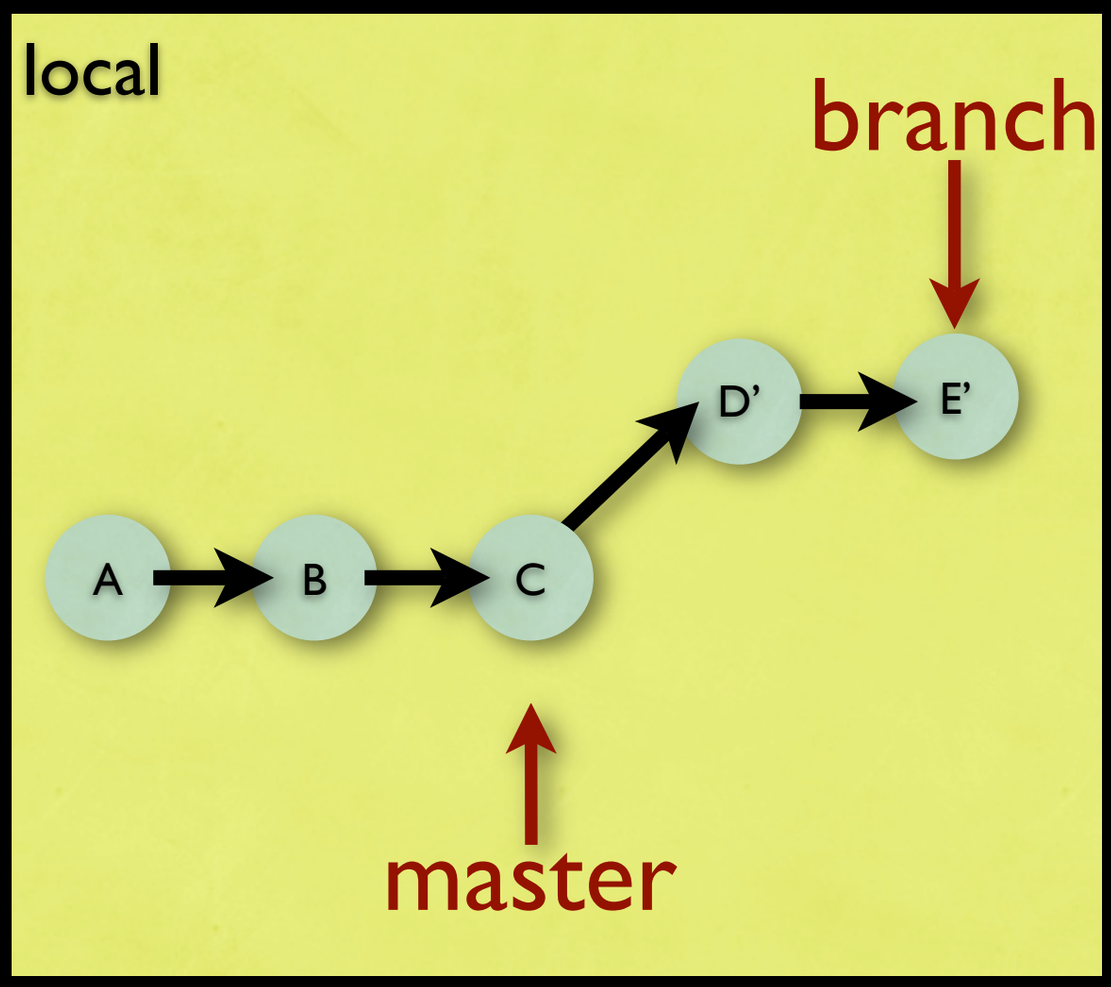
Local - Rebasing

- "Cut and paste" a branch onto another
  - `git rebase onto [source]`
- Apply patches in order
- Use `onto` as new branch point
- Prompt on patch failure



Local - Rebasing

- "Cut and paste" a branch onto another
  - `git rebase onto [source]`
- Apply patches in order
- Use `onto` as new branch point
- Prompt on patch failure



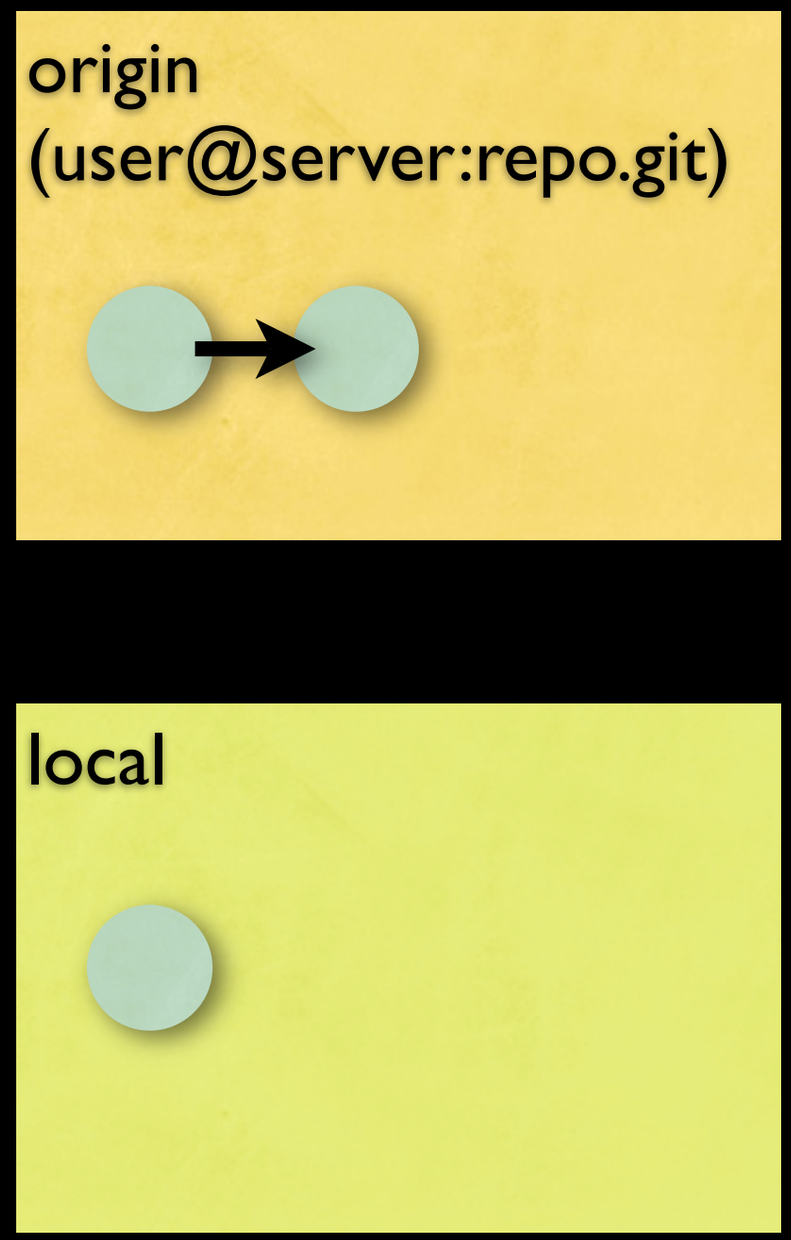
Interactive Rebasing

- Interactive mode lets you "refactor" commits
  - pick - use the commit as is
  - squash - combine with previous
    - edit - stop during rebase to edit files or commit message
- Get list in editor; rearrange; remove commits; set command on each patch
- I find normal rebase followed by interactive to refactor is easiest

Merge vs. Rebase

- Rebasing
  - Keeps a linear history
  - Rewrite history - untested commits!
  - Use for: short-term conflicts
- Merging
  - Adds merge commit
  - Maintains parallel history
  - Use for: long term parallel development

Workflows - Sharing

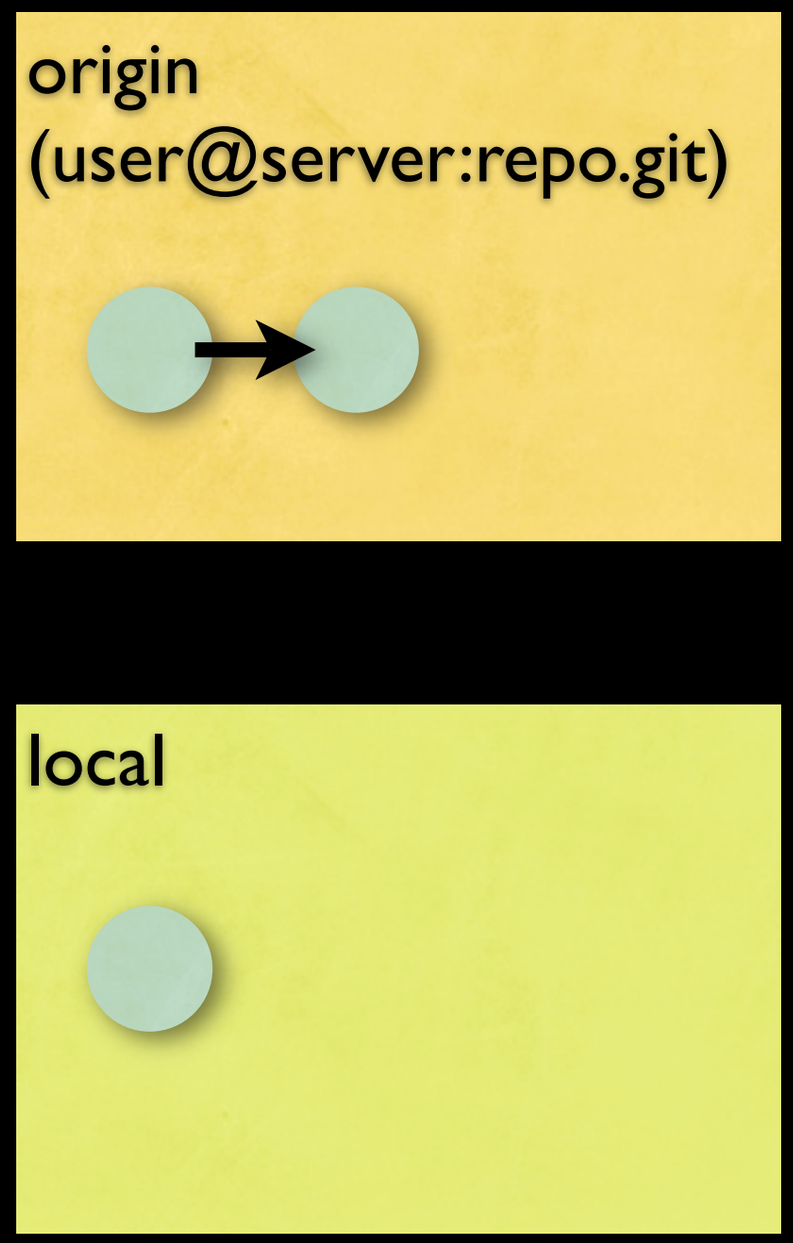


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories

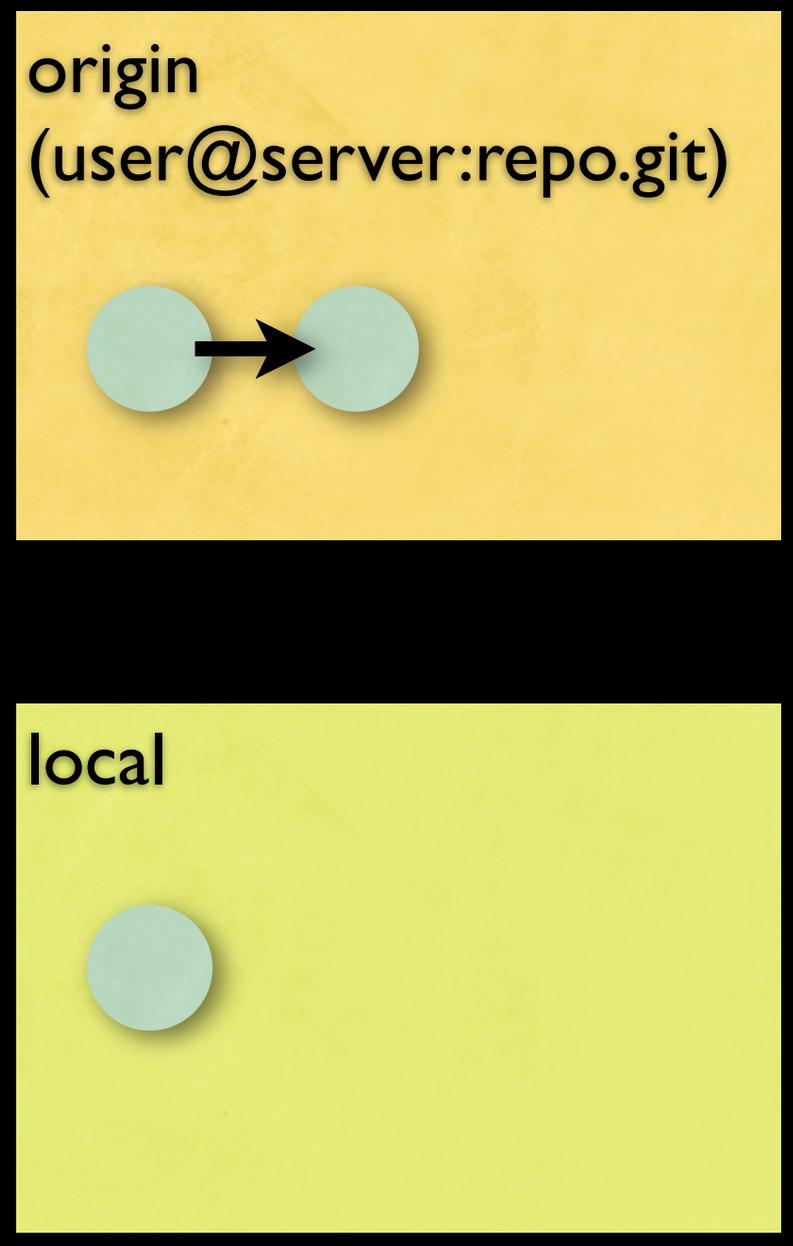


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`

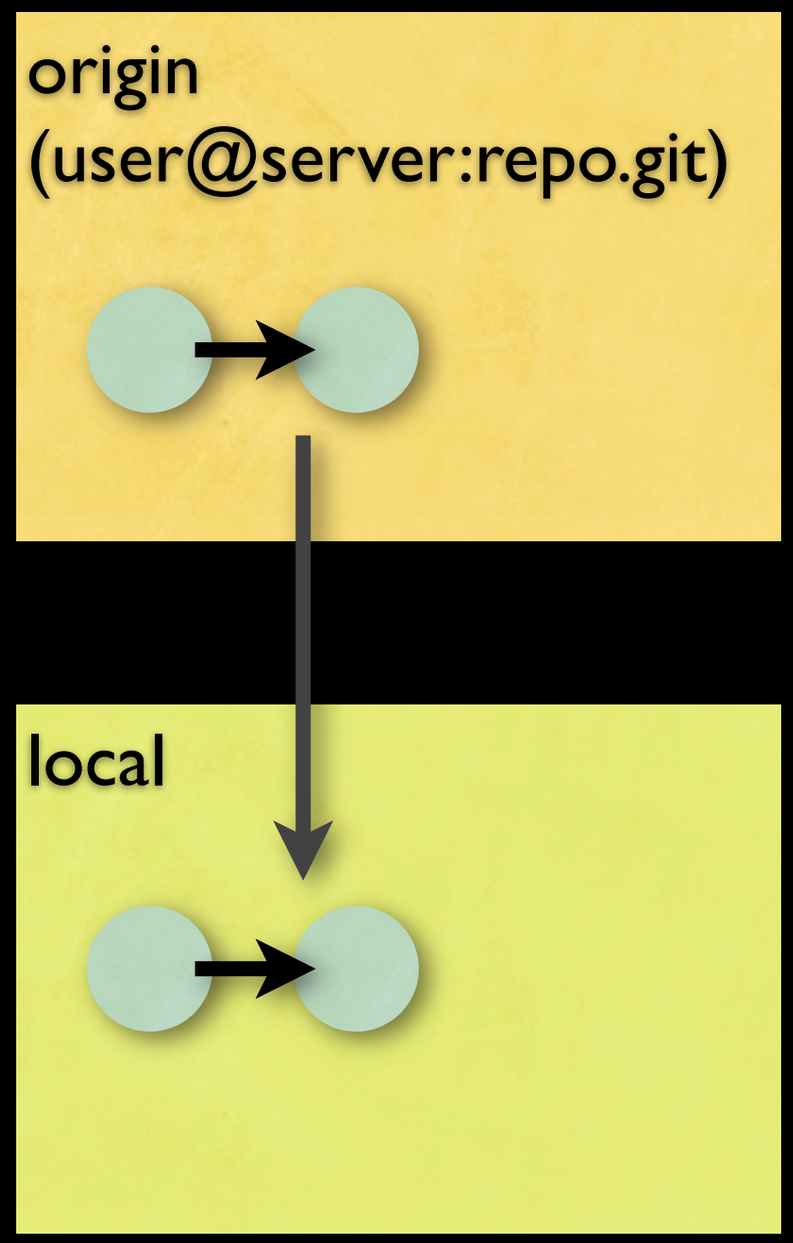


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`

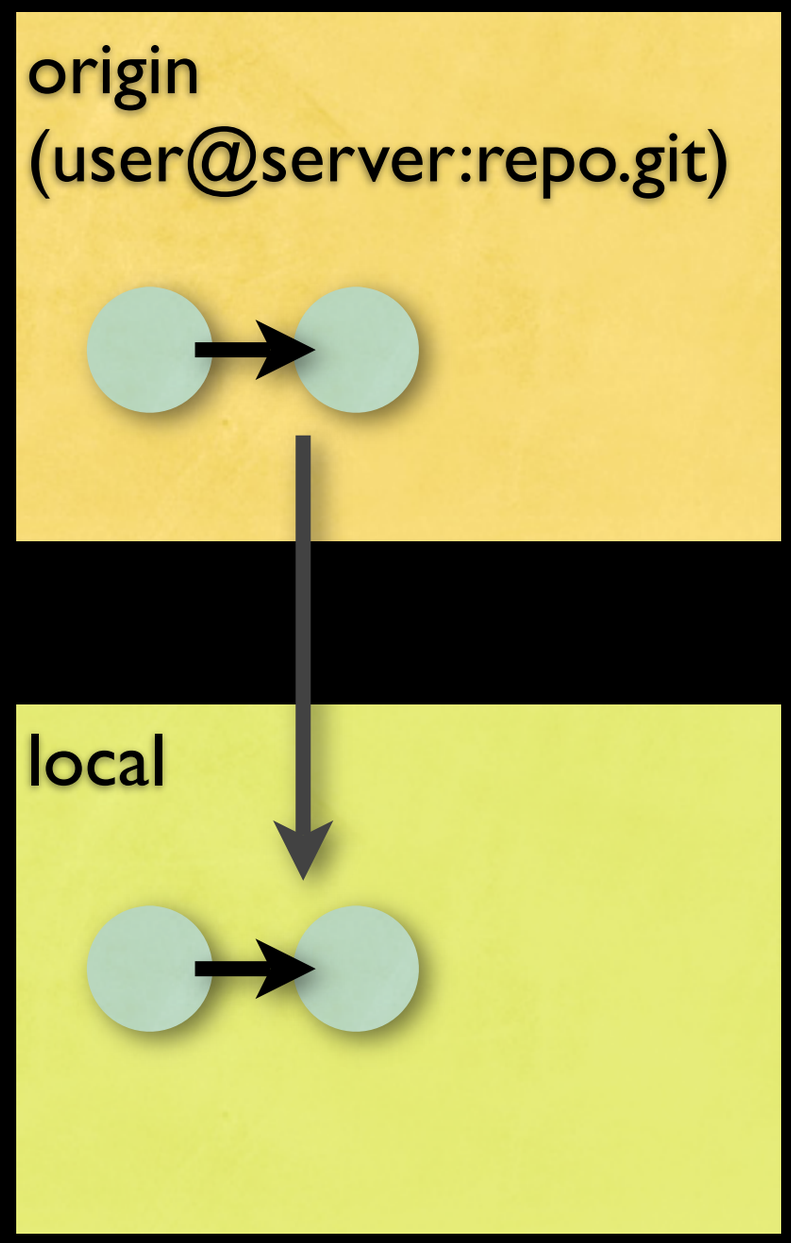


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`
  - `git push origin [src:dest]`

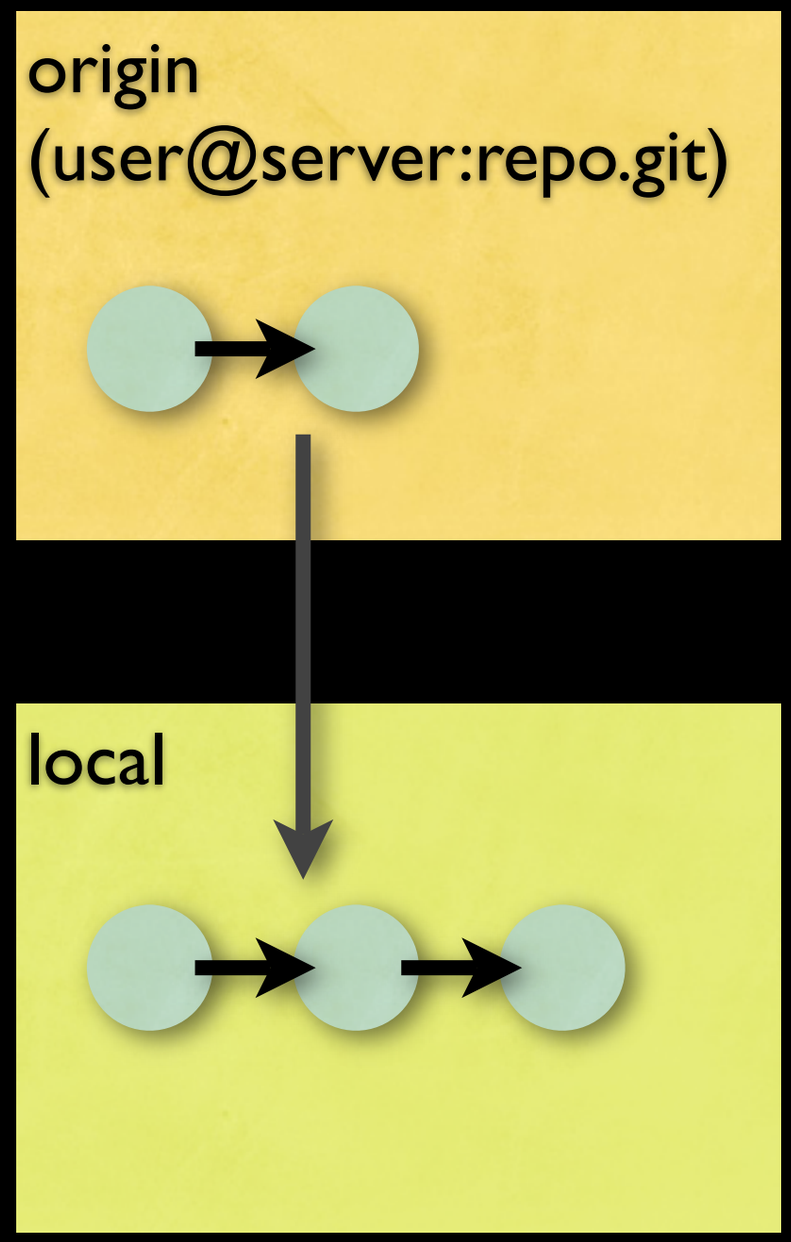


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`
  - `git push origin [src:dest]`

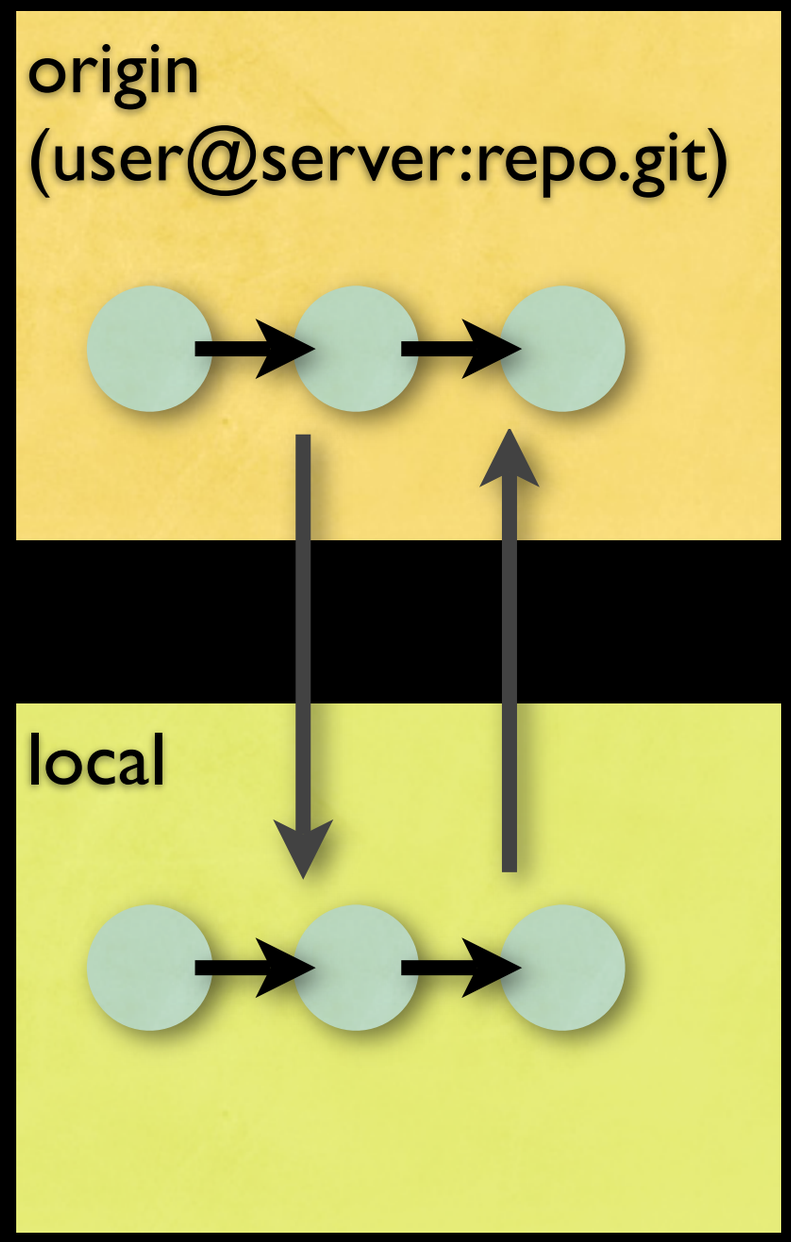


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`
  - `git push origin [src:dest]`

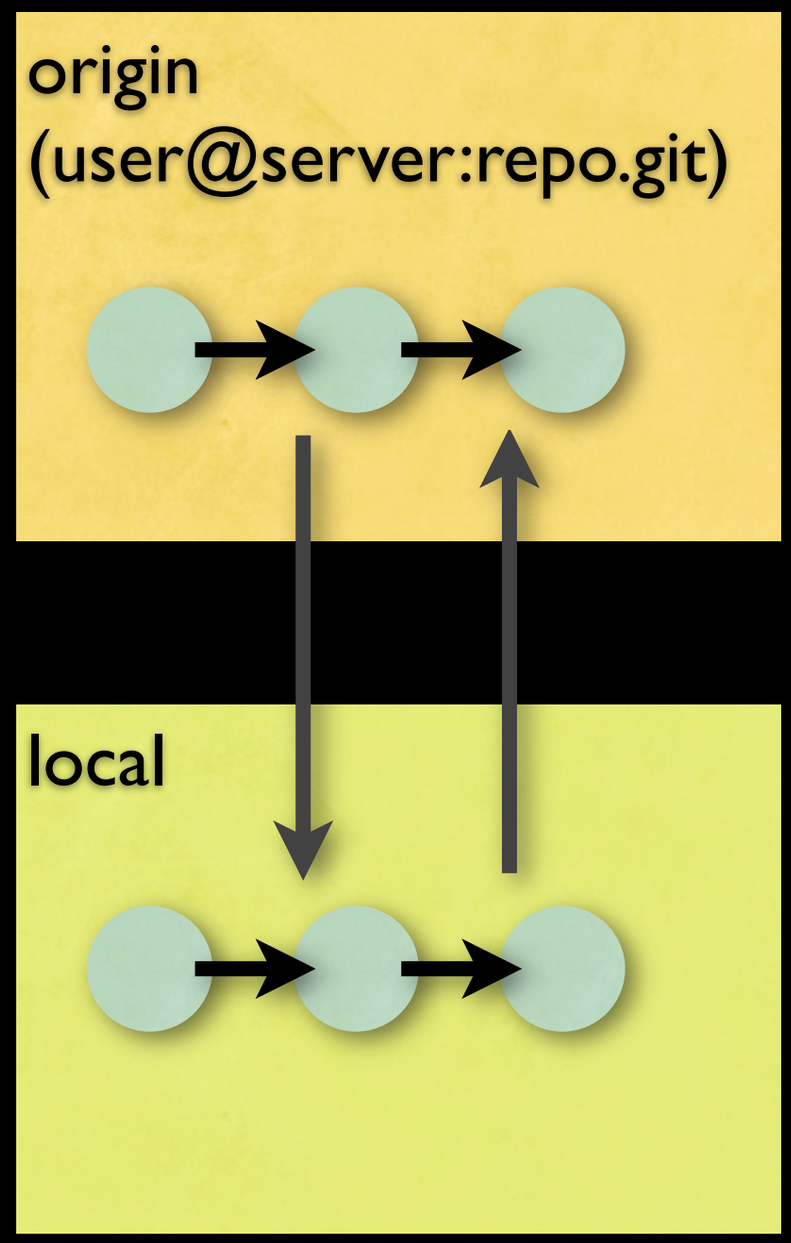


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`
  - `git push origin [src:dest]`
- Or just ship patch[sets]

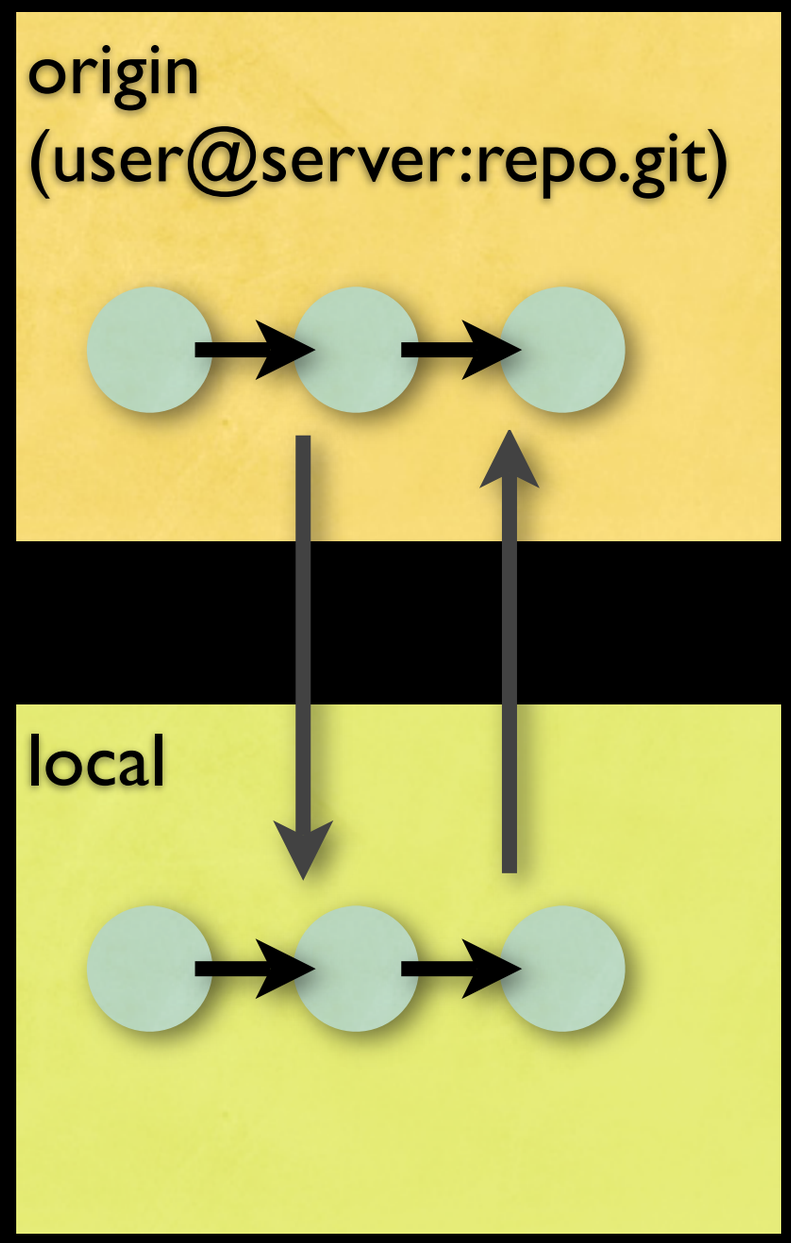


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Sharing

- Sync between repositories
  - `git pull origin [src:dest]`
  - `git push origin [src:dest]`
- Or just ship patch[sets]
  - `git-format-patch, git-am`

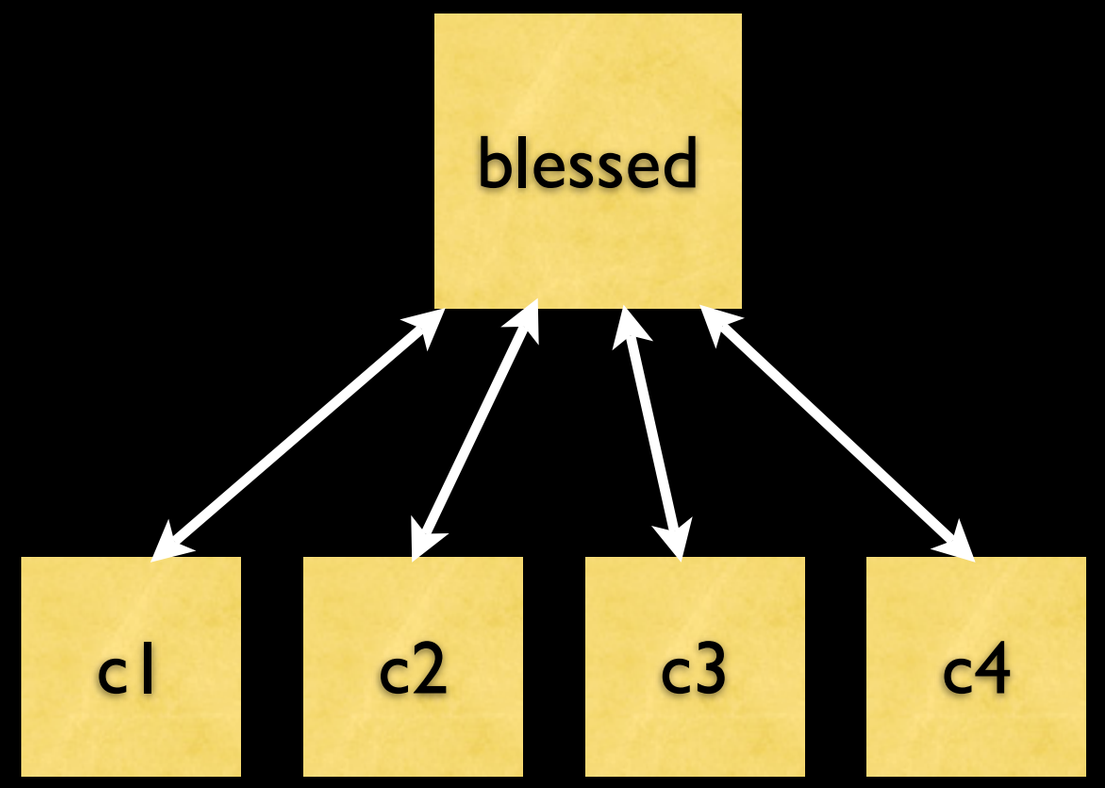


Of course we still want to be able to exchange updates or new revisions, and we do that by syncing. In git this is called pushing and pulling. In both, we specify the remote server, and optionally the source and destination branches. We'll get to branches soon. In this example we have a **remote** called origin, which is just shorthand for a server and account.

Of course, we can always follow the really old-fashioned approach and get patches from git which we can then email. Git has very nice support for this because a lot of kernel development is done this way. Ideally you shouldn't be using this, so we won't cover it.

Workflows - Centralized

- Blessed repository
- Centralized, account per committer
- Push/pull equivalent to commit/update

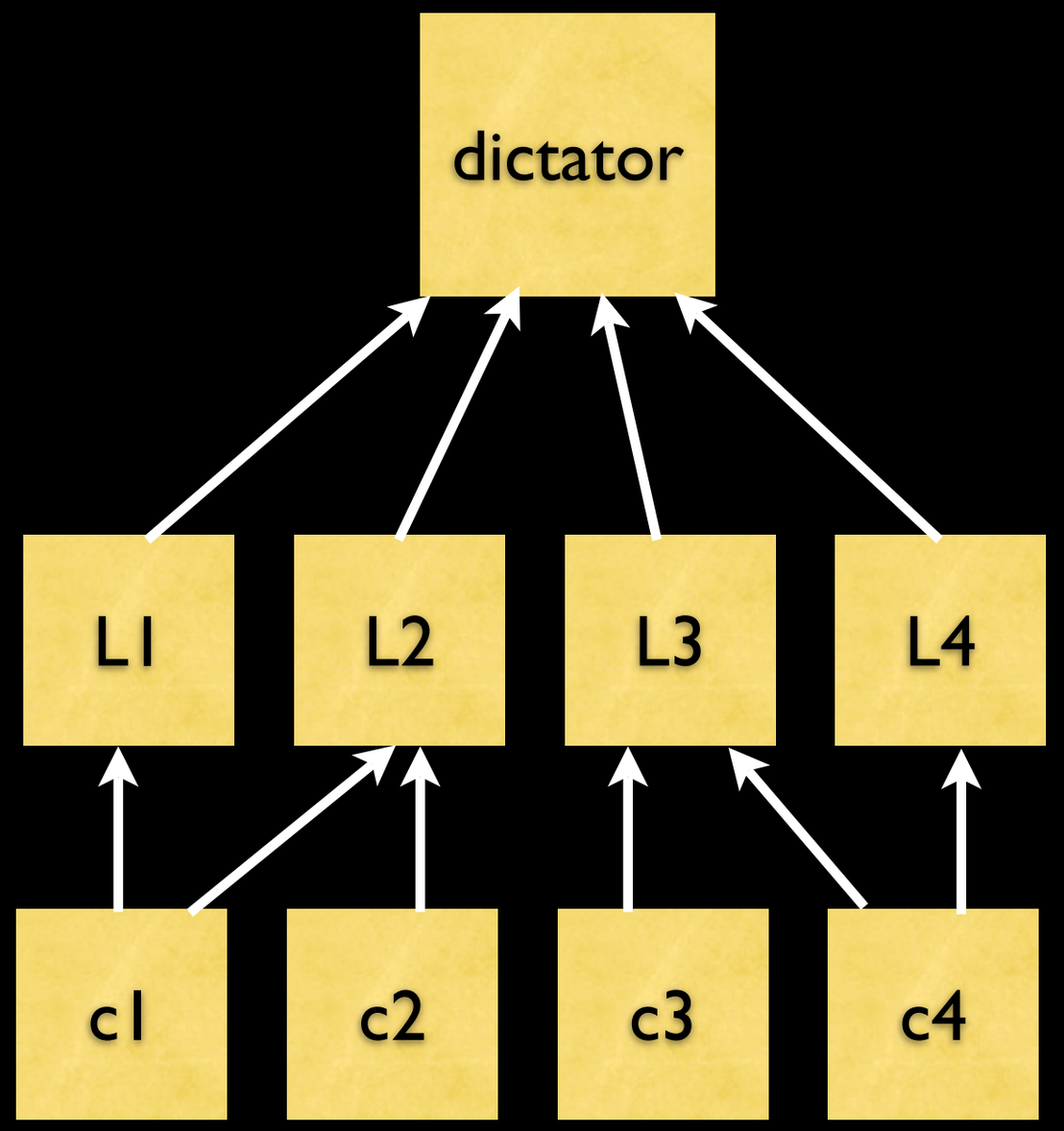


Ok, so we know how to make changes, push and pull between repositories, but then how do we **actually** work with others?

Well, let's cover the centralized style first since its one of the easier approaches, and still quite common. In this model, there is a blessed, centralized repository. All committers need to write access, i.e. accounts to the repository on its server. Each committer clones that repository and simply pulls/pushes to it. No communication occurs between committer repositories. This ends up being essentially equivalent to svn commit and update except that more than one commit can occur in a push/pull.

Workflows - Lieutenants

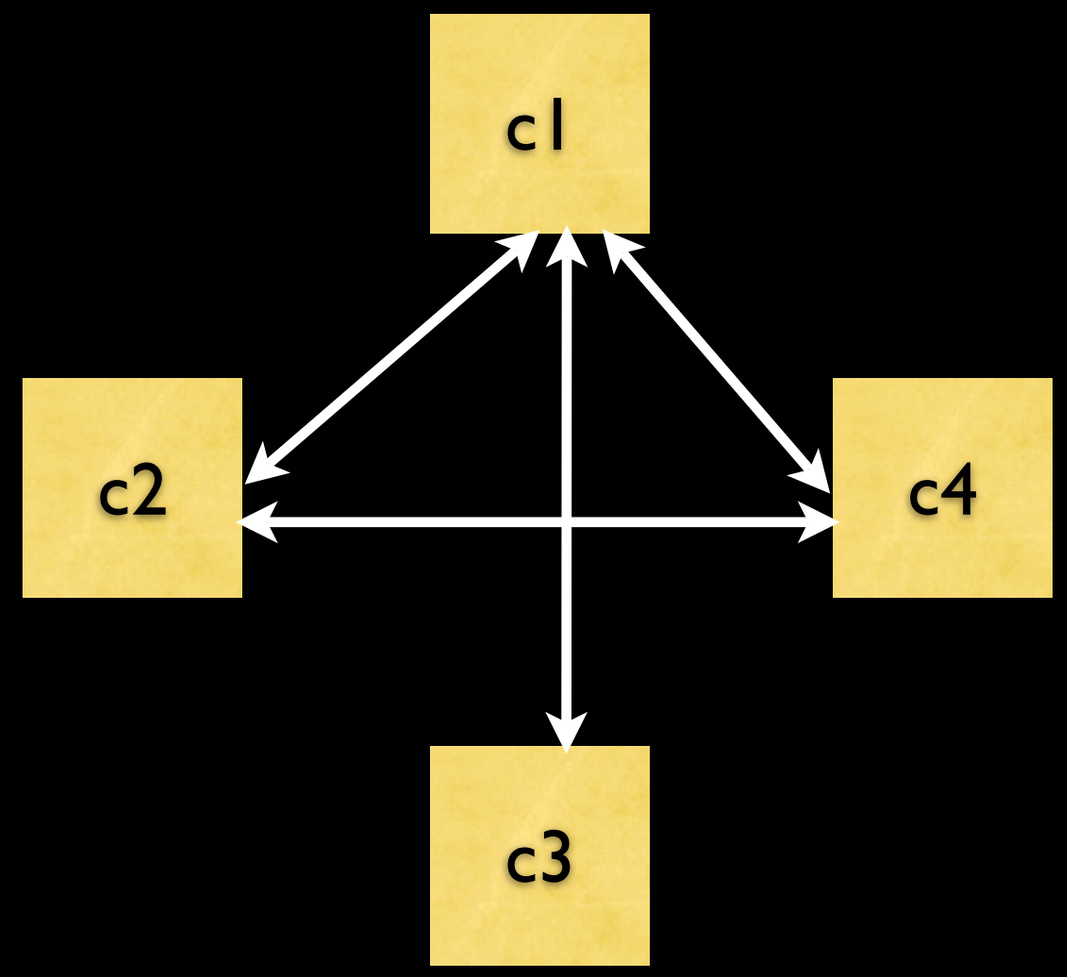
- Blessed dictator repository
- Pull up
  - Dictator pulls from lieutenants
  - Lieutenants pull from contributors



We can extend this model a bit to get a more scalable solution. We still have a blessed centralized repository which most people use. This is “the project.” We effectively have 1 committer (or maybe lieutenants can also commit directly). The dictator pulls from lieutenants. The lieutenants, however, may pull from anybody to get patches into the project. This isn’t too different except we’ve really flipped the arrows -- instead of pushing into the blessed repository, we flip things around and pull into the blessed repository. This is top-down instead of bottom-up.

Workflows - Distributed

- Repositories aren't special
  - Push/pull from anybody
- In practice, centralized repositories emerge



The distributed workflow doesn't treat any repository as special. This is the bazaar model. In practice this never happens. Somebody is a leader and their repository is de-facto special. But it is **only** de facto since everybody can get that repository and try to get people to work against theirs. Note that this model is always possible, the previous models are really just policies we apply on the distributed version control model for our own sanity.

Other nice benefits

- Fault tolerance
  - Entire history of Linux kernel is cloned in thousands of locations
- Private work, rebase to make it look pretty
- Can do most work without hitting the network
- When on network, usually faster (less chatty)

Not a panacea

- Large files can be a problem
  - Large initial clone
    - Or need to prune history for efficiency
- Private sharing can be a pain (auth)
  - Often forces centralized model
  - Or let somebody else do it (e.g. Github)

Additional Resources

- Git Homepage [http://git-scm.com/]
- Github [http://www.github.com/]
- GitReady [http://www.gitready.com/]
- Git for Computer Scientists

  [http://eagain.net/articles/git-for-computer-scientists/]