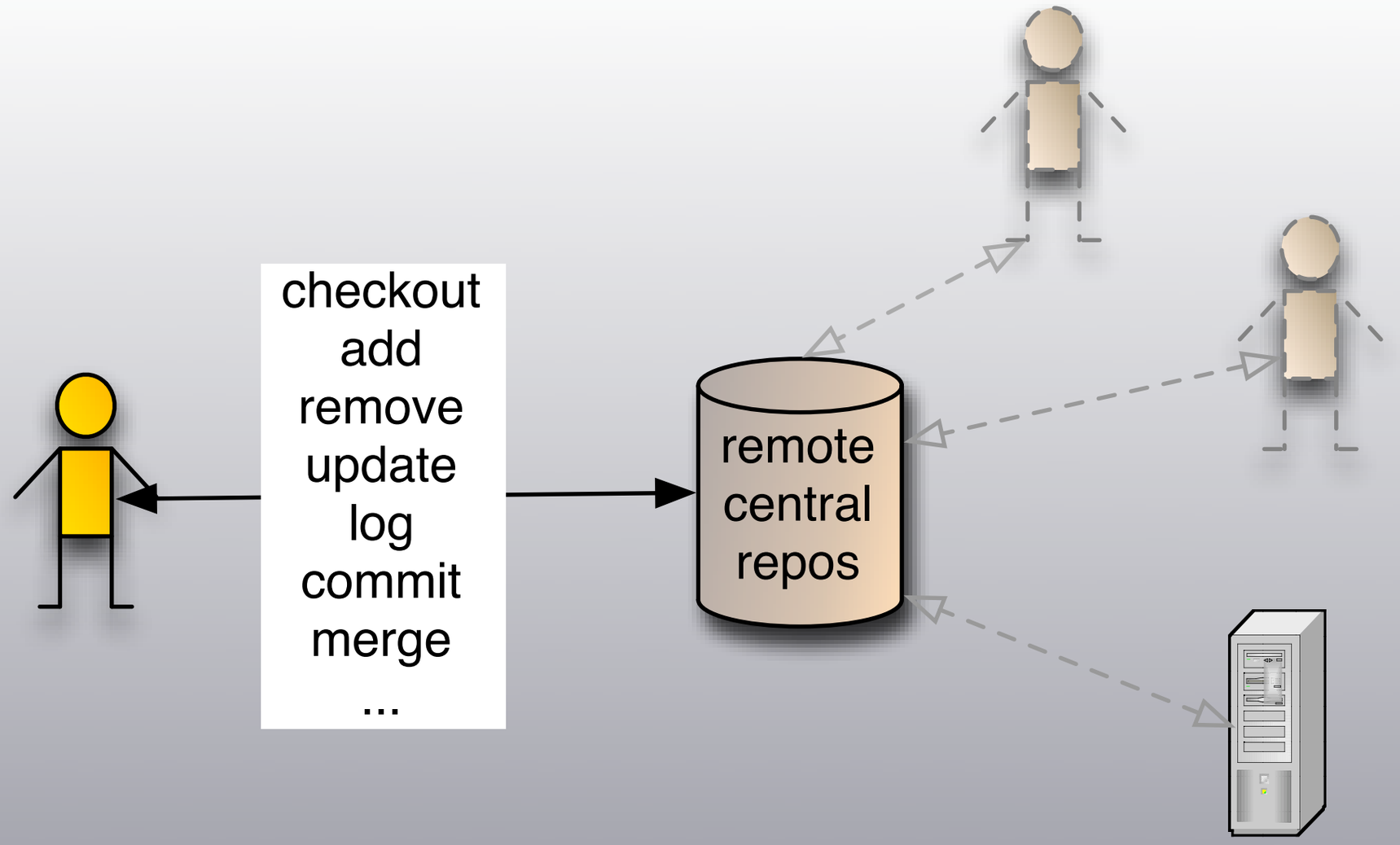
# Distributed Source Control With Mercurial

## Or, How I Learned to Stop Worrying and Love the Merge

Ted Naleid - Lead Developer at Carol.com

# Overview

- Introduction
- SVN (centralized VCS) vs DVCS
- Current Popular DVCS Alternatives
- Mercurial (hg) examples
- Hands on with hg




# Centralized VCS (SVN)

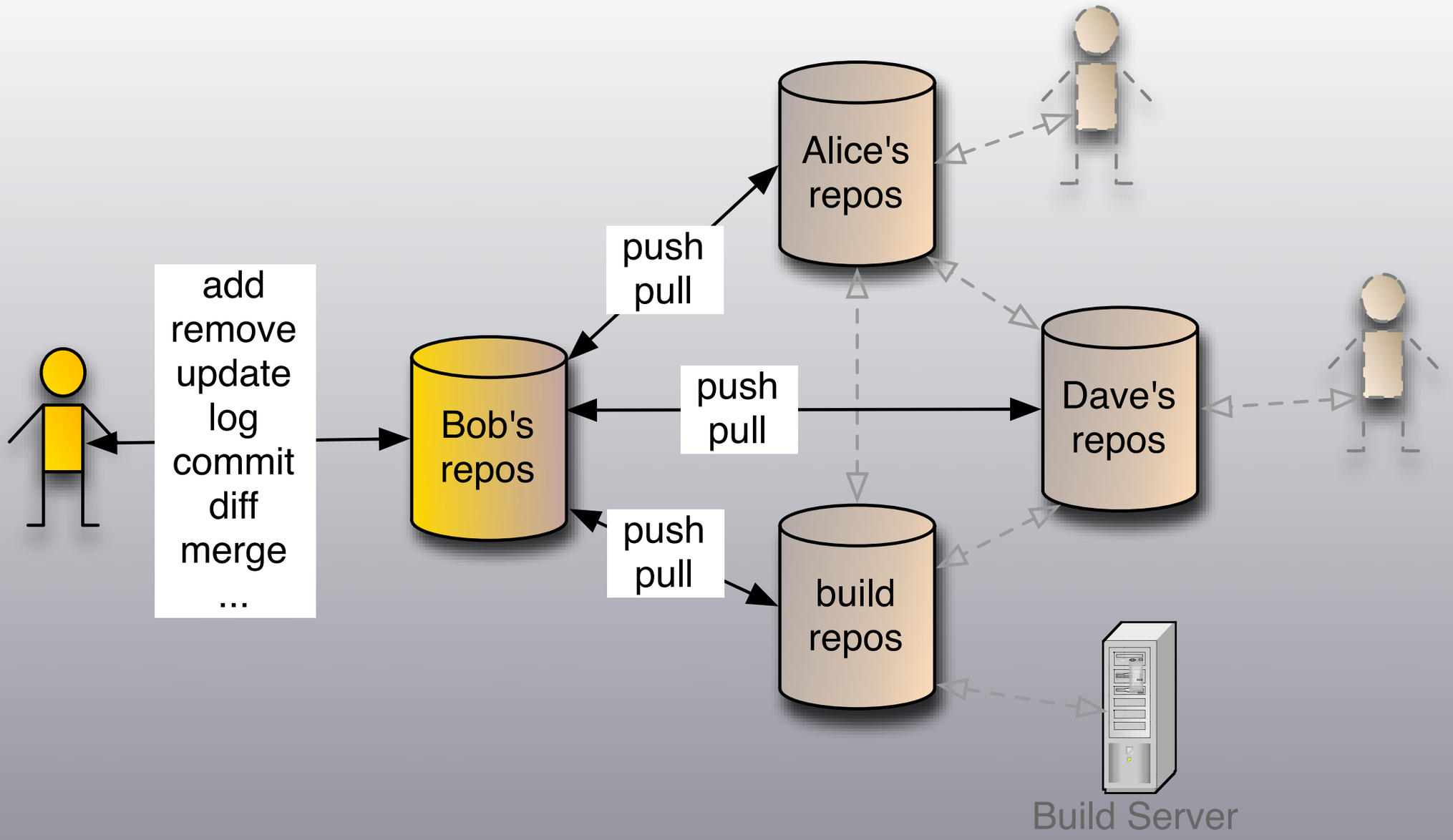
Everybody is relatively familiar with centralized version control systems.

SVN is only one of these, but others that are used quite a bit include CVS, Perforce, PVCS, and (ick!) ClearCase.

Centralized VCS systems have been around in one form or another for decades and have been relatively prevalent in business software development for about the last 15 years.

In a centralized system, there is a server that people are granted access to.  Any changes that you want to save, need to be checked in.
This server is the same server that continuous integration systems will pull from.  The only way to get a change from Bob to Alice is to check it in so that everyone gets the file (without going through patch files).

You can create branches for development, but those branches are full copies on the server that everyone has access to.  It's painful enough that I bet most people in the room probably have never created their own branch.




# Distributed VCS (Mercurial)

In a distributed version control system, everyone has a full copy of the repository, this includes full history for each of the files.

Every repository is “equal”, though some repositories could be used in special ways (ex: the build machine’s repository).

As the diagram shows, it’s also easy for smaller teams to collaborate on changes in this model before pushing code to designated repositories (like the build server’s repository). Bob can collaborate on code with Alice without having to affect either Dave or the build server.

Bob is also able to add/remove code locally (thus saving state) without affecting anyone else. He’s also able to clone his repository to try out some experimental work.

Since it’s all local, it’s just as quick and easy as copying a directory. If an experiment works, Bob can simply push that experimental code back into his main repository (or just switch over to the new one).

# SVN Limitations

# DVCS/Mercurial Strengths

# Branching is easy, but merging is painful

# Branching is easy, merging is (relatively) easy

Subversion's merge isn't history aware.
Users have to track which revisions have been merged between branches.
Like any manual process, this leads to mistakes/problems.

Merging is never completely painless, but the merge tools are much more aware of history

The tools also encourage frequent checking in as well as updating (since it's easy to revert state), so if you do run into a merge issue
it's likely that you've got a small amount of code to merge rather than a huge "big bang" update.

# Active net connection required to interact

# If your computer is on, you have access to the repository

Can’t do anything without a connection to the central repository, branch, diff, see the current checked in version, etc.

Users are unable to save state if there are any issues with the repository, or if the build is broken and they don’t want to “pile on” their changes into a broken build.

Unable to share changes with others without sharing with everyone

(including the build server)

Sharing changes with selected people is easy

(hg serve/hg push/hg pull)

# Fails to merge changes when something is renamed

# Aware of file history and can merge into renamed file

If something gets renamed, the common method is to have everyone check any changes in before you do the rename or else they manually need to move their changes over.

.svn files are littered throughout your source tree

Single .hg directory at root of source tree

This can make recursive grepping/searching easier as you don't need to exclude all of those .svn directories, just the root .hg dir.

# Slow over-the-wire performance

# Fast performance; you're working on local filesystem

As the repository continues to grow, this will become more and more of a problem.

This also enables activities and commands that aren't possible in a client/server environment.

Mercurial has a "grep" command that lets you do a text search through your repository and it will show you the last revision that a piece of text was there.

More and more tools are also appearing that are taking advantage of having all of this history info right on a local hard disk with some nice visualization tools.

# Discourages experimentation

# Cheap to create/throw away local experimental branch

Any branches in SVN are there forever and are visible to everyone (cluttering things up).

In a DVCS, it's simple to clone your repository and give something else a try.  If it works, you can merge it in otherwise, you can throw it away.

# SVN Strengths

# DVCS/Mercurial Limitations

Familiar to most developers

Most developers will need to learn to think differently

We’re all mac users here though, right?  So we already know how to “think different” :)

# Relatively easy to grasp

# Better understanding of version control concepts required

Many developers don't really even understand SVN all that well to begin with, just enough to check code in/out.

I think that understanding version control is one of the core things that a developer should be able to do.

As the old saw goes, "Those who cannot remember the past are condemned to repeat it". This is especially true when it comes to version control.

Everyone knows where the trunk is because there's only one server

Need to define and adhere to convention to know where the trunk is

Without convention, things could spiral out of control with multiple forks of the source tree.

Not really that big of a deal in practice, just part of the mind switch and something people need to think about.

# Well established tool support and integration

# Tool support and integration isn't quite as far along

Tool support is better than expected for Mercurial, integration with Eclipse and IntelliJ is complete (though I haven't done much with either).

There is also much better potential for tools to get _really_ good as the entire VCS is local and any hits to it only impact you.  You don't need to worry about expensive operations as much as you won't be slowing everyone else down.

# There is a winner among free, centralized VCS systems: Subversion

# DVCS systems are still new and a clear winner has not been established

There have been 2 generations of DVCS systems.  The first includes Arch and Monotone.  First gen is generally considered slow and never really caught on in a big way.

The second generation started with Darcs and the Bitkeeper debacle in the Linux core, and now has added Mercurial, Bazaar, and Git.  There has been a lot of buzz around these systems in the past couple of years.  These are the systems that I'll concentrate on.

# Popular 2nd Gen DVCS Systems

Git - Bazaar - Mercurial

I'm covering the 3 "big" ones in 2008.  These are all newer 2nd gen DVCS systems, there are others like Arch, Monotone, BitKeeper, and Darcs that I'm not covering.

They all have smaller user bases, and I'm truthfully not that familiar with any of them.

# Git

Created by Linus Torvalds in 2005 after the BitKeeper "debacle"


BitKeeper was used for the linux kernel, but then they ended support for the Free Use version of the software in early 2005.

There are many internet threads about this and most of the wikipedia page on BitKeeper deals with this topic.

# Git - Website & Hosting

- Website: http://git.or.cz/
- Hosting:
  - http://github.com
  - http://repo.or.cz

# Git - Use in the Wild

- Linux Kernel
- One Laptop Per Child (OLPC)
- Ruby on Rails
- Lots of other Ruby stuff
- Written mostly in C

# Git - Common Wisdom

- Fastest of the DVCS systems
- Unfriendly to non-Linux/Unix systems
- Complex, with ~150 commands added to path
- Very popular in the Linux/Ruby communities
- Probably the most “buzz” off all DVCS systems right now

# Bazaar (bzr)

Created by Canonical, Ltd. (creators of Ubuntu) in 2005

Started just slightly before the BitKeeper debacle.

# Bazaar - Website & Hosting

- Website: http://bazaar-vcs.org/
- Hosting: https://launchpad.net/

# Bazaar - Use in the Wild

- Ubuntu
- Drupal
- Fairly big in Python community
  - it's written mostly in Python

# Bazaar - Common Wisdom

- Has gone through lots of revisions/changed formats
- Slowest of the 3
- Migration of an existing SVN repository is REALLY slow
- Made to be friendly, similar to SVN
- Smallest market share

Supposedly, the speed of Bazaar has improved quite a bit and there were just some early revisions that gave it a bit of a bad name.

Not as slow as Darcs, Arch, Monotone, and some of the other less popular DVCS systems though.

Mozilla tried a migration from SVN and it took over a month of constant running for Bazaar to create the new repository.

# Mercurial (hg)

Created by Matt Mackall in 2005 after the BitKeeper "debacle"

# Mercurial - Website & Hosting

- Website: http://www.selenic.com/mercurial/
- Hosting:
  - http://freehg.org/
  - http://sharesource.org/

# Mercurial - Use in the Wild

- OpenJDK (Java)
- OpenSolaris
- Mozilla
- NetBeans
- Many others (largely Java/Python related)
  - Like Bazaar, it's mostly written in Python

# Mercurial - Common Wisdom

- Similar syntax to SVN
- Slightly slower than Git, but faster than Bazaar
- Good cross-platform support
- Getting good support from large Java projects (OpenJDK, NetBeans, etc)
- Lower maintenance and easier learning curve than Git

# Why did I choose Mercurial over Git/Bazaar?

# Similar commands to SVN

- hg add
- hg remove
- hg update
- hg log
- hg status
- It's like SVN but with the ability to "push" and "pull"

# Better cross platform support and growing tool integration

Plugins for IntelliJ and Eclipse.

Also TortiseHg for windows (similar to TortiseSvn)

Also has a plugin architecture that allows for extensions and other diff/merge programs to be used.

# No “packing” of the repository is necessary

Git has a reputation for the size of the repository needing regular manual maintenance through “packing” and this has bit a number of new developers.

The less manual things and maintenance a dev needs to perform, the better.

# Local revision numbers are “friendly”

In git, all you have to refer to are SHA-1 hashes.

Mercurial revision numbers aren’t perfect though, as it’s simply a local key to a real hash underneath the covers.  Just like a database ID key, if you look at another database, revision 14 is probably a different patch.

# It's used by a number of big Java projects

OpenJDK, OpenSolaris, NetBeans

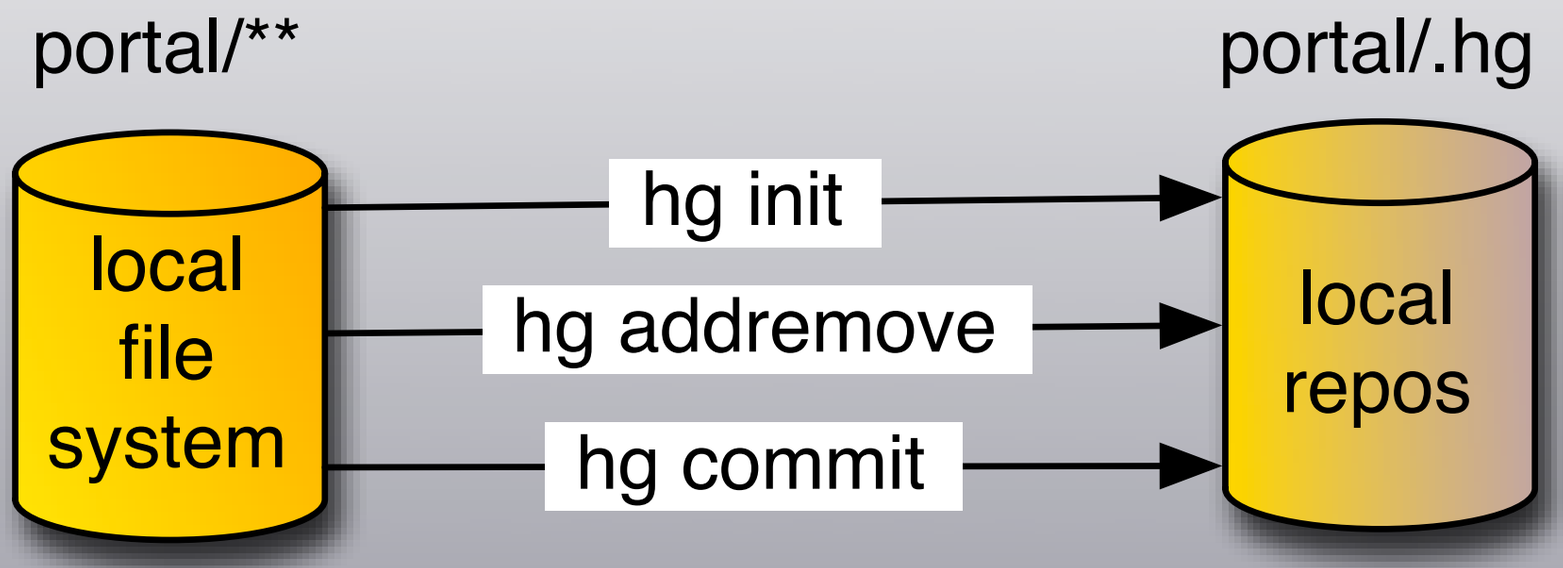
It's the first one I tried :)

# All 3 of these tools look fantastic.

If any one of the 3 was the only thing available, I'd be happily using it and likely wouldn't be complaining.

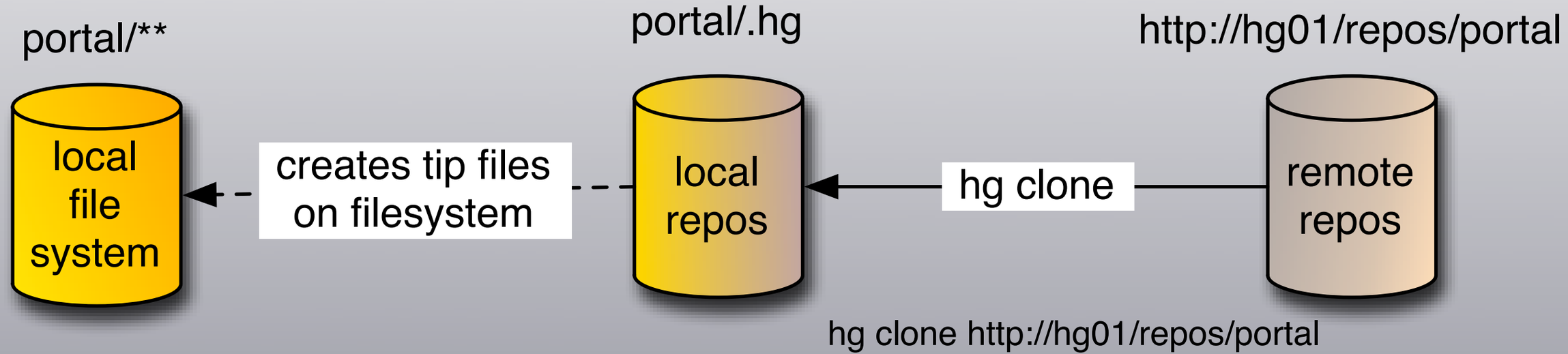
# Mercurial Usage Examples

How do I do X?



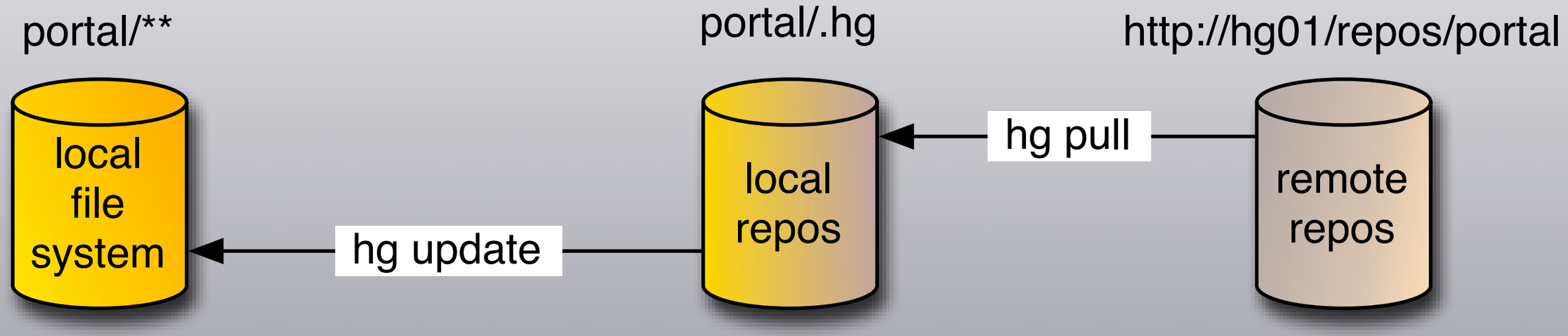

# Create a new repository

inside any directory (example above “portal”), you can execute the commands shown to create a local repository that you can check stuff into or that anyone else could potentially clone or contribute to.




# "Checkout" an Existing Repository

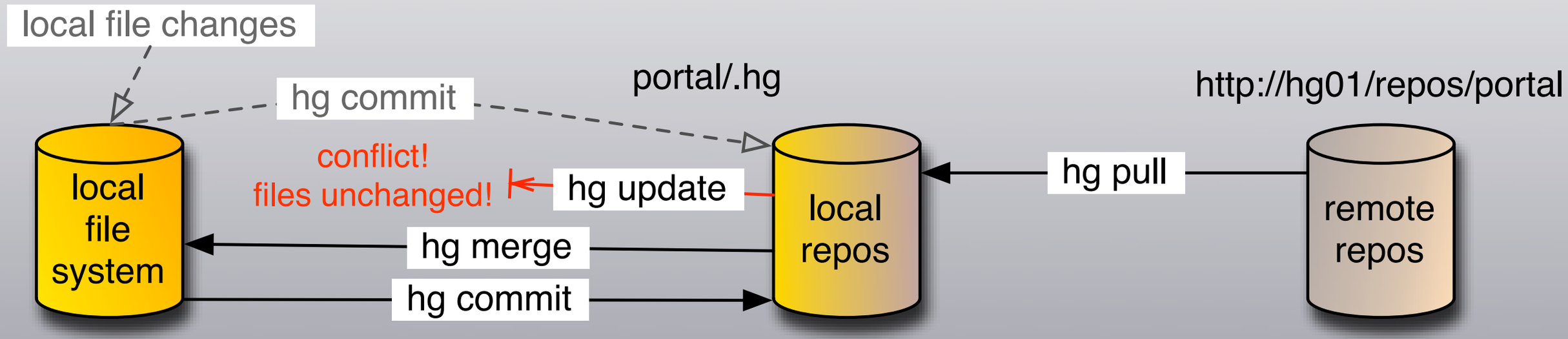
The copy that you create has all of the history in the source repository.




"hg pull -u" will do this in one command

# Pull down the latest changes

(no conflicts with local changes)

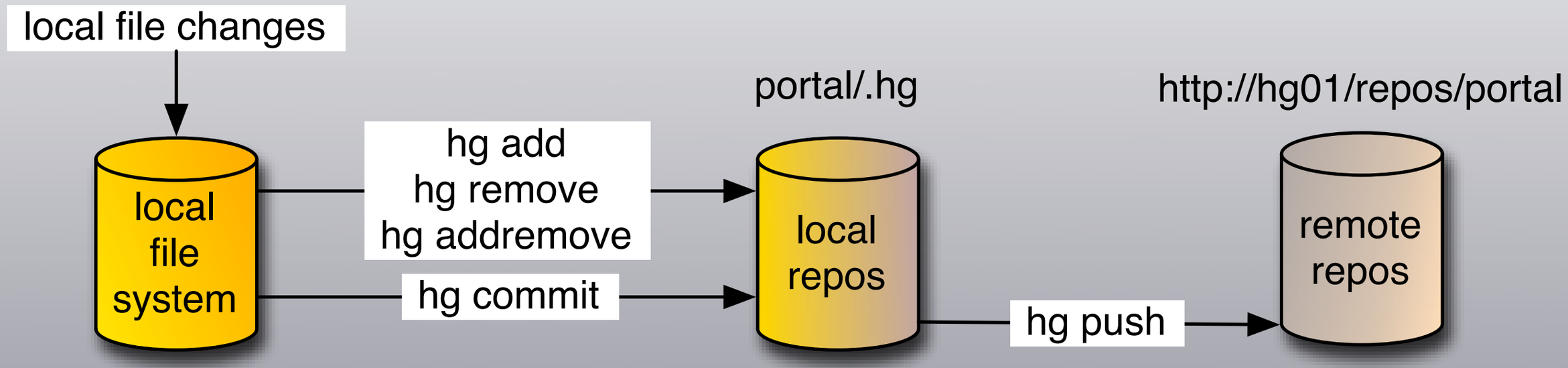



"hg fetch" will pull->update->merge->commit in one command

# Pull down the latest changes

(conflicts detected with local changes)

“hg fetch” does all of that in one command, this just shows the details




# Push changes to another repository

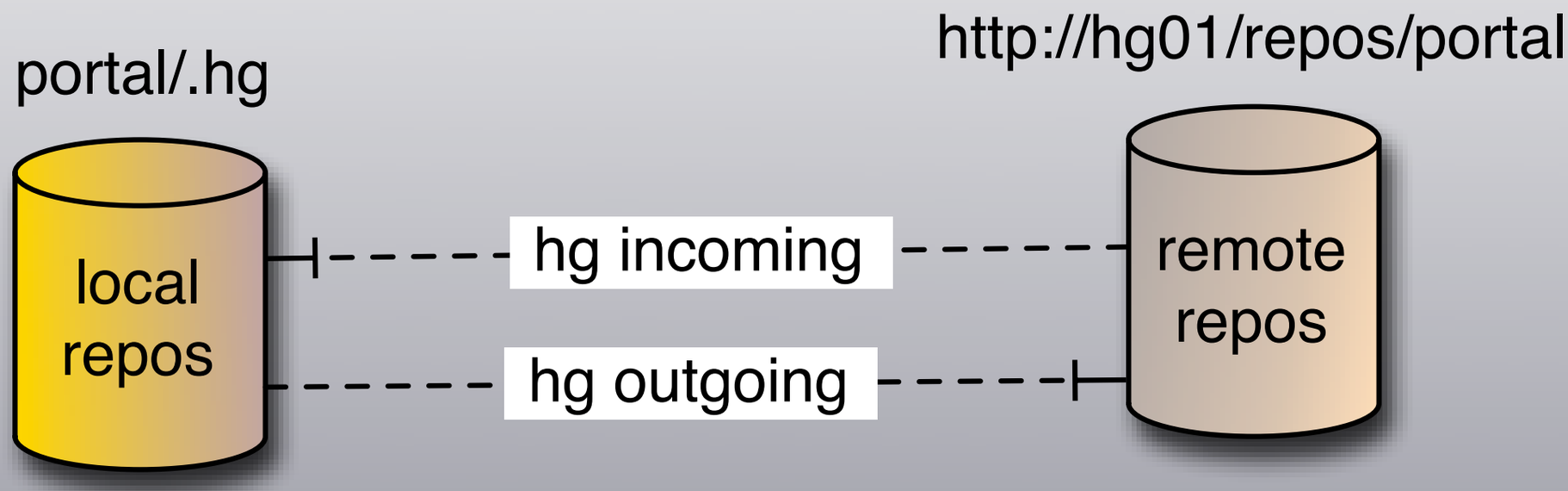
(by default, push will refuse to run if it would require a merge)

If a push would increase the number of “heads”, this indicates that the one trying to do the push has forgotten to pull and merge changes before pushing.

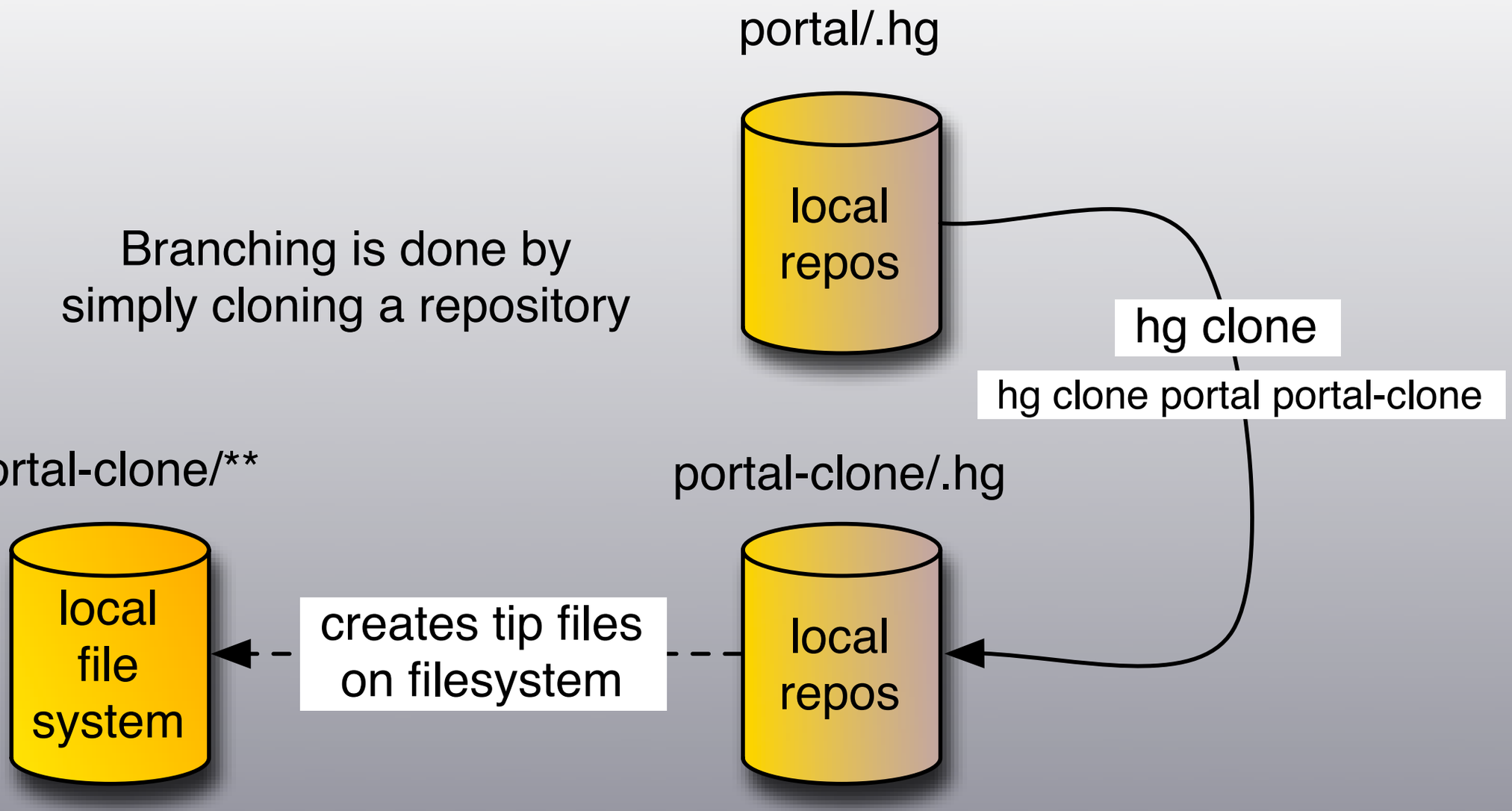
(Multiple heads means that there would be a conflict and 2 versions of an existing file at the same time, you can do an “hg heads” to see the current list of heads)

This push will NOT update the files on the remote filesystem (which is a good thing). The remote repository would get the changes on the filesystem at the next update.

This makes it possible for things like a build server not getting it’s files switched out from underneath it during a build if someone pushes something new out.



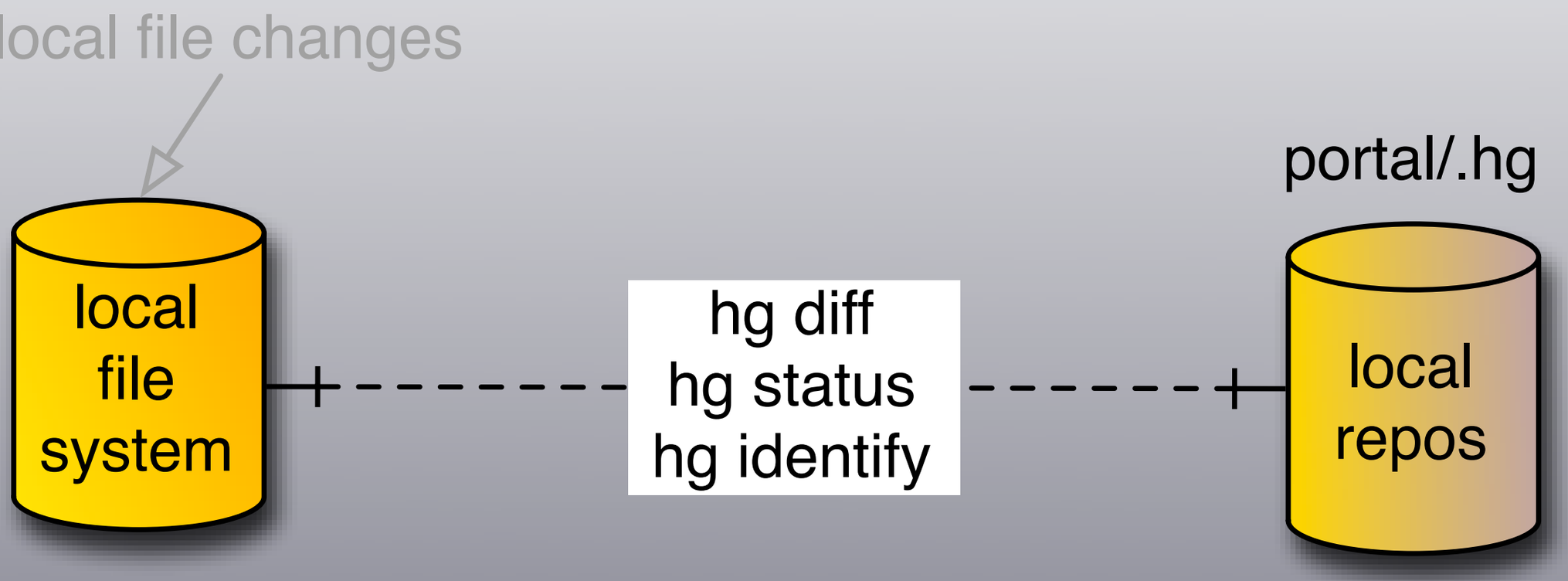

# Do a push/pull dry run




# Create a new “branch”

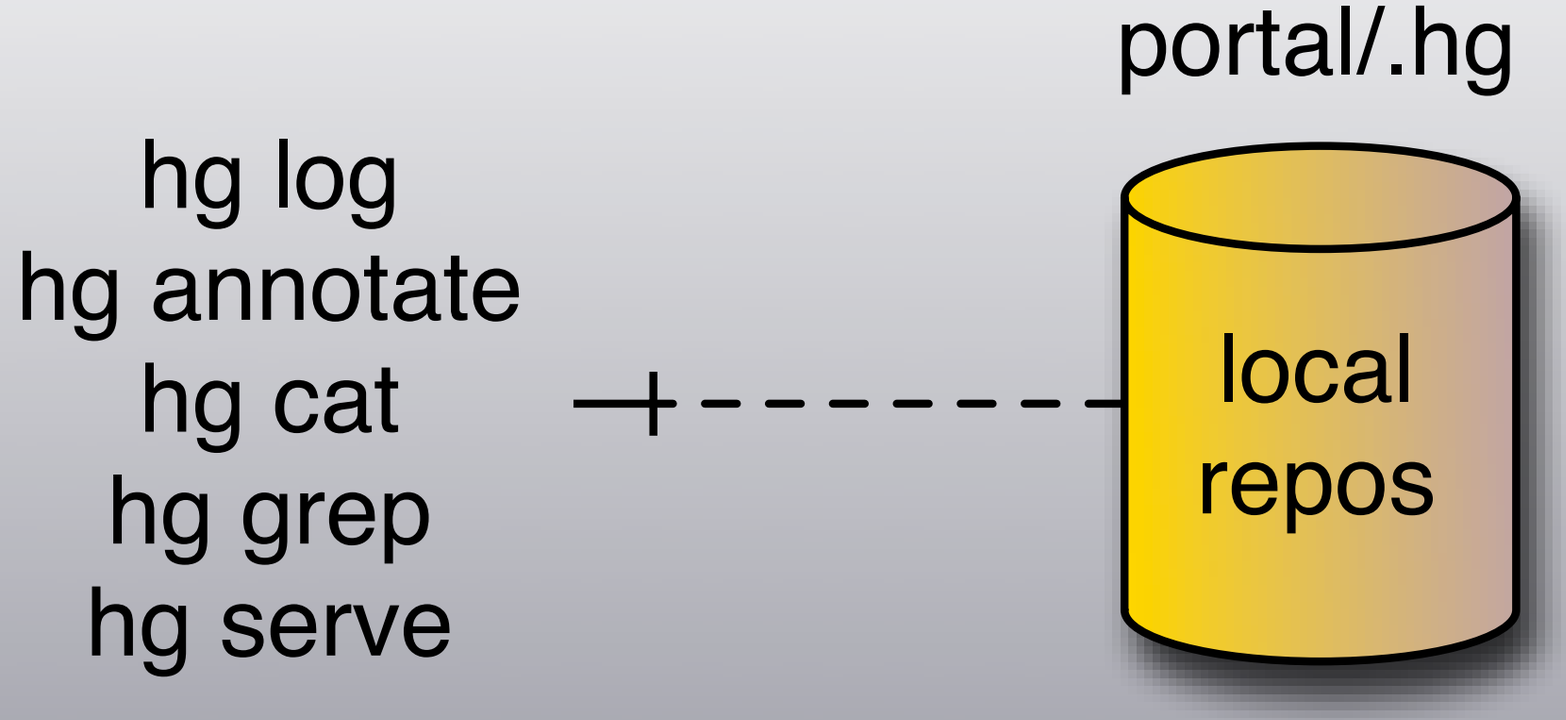
(experimenting is cheap and easy)

There is also an “hg branch” command to create what most people think of as a branch, but this isn’t the recommended way of working for “big releases”. It’s easier and more obvious to track clones of the repository, and you can also push/pull changes between local repositories if you need to, so you aren’t losing anything.




# Compare file system with repository

hg status and hg diff work very much like SVN
hg identify will show you the version of your working copy, you can compare this with “hg tip” to see if it’s the same number and if you have the latest stuff in your repo in your working copy on disk.
“hg identify -n” is useful as it shows you the local revision number of the working copy on the file system




# Query repository for info

There are many others if you dig in, but these are some of the basics.

“hg serve” starts up a web server that you can access at http://localhost:8000 by default

# Hands on with Hg

Go through slides of what we'll do and then do live-coding session in the terminal to show details of each thing.

# Creating new repository

create new grails project (using grails 1.0.3):
mkdir -p ~/temp && cd ~/temp
grails create-app HgTest
cd HgTest
grails create-domain-object Book
grails create-domain-object Author
mate .
// Edit files to this:

Author.groovy:

```
class Author {
        String name
        static hasMany = [books:Book]
        String toString() { name }
}
```

Book.groovy:

```
class Book {
        String title
        static belongsTo = [author: Author]
        String toString() { title }
}
```

BootStrap.groovy (inside def init):

```
def author = new Author(name:"Ray Bradbury")
author.addToBooks(new Book(title: "Fahrenheit 451"))
author.save()
```

grails generate-all Book
grails generate-all Author
grails run-app
go to http://localhost:8080/HgTest/ and try out application
Now create repository with new code:
hg init
hg addremove
hg status
hg commit -m “initial checkin of HgTest”
hg serve // show files at http://localhost:8000

# Cloning repository/ branching

The easiest way to create a branch is done simply through cloning the repository.  This is the recommended way to do “big picture” branches (such as new revisions).

Open up new terminal window.
cd ~/temp
hg clone HgTest HgClone
cd HgClone
grails run-app  // show that it works just like the old one does, Ray Bradbury is there
mate . ../HgTest
// edit BootStrap.groovy so that the book is Celsius 232.778
hg diff
hg commit -m “I’m feeling european today”
hg log

There is more normal “branch” support within a single repository, the HgBook refers to this as something more for “power users” and I agree that it makes things less clear.  Cloning repositories is easy, and it’s very clear based on the directory that you’re in which repo you’re working on.

# Pushing changes to another repository

// show hg log of HgTest, only one commit there
hg push
cd ../HgTest
hg tip  // see commit message
// view file on disk, it hasn't changed yet
hg update
// now it has

This is useful as you can push changes to a repository without messing up what it's currently got in progress.
Once they do an update they'll get the latest changes on the local filesystem (similar to checking into an SVN repo now).

# Merging conflicting changes

```
// in HgTest
change author to “Theodore Sturgeon”
change book to “Microcosmic God”
hg commit -m “changing to short story”
hg log
show the log with #2 revision

cd ../HgClone
change author to “Daniel M. Pinkwater”
change book to “Lizard Music”
hg commit -m “dmp”

hg pull
hg heads
hg update // conflict!
hg merge
hg commit -m “keeping microcosmic god”
hg log  // 2 parents to file
```

# Creating a tag

hg tag foo
// tags are not copies like they are in subversion, they are simply revision aliases
hg log -r foo

hg tag -r 1 european
hg log -r european
hg tip

Tags are simply stored in a file in the root of the project called .hgtags
cat .hgtags

These tags will be pushed/pulled to any repository that you interact with.  If you just want a marker for your own use, you can create a “local” tag:
hg tag -l my_local_tag

# Searching through history

hg grep Bradbury
hg grep Pinkwater

# Viewing repository through a web browser

hg serve
// show 2 parents to merged file

# A quick way to stick your toes in and try Mercurial out: use it as a "Super Client" for SVN

It's nice to be able to very easily version filesets.  Even if a company doesn't switch to Mercurial, you can still use it locally to check stuff in before grabbing new things from the server that you think could break you.

Then it's easy to get back to your original state.  I used this during a big grails upgrade.

Go into any directory that you've checked out of SVN.

hg init
hg addremove
hg commit -m "initial checkin"
hg serve  // go to http://localhost:8000 to see the files you've checked in

now you can svn update without worry about getting back to your old state, you can also check in before trying something crazy out.

You now have a mercurial repository that you can check in to whenever you want (such as just before doing an update).

You can share it with others using "hg serve" to let them "hg clone" your repository.

# Web Resources

- Choosing a distributed version control system

  http://www.dribin.org/dave/blog/archives/2007/12/28/dvcs/

- Understanding Mercurial

  http://www.selenic.com/mercurial/wiki/index.cgi/UnderstandingMercurial

- Version Control and the “80%” (DVCS counterpoint)

  http://blog.red-bean.com/sussman/?p=79

- Mercurial Book

  http://hgbook.red-bean.com/hgbook.html

- Video of Bryan O’Sullivan (creator of the Mercurial Book)

  http://video.google.com/videoplay?docid=-7724296011317502612

# Questions?