Revision Control

THE LINUX FOUNDATION

Keeping Track of Changes

- As you work on a project changes accumulate; as you fix bugs, add new features and optimize, your code base will diverge dramatically from where you began
- Sometimes you add a feature or fix a bug and think everything is working perfectly, but then such wistful thinking meets reality when a user (perhaps even yourself) employs your application in a way you did not foresee
- Or you hit what was thought to be an impossibly rare race condition when you port to a new platform with faster hardware and multiple CPUs
- At such a point you may have progressed considerably past the point where the problem was introduced, and you will have to locate the point in your development where things went wrong, fix the problem, and bring all the other changes forward to the current state

Manual Revision Control Methods

- Every developer has been sloppy about this at some point, particularly when they are the only one working on the project and code base
- While there is no substitute for a clean and intelligent design that has a lot of modular components, a design which is easy to break down and reassemble once pieces have been fixed, there are tools which can be used to greatly facilitate efficient cycles of development
- The simplest method is to simply keep regular backup snapshots of the work, and revert to them as problems arise - we have all done this
- But it is a manual process, and as the size of the project grows it becomes more and more unwieldy

Tightening and Tracking Control Better

- You will wind up differencing your current code base with earlier ones; why not just store the history of changes, and the differences instead of complete snapshots? (if nothing else, you will save disk space)
- More importantly you can track the changes directly instead of inferring them from the endpoint and starting point
- As other developers are added to the project the manual method also becomes difficult to manage:
  - One developer has to remain the master who handles all actual changes
  - Other developers cannot commit changes directly unless you want to confuse the result
  - Difficult to manage simultaneous contributions

Distributed Development

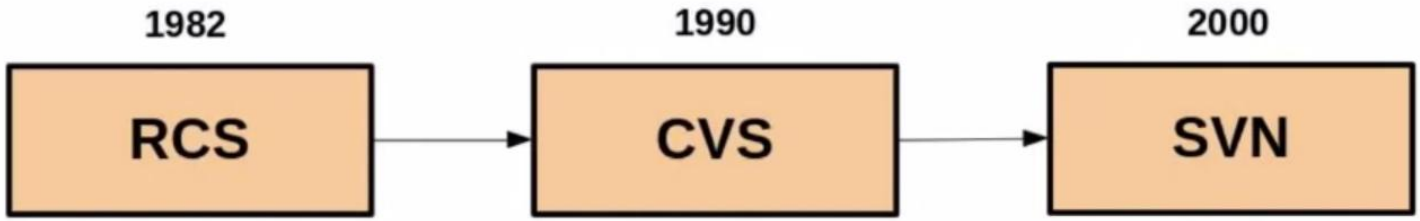
- Massively distributed projects (such as the Linux kernel) can involve thousands of contributors, and even the notion of a central master repository of all wisdom can become a hindrance
- In order to organize updates and facilitate cooperation, many different schemes are available for source control
- Standard features of such programs include the ability to keep an accurate history, or log of changes, be able to back up to earlier releases, coordinate possibly conflicting updates from more than one developer, etc.

Revision Control Systems

Converting Between Different Systems

- It is possible to bring projects which use other revision control systems into the git universe
- While it is always possible to simply taking the working copy of the project files, ignore the revision control information, and create a new repository, this throws away the previous history of changes, and the possibility of reverting to an earlier state
- There are tools for importing projects into git; indeed we will do exercises showing how to do this for CVS and Subversion
- It is also possible to go in the opposite direction, to export a git project to another system
- You might do this because you want to switch your revision system to one of the other choices; more likely you would do this because you need to mirror your git repository so those who have access only to one of the older systems can synchronize with it

Major Revision Control System History



Revision Control System (RCS)

- An old and widely used source code system is provided by **RCS**
- It is easy to convert projects using the earlier SCCS system to using RCS
- **make** has built-in rules for RCS which makes it easy to use in development projects
- **emacs** can be used tightly with RCS using the vc mode and related functions, e.g. vc-ediff
- Conventionally a sub-directory named RCS (or rcs) is situated under the directory containing the files under control; this is the repository
- RCS forces developers to work in a serial fashion:
  - When a file is unlocked no one can commit changes to a file, but the lock is available to anyone with access to the RCS repository
  - When a file is locked, only the holder of the lock can commit changes to the file
- As we shall see, more modern systems permit better cooperative development

Concurrent Version System (CVS)

- CVS is more powerful than RCS and does a more thorough job of letting multiple developers work on the same source concurrently
- However, it has a steeper learning curve, as it has many more commands and possible environment customizations
- All files are stored in a **centralized repository**
- One never works directly with the files in the repository; instead you get your own copies into your working directory, and when you are finished with changes you check (or commit) them back into the repository
- Repositories can be on the local machine or on a remote platform anywhere in the world

Concurrent Version System (Cont.)

- The repository location may be specified either by setting the **CVSROOT** environment variable, or with the **-d** option; i.e. the two following lines are equivalent:

```
$ cvs -d /home/coop/mycvs init
$ export CVSROOT=/home/coop/mycvs ; cvs init
```

- A number of users can make changes to a given module at once
- There are a number of commands to help with resolving differences and merging work
- As for RCS, emacs can be used tightly with CVS using the vc mode and related functions

Subversion

- Subversion is an open source project designed to be the successor to CVS, and which has gradually replaced it in many projects
- In order to keep the transition smooth, the Subversion interface resembled that of its predecessor, and made it easy to migrate CVS repositories to Subversion
- Directories, copies and renames are versioned, not just files and their contents; metadata associated with files (such as permissions) can also be versioned
- Revision numbering is in a per-commit basis, not per-file, and atomicity of commits is complete; unless the entire commit is completed no part of it goes through

Subversion (Cont.)

- A standalone Subversion server can be set up using a custom protocol, which runs as an inetd service or as a daemon, offering authentication and authorization; one can also use Apache to set up a http-based server
- There are many enhancements that improve efficiency and reduce storage size requirements; in particular, costs are proportional to the size of the change set, not that of the data set
- Binary files can be handled, and there are built in tools for the mirroring of repositories

git

- The Linux kernel development system has special needs in that it is widely distributed throughout the world, with hundreds (even thousands) of developers; it is all done very publicly, under the GPL
- For a long time there was no real source revision control system
- Then major kernel developers went over to the use of **BitKeeper**, which while a commercial project granted a restricted use license for Linux kernel development
- However, in a very public dispute over licensing restrictions in the spring of 2005, the free use of BitKeeper became unavailable for Linux kernel development
- The response was the development of **git**, whose original author was **Linus Torvalds**

git (Cont.)

- Technically, git is not a source control management system in the usual sense, and the basic units it works with are not files
- It has two important data structures: an **object database** and a **directory cache**
- The object database contains objects of three varieties:
  - **Blobs**: chunks of binary data containing file contents
  - **Trees**: sets of blobs including file names and attributes, giving the directory structure
  - **Commits**: changesets describing tree snapshots
- The directory cache captures the state of the directory tree
- By moving away from a file by file based system, one is better able to handle changesets which involve many files

git and Distributed Development

- Many projects have developers working separately, united by a common version control system; this is possible even when there is a central authoritative repository, such as with CVS
- What makes git different is that on a technical level, there is no such thing as central authoritative repository; the underlying framework is actually peer-to-peer in nature
- The importance and central role played by one particular location is political and sociological, not technical
- Distributed development is not defined by various parts of a project being worked on separately, with a partition between different host repositories
- In git, distributed development means every repository is authoritative, and contains not just one part of the project, but the entire code base

git and Distributed Development (Cont.)

- git has no politics, and has no preferred model for organization
- The hierarchy of the development community can be very centralized and top down, or it can be very flat
- The exchange of information and changes can be pyramidal, with a number of development trees coalescing into a top level one, or it can be very egalitarian
- git is a tool, not a rigid method, and it can be used in any way that fits the needs and philosophy of a project

Concepts and Design Features

THE LINUX FOUNDATION

Concepts

- Many basic git commands resemble those of other version control systems
- Operations like adding files, committing changes, differencing with earlier versions, and logging the history are common to any serious source control system
- However, underneath the hood git is quite different than many of its antecedents
- For example, in git a file is not an essential object
- Common operations which may be somewhat cumbersome in other systems, such as renaming a file, are remarkably simple when using git
- The long hexadecimal numbers associated with commits are also somewhat strange; they are more than identifiers and they incorporate checksums that are computed from the contents of the repositories and the changes that are made

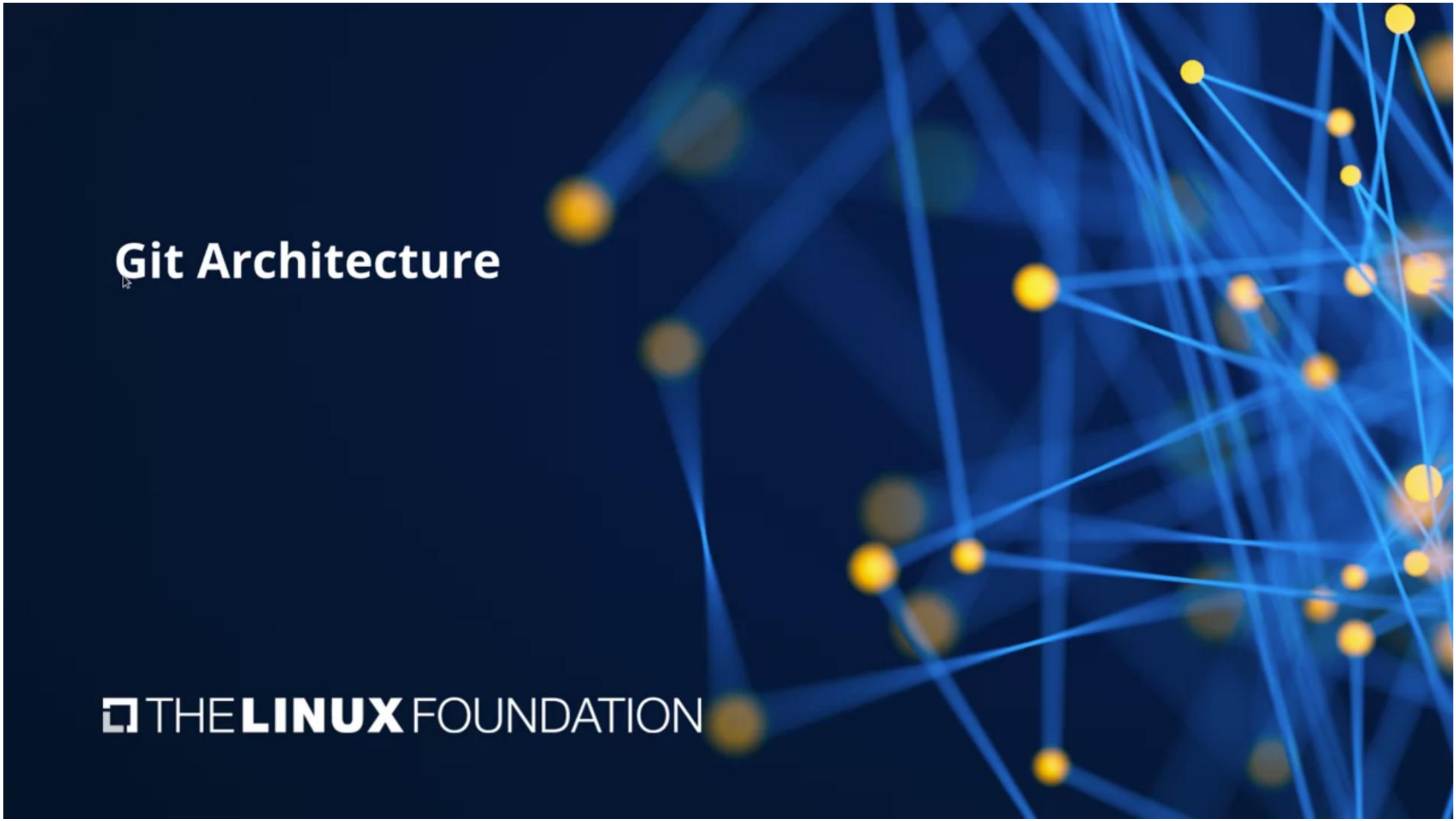
Design Features

- git grew out of the Linux kernel development community and was designed to meet its particular needs
- As time goes on, it has been adopted by additional projects, but its basic structure and motivation shows its roots
- Here we list the design features, many of which could not be found in pre-existing version control systems, which inspired the development of git:
  - **Facilitate distributed development**: developers had to be able to work in parallel without constant re-synchronization and without needing to constantly contact a central repository
  - **Scale to handle large numbers of developers**: the Linux kernel community has literally thousands of developers; this had to be handled both efficiently and reliably
  - **Attain high speed and maximal efficiency**: it is important to avoid copying unnecessary information, use compression, not bottle up networks, and be able to handle varying network latencies

Design Features (Cont.)

- **Maintain strong integrity and trust**: security demands that no unauthorized alterations can be inserted, and that repositories are authentic, not impostors; cryptographic hash functions are used for this purpose
- **Keep everyone accountable**: all changes must be documented and ascribed to whomever did them; there is always a trail left behind that can be displayed
- **Keep immutable data in the repository**: information, such as the history of the project, can not be changed
- **Make atomic transactions**: when changes are made they all must go through, or none do; this avoids leaving the repository in an uncertain or corrupted state
- **Support branching and merging**: git supports parallel branches of development, and has very robust methods for merging branches when it is time
- **Make each repository independent**: each repository has all the history in it; there is never a need to consult a central repository
- **Operate under a free unencumbered license**: git is covered by the GPL version 2

Git Architecture



THE LINUX FOUNDATION

Repositories

- The **repository** is a database
- It contains every bit of information required to store a project, manage revisions, and display its history
- The git repository contains not just the complete working copy of all the files that comprise the project's contents; it also contains a copy of the repository itself
- Each repository also contains a set of configuration parameters, such as the author's name and email address

Repositories (Cont.)

- If you clone a git repository (make another copy of it for someone else, or even yourself) this configuration information is not carried forth; such information is instead maintained on a per-site, per-user, per-repository basis
- There are two important data structures that are maintained within a repository:
  - The **object store** contains the guts of the project, a set of discrete binary objects
  - The **index** is a binary file that changes dynamically as the project morphs; it contains a picture of the overall project structure at any given time

Objects

- Git places four types of objects in its **object store**
- All together, they contain everything known about the project and how to construct both its current state and its previous history:
  - **Blobs** (**B**inary **L**arge **Ob**jects): this is an opaque construct that contains a version of a file's content; it does not contain the file's name or any other metadata, just the content
  - **Trees**: these record blob identifiers, pathnames, file metadata, etc., for files in a directory; they can also refer to subdirectories and objects within them
  - **Commits**: every time a change is made to the repository a commit object is created containing the metadata that fully describes the change; each commit points to a tree object that gives a complete snapshot of the project; commits have parents and children; except for the initial (or root) commit which is parent-less
  - **Tags**: these assign human friendly names to those horrendously long hexadecimal numbers used internally by git

Content vs. Pathnames

- Unlike most other version control systems, git tracks content, not files
- It does not store information based on the file or directory names, or directory structure and file layout
- This is associated information, not directly stored in the blobs
- For instance, suppose you have two files in two directories with different names
- git stores only one binary blob associated with the content
- The content is associated with a unique hexadecimal string
- If either file is changed, its identifier changes, and now git associates a new blob with the new content
- Comparisons are very fast because git need only compare the 160-bit identifiers, rather than actually compare the blobs, much less the actual files

Content vs. Pathnames (Cont.)

- As files change new, uniquely identified, binary blobs are created for each version
- Other version control systems often keep a copy of a file at some point in time together with revision histories and have to play back (or forward) to reconstruct a given version of the file
- git can do all this more quickly because it always has knowledge of the complete content of any version of a file; in other words, in git's strange way of doing things, a file's history can be computed by tracking changes in content, or blobs, not files
- Filenames are treated as metadata distinct from file contents, and stored in tree objects and indexes
- The contents of the `.git` subdirectory tree look nothing at all like the working directories they represent
- This is totally different than the way older revision systems like RCS or CVS are set up

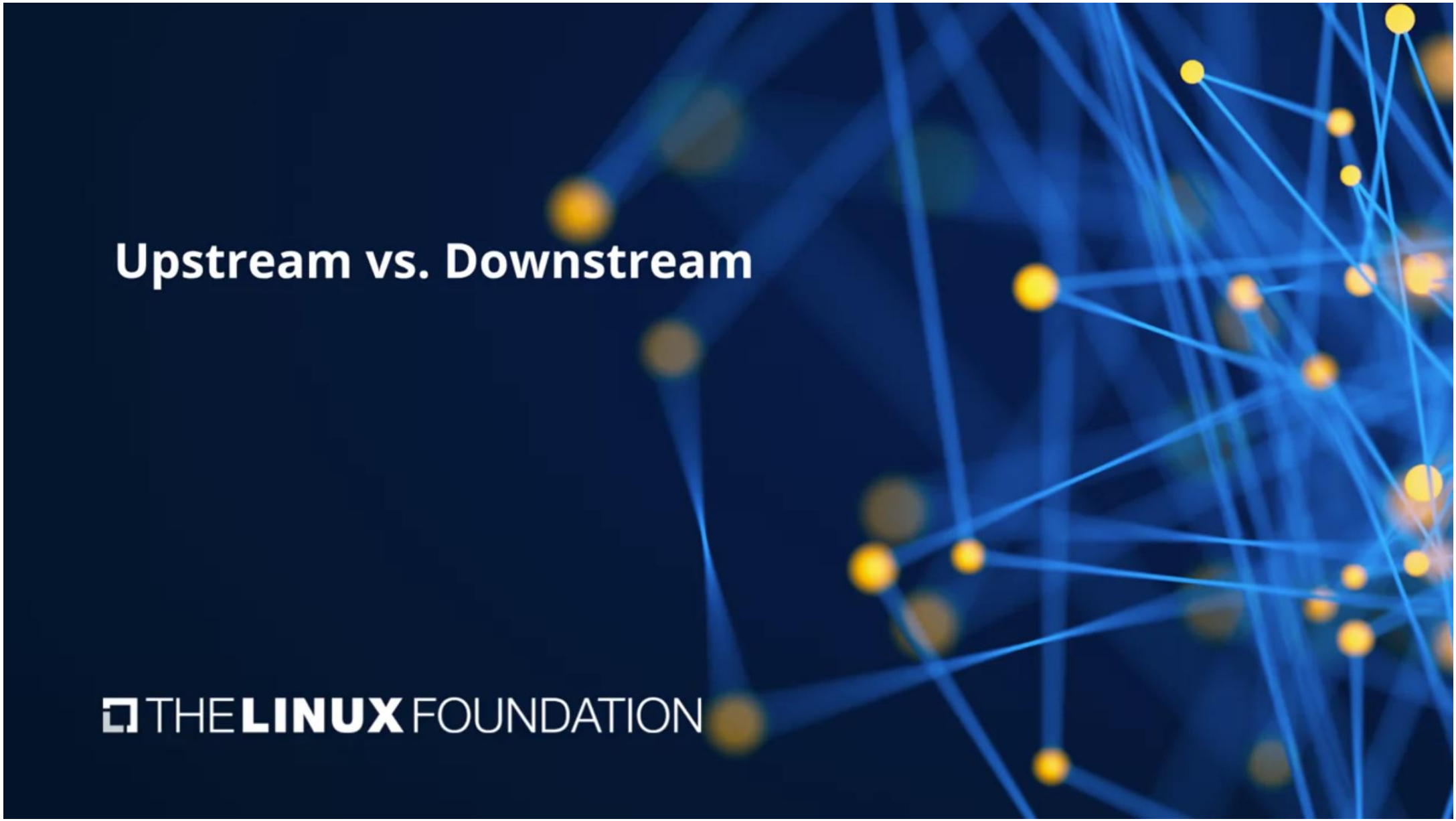
Committing vs. Publishing

THE LINUX FOUNDATION

Committing vs. Publishing

- The steps of committing and publishing are quite distinct in using git, as compared with other revision control systems
- A **commit** is a local process; it merely means saving the current state of your working project files in your local repository, thus setting up a convenient marker in the development history
  - Whether you commit often with small changes, or infrequently with large change sets is your decision
  - The commit operation requires no network access; you are free to reorganize your commits in your working repository before you make them public; you can pick the points to publish with care, so they are logical and well-defined
- **Publishing** means sharing your changes by making them public
  - This can be done either by a making a push or letting others pull, or by use of patches; this effectively freezes the repository history

Upstream vs. Downstream



THE LINUX FOUNDATION

Upstream vs. Downstream

- The parent repository is often considered **upstream** and the repository you are working on, at one point cloned from the parent, is often considered **downstream**
- Once again there is nothing in git itself which makes this distinction, akin to a server/client relationship, since git has a fundamentally peer-to-peer architecture
- Conceptually, any repository to which you send changes is considered upstream; any repository which is based on yours is considered downstream
- There can be multiple levels involved if one has sub-trees, or feeders
- One repository can have both downstream and upstream relationships; for instance, it might be the focus of a particular subsystem and the entire project repository can be upstream, while individual sub-projects can be downstream

Forking

THE LINUX FOUNDATION

Forking

- A project **forks** when someone takes the entire project and goes off in another direction; this is sometimes called branching
- In git this term has a different semantic meaning as there can be multiple branches within a given repository
- Technically, every time one clones a repository with git, one creates a fork, but these are not meant to be permanent bifurcations
- There are a number of reasons a project may fork: disputes among developers about the project direction or licensing issues, or political and personal conflicts
- What makes git so useful is that its robust merging capabilities make healing a fork quite simple
- Every time a branch is merged into the main repository the fork is healed; even forks which are quite old can often be merged back in with grace and ease

File Categories

THE LINUX FOUNDATION

File Categories

As far as git is concerned, files in your project directories fall into three categories:

- Tracked files
- Ignored files
- Untracked files

Tracked Files

- **Tracked files** are those that are already in the repository in their current working state, or have been staged; i.e. changed but not yet committed

Ignored Files

- **Ignored files** are invisible to git
- Each directory may contain a file named **`.gitignore`** which lists either specific files or name patterns which are to be ignored
- For instance many temporary files could be ignored with a specification like **`*.o`**
- The **`.gitignore`** file in any directory applies recursively to all subdirectories below it, so if you specify a filename or pattern in the main directory, all files fitting the description will be invisible

Ignored Files (Cont.)

- It is also possible to override the rules by prefixing with **!**; for example, if your **.gitignore** file includes:

```
*.ko
!my_driver.ko
```

- The second specification overrides the first more general one, and the file **my_driver.ko** will be tracked

Untracked Files

- **Untracked files** are anything that does not fit in the previous two classifications
- All files in your current working directories are examined and anything that is not tracked or ignored is considered untracked
- This may be files that have not been added yet, they may be temporary files, etc., but if you commit the entire directory, they will become tracked files

Making a Commitment

THE LINUX FOUNDATION

Commit Process

- The **commit process** is familiar from other version control systems
- However, the way git handles this process is very particular
- When a commit is made, a commit object is created from the files in the index, and the commit object is placed in the object store
- The files themselves are in the object store already when they are placed in the index
- Any new files since the last commit result in new blobs and any new directories result in new trees; any unchanged object is simply re-used
- Thus minimal new storage is required

Commit Process (Cont.)

- Furthermore this process is very fast because blobs do not have to be compared directly; all git has to do is compare their hexadecimal identifiers to see if they are identical
- In particular if the hash describing a directory has not changed, nothing in any of its subdirectories has changed either
- The commits are connected in an ancestral tree
- You should pick well-defined points, but whether you do many small commits or fewer larger ones is your choice
- Note however, that git does handle large numbers of small commits very efficiently, and when one uses the bisection tool, it can speed up finding points where bugs and regressions were introduced

Branches

What Is a Branch?

- At some point it may become necessary to move off the main line of development to create an independent **branch**
- For example, when a major release of a product comes out one may want to establish a maintenance release branch and a development branch:
  - The maintenance release branch may be restricted to fixing bugs and serious performance bottlenecks, and plugging security holes
  - Besides including these changes, the development branch would also carry forth incorporation of new features and optimizations, not intended for public release until the next major product version is ready
- A separate branch isolates a particular area of development or work on a very serious bug, without being burdened by the noise introduced by other simultaneous changes
- What makes git so useful is that both branching and inverse merging processes are easy, and the whole infrastructure has been structured with this in mind

What Is a Branch? (Cont.)

- This was inspired by the way the Linux kernel development community evolved
- Many parallel branches exist for things like network drivers, networking and USB
- In the git way of doing things, while it is the collection point for changes and is in some sense the most important branch, in a technical sense it is no different than any other branch (its importance is social, political, and tactical, not structural)
- One branch can track another while proceeding independently; it can be brought up to date with changes in the other branch without having to start over; the differences between the parallel branches can be kept as small as possible as they both advance
- While you can have many branches in a project as part of the same repository, you can only have one active (i.e. current) at a time
- The files in your working directories are those from that branch; if you switch branches, the files will change

Branch Names vs. Tags

- Sometimes people get confused about the difference between **branch names** and **tags**
- In fact it is possible to use the same string for both as they operate in different namespaces, but it does require some care and is not recommended for the non-expert
- A branch name represents a line of development:
  - Usually the main line of development is known as the master branch, as it is called by default
  - As time goes on other branches will be born, given names, and develop in parallel
  - They might have names like **`devel`**, **`debug`**, or **`stable`**
  - While the contents of the branch will change as time goes on and new commits are made, the name of the branch will not normally change, except in the case where another branch is embarked upon

Branch Names vs. Tags (Cont.)

- Tags on the other hand, represent a stage of a particular branch at one point in its history (unless you are asking for a lot of pain you would never change the name of a tag in the future; they should be **immutable**)
- If two branches share a common ancestor they will share tags that predate the separation
- If two parallel branches adopt the same name for a tag after they separate, the merging process will have to deal with this
- But remember the tag can just be thought of as a nickname for one of those long hexadecimal strings which are the fundamental commit identifiers

What Is Merging?

THE LINUX FOUNDATION

Merging

- All but the most primitive of revision control systems have to deal with the problem of **merging**
- This happens when more than one developer have been working on the project simultaneously, and the need arises to bring their changes back into a unified code base
- If the changes do not conflict with each other directly; i.e. they work on different files or on different parts of the same files, merging the work is not essentially difficult
- For example, if there are two change sets one could simply apply one and then the other; the order should not matter
- The only technical refinement is that after one set of changes there may be a shift of lines where the second set of changes has to work, due to insertions or deletions made by the first one

Merging

- Even in this simple case of non-overlapping change sets, more subtle problems can arise of the type that only humans are likely to notice
- Complications can easily be introduced, for example, if two distinct change sets both try to fix the same bug, but do it in different places with different methods
- In such cases using a git methodology of having many small patch sets and commits, combined with the use of tools like bisection, can help to rapid resolution of such subtle problems
- In the case of conflicting change sets, git has been designed from the outset to have strong tools for conflict resolution
- Merging can be seen as the inverse process to branching
- Without an efficient automated process for rejoining, we would have a much weaker revision control system

Working with Distributed Repositories

THE LINUX FOUNDATION

Working with Distributed Repositories

- Many other revision control systems are built on the notion of a central, authoritative repository which is the hub that individual developers work with
- An essential difference in git's architecture is that no one repository retains such a central role, at least not structurally
- Maintaining separate repositories make sense in at least three situations:
  - A developer is working autonomously
  - Developers are separated across a large (even global) network; a local group of developers may share their own repository to collect changes of immediate import and interest
  - There are divergent projects or sub-areas that are worth developing deeply on their own, with a mind to eventual merging of changes that are found to be beneficial to the main project
- git has a strong infrastructure dedicated to branching and merging multiple repositories, it can handle relatively easily the complicated logistics of distributed development

Operations Involved in Handling Remote Repositories

- The essential operations involved in handling remote repositories are:
  - **Cloning**: establish an initial copy of a remote repository, and place in your own object database; the essential command is **`git clone`**
  - **Pulling**: bring home changes from the remote repository keeping your tracking branch up to date; the essential commands are **`git pull`** and **`git fetch`**
  - **Pushing**: submit your changes to the remote repository; the essential command is **`git push`**
  - **Publishing**: making your repository available for others to clone, and pull from, and perhaps to push to
- git uses tracking branches to handle content in remote repositories
- These are local branches that serve as proxies, or references, for specific branches in remote repositories

Bare Repository

- There is a special kind of repository called a **bare** repository (normally you work in a development repository)
- A bare repository has no working directories and is not used for development and has no checked out branches; it is used only as an authoritative place to clone and fetch from and push to
- It is created with the **`--bare`** option to the clone command

Why Use Patches?

THE LINUX FOUNDATION

Why Use Patches?

- The first reason for doing this is to encourage review before changes are merged
  - For many projects, such as the Linux kernel, there are well established mailing lists and discussion groups
  - By enabling more eyeballs to view the patch, suggest or make changes and test before final submission, better development can be achieved
- A second reason is not all developers are using git
  - By reverting to the patch method these developers can also play with proposed change sets
- A third reason is that even if developers are using git, there may be interference, such as corporate firewalls, that interfere with using the git, ssh and even the http protocols
  - Email is likely to provide a method for bypassing these restrictions
- Fortunately, git has all the builtin infrastructure for handling patches and even for handling direct emailing


Press Esc to exit full screen
Emailing
THE LINUX FOUNDATION

git send-email

- Not surprisingly the command to directly email a patch is **`git send-email`**
- To send a message to the Linux kernel mailing list you would do:

```
$ git send-email -to linux-kernel@vger.kernel.org 001-first-commit.patch
```

- When doing this, you will be prompted for some information, such as who the message is coming from
- Additional information such as your smtp mail agent configuration can also be specified, as well as additional recipients of the emailed patch
- Whether or not this will work without adjustment on your system's mail setup (such as a possible re-configuration of sendmail) and how to fix any problems is beyond our scope here, as it can be difficult depending on your system configuration, and it might be impossible to do without being a privileged user

Other Email Clients

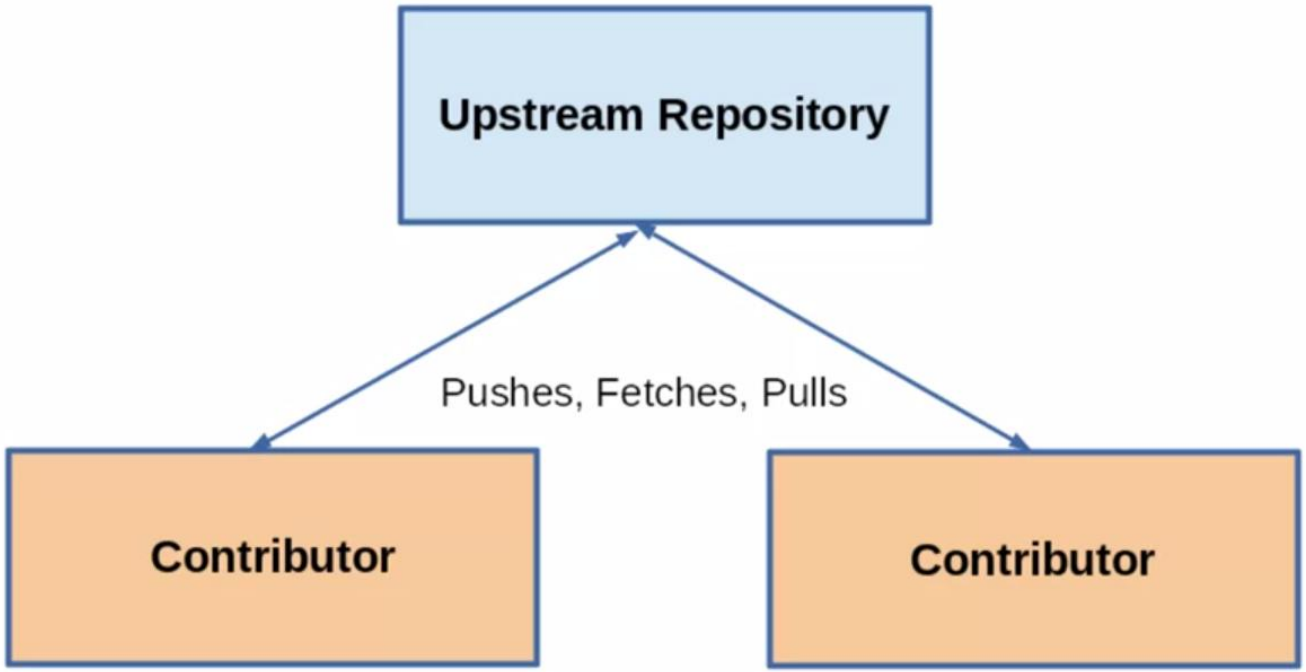
- There is no need to use the **`git send-email`** command; you can use whatever email client you are accustomed to, such as **thunderbird**, **evolution**, **mutt**, etc.
- But if you do use a conventional email client you must be careful to not screw up the patch in the mailing process:
  - Turn off any html encoding and send plain ASCII text
  - Turn off any line-wrapping or flowing
- Generally, in-lining the patches directly into the email message is preferred by most developers, rather than using attachments (which besides requiring extra steps to view) can expose other email client idiosyncrasies
- Even for very long patches attachments may not be a good idea

Gerrit

Models of Distributed Development

- Git has well-established methods of workflow
- It is pretty flexible and projects can choose quite different approaches, but it is always based on a cycle of:
  - Make changes in the code, probably in a development branch
  - Commit those changes to the branch; there may be one or more changes (patches) per commit
  - Publish those changes through a push or a pull request
  - The changes will be reviewed and merged if necessary or sent back for further work
- This method works well if you have a basic pyramid view of things where each subsystem has a maintainer managing their piece of the work, and that manager:
  - Has ultimate authority over the changes to that subsystem before passing them up the pyramid for ultimate review and merging

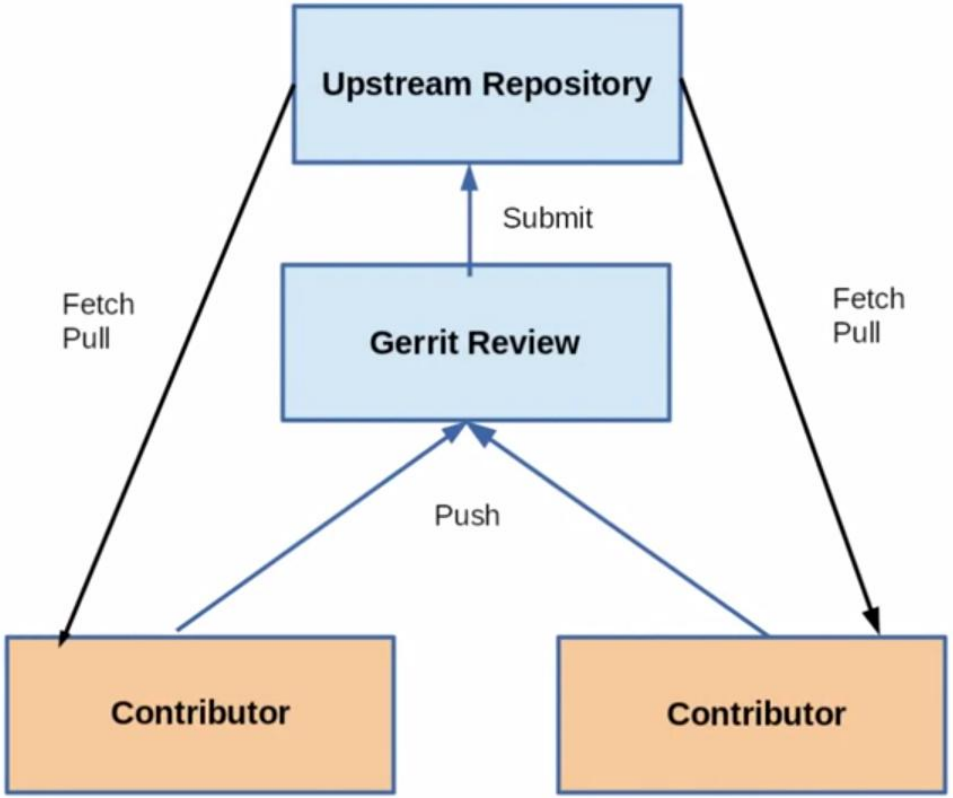
Simplified Git Workflow



Gerrit

- **Gerrit** comes in when you want to have more dispersed view
- While such a workflow is not exactly new (most projects have multiple reviewers with some structure for who makes the ultimate decisions) the Gerrit architecture is designed to formalize this procedure
- This works best when there is one change per commit, rather than a block of them, as it makes it easier to review and modify/reject/accept each one on its own merits

Git Workflow with Gerrit



Review Process

- From the preceding diagram, one can see that Gerrit introduces a reviewing layer that lies between the contributors and the upstream repository
  - Contributors submit their work (one change per submission is best) to the reviewing layer
  - Contributors pull the latest upstream changes from the upstream layer
  - Reviewers one the ones who submit work to the upstream layer
- The reviewers evaluate pending changes and discuss them
- According to project governing procedures they can grant approval and submit upstream, or they can reject or request modifications
- Gerrit also records comments about each pending request and preserve them in a record which can be consulted at any time to provide documentation about why and how modifications were made