we will now discuss version control systems. Many of you out there are familiar with **centralized version control systems**like Subversion (SVN), CVS, and Perforce, while others have jumped straight into the [**distributed version control**](https://bitbucket.org/product/version-control-software) worlds of [Git](http://www.atlassian.com/git/) and Mercurial. There are many other flavors of centralized and distributed version controls out there – each with there own advantages and disadvantages.

Centralized Version Control

There are many version control systems out there. Often they are divided into two groups: “centralized” and “distributed”.

Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy.

“Committing” a change simply means recording the change in the central system. Other programmers can then see this change. They can also pull down the change, and the version control tool will automatically update the contents of any files that were changed.

Most modern version control systems deal with “changesets,” which simply are a groups of changes (possibly to many files) that should be treated as a cohesive whole. For example: a change to a C header file and the corresponding .c file should always be kept together.

Centralized version control solves the problems described in the [previous post on What is Version Control?](http://atlassian.com/blog/2012/02/version-control-diffs-patches/). Programmers no longer have to keep many copies of files on their hard drives manually, because the version control tool can talk to the central copy and retrieve any version they need on the fly.

Some of the most common centralized version control systems you may have heard of or used are CVS, Subversion (or SVN) and Perforce.

A Typical Centralized Version Control Workflow

When you’re working with a centralized verison control system, your workflow for adding a new feature or fixing a bug in your project will usually look something like this:

* Pull down any changes other people have made from the central server.
* Make your changes, and make sure they work properly.
* Commit your changes to the central server, so other programmers can see them.

Distributed Version Control

In the past five years or so a new breed of tools has appeared: so-called “distributed” version control systems (DVCS for short). The three most popular of these are Mercurial, [Git](http://www.atlassian.com/git/) and Bazaar.

These systems do not necessarily rely on a central server to store all the versions of a project’s files. Instead, every developer “clones” a copy of a repository and has the **full** history of the project on their own hard drive. This copy (or “clone”) has *all* of the metadata of the original.

This method may sound wasteful, but in practice, it’s not a problem. Most programming projects consist mostly of plain text files (and maybe a few images), and disk space is so cheap that storing many copies of a file doesn’t create a noticable dent in a hard drive’s free space. Modern systems also compress the files to use even less space.

The act of getting new changes from a repository is usually called “pulling,” and the act of moving your own changes to a repository is called “pushing”. In both cases, you move changesets (changes to files groups as coherent wholes), not single-file diffs.

One common misconception about distributed version control systems is that there *cannot* be a central project repository. This is simply not true – there is nothing stopping you from saying “this copy of the project is the authoritative one.” This means that instead of a central repository being *required* by the tools you use, it is now optional and purely a social issue.

Advantages Over Centralized Version Control

The act of cloning an entire repository gives distributed version control tools several advantages over centralized systems:

* Performing actions other than pushing and pulling changesets is *extremely* fast because the tool only needs to access the hard drive, not a remote server.
* Committing new changesets can be done locally without anyone else seeing them. Once you have a group of changesets ready, you can push all of them at once.
* Everything but pushing and pulling can be done without an internet connection. So you can work on a plane, and you won’t be forced to commit several bugfixes as one big changeset.
* Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone.

Disadvantages Compared to Centralized Version Control

To be quite honest, there are almost no disadvantages to using a distributed version control system over a centralized one. Distributed systems do *not* prevent you from having a single “central” repository, they just provide more options on top of that.

There are only two major inherent disadvantages to using a distributed system:

* If your project contains many large, binary files that cannot be easily compressed, the space needed to store all versions of these files can accumulate quickly.
* If your project has a very long history (50,000 changesets or more), downloading the entire history can take an impractical amount of time and disk space.

The authors and contributors of modern distributed version control systems are working on solving these problems, but at the moment, no bundled, built-in features solve them.

Conclusion

Version control systems aim to solve a specific problem that programmers face: “storing and sharing multiple versions of code files.” If you’re a programmer of any kind and you don’t use any kind of version control, you should start right now. It will make your life easier.