Computing Activity Protocol

May 5, 2018

Sat Apr 13 29:30:12 CEST 2018

Managing large binary files alongside git

Problem 1. Only using standard `git` isn't elegant for managing and versioning binary files in addition to text-based files. For binary files, `diff`s most often don't make sense and there's no general behaviour or rules for how they would be merged. If there's no reasonable way to merge, the files have to be stored whole. Since binary files are usually way larger than pure source code files, saving them as whole blobs for each commit enourmously increases the size of the core git repository.
The solution must involve the storage of a set of binary files (whole chunks, no diffs) for each commit that produced new ones, outside the core git repository, but still associatable with each commit.
What I specifically want at this point is to export a pdf (e.g. current version from latex file), and misc. exported figures (pdf or png) for different versions of a program.
This would be great for collaboration, since if the binary files aren't integral to the git repository itself, a supervisor doesn't have to deal with an unprofessionally cluttered up and ever growing git repo containing binary files, but rather just `get`s or `drop`s only the necessary binary files (demo-files, figures, pdfs) to review at will.

Solution There exist several approaches to do this:

1. I tried `git-media` first, which turns out to not be developed actively any longer and therefore I abandoned it.

2. `git-annex` seems to be promising, and that's what I've implemented.

`git-annex` is a decentralized system, where binary files can be added to a commit. Those binaries are then stored in a special subfolder somewhere in `.git/`, and symlinks are automatically put at the right positions in the working tree. Core `git` only tracks and checks out the symlinks (those are usually text files with a hash of the associated binary file).
I set it up with one client (non-bare repo) and one server (bare repo), where the client pushes changes to the server and syncs the binary files (`rsync` by default, and over ssh, in this case is `rsync` not a special remote) with a call to `git annex sync --content`.
Details can be found in the walkthrough on the official `git-annex` webpage `http://git-annex.branchable.com/walkthrough/`.

Problem 2. There are differences between a plain ol' `remote` and a `git-annex` special remote. Special remotes are places, where the contents of the binary files are stored (optionally encrypted), while git annex can still operate on them much as it would on any other remote. Such special remotes support also nice integration with a whole variety with widely used web services such as Dropbox, Amazon S3, or your own ssh rsync server like. Using a special remote is sensible to do if your core git repo is stored on GitHub, which doesn't yet support `git-annex`. ("Once you've initialized a special remote in one repository, you can enable use of the same special remote in other clones of the repository."[1])

Sat Apr 14 20:01:02 CEST 2018:

Blender As A Python Module

As I want to be able to use Blender's rendering and logic capabilities to make scientific illustrations in a programmatic way, I read a few paragraphs about the Blender/Python API (`https://docs.blender.org/api/blender_python_api_2_68_release/contents.html`). I tried writing python scripts with blender GUI open at the same time in it's Text Block and interactive console mode, but found debugging to be very inefficient and mentally exhausting, because one had to constantly reload scripts and re-open blender. I now learned (`https://docs.blender.org/api/blender_python_api_2_68_release/info_tips_and_tricks.html`) that in a text block you can re-load a script that you edit in an external Editor/IDE. That still uses Blender and launches an insulated python interpreter within blender, enabling the use of the `import bpy` library. Sadly though, when writing custom scripts from outside blenders built-in interactive console, there is no code completion, because `bpy` sources aren't accessible from the outside (for some reason). For python programmers who want to run a python script as the main process and call the blender library `import bpy` as a module from within python (as opposed to the standard way, where blender calls python from within it's process), there is a way to build blender from source with slightly modified options for CMake (`https://wiki.blender.org/index.php/User:Ideasman42/BlenderAsPyModule`), so that after proper configuration you can use only python scripts to call blender and produce an output, and also providing the source and code completion from outside the blender GUI. I still have to figure out, in what ways the behaviour differs for both workflows (combination of GUI and interactive console vs only scripting).
I now pulled all of blenders sources (`https://git.blender.org/blender.git`) and built dependencies according to the instructions at `https://wiki.blender.org/index.php/Dev:Doc/Building_Blender/Linux/Ubuntu/CMake`. Especially installing all dependencies (automatically with the provided `install_deps.sh` script) took a long time and some things that are already installed may have been built from source code again. I am wondering if a `make install` will then also override the already installed things (like numpy) and if that will cause a broken installation or redundancies.

[1] `http://git-annex.branchable.com/walkthrough/#index12h2`

The next step is to build Blender using CMake. I'm not quite sure, where the provided CMake options are to be inserted. Also, I don't know where all the `bpy` sources are going to be added to (probably the usual local or system-wide installation directories). Also, for Jedi-Vim I'm not sure if it will find blenders sources for code-completion and syntax checking.

OK, I now know what CMake is, great (It's a generator for buildsystems, it can generate GNU-Makefiles to be executed with `make`, btw. there are other buildsystems apart from GNU `make` and on other platforms, not just Linux). You can specify options like this:
`cmake -DWITH_PYTHON_INSTALL=OFF -DWITH_PLAYER=OFF -DWITH_PYTHON_MODULE=ON ../blender`
This will setup the appropriate GNU Makefile and `make` should work as expected.

Mon Apr 16 14:59:56 CEST 2018

Setting up a virtual lab environment for system administration tests

It is important to have a lab environment/network to *play around* with installations and configurations. Testing in a lab system removes a large part of the risk when it comes to making changes in an already working system. Since the samba server would be linux server with multiple windows and unix clients, the lab network needs to consist of at least one host acting as the samba server (here: non-virtual ubuntu) and two clients (here: two windows virtual machines). This virtual lab needs to be properly configured first (making virtual snapshots from which you can quickly spin up clients with custom configurations, here VirtualBox is used).
TODO:

- Remove current double-boot and install one private Ubuntu (500 GB) and one *Akaflieg Testing* Ubuntu (300 GB) (I doubt that I have enough memory and power to run 4 virtual machines alongside each other) alongside with their respective swap partitions (each 15 GB) and a seperate empty FAT32 partition, for file sharing, storing virtual machines, etc. (the rest, ca. 1 GB). The *Akaflieg Testing* Ubuntu (AFT Ubuntu) is used to be able to have a fresh system for testing out installations and is exclusively used to make experiments.

- Get Windows 7 and Ubuntu virtual machines with basic installations (cygwin, ssh, git, rsync) up and running in VirtualBox (with guest-additions). Then clone them and assign different hostnames and ip adresses to distinguish them from each other. Make sure that they are all connected to each other (ping ip's).

- Install samba server on AFT Ubuntu and