GitP4Transfer - Git to P4 Migration

Perforce Professional Services

2023-08-01

Table of Contents

Chapter 1. GitP4Transfer.py

1.1. Overview

This is a functional script to import a git repo with LFS commits for a single branch (e.g. master/main) into Helix Core.

It includes some ideas from `git-p4` and `git-filter-repo.py`.

1.2. Pre-requisites

- Install recent version of git (2.x)
- Install Python 3.8+ and modules p4python

1.2.1. Install Git/Git LFS

Easiest to install these from Wandisco to get recent versions (need 2.x and not 1.8 for example):

```
sudo yum install http://opensource.wandisco.com/centos/7/git/x86_64/wandisco-git-release-7-2.noarch.rpm
sudo yum install git git-lfs
git --version
```

1.2.2. Clone repo

This assumes a filesystem such as `/hxdepots` with plenty of free space.

```
cd /hxdepots
mkdir work
cd work
git clone <url of your git repo>
```

1.2.3. Fetch all LFS objects

1. First ensure that git LFS credentials are stored

   ```
   git config --global credential.helper store
   ```

   ```
   $ git branch
   * master
   ```

   ```
   $ git lfs fetch --all
   ```

```
fetch: 163739 object(s) found, done.
fetch: Fetching all references...
Username for 'https://git.example.com': fred.bloggs
Password for 'https://fred.bloggs@example.com':
Downloading LFS objects:   4% (6561/163738), 9.1 GB | 100 MB/s
```

2. After the above you can Ctrl+C to abort because credentials should be in place.

```
cat ~/.git-credentials
```

3. If you want to check, the re-run the command and you should not be prompted.

```
git lfs fetch --all
```

4. Finally you can spawn the full fetch of all LFS versions (which often takes hours depending on size of your repo):

```
nohup git lfs fetch --all > ../fetch.out &
```

```
perforce@ip-10-0-0-151 gitrepo]$ cat ../fetch.out
fetch: 163739 object(s) found, done.
fetch: Fetching all references...
```

5. Check for LFS files not found too - all files less than 140 bytes in size are possible candidates to be checked:

```
find .git/lfs/objects/ -type f -size -140c
```

6. LFS files which have not been replaced with their proper contents will be similar to this sort of format:

```
version https://git-lfs.github.com/spec/v1
oid sha256:8923f38904c1ae21cd3d3e6e93087c07fda86fe97ee01d8664bb95fc20cd1de
size 858449
```

If such files are found, you will need to determine why they were not fetched and try to fix that to get proper LFS contents downloaded.

1.2.4. Install Python3.8

Unfortunately 3.6 is missing some required changes in the `subproc` library, so you may ßneed to build from source. Ubuntu is similar (but different dependencies to install first!)

```
yum install wget yum-utils make gcc openssl-devel bzip2-devel libffi-devel zlib-devel
VER="3.8.12"
wget https://www.python.org/ftp/python/$VER/Python-$VER.tgz
tar zxvf Python-$ver.tgz
cd Python-$ver
./configure
make install
```

1.2.5. Install GitP4Transfer.py

We install dependencies and then the script itself.

1. Run the following as `root`:

```
cat << EOF > /etc/yum.repos.d/perforce.repo
[Perforce]
name=Perforce
baseurl=http://package.perforce.com/yum/rhel/7/x86_64/
enabled=1
gpgcheck=1
EOF

rpm --import https://package.perforce.com/perforce.pubkey

yum install perforce-p4python3
```

2. As normal user, e.g. `perforce`:

```
pip3 install --user requests ruamel.yaml
```

3. Clone the gitp4transfer repo

```
git clone https://github.com/perforce/gitp4transfer.git
```

4. Ensure dependencies setup

```
cd gitp4transfer
python3 GitP4Transfer.py -h
```

The above should produce output like:

```
$ ./GitP4Transfer.py -h
usage: GitP4Transfer.py [-h] [-c CONFIG] [-n] [-m MAXIMUM] [-r] [-s] [--sample-
config] [--end-datetime END_DATETIME]
```

```
NAME:
    GitP4Transfer.py

DESCRIPTION:
    This python script (3.8+ compatible) will transfer Git changes into a Perforce
    Helix Core Repository, somewhat similar to 'git p4' (not historical) and also
    GitFusion (now deprecated).

    This script transfers changes in one direction - from a source Git server to a
    target p4 server.
    It handles LFS files in the source server (assuming git LFS is suitably
installed and enabled)

    Requires Git version 2.7+ due to use of formatting flags

    Usage:

        python3 GitP4Transfer.py -h

    The script requires a config file, by default transfer.yaml. An initial example
can be generated, e.g.

        GitP4Transfer.py --sample-config > transfer.yaml

    For full documentation/usage, see project doc:

        https://github.com/rcowham/gitp4transfer/blob/main/doc/GitP4Transfer.adoc

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Default is transfer.yaml
  -n, --notransfer      Validate config file and setup source/target workspaces but
don't transfer anything
  -m MAXIMUM, --maximum MAXIMUM
                        Maximum number of changes to transfer
  -r, --repeat          Repeat transfer in a loop - for continuous transfer as
background task
  -s, --stoponerror     Stop on any error even if --repeat has been specified
  --sample-config       Print an example config file and exit
  --end-datetime END_DATETIME
                        Time to stop transfers, format: 'YYYY/MM/DD HH:mm' - useful
for automation runs during quiet periods e.g. run overnight but stop first thing in
the morning

Copyright (C) 2021-22 Robert Cowham, Perforce Software Ltd
```

5. Setup config file

```
python3 GitP4Transfer.py --sample-config > transfer_config.yaml
```

6. Create appropriate target depot, e.g. `//git_import/repoA/master` and ensure setup in config file.
7. Do a test of config:

   ```
   python3 GitP4Transfer.py -c transfer_config.yaml -n
   ```

   Validate log files for success.

8. Consider setting up `p4 typemap` as appropriate for your import (e.g. for Unreal Engine or Unity)
9. Do a first test of one commit (note this is often quite a big commit so may still take a while!)

   ```
   python3 GitP4Transfer.py -c transfer_config.yaml -m1
   ```

10. If the above works, kick off a full transfer and monitor log/output file:

    ```
    nohup python3 GitP4Transfer.py -c transfer_config.yaml > out &
    ```

1.2.6. Note about temp branch

The script works by replaying each commit. To do this it executes:

```
for each commitid in reverse order:
    git switch -C p4_exportBranch <commitid>
    parse the output of git diff-tree against previous commit
    run various p4 commands
```

As a result, expect the new branch `p4_exportBranch` to be created and continually updated. This is effectively a dummy branch.

When the script has finished you may need to: `git checkout master` or similar to reset to your current branch.

if the script fails, then the active branch is going to be the temp one - don't assume it is HEAD/master!

1.2.7. Things to do

- Adjust unknown_git user
- Date times for changes update
- Interleave in date/time order
- More informative commit messages

1.2.8. Branch diffs

Generated by:

```
git log --first-parent --oneline master > ../b_master.txt
```

Chapter 2. gitp4transfer - tool for Git and Plastic SCM migrations

This uses git's `fast-import` (or `fast-export`) file format which is more or less supported as an export format for Plastic SCM (with caveats below!)

These files contain a complete record of the main information from a git repository:

- file contents (blobs) - the actual content of each unique file (e.g. a git SHA1 entry)
- commits (including attributes such as user/datetime/message etc) and with lists of actions within that commit:
  - modify (new or edited file)
  - delete (of file or directory)
  - rename (of file or directory)
  - copy (of file or directory)
- branch indications (per commit)

The format of these files is documented here: https://git-scm.com/docs/git-fast-import

Simple examples are included in the test module: main_test.go

```
blob
mark :1
data 9
contents

blob
mark :2
data 10
contents2

reset refs/heads/main
commit refs/heads/main
mark :3
author Robert Cowham <rcowham@perforce.com> 1680784555 +0100
committer Robert Cowham <rcowham@perforce.com> 1680784555 +0100
data 8
initial
M 100644 :1 src/file1.txt
M 100644 :2 src/file2.txt
```

2.1. Overall Approach

The `gitp4transfer` tool can be run on a `git fast-export` file, and the tool creates:

* a `p4d` journal format (using schema for p4d 2004.2 release)
* individual librarian files (equivalent of git blobs) in a directory tree corresponding to p4d depot structure

These are used to create a Perforce Helix Core repository:

* `p4d -jr <journal>` to create db.* metadata files
* `p4d -xu` to upgrade the db.* to format of the actual `p4d` version schema
* `p4 storage` command is run to perform the 2019.1 `p4d` storage upgrade process
* `p4 verify -qu` command is run to perform an update of the MD5 checksums for all depot files

Optionally:

* `p4d -xi` can convert the repository to unicode mode
* you can specify that a case-sensitive Plastic Repository is converted to `p4d` repository running in "case-insensitive" mode (at the risk of the loss of some actions, e.g. those which only differ in case)

All of the above operations are wrapped by the `run_conversion.sh` script (Section 2.4, “Using the shell wrapper run_conversion.sh”) with its various parameters.

2.1.1. Resulting converted repository and how to use it

The result of a successful repository converted by `gitp4transfer` contains:

* a top level depot with all branches
* branching is "sparse" meaning that only files changed on actual git branches are populated withint the resultind p4d repo
    * therefore you will need to review

Using the result can be done in two main ways:

* As a standalone p4d repository
* As a basis for using p4 DVCS commands (p4 push/fetch) to transfer the converted contents into an already existing repository
    * Note the usual caveats about the requirements for compatible settings for case sensitivity, unicode etc.

2.2. Approach for Plastic SCM

Plastic’s git export capability writes files which are only directly git-compatible for simple repos. The following issues occur in practice:

* branch names contain spaces (auto-converted to underscores "_")
* multiple operations in the same commit can over-write each other - especially common when merging commits between branches. Examples include:

- modify of a renamed file (so called "dirty rename")
- delete of the target of a renamed file (which means the rename is not actioned)
- delete of an already deleted file
- rename of a renamed file
- many, many more! These are "documented" as part of the test suite

The above are complicated by the fact that you can perform actions on a directory (e.g. rename "src" to "targ" which affect all the files current present in that directory "src"). Also a "directory action" such as rename or delete, may override individual file actions which came previously in the same commit! This is especially common when you look at commits which are the result of "merges" from other branches.

The net result is a lot of complexity and iterating through all the "file actions" in a commit, trying to decide which ones are still valid, and which are invalid because they have been over-ridden by a later action.

Plastic's "creativity" in this area (actions over-riding other actions in same commit) means that there likely to be scenarios still not yet correctly covered by the current tool.

2.2.1. Creating Plastic Git Export file

Create workspaces:

```
cm wk mk SomeRepo /data/work/SomeRepo SomeRepo@some_server@cloud
cd /data/work/SomeRepo
```

Show rough stats:

```
[robert@testhost SomeRepo]$ cm status
/main@SomeRepo@some_server@cloud (cs:49123 - head)
```

Other details:

```
[robert@testhost SomeRepo]$ cm find changeset > ../cm_SomeRepo.txt

[robert@testhost Onward2Repo]$ tail -3 ../cm_SomeRepo.txt
1112339  49123    /main/beta/dev/Art 07/07/2022 09:29:51 fred@example.com Update files

Total: 49012
```

Note the number of the final changelist and the total may not be the same.

2.2.2. Exporting of data

From within the workspace directory, launch the Plastic `cm fast-export` command - this creates a complete single git `fast-export` format file containing all file revisions and all commits.

```
nohup cm fast-export --export-marks=marks.cm SomeRepo@some_server@cloud ../git-
SomeRepo > ../out_SomeRepo &
```

This can take a while (hours), and also can produce rather large export files (e.g. many TB in size!)

2.3. Running gitp4transfer

Once you have created the git export, you can run the `gitp4transfer` tool to process it. While it does work on Windows, it is usually run on Linux (or Mac).

The latest release of this tool is available here: https://github.com/rcowham/gitp4transfer/releases

Basic help and options are shown below. It is normally expected that you don't run the tool directly but use the wrapper script Section 2.4, "Using the shell wrapper run_conversion.sh" (which does expect that `gitp4transfer` is in the $PATH)

```
$ ./gitp4transfer -h
usage: gitp4transfer [<flags>] [<gitimport>]

Parses one or more git fast-export files to create a Perforce Helix Core import

Flags:
  -h, --help                  Show context-sensitive help (also try --help-long and
--help-man).
  -c, --config="gitp4transfer.yaml"
                              Config file for gitp4transfer - allows for branch
renaming etc.
  -d, --import.depot="import"    Depot into which to import (overrides config).
      --import.path=IMPORT.PATH  (Optional) path component under import.depot
(overrides config).
  -b, --default.branch="main"    Name of default git branch (overrides config).
      --case.insensitive         Create checkpoint case-insensitive mode (for Linux)
and lowercase archive files. If not set, then OS default applies.
      --convert.crlf             Convert CRLF in text files to just LF.
      --dummy                    Create dummy (small) archive files - for quick
analysis of large repos.
      --dump                     Dump git file, saving the contained archive contents.
  -a, --dump.archives            Saving the contained archive contents if --dump is
specified.
  -m, --max.commits=0            Max no of commits to process (default 0 means all).
      --dryrun                   Don't actually create archive files.
      --archive.root=ARCHIVE.ROOT
                                 Archive root dir under which to store extracted
archives.
```

```
      --graphfile=GRAPHFILE      Graphviz dot file to output git commit/file structure
to.
      --journal="jnl.0"          P4D journal file to write (assuming --dump not
specified).
      --debug=0                  Enable debugging level.
      --parallel.threads=0       How many parallel threads to use (default 0 means no
of CPUs).
      --debug.commit=0           For debugging - to allow breakpoints to be set - only
valid if debug > 0.
      --version                  Show application version.

Args:
  [<gitimport>]  Git fast-export file to process.
```

2.3.1. Configuration YAML file

This is a fairly simple format:

```
# import_depot: the p4d depot into which all data is imported
import_depot:       import

# import_path: an optional extra path within the import_depot, e.g.
#   "git/repo1" => //import/git/repo1/<branch>/<git path>
import_path:

# default_branch: the name of the git default branch (typically "main" or "master")
default_branch:     main

# branch_mappings: an array of name/prefix pairs which "rename" the original branches
#   into the p4 repository.
#   Note that the name is a go regex, and the prefix is the equivalent of import_path
above
# - name:   main.*
#   prefix: trunk
# - name:   .*
#   prefix: dev
branch_mappings:

# typemaps: an array of strings representing (simplified) typemap lines.
#   Note type defaults to text+C but various binary formats are correctly
#   detected already, and identified as "binary" or "binary+F"
typemaps:
- binary //....uasset
- binary //....exe
```

2.4. Using the shell wrapper run_conversion.sh

This is a Bash script which provides a convenient wrapper to perform conversions and subsequent repository building actions automatically.

```
$ bash ./run_conversion.sh -h
USAGE for run_conversion.sh:

run_conversion.sh <git_fast_export> [-p <P4Root>] [-d] [-c <configfile>] [-dummy] [-
crlf] [ [-insensitive] | [-sensitive] ]
    [-depot <import depot>] [-graph <graphFile>] [-m <max commits>] [-t <parallel
threads>]

   or

run_conversion.sh -h

    -c           <configfile> name of Yaml config file to control conversion (means
parameters below don't need to be provided)
    -d           Debug
    -depot       <import depot> - Depot to use for this import (default is 'import')
    -crlf        Convert CRLF to just LF for text files - useful for importing Plastic
Windows exports to a Linux p4d
    -unicode     Create a unicode enabled p4d repository (runs p4d -xi)
    -dummy       Create dummy archives as placeholders (no real content) - much faster
    -graph       <graphfile.dot> Create Graphviz output showing commit structure (see
also 'gitgraph' utility which is more flexible)
    -insensitive Specify case insensitive checkpoint (and lowercase archive files) -
for Linux servers
    -sensitive   Specify case sensitive checkpoint and restore - for Mac/Windows
servers (for testing only)
    -m           <max commits> - Max no of commits to process (stops after this number
is reached)
    -t           <parallel threads> - No of parallel threads to use (default is No of
CPUs)
    -p          <P4Root> - directory to use as resulting P4Root - will default to a
tmp dir if not set
    <git_fast_export> The (input) git fast-export format file (required)

Examples:

./run_conversion.sh export.git
./run_conversion.sh export.git -p P4Root

nohup ./run_conversion.sh export.git -p P4Root -d -c config.yaml > out1 &
```

2.4.1. Example conversions

Typical conversion process is:

- Run `run_conversion.sh` in the background (can take hours) on an export file
- Review output, concentrating particularly on:

  - Any verify errors (indicates some sort of logic error in `gitp4transfer`)
  - Run `p4` commands against the created `p4d` instance to review output
  - Run a `p4d` in the output directory and then use `P4V` to review the output

Once things are looking reasonable, you need to test the resulting conversion. E.g.

- Use Plastic `cm` to create a workspace with latest known state (on a specific branch)
- Create a `p4` workspace for the resulting conversion (same branch), e.g. with root `/data/work/p4ws`
- Sync the workspace
- Run a local diff between the two and look to understand any differences in file contents, e.g.
  - `diff -qr /data/work/p4ws /data/work/SomeRepo`

Note that sometimes it is easier to "fix forwards" if there are only a few errors, by manually copying those files over and submitting them.

2.5. Useful Utilities

These are useful tools which help to understand the repository structure, debug issues with the conversion, and to test things like resulting p4 structure without worrying about the contents of all files (because processing of all file contents can take many hours vs a few minutes for filtered views)

2.5.1. gitfilter

This tool allows you to process a git fast-export file and produce a new version which filters out the actual file contents. Instead it replaces the contents of every `blob` file with its unique ID.

The advantages of this utility are:

- resulting git export file is hugely smaller (e.g. TB → a few GB) and can be processed much quicker
- it has options to produce a filtered export file which only includes file actions matching specific regex patterns (allows for even more hugely reduced export files and thus enables interactive debugging of problems which might otherwise)

```
$ ./gitfilter -h
usage: gitfilter [<flags>] [<gitimport>] [<gitexport>]

Parses one or more git fast-export files to filter blob contents and write a new one

Flags:
  -h, --help                     Show context-sensitive help (also try --help-long and
--help-man).
  -r, --rename                   Rename branches (remove spaces).
  -f, --filter.commits           Filter out empty commits (if --path.filter defined).
  -m, --max.commits=MAX.COMMITS  Max no of commits to process.
```

```
      --path.filter=PATH.FILTER  Regex git path to filter output by.
  -d, --debug=DEBUG              Enable debugging level.
      --debug.commit=0           For debugging - to allow breakpoints to be set - only
valid if debug > 0.
      --version                  Show application version.

Args:
  [<gitimport>]  Git fast-export file to process.
  [<gitexport>]  Git fast-import file to write.
```

Examples

```
# Filter a file with debug output
nohup ./gitfilter --debug 1 --rename big_file.git filtered_file.git > out1 &

# Same as above with only 500 commits
nohup ./gitfilter --debug 1 --rename --max.commits 500 big_file.git filtered_file.git
> out1 &

# Filter with all file history ommitted except for specified regex paths
nohup ./gitfilter --debug 1 --rename --path.filter "file1/.txt|file2\.txt"
big_file.git filtered_file.git > out1 &
```

2.5.2. gitgraph

This utility produces Graphviz `*.dot` files which can create a graphical view (e.g. .svg which can be opened in a browswer) of the git commit structure.

While a graph.dot which contains 50k nodes is not easily processed by Graphviz in any sensible amount of time (think hours), with the filtering options (specifying start and end commits) and using a file which has been output from Section 2.5.1, “gitfilter” you can create a quick visual representation of the git branching structure and see things like merges between branches etc.

Very useful for some debugging/understanding of branching relationships.

```
$ ./gitgraph -h
usage: gitgraph [<flags>] [<gitexport>]

Parses one or more git fast-export files to create a graphviz DOT file

Flags:
  -h, --help             Show context-sensitive help (also try --help-long and --help
-man).
  -m, --max.commits=0    Max no of commits to process (default 0 means all).
  -o, --output=OUTPUT    Graphviz dot file to output git commit/file structure to.
  -f, --first.commit=0   ID of first commit to include in graph output (default 0 means
all commits).
  -l, --last.commit=0    ID of last commit to include in graph output (default of 0
means all commits).
```

```
  -s, --squash          Squash commits (leaving branches/merges only).
      --debug=0         Enable debugging level.
      --version         Show application version.

Args:
  [<gitexport>]  Git fast-export file to process.
```

Example usage

```
# Graph first 500 commits
./gitgraph --max.commits 500 --output graph.dot export.git

# Graph commits between start and end, ommitting any commits without either parent or
merge
./gitgraph --first.commit 10234 --last.commit 21321 --output graph.dot export.git

# For the above you can create SVG files which can be opened with any browser
# Install Graphviz dot as per your operating system

dot -Tsvg graph.dot > graph.svg
```

2.6. Troubleshooting

2.6.1. Verify Errors

If the resulting conversion contains verify errors, then it implies a logic error in the tool which is not coping with some not previously encountered Plastic/git scenario.

The resolution is to consult with the author of the tool:

- Create a log file with debug on
- Consider providing a `gitfilter` produced git export (contains filenames but no data)