Mercurial: Add functionality to store an unresolved merge-state

Organization: Python Software Foundation

Table of Contents:

Sub-Org Info

I want to work with **Mercurial**.

Version Control Systems are an inevitable part of software development workflow. It's necessary to track changes to the source code and keep the written code safe and easy to run without dealing with problematic cases upon adding new features. **Mercurial** is a free, distributed source control management tool. It efficiently handles projects of any size and offers an easy and intuitive interface.

About Me

Basic Information

Name: Navaneeth Suresh

Email: navaneeths1998@gmail.com

IRC: navaneeth at freenode.net

Phabricator: navaneeth.suresh

Bitbucket: navaneethsuresh

Blog: http://themousepotato.github.io/

Phone: +91-8281169538

Timezone: IST (UTC +5:30)

Personal Background

I am a second-year undergraduate student at IIT Kharagpur [1], India. I'm pursuing a degree in Ocean Engineering and Naval Architecture. I've been coding in Python for five years and in C and C++ for over three years and am proficient in all three of them now.

I used to write programs in Python quite often as it is easy and fast to code. Compared to other programming languages, its readability is the most exciting thing I have come across.

I am quite familiar with version control systems. I have used CVS while working as a webmaster [2] at the GNU Project and both Mercurial and Git for maintaining personal projects as well as organization based projects.

Contributions

[MERGED] Patch: branches: add -r option to show branch name(s) of a given rev (issue5948)

The default behavior of `hg branches` command doesn't allow to accept revsets. This patch changed its behavior to accept revsets.

[MERGED] Patch: log: fixes line wrap on diffstat (issue5800)

When used with both `--graph` and `--stat`, `hg log` command had a problem of wrapping the text. This problem was solved by exploiting the `graphwidth` parameter.

[MERGED] Patch: histedit: add warning message on editing tagged commits (issue4017)

After `histedit`, the tags associated with the changesets are lost. This patch added a prompt when the user tries to edit the history of changesets with tags using `hg histedit`.

[MERGED] Patch: tags: avoid generating commit for an already deleted tag (issue5752)

This made hg tags --delete <tagname> to stop creating commits for an already deleted tag.

[MERGED] Patch: diffstat: make --git work properly on renames (issue6025)
On using both --stat and --git, hg log command showed only the current name of a renamed file. This patch made it show both the names to keep it identical with git's log command output.

[UNMERGED] Patch: commit: remove ignore whitespace option on --interactive (issue6042)
Tried to work on an issue in which interactive mode ignored whitespace changes on doing hg commit with such a config on diff. This was later fixed by @spectral in rHG66399f2e92aa.

[MERGED] Patch: revset: add expect to check the size of a set
This added a new function expectsize() to validate the size of a revset. This was suggested by the Mercurial community and was posted in WeShouldDoThat [2].

[MERGED] Patch: mq: disable qrecord during histedit (issue5981)
During a histedit, qrecord wasn't disabled initially, and this can lead to deadlock like situations. This patch prevented qrecord from working during histedit.

[MERGED] Patch: uncommit: don't allow dirty working copy with PATH (issue5977)
This patch made hg uncommit to require experimental.uncommitondirtydir config option when tried to uncommit on a dirty PATH similar to a dirty dir.

[MERGED] Patch: uncommit: add flag --allow-dirty-working-copy
Added --allow-dirty-working-copy flag as an alias for experimental.uncommitondirtydir config option.

[MERGED] Patch: unshelve: disable unshelve during merge (issue5123)

Doing hg unshelve during an uncommitted merge can delete the second parent of the dirstate. This patch made unshelve to abort during an uncommitted merge.

[WIP] Patch: mq: make unshelve to apply on modified mq patch (issue4318)
This patch tries to abort an unshelve on a modified mq patch by adding checks in rebase and mq.

[WIP] Patch: patch: stop aborting when add/rename/copy files on --interactive (issue5727)
This patch tries to abort on interactive mode when tried to add/rename/copy files.

I will be adding more patches based on my project after submitting the proposal.

The Project

Abstract

Title: Add functionality to store an unresolved merge-state

Description

Merge conflicts are part of every version control system structure. There can be situations in which some changes are necessary for a piece of software to function properly but, an unexpected merge conflict may lag the development workflow. The user may want to store a partially resolved merge-state if they have to fix an urgent bug. In the current scenario, they are only allowed to either fully resolve conflicts or abort the operation that led to conflicts by discarding the partially resolved state with conflicts. This project is about adding functionality to store an unresolved merge-state to help the user on such occasions. This lets the user do the required tasks at the moment in the same repository and get back later to the same merge-state and resume resolving conflicts. This project also adds functionality to let someone else do the conflict resolution for the user if they want by committing the conflicts and sharing it with other users. The feature of storing conflicts as commits is implemented in Pijul version control system.

Conflicts in Pijul

Pijul is a version control system inspired from a paper [3] on category theory of patches. Unlike Mercurial, it's exclusively based on patches rather than snapshots. I read more about Pijul from Joe Neeman's blog [4]. He says that unlike other VCSes, Pijul doesn't have conflicts instead, it has what he calls as graggles inspired by pushouts [5]. This internal representation of conflicts makes Pijul easier to handle conflicts. Interestingly, Pijul doesn't complain about conflicts(it's a known bug), and it shows a similar output on conflicts like other VCSes when they need you to resolve a merge conflict. This may make us think it's even worse than other VCSes. But, this is because Pijul is trying to reproduce a file which our operating system and editor can understand from the graggle it maintains internally. Pijul doesn't allow us to see this internal representation by default. But, we can see it using a debug command which dumps the entire graph into a file in graphviz' dot format [6], one for each branch of the repository, called debug_<branchname>. I have documented the steps Pijul follows to store conflict as commits after completing the merge here [7].

Imerge Extension

Imerge extension [8] lets the user split a merge into pieces. Unlike core hg merge command, this records the names of all files with conflicts and allows the user to merge these files and mark as resolved on a successful merge. After resolving all files, the merge is completed and the user can commit this merge. It supported hg imerge save <file> and hg imerge load <file> to store/restore merge-states by writing and reading data to/from an external file. However, it stopped working after the changeset 368a4ec603cc (merge: introduce mergestate) and was replaced by hg resolve command in 1.2 and later. Its code lies here [9]. This project might involve importing some code from this extension.

Overview

Why not shelve the conflicts to unshelve for later after getting things done?

Usually, when we have a dirty working directory and need to commit changes not related to the current uncommitted changes, we go for shelving the uncommitted changes and unshelving the changes after getting necessary things done. But, once we are in an unresolved merge-state, however it happened, we are in a merge-state. We have to resolve that merge-state before we are allowed to do anything else. Shelve will throw an error and abort here as the dirstate has two parents.

However, a possible workaround can be achieved by modifying the existing code of shelve to handle merge-states when explicitly mentioned. The abort on shelve._docreatecmd() and mergeutil.checkunresolved(ms) on localrepo.commit() can be removed only when the user wants to store conflicts. This will consider only the first

parent of the dirstate and shelve/unshelve changes accordingly. I have tried this and got this working using conflict markers enabled. But, this won't store any data about conflicts and the user cannot distinguish conflicted files from normal files. Grep using conflict marker syntax may work. But, that cannot be recommended and we don't want Mercurial to lose its intuitive behavior here.

Using `hg store-conflicts` to store an unresolved merge-state

A new command should be introduced to store the unresolved merge-state. Updating the shelve's behavior to handle conflicts will involve much refactoring. This project also consists of adding functionality to share conflicts among users. This won't be possible if we keep it under shelve by the UI similar to `hg shelve --with-conflicts` considering it will violate the default behavior of shelve.

Aborting the merge after conflicts had occurred and stored

This should change the repo to the state just before the merge and make the working directory clean. This action will be performed immediately after storing the unresolved merge-state. For the support of `merge` command, `hg merge --abort` is implemented by Pulkit in D1829**.** I can partially import code from that. For multi-step commands like `rebase` and `histedit` the behavior will slightly differ.

Using `hg restore-conflicts` to restore an unresolved merge-state

It will get back to the state which had conflicts when the user tried to merge before using the data stored using `hg store-conflicts` from the top of the user's new changesets.

Adding support to transmit stored conflicts between repositories

This should be the end-goal of this project. In the current scenario, the person who has conflicts in his repo is the only one who will be able to resolve it and commit changes to continue the workflow. If the repo has enormous conflicts, it will be a hard time for one person to fix the whole amount of conflicts. Things will even get worse if they are not aware of the code that other people have written. This functionality enables a user to share the conflicts they had committed with others and they can also help in resolving the conflicts having committed an unresolved state using `hg store-conflicts` by the user themselves.

Potential Problems

Even though we can see Pijul storing conflicts using its internal representation, it's not possible for Mercurial to store the conflicts in the same way as Pijul does as said by Pulkit, Jordi and Nathan. This is because Mercurial is based on snapshots whereas Pijul is based on patches. However, Mercurial does patches with MQ but, that's kind of unloved now says Jordi. After a merge that led to conflicts, right now, the merge-state is being stored at .hg/merge directory. It doesn't allow a user to work on the current working directory without resolving the conflicts and marking it as resolved by making it dirty and a dirstate having two equivalent parents. This project can also make an unresolved merge-state store as commits and allow it to share between users by pushing to the remote repo. Phase concepts should be wisely utilized here.

I will be working with storing data in changeset level using both internal and draft phases. Now, only shelve changesets are written in the internal phase. I read about obsolete changesets and thought of using that for hiding changesets. But, later realized that obsolescence markers are not meant as a generic hidden mechanism for local temporary nodes and should not be used that way. Hiding changesets using internal phase is recommended by the community. However, I might encounter compatibility issues. So, I have to store the changesets with secret phase when internal phases are not supported. On writing every read/write/traversal mechanism, I should consider performance issues and must be doing performance testing quite often.

For a hg user who is not aware of the mechanism that is happening internally, storing and restoring an unresolved merge-state may appear similar to what shelve and unshelve does(leaving the storing and sharing conflicts between users as commits part) in an ideal situation. But, shelve can only apply on uncommitted changes when the dirstate has only one parent. Now, when we are in a merge-state hg tries to pretend that p1() and p2() are both equally valid parents and how we can have normal, non-merge commits with null p1() and non-null p2(). Also, when tried to unshelve, unshelve assumes that first parent is our original parent and second is the temporary "fake" commit we are unshelving.

Goals

1. Store conflicts as commits by changing the default behavior of `localrepo.commit()` to handle the unresolved merge-state.
2. Introduce `hg store-conflicts` and `hg restore-conflicts` commands by providing support to store/restore conflicts caused by `hg merge` command.
3. Add support for multi-step commands like `rebase`, `histedit`.
4. Add `--with-conflicts` flag to `hg push` and `hg pull` commands to share stored conflicts among users.
5. Write documentation and tests for every code written.

The Plan

Adding a method for storage

Jordi gave an idea that if we want to store an unresolved merge-state as a commit, it would have to be some out-band thing like other .hg* things we store like .hglargefiles or .hgignore or .hgtags. I studied how these are stored and tracked by hg. Came across some wiki pages ([10], [11]) and realized that I have to consider some cases while storing it as commits. Other than the reason that hg storing commits as snapshots, one of the significant things is the append-only nature of Mercurial's version history. Since the Mercurial's HTTP protocol assumes that history is append-only when transmitting between repositories, no mutable properties can be stored as such. Also, by considering performance and scalability, append-only operations are only used. Pierre-Yves David suggested that for a commit with merge conflict, we need to guarantee they will not be part of the public history as the conflict has to be resolved before we change the phase to public. Also, he pointed out that keeping a "flag" at the changeset level(or manifest level or both) to point a file that has conflict would be nice. I thought of working with diffs to save storage size. Now, we have the whole file content stored in a merge-state. Pierre-Yves told that we need to worry about their content a bit less because of the non-public phase. Also, storing the file content in each file revlog ensures efficient storage. Phases in hg allow us to commit as draft and use it as a mutable commit. Other known phases are public and secret.

However, our priority is to store it locally and make things working. I am thinking to commit the unresolved merge-state and record the changeset with the extra mapping

{'internal': 'conflict'}. This will be a hidden changeset and won't be visible to the user directly in DAG by commands like hg log -G. After getting things working with internal phase, I will work on sharing changesets with users as draft phase. Now, localrepo.commit() doesn't allow us to commit an unresolved merge-state. The first step to store it as commit would be modifying this existing code to handle the unresolved merge-state as commits. localrepo.commit() will recognize the changeset as conflict from the extra mapping. Thus, this abort can be prevented here by adding a condition before mergeutil.checkunresolved(ms). After storing the merge-state as commit, the dirstate must be same as the state just before the merge. To get this to work, this function should be called from localrepo.commit() having the function definition written in merge.mergestate so that we can have the stored merge-state read even after committing the conflicts. This is really important not to lose the merge-state. I have the following code in mind to update the dirstate back to the old state:

```
ms = mergemod.mergestate.read(repo)
if ms.active():
    # there were conflicts
    node = ms.localctx.hex()
    repo.ui.status(_("storing conflicts, updating back to"
                     " %s\n") % node[:12])

    update(repo, node, branchmerge=False, force=True)
else:
    error.Abort(_("no known conflicts found to store"))
```

Deciding things need to be stored

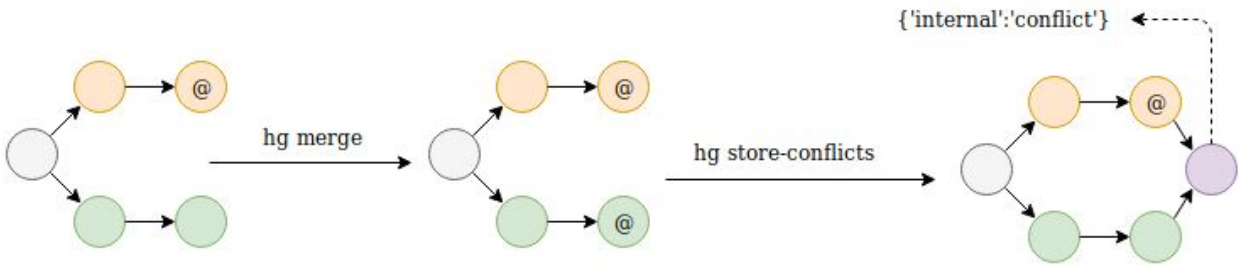
It's necessary to store the requisite amount of information for later restoration. The conflict changeset should store the revisions the user is merging, the list of files with conflicts and copies of the files the user has resolved so far. Imerge extension's fields of storage format can be used. These fields are p1, p2, conflict count, conflict filenames and resolved filenames. conflict filenames are tuples of localname, remoteorig and remotenew.

unresolved = [f for f in ms if ms[f] == 'u'] will give the list of unresolved files in the merge-state. After a merge that led to conflicts, *.orig files are generated in the repo which will have the content of the files just before the merge.

Scenario for storing conflicts

1. Transaction is open.
2. Unresolved merge-state is committed.
3. Commit is recorded as a conflict-changeset.
4. Apply hiding on this unresolved merge-state.
5. Transaction is committed.

Thus, we will have the following DAG after storing merge conflicts:

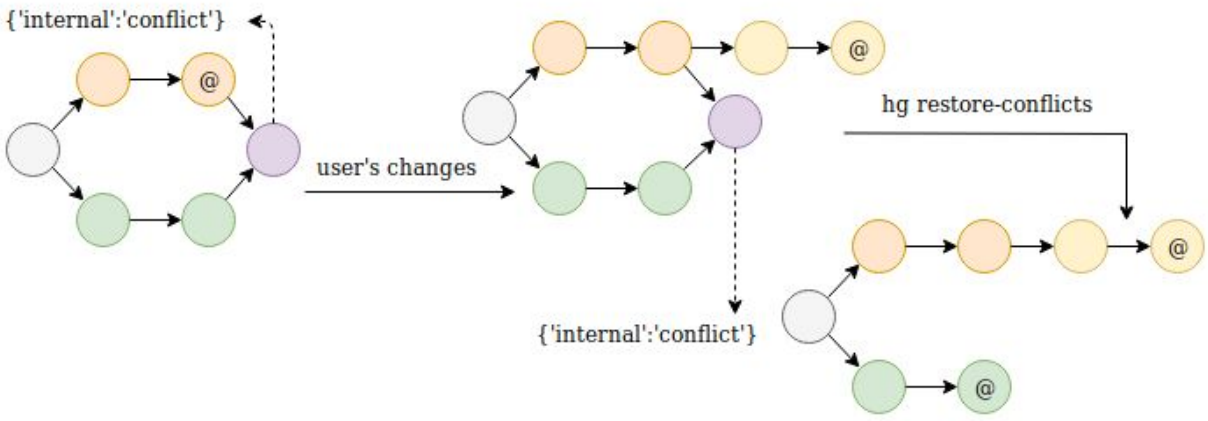


Here, @ indicates the current state of the repo.

I am planning to make hg store-conflicts and restore-conflicts commands perform transactions atomically. Even if the transaction gets aborted, there won't be a partially completed state. Either it's not started or fully completed. The current hg merge command's behavior is however non-atomic and was reported as a bug here [12]. The data will be stored in .hg/conflicts in a similar way shelve does.

Restoring unresolved merge-state from storage

The following DAG illustrates the reflection on it by this functionality:



Here, the user adds two new changesets to the repo after storing conflicts and tries to restore conflicts. This two branches/bookmarks/features will be merged now to result again in conflicts but, by restoring the partially resolved part of the user. Partially resolved merge-state can be restored by rebasing the conflict changeset on the top of this merge. This will involve changing the default behavior of rebase to work on top of merge-states but, only when the source is a conflict changeset.

A discussion on supporting histedit on merge can be found here [13]. As said by Augie, it's disabled because the current UI has no way to express non-linearity. If we have to perform rebase on merge-states, the UI should support non-linearity. On certain constraints, this can be implemented.

merge.graft has keepparent option to keep the second parent when it rewrites the dirstate on merge commits since this [14] patch has landed. A similar thing can be done with rebase also to not lose the second parent.

Scenario for restoring conflicts

1. Transaction is open.
2. Merge old branches.
3. Uncommitted merge changes with conflicts are put in a temporary commit.
4. This is recorded as a conflict-changeset.
5. Access the stored conflict-changeset which contains changes in the old partially resolved merge-state.
6. hg rebase -r <old-conflict> -d <tmp-conflict>
7. Restore working copy parents.
8. Hide <tmp-conflict> and <rebased>
9. Close transaction.

Adding support for merge command

After storing an unresolved merge-state, support for merge command should be implemented first. For the user who gets into conflicts when tried to merge, the error message should also be suggesting the functionality to save conflicts and return back to a clean working directory. The default behavior of storing/restoring unresolved

merge-states will be implemented by considering merge command. So, most of the work will be already done after writing storing/restoring functionalities.

Adding support for multi-step commands

For multi-step commands like rebase, histedit, we need to store their state files also. We can preserve the same format it uses now as implemented in state.py. Now, all the data for the state is stored in the form of key-value pairs in a dictionary using cbor to serialize and deserialize data while writing and reading from disk. The same behavior should be enough for adding support for multi-step commands to store their unresolved merge-state. This data about the state can be committed along with the merge-state and stored in .hg/conflicts.

Adding support for sharing stored conflicts among users

Commits with internal phase cannot be transmitted across repositories. After getting all things working with internal phase for commits with conflicts, their storage method can be rewritten with draft phase because, for this functionality, the conflicts must be stored as commits with their phase not being public but, can be still shared. We can store it with draft as phase and with the extra mapping {'internal': 'conflict'}. By default, changesets pushed and pulled are public. So, a changeset with draft phase living in the local repo can change its phase to public when pushed to the remote repo. This can be prevented by preserving the phase as draft by identifying the extra mapping for conflict in the changeset. Currently, hg has a flag at the manifest level and extra mapping in the changeset level.

From the UI point of view, I will be adding a --with-conflicts flag to hg pull and hg push commands to transmit conflict commits among repositories. hg push will push the conflict commit as draft and hg pull will pull it as draft, then make it as internal phase for the hg restore-conflicts command to work.

Documenting changes

Since there will be enough changes to the UI and the general behavior caused by the new hg store-conflicts and hg restore-conflicts commands, it's necessary to document

every piece of code that I will be writing. I will keep on updating documentation as I progress writing code. After getting the documentation within the code done with project implementation, I'll update the wiki [15] by creating a page on `store-conflicts` and edit other pages that might change their behavior.

Writing tests

Since there are a lot of corner cases involved with merge-states and the project consists of the modification of existing UI, writing tests is an inevitable part of this project. This will help the community to track the bugs that we might get encounter on adding more functionalities. I have read Mercurial's wiki for writing tests [16] and have already sent many patches based on regression tests so that I won't face any problem writing them.

Timeline (Tentative)

Community Bonding period (May 6 - May 27)

I've been going through the codebase while fixing the existing issues and planning this project. However, I might not be covered every part of it for implementing this project. The community bonding period will be utilized wisely for understanding the codebase enough especially for shelve and merge code. I'll investigate more about dealing with internal phases for hiding changesets. I will be in touch with mentors and will be seeking help from them whenever needed.

Week 1 - Week 2 (May 28 - Jun 10)

I will investigate more about merge-state and work on storing its unresolved state as commits. This will involve changing the default behavior of `localrepo.commit()` to handle unresolved merge-state. I will start building the UI for `hg store-conflicts` from this time.

Week 3 - Week 4 (Jun 11 - Jun 24)

This period can be utilized to add the functionality of restoring the unresolved merge-state stored. This will also involve developing helper functions to make merge/rebase handle unresolved merge-states.

Phase 1 Evaluation

Week 5 - Week 6 (Jun 25 - Jul 8)

Hopefully, conflicts can be stored and restored by this time. I will work on `merge` command to add support to this functionality. This will also involve making minor changes to the merge code, changes in the commit messages thrown on conflicts, etc.

Week 7 - Week 8 (Jul 9 - Jul 22)

I will start by analysing the working of multi-step commands including `rebase` and `histedit` performing an internal merge to give the conflicted state. I will work on implementing support for these commands to store their unresolved merge-state with their state files. This will also consist of adding their respective behavior on `--abort` after storing conflicts and state files.

Phase 2 Evaluation

Week 9 - Week 10 (Jul 23 - Aug 5)

I will be working on exchanging conflicts between users from this time. This is the end-goal of the project. However, this project gives more emphasis on storing the conflict changeset locally. This will involve working with different phases and changing the behavior of `hg pull` and `hg push` commands to handle conflict changesets.

Week 11 - Week 12 (Aug 6 - Aug 19)

This period will solely be given for documentation of the `hg store-conflicts`, `restore-conflicts` and related changes happened to other parts of the codebase. I'll add working examples to a dedicated file and also update the Mercurial wiki.

Week 13 (Aug 20 - Aug 26)

I will keep this as a buffer period if anything goes wrong with this timeline. If everything goes well, I will have achieved my target until now. Also, there are some bug fixes which are not related to this project that I sent. I will continue fixing them in this period.

Post GSoC

Contributions to Mercurial helped me to learn a lot of things. People like Yuya, Martin, et al. posting nit-picks always motivated me to improve my pythonic way of writing code and look at efficiency issues quite often. I was really afraid to use git when I used it to edit history and was impressed by hg history editing commands and its intuitive nature. I'll introduce my friends in college to Mercurial via an Open Source Society [17] in my college that I'm part of by conducting workshops. I will continue contributing to Mercurial even after GSoC. I will be fixing bugs that might have occurred due to the code that I've written and try to work on ideas posted in WeShouldDoThat page. Also, I quite liked HgWebPlan [18]. I am planning to write a REST API which will make repository mutations possible in hg. I will be happy to stay active in IRC and would love to work on community prioritized issues.

How do the mentors keep track of my work?

I'll be posting the patches to the Phabricator [19] for reviews so that the mentors can have a better idea of the work I'm doing. If necessary, I can also send RFC patches to the mailing list [20] for reviews. I'll also be writing weekly blogs where I'll put up my progress and experiences. Besides this, I regularly stay active on IRC where I can interact with fellow developers in Mercurial.

Other Commitments

I don't have any other commitments per se as I will be in my college vacations during most of the GSoC period. Even after opening my college in mid-July, I don't have any exams scheduled in the GSoC period (reference: institute academic calendar [21][22]). So, I won't face any problem with managing my tasks. I am only applying for Mercurial for GSoC 2019 and I will easily be able to give around 40 hours a week to my GSoC project in Mercurial.