**[Git's rejected push error](https://blog.plover.com/prog/git-ff-error.html)**

On Saturday I posted [an article explaining how remote branches and remote-tracking branches work in Git](https://blog.plover.com/prog/git-remote-branches.html). That article is a prerequisite for this one. But here's the quick summary:

When dealing with a branch (say, master) copied from a remote repository (say, remote), there are *three* branches one must consider:

1. The copy of master in the local repository
2. The copy of master in the remote repository
3. The local branch origin/master that records the last known position of the remote branch

Branch 3 is known as a “remote-tracking branch”. This is because it tracks the remote branch, not because it is itself a remote branch. Actually it is a local copy of the remote branch. From now on I will just call it a “tracking branch”.

The git-fetch command (green) copies branch (2) to (3).

The git-push command (red) copies branch (1) to (2), and incidentally updates (3) to match the new (2).

The diagram at right summarizes this.

We will consider the following typical workflow:

1. Fetch the remote master branch and check it out.
2. Do some work and commit it on the local master.
3. Push the new work back to the remote.

But step 3 fails, saying something like:

! [rejected] master -> master (fetch first)

error: failed to push some refs to '../remote/'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

In older versions of Git the hint was a little shorter:

hint: Updates were rejected because the tip of your current branch is behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

Everyone at some point gets one of these messages, and in my experience it is one of the most confusing and distressing things for beginners. It cannot be avoided, worked around, or postponed; it must be understood and dealt with.

Not everyone gets a clear explanation. (Reading it over, the actual message seems reasonably clear, but I know many people find it long and frighting and ignore it. It is tough in cases like this to decide how to trade off making the message shorter (and perhaps thereby harder to understand) or longer (and frightening people away). There may be no good solution. But here we are, and I am going to try to explain it myself, with pictures.)

In a large project, the remote branch is always moving, as other people add to it, and they do this without your knowing about it. Immediately after you do the fetch in step 1 above, the tracking branch origin/master reflects the state of the remote branch. Ten seconds later, it may not; someone else may have come along and put some more commits on the remote branch in the interval. This is a fundamental reality that new Git users must internalize.

**Typical workflow**

We were trying to do this:

1. Fetch the remote master branch and check it out.
2. Do some work and commit it on the local master.
3. Push the new work back to the remote.

and the failure occurred in step 3. Let's look at what each of these operations actually does.

**1. Fetch the remote master branch and check it out.**

git fetch origin master  
git checkout master

The black circles at the top represent some commits that we want to fetch from the remote repository. The fetch copies them to the local repository, and the tracking branch origin/master points to the local copy. Then we check out master and the local branch master also points to the local copy.

Branch names like master or origin/master are called “refs”. At this moment all three refs refer to the same commit (although there are separate copies in the two repositories) and the three branches have identical contents.

**2. Do some work and commit it on the local master.**

*edit…*  
git add …  
git commit …  
…

The blue dots on the local master branch are your new commits. This happens entirely inside your local repository and doesn't involve the remote one at all.

But unbeknownst to you, something else is happening where you can't see it. Your collaborators or co-workers are doing their own work in their own repositories, and some of them have published this work to the remote repository. These commits are represented by the red dots in the remote repository. They are there, but you don't know it yet because you haven't looked at the remote repository since they appeared.

**3. Push the new work back to the remote.**

git push origin master

Here we are trying to push our local master, which means that we are asking the remote repo to overwrite its master with our local one. If the remote repo agreed to this, the red commits would be lost (possibly forever!) and would be completely replaced by the blue commits. The error message that is the subject of this article is Git quite properly refusing to fulfill your request:

! [rejected] master -> master (fetch first)

error: failed to push some refs to '../remote/'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

Let's read through that slowly:

Updates were rejected because the remote contains work that you do not have locally.

This refers specifically to the red commits.

This is usually caused by another repository pushing to the same ref.

In this case, the other repository is your co-worker's repo, not shown in the diagram. They pushed to the same ref (master) before you did.

You may want to first integrate the remote changes (e.g., 'git pull ...') before pushing again.

This is a little vague. There are many ways one could conceivably “integrate the remote changes” and not all of them will solve the problem.

One alternative (which does *not* integrate the changes) is to use git push -f. The -f is for “force”, and instructs the remote repository that you really do want to discard the red commits in favor of the blue ones. Depending on who owns it and how it is configured, the remote repository may agree to this and discard the red commits, or it may refuse. (And if it does agree, the coworker whose commits you just destroyed may try to feed you poisoned lemonade, so use -f with caution.)

See the ['Note about fast-forwards' in 'git push --help'](https://git-scm.com/docs/git-push#_note_about_fast_forwards) for details.

To “fast-forward” the remote ref means that your local branch is a direct forward extension of the remote branch, containing everything that the remote branch does, in exactly the same order. If this is the case, overwriting the remote branch with the local branch is perfectly safe. Nothing will be lost or changed, because the local branch contains everything the remote branch already had. The only change will be the addition of new commits at the end.

There are several ways to construct such a local branch, and choosing between them depends on many factors including personal preference, your familiarity with the Git tool set, and the repository owner's policies. Discussing all of this is outside the scope of the article, so I'll just use one as an example: We are going to *rebase* the blue commits onto the red ones.

**4. Refresh the tracking branch.**

git fetch origin master

The first thing to do is to copy the red commits into the local repo; we haven't even seen them yet. We do that as before, with git-fetch. This updates the tracking branch with a copy of the remote branch just as it did in step 1.

If instead of git fetch origin master we did git pull --rebase origin master, Git would do exactly the same fetch, and then automatically do a rebase as described in the next section. If we did git pull origin master without --rebase, it would do exactly the same fetch, and then instead of a rebase it would do a merge, which I am not planning to describe. The point to remember is that git pull is just a convenient way to combine the commands of this section and the next one, nothing more.

**5. Rewrite the local changes.**

git rebase origin/master

Now is the moment when we “integrate the remote changes” with our own changes. One way to do this is git rebase origin/master. This tells Git to try to construct new commits that are just like the blue ones, but instead of starting from the last black commit, the will start from the last red one. ([For more details about how this works, see my talk slides about it.](http://perl.plover.com/yak/git-rebase-wtf/)) There are many alternatives here to rebase, some quite elaborate, but that is a subject for another article, or several other articles.

If none of the files modified in the blue commits have also been modified in any of the red commits, there is no issue and everything proceeds automatically. And if some of the same files are modified, but only in non-overlapping portions, Git can automatically combine them. But if some of the files are modified in incompatible ways, the rebase process will stop in the middle and asks how to proceed, which is another subject for another article. This article will suppose that the rebase completed automatically. In this case the blue commits have been “rebased onto” the red commits, as in the diagram at right.

The diagram is a bit misleading here: it looks as though those black and red commits appear in two places in the local repository, once on the local master branch and once on the tracking branch. They don't. The two branches share those commits, which are stored only once.

Notice that the command is git rebase origin/master. This is different in form from git fetch origin master or git push origin master. Why a slash instead of a space? Because with git-fetch or git-push, we tell it the name of the remote repo, origin, and the name of the remote branch we want to fetch or push, master. But git-rebase operates locally and has no use for the name of a remote repo. Instead, we give it the name of the branch onto which we want to rebase the new commits. In this case, the target branch is the tracking branch origin/master.

**6. Try the push again.**

git push origin master

We try the exact same git push origin master that failed in step 3, and this time it succeeds, because this time the operation is a “fast-forward”. Before, our blue commits would have replaced the red commits. But our rewritten local branch does not have that problem: it includes the red commits in exactly the same places as they are already on the remote branch. When the remote repository replaces its master with the one we are pushing, it loses nothing, because the red commits are identical. All it needs to do is to add the blue commits onto the end and then move its master ref forward to point to the last blue commit instead of to the last red commit. This is a “fast-forward”.

At this point, the push is successful, and the git-push command also updates the tracking branch to reflect that the remote branch has moved forward. I did not show this in the illustration.

But wait, what if someone else had added yet more commits to the remote master while we were executing steps 4 and 5? Wouldn't our new push attempt fail just like the first one did? Yes, absolutely! We would have to repeat steps 4 and 5 and try a third time. It is possible, in principle, to be completely prevented from pushing commits to a remote repo because it is always changing so quickly that you never get caught up on its current state. Repeated push failures of this type are sign that the project is large enough that repository's owner needs to set up a more structured code release mechanism than “everyone lands stuff on master whenever they feel like it”.

An earlier draft of this article ended at this point with “That is all I have to say about this.” Ha!

**Unavoidable problems**

Everyone suffers through this issue at some point or another. It is tempting to wonder if Git couldn't somehow make it easier for people to deal with. I think the answer is no. Git has multiple, distributed repositories. To abandon that feature would be to go back to the dark ages of galley slaves, smallpox, and SVN. But if you have multiple distributed anythings, you must face the issue of how to synchronize them. This is intrinsic to distributed systems: two components receive different updates at the same time, and how do you reconcile them?

For [reasons I have discussed before](https://blog.plover.com/prog/git-merge.html), it does not appear possible to automate the reconciliation in every case in a source code control system, because sometimes the reconciliation may require going over to a co-worker's desk and arguing for two hours, then calling in three managers and the CTO and making a strategic decision which then has to be approved by a representative of the legal department. The VCS is not going to do this for you.

I'm going to digress a bit and then come back to the main point. Twenty-five years ago I taught an introductory programming class in C. The previous curriculum had tried hard to defer pointers to the middle of the semester, as K&R does (chapter 7, I think). I decided this was a mistake. Pointers are everywhere in C and without them you can't call scanf or pass an array to a function (or access the command-line arguments or operate on strings or use most of the standard library or return anything that isn't a number…). [Looking back a few years later I wrote](https://groups.google.com/d/msg/comp.lang.perl.misc/z0rObxz4CLw/1xHbGEuni7UJ):

Pointers are an essential part of [C's] solution to the data hiding problem, which is an essential issue. Therefore, they cannot be avoided, and in fact should be addressed as soon as possible. … They presented themselves in the earliest parts of the material not out of perversity, but because they were central to the topic.

[I developed a new curriculum that began treating pointers early on](https://github.com/mjdominus/intro-c-programming), as early as possible, and which then came back to them repeatedly, each time elaborating on the idea. This was a big success. I am certain that it is the right way to do it.

(And I've been intending since 2006 to write an article about K&R's crappy discussion of pointers and how its deficiencies and omissions have been replicated down the years by generation after generation of C programmers.)

I think there's an important pedagogical principle here. A good teacher makes the subject as simple as possible, but no simpler. Many difficult issues, perhaps most, can be ignored, postponed, hidden, prevaricated, fudged, glossed over, or even solved. But some must be met head-on and dealt with, and for these I think the sooner they are met and dealt with, the better.

Push conflicts in Git, like pointers in C, are not minor or peripheral; they are an intrinsic and central issue. Almost everyone is going to run into push conflicts, not eventually, but right away. They are going to be completely stuck until they have dealt with it, so they had better be prepared to deal with it right away.

If I were to write a book about Git, this discussion would be in chapter 2. Dealing with merge conflicts would be in chapter 3. All the other stuff could wait.

That is all I have to say about this. Thank you for your kind attention, and thanks to Sumana Harihareswara and AJ Jordan for inspiration.