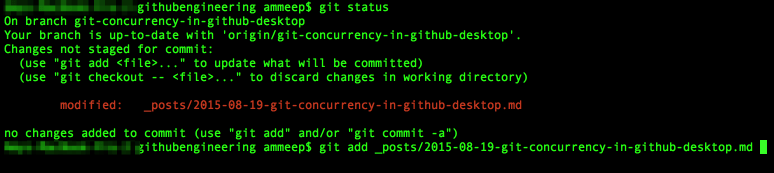
**SOFTWARE ENGINEERING 2**

**GIT**

**CONCURENCY MANAGEMENT**

In [GitHub Desktop](http://desktop.github.com/), many background threads will  
read or write to the same Git repository, at the same time.

However, git is typically not used in a concurrent fashion. When using git via  
the command line, operations are executed in a sequential manner. Read or write  
operations are performed against git, independently of each other.  
  
During the build of GitHub Desktop, we discovered executing git commands serially  
was a one-way ticket to an unresponsive app. For example, waiting  
to load diffs until after we’ve counted the number of commits in the history  
would result in a slow and unresponsive application.

To maintain correctness and a responsive user interface, we needed a  
solution to concurrency control.

**Git, libgit2 and concurrency**

GitHub Desktop has two methods of interacting with a git repository.

* Calling into C implementations of the Git core methods via [libgit2](https://github.com/libgit2/libgit2)
* Shelling out to the git command line interface

We would like to use libgit2 for all of our git operations because it is faster and easier to program with. Unfortunately it is not yet a complete implementation, so we use the CLI to fill in the missing functionality.  
This poses an interesting problem, in that both git and libgit2 have different  
approaches to concurrency control.

Git implements a pessimistic approach to concurrency control. Lock files are used to prevent concurrent access to the underlying git objects on disk. When performing an operation against a git object, git will create a \*.lock  
file inside the .git directory. This signals that the \* object is locked for  
use. Further operations are prevented until the lock is released and the \*.lock file is deleted.

By contrast libgit2 cannot guarantee objects can safely be shared between threads. Mutable data structures in libgit2 are not thread safe, and operations must be performed carefully. The libgit2 API allows you to compose granular operations together, and granular locking would come at a performance cost. Libgit2 data structures are rarely used in isolation, and concurrency control should be implemented at the level over a collection of fine grained operations or a single unit of work.

**A new concurrency model**

GitHub Desktop ships as a native application on both Mac and Windows. The Mac app is implemented in Objective-C, while the Windows app is implemented in C#. Both platforms are implemented in a reactive style, using  
[Microsoft’s Reactive Extensions (Rx)](https://github.com/Reactive-Extensions)  
and [our own ReactiveCocoa (RAC)](https://github.com/ReactiveCocoa/ReactiveCocoa).  
This allows the composition of background tasks, such as executing git operations.  
All git operations are executed asynchronously and across thread boundaries.

To ensure GitHub Desktop executed git operations in a safe, and yet performant manner, we needed a new concurrency model that enabled us to:

* Organize work at the level of asynchronous Observables (Rx) and Signals (RAC)  
  instead of synchronous blocks of code.
* Perform most operations concurrently.
* Retain the ability to perform destructive operations serially and exclusively, as required by Git or libgit2

**Concurrent and exclusive locks**

Each high level operation GitHub Desktop performs can be thought of as a unit of work.  
A single unit of work can be made up of many fine-grained operations. Our units  
of work can be categorized as either:

* Concurrent operations
* Exclusive operations

Concurrent and exclusive operations don’t always have  
a 1:1 relationship with reading and writing to the underlying repository.  
For example, it is safe to write [Git refs](https://git-scm.com/book/en/v2/Git-Internals-Git-References) concurrently with other work,  
because a ref update is atomic. On the other hand, some read operations may  
update caches in an unsafe way, and so those need to be performed exclusively.

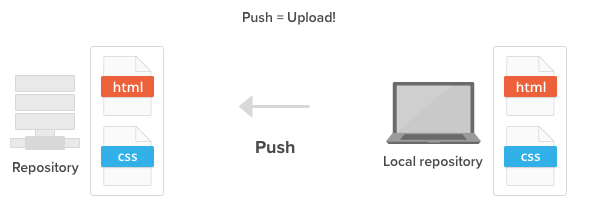
GitHub Desktop uses an AsyncReaderWriterLock as a queue, upon  
which concurrent operations can either be run exclusively or in parallel. Exclusive operations behave  
like a barrier, waiting for previously-enqueued work to complete before beginning,  
and themselves finishing before any further work starts.

# **GIT SYNCHRONIZATION**

Remote repositories allow us to share our changes with other members of the team. They can be on a private server, on a different computer than yours, or hosted using a service like Backlog. Wherever yours is hosted, you'll need to be able to sync your local repository with the remote repository frequently. You'll do this using three commands: git push, git pull, and git merge.

## Git push

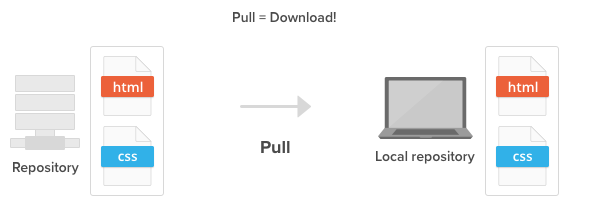
In order to start sharing changes with others, you have to push them to a remote repository using the "push" command. This will cause the remote repository to update and synchronize with your local repository.

Push your local changes to a remote repository.

## Git pull

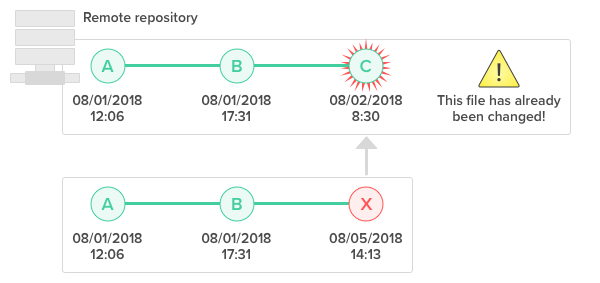
Whenever somebody pushes their changes to the shared remote repository, your local repository becomes out of date. To re-synchronize your local repository with the newly updated remote repository, simply run the git pull operation.

When the pull is executed, the latest revision history will download from the remote repository and import to your local repository.

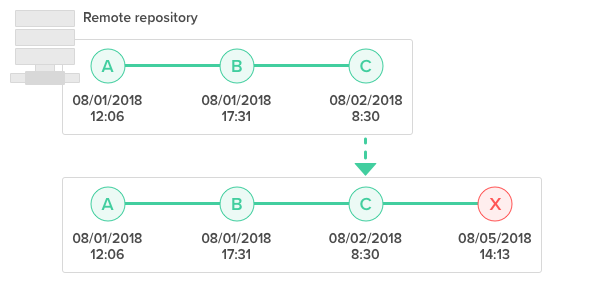
Pull changes from a remote repository to your local repository.

## Git merge

Your push to the remote repository will be rejected if your local repository is out of date, possibly because there are some updates on the remote repository that you do not have locally yet.

You are unable to push to the remote repository if your local repo is out of date.

If that is the case, you'll have to use the git merge command to grab the latest change from the remote repository before you are allowed to push. Git enforces this to ensure that changes made by other members get retained in the history.

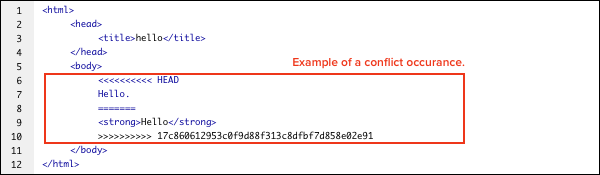
You must merge the latest changes before pushing.

During a "merge", Git will attempt to automatically apply those history changes and merge them with the current branch. However, if there is a conflict in changes, Git will throw an error prompting you to resolve the conflict manually.

## Resolve merge conflicts

When merging two branches, you may come across a conflict that needs resolving before you can properly complete the merge. For example, when two or more members make changes on the same part of a file in the two different branches (remote and local branches in this case), Git will not be able to automatically merge them.

When this happens, Git will add some standard conflict-resolution markers to the conflicting file. The markers help you figure out which sections of the file need to be manually resolved.

Example of a conflict occurrence.

In our example, everything above "=====" is your local content, and everything below it comes from the remote branch.

You must resolve the conflicting parts as shown below before you can proceed with creating a merge commit.

