Upgrade strategy report

Igalia
for Ecosia

A Coruña (Spain)
20th August 2020

Contents

1 Introduction

1.1 About the document

This document contains the results and recommendations from Igalia, for the continuous integration procedures of Ecosia Chromium downstream fork. The project was developed in August, 2020.

Maintaining a Chromium downstream tree is a challenging effort. The repository is huge, and the effort of hundreds of contributors makes it move fast. This document tries to give some ideas, from Igalia previous experience maintaining other downstream forks of Chromium.

It should be used as a reference, so Ecosia can take the hints or tricks that work better in their environment and constraints.

1.2 Credits

Most of the contents for this document have been written by:

- Jose Dapena <jdapena@igalia.com>
- Henrique Ferreiro <hferreiro@igalia.com>

Specific help was provided by:

- Alexander Dunaev <adunaev@igalia.com>: Google background services removal.
- Pablo Saavedra <psaavedra@igalia.com>: continuous integration services recommendations.

1.3 Branches terminology

1.3.1 Upstream

Branches, tags, code, that comes strictly from the upstream project.

1.3.2 Downstream

What becomes the result of applying the Ecosia project changes on top of the upstream (the combination of upstream and Ecosia changes).

1.3.3 Baseline

For a downstream content (tag or branch), the commit in the branch that it is based on that contains only upstream contents.

1.4 Types of delta

1.4.1 Total delta

This is the full difference between the baseline we consider and all the downstream changes.

When we do not specify the type of delta, we are referring to total delta.

1.4.2 Modifying delta

The set of changes downstream that are modifying, removing or renaming files that exist in the baseline.

1.4.3 Independent delta

The set of changes downstream that are not modifying delta (so they are new files only existing downstream).

1.5 Upgrade models

1.5.1 Current model

Ecosia is right now deploying a model that takes months to release. Even when several improvements have been done in the procedure, repository structure and codebase, it is not expected to be enough for providing users with frequent upgrades.

1.5.2 Fast upgrade

Fast upgrade is the initial goal of the procedure improvements proposed. The idea is being able to reduce the time to upgrade, so it takes less than 6 weeks (the upstream release frequency).

Validation needs to be partially based in automation, combined with manual testing. Upgrade should takes less than 6 weeks, and ideally less than 2 weeks from upstream first stable release of a major version.

1.5.3 Continuous upgrade

Continuous upgrade is the ability to track in hours or a few days the upstream releases.

It is expected to be strongly based in automation, so verifying a build is functional takes less than a day.

Most of the releases should happen in less than a day from Chromium upstream release.

2 Analysis of delta

In this chapter we review the delta with upstream, that is a basic input for frequent upgrade.

It is based on the status of Ecosia tree in August 3rd, 2020.

2.1 Raw delta

Total delta in GIT to the baseline.

```
$ git diff 1956716893e9e91bb316d4fa59a991458b956ff0 --shortstat
 654 files changed, 24157 insertions(+), 16540 deletions(-)

$ git diff 1956716893e9e91bb316d4fa59a991458b956ff0 --dirstat
   3.6% chrome/android/java/res_ecosia/drawable-hdpi/
   6.3% chrome/android/java/res_ecosia/drawable-xhdpi/
  13.0% chrome/android/java/res_ecosia/drawable-xxhdpi/
  21.7% chrome/android/java/res_ecosia/drawable-xxxhdpi/
   3.0% chrome/android/java/res_ecosia/
   6.3% chrome/android/java/strings/translations/
   6.9% components/strings/
   3.5% scripts/ecosia_res/assets/drawable-xhdpi/
   7.0% scripts/ecosia_res/assets/drawable-xxhdpi/
  12.0% scripts/ecosia_res/assets/drawable-xxxhdpi/
   3.8% scripts/ecosia_res/assets/
   9.5% third_party/
```

2.2 Modified files delta

Delta of only modified files

```
$ git diff --diff-filter=M 1956716893e9e91bb316d4fa59a991458b956ff0
  --shortstat
 308 files changed, 15761 insertions(+), 16490 deletions(-)

$ git diff --diff-filter=M 1956716893e9e91bb316d4fa59a991458b956ff0
  --dirstat
  45.1% chrome/android/java/strings/translations/
   3.0% chrome/android/
  49.7% components/strings/
```

So, a huge part of the conflicting delta is actually translation files:

```
$ git diff --diff-filter=M 1956716893e9e91bb316d4fa59a991458b956ff0
  --shortstat chrome/android/java/strings/translations/
 52 files changed, 7540 insertions(+), 7540 deletions(-)
$ git diff --diff-filter=M 1956716893e9e91bb316d4fa59a991458b956ff0
  --shortstat components/strings/
 156 files changed, 6997 insertions(+), 6997 deletions(-)
```

That tells us:

- Files changed: 308 - 52 - 156 = 100
- Insertions: 15761 - 7540 - 6997 = 1224
- Deletions: 16940 - 7540 - 6997 = 2403

We see in `--diff-filter=M` that the search engine prepopulated data is +2-1269.

So basically, the code-specific delta is pretty small, and mostly Java code in the Chrome browser.

```
$ git diff --shortstat 1956716893.. chrome/android/java/src/org/chromium/
58 files changed, 173 insertions(+), 251 deletions(-)
```

2.3 Kind of modified files

Pattern	Files changed	Total files	Lines changed	Total lines
XTB translation files	208	208	29074	29074
C++ files	4	4	1280	1280
GRD files	4	4	764	764
Java files	65	110	458	3847
Markdown	1	4	424	530
XML resources	15	44	100	1471
GN / gclient / DEPS files	4	14	114	326
PNG images	0	157	-	-
SH scripts	0	5	0	111
Python scripts	0	4	0	231
AAR/Info/JAR	0	71	0	166
TOTAL	308	654	32251	40697

2.4 Conclusion

Conflicting delta is very small, with two main kind of types:

- Translations: these can be problematic as they are rewritten. And the way to resolve the conflict implies obtaining translations. Adding stubs provisionally should simplify that. I.e. a translation prefixed with `TODO(john.doe)`. Then later translations can be added.
- Java files: changes in existing files are very small.

Main concern here is about the non conflicting change:

- New changes in Java code that add Chromium branding that should not be seen in Ecosia browser. A solution could be intentionally renaming the chromium branding so each time new compiled code uses it, compilation breaks.
- New strings added somewhere that use Chromium branding. A possible fix for this is adding a compilation step that filters generated translations to detect if there is Chromium brand name, and make compilation fail.

3 Repository policy

3.1 Current status

Currently, the Ecosia repository contains:

- Upstream chromium till m69 Ecosia baseline.
- Patches applied on top of different upstream baselines (up to m69).
- An orphaned commit with the snapshot of upstream chromium m78, and all the patches of Ecosia on top of that.

3.2 Proposal 1: new snapshots-only repository

This repository would contain only snapshot commits as needed, keeping a structure that respects upstream branches structure. At least the branch point commit of each used major release should be used, then new snapshots would be applied for getting new point releases.

3.2.1 Branches and tags structure

The `upstream` branch would contain the snapshots of the branch points only:

- `upstream`
  - Initial commit (empty).
  - Snapshot of m78 baseline (tagged as `upstream-78.0.3904.0`).
  - Snapshot of m85 baseline (tagged as `upstream-85.0.4183.0`).

Then, for each upstream branch point, we'd create a branch, and apply different snapshots for the baselines we are going to use. I.e. for m78:

- `upstream-78` (branch point would be the tagged commit `upstream-78.0.3904.0`)
  - Snapshot of `78.0.3904.96` (tagged as `upstream-78.0.3904.96`)
  - Snapshot of `78.0.3904.126`, ...

Ecosia work should happen always on top of one of these tags, and the branch name should refer to that.

- `ecosia-78.0.3904.96` (branch point would be `upstream-78.0.3904.96`)
  - Commit 1
  - Commit 2

  - …
  - Commit N (tagged release `78.0.3904.96.ecosia.R`)

I.e. let's assume first snapshot will be the baseline for `78.0.3904.0` (the branch point for Ecosia 78 current work). The `R` part will be the Ecosia version and will be incrementally added independently of the upstream release it is based on.

3.2.2 Git commands

It should be possible, for any Ecosia release, to determine the diff to the baseline:

```
$ git diff upstream-78.0.3904.96..ecosia-78.0.3904.96
```

Or rebasing to a new point release, so all the conflicts are resolved:

```
$ git checkout ecosia-78.0.3904.96      # Latest working branch
$ git checkout -b ecosia-78.0.3904.126  # New working branch
$ git rebase --onto upstream-78.0.3904.126 upstream-78.0.3904.96 ecosia-
78.0.3904.126
```

The rebase command is going to apply the list of changes between `upstream-78.0.3904.96` and `ecosia-78.0.3904.126` on top of `upstream-78.0.3904.126`, and that will be the `HEAD` of `ecosia-78.0.3904.126`.

Or rebase to a new major release:

```
git checkout ecosia-78.0.3904.126   # Latest working branch
git checkout -b ecosia-85.0.4183.0  # New working branch name
git rebase --onto upstream-85.0.4183.0 upstream-78.0.3904.126 ecosia-
85.0.4183.0
```

Though, before rebasing to a new major release, or periodically in point release rebases, it would be interesting to create squashed branches (see more about this later). In that case, the latest working branch used would be the squashing branch. I.e. for a minor release:

```
$ git checkout ecosia-78.0.3904.96-squashed      # Latest squashed branch
$ git checkout -b ecosia-78.0.3904.126           # New working branch
$ git rebase --onto upstream-78.0.3904.126 upstream-78.0.3904.96 ecosia-
78.0.3904.126
```

3.2.3 How to create a snapshot

A script would be created that prepare the upstream snapshot.

A possibility would be having an additional remote for upstream in the machine creating the snapshots:

- `git diff --binary` of upstream changes
- `git checkout` old baseline
- `git checkout -b` new baseline
- `git apply` upstream changes patch
- `git add -A`
- `git commit`

Something like this would mostly work (though specific parameters need to be found for avoiding modifying end of line characters from upstream).

3.3 Proposal 2: use upstream

It would be keeping a downstream copy of upstream repository, up to date with the branches we need… But applying the same criteria of branch names. It needs adaptations, as upstream does not provide branches, and tags have a different criteria.

Apart from that, the rebasing strategy would be implemented in a similar way.

3.4 Mitigating the impact of the repository size

Even if using the upstream repository, there are some actions that can be performed to avoid huge downloads and reduce disk space usage.

3.4.1 Using shallow clones

To avoid downloading all history, `git` supports shallow clones, that is, cloning the repository history up to some depth:

```
$ git clone --depth=N
```

`gclient` also supports using `--depth=1` in all subprojects by using `gclient sync --no-history`.

When access to older history is needed, `git fetch` can be called with the same `--depth` parameter. Ultimately, if all history is required, use `git fetch --unshallow`.

3.4.2 Limiting Chromium dependencies

`gclient` manages dependencies for multiple OSs and configurations. Besides making sure that only the needed OSs are used, one way of reducing the downloaded data is using some options to omit certain features. The following `.gclient` snippet will do that for a recent version of Chromium:

```
$ cat .gclient
solutions = [
  {
    ...
    "custom_vars": {
      "checkout_configuration": "small",
      "checkout_nacl": False,
    },
  },
]
target_os = ["android"]
```

In the future, other `checkout_` variables worth checking may be added to the project.

Beyond this options, make sure to always use `gclient sync -D` when upgrading between major releases, so that unused third party code is removed from the tree.

3.5 Specific mitigations for repository size in GitHub

Pushing a new copy of Chromium to GitHub can be problematic. In special if the resulting repository is expected to be private.

Nowadays, GitHub repository size is expected to be mostly unlimited. But that does not mean we can easily push Chromium to a private repository. First check the official policy covering part of the rules. Then:

- From our experience, there is a non explicit GitHub push size limit. Not per commit, but per push. A too much big push will fail. We observed this could be around 1GB, but it is not clear how big this limit is now.
- There is a limit in file size of 100MB.
- Pushing a huge repository takes bandwidth and time.
- There was a repository size limit of 100GB. But this is not strictly enforced anymore. Though, there is the recommendation to keep repository size below 1GB (and strong recommendation to have it below 5GB).

Some metrics can be extracted from the repository using the tool git-sizer (available in Ubuntu and Debian package `git-sizer`). The critical numbers are commit and blob maximum sizes, biggest checkout size, and the blobs overall size.

As a private repository cannot be forked from a public one, the repository needs to be pushed completely from Ecosia side.

These are several strategies that can help to get a full Chromium repository in a private copy:

- Push using upstream tags in master progressively until you have the full repository. I.e. push Chromium 30.x, then 60.x, then 70.x, 80.x, 85.x. At this point you should already have a git mirror of Chromium that can be private.
- In the case of using snapshots, you may hit the limit for a single push. To workaround that, you can split the snapshot in several commits. I.e. one for Chrome's `src` but `third_party`, then one for `third_party` but `blink`, and the last one for `third_party/blink`.
- There is a last possibility, that is just developing in a public repository. Then you can fork directly from the official Chromium mirror.

4 Patches strategy

4.1 Merge vs Rebase

We strongly recommend rebase over merge. It makes life easy for git algorithms to analyze changes, and further rebases. It also allows to progressively simplify history of commits in further rebases.

4.2 Squashing

Keeping a clear list of patches to apply in the future is helpful, to simplify future rebases, and avoid needing to rebase intermediate versions of the same work. Here we propose some recommendations to keep the number of patches lower.

4.2.1 Fixups

When a commit is usually a fix of an existing commit (not new functionality, just a commit that makes a previous commit work in a better way, or just fixes a bug), it is better to use explicitly a fixup commit.

Git creates that automatically doing `git commit --fixup COMMIT_ID` (being `COMMIT_ID` the hash of the commit the new commit will modify). It will generate a commit with a single line using the original commit message.

This should NOT be the final commit message. It should be amended, keeping the summary line as proposed by `--fixup`, but add all the relevant information in the commit (explain the bug it fixes, etc, etc).

4.2.2 Analysis of dependencies

When squashing a large number of commits, identifying which ones can be squashed together without efforts can be useful.

The idea is looking for clusters of interrelated patches, and then, analyze each cluster to decide the order they should be applied, and if any squashing is interesting.

For that purpose we recommend the tool `git-deps`: `https://github.com/aspiers/git-deps`

It allows to easily identify for a series of commits, the dependencies they have.

Knowing the dependencies is specially important also on the rebase process, as it allows to parallelize work: two clusters of patches depending only on the upstream tree, but not between them can be applied independently in any order we want. So they can be assigned to different developers in case manual porting is needed.

4.2.3 Tagging commits

For the purpose of simplifying the commit structure, it could be useful to add tags to the commit messages, so related patches could be identified. This activity can be useful after the analysis of the commits dependencies, once we figure out which clusters we have, and the area they belong to.

The idea is that each commit summary should be prefixed with a tag for the area it belongs to. This way, any rebase activity will show in the commit line the areas, so the rebase and squash strategy can move towards putting all the commits of an specific area together, or in small groups.

Tags should be:

- Short
- Hard to write in different ways for the same word
- Use spaces or underscores or dashes, but do not mix them.
- Map them to the ticket system labels if possible.

A file explaining what each tag means can be added to the tree (i.e. as part of README.md or any other development document).

An example: for commits related to translations, we could use `i18n`. Then, when squashing we could try to get them together.

Imagine we have this hypothetical commit history:

```
1234567 [i18n] Translate to Esperanto
4432111 [other] Other unrelated stuff
abcdef1 fixup! [i18n] Translate to Esperanto
4567890 [i18n] Fix dialogs in Esperanto
2341234 Revert: [i18n] Fix dialogs in Esperanto
1122334 [i18n] Fix all dialogs in Esperanto
4124121 [other] More work on other unrelated stuff
```

With tags, fixups and reverts analyzed, we would end up with something like this:

```
321654b [other] Other unrelated stuff
6172839 [other] More work on other unrelated stuff
4141515 [i18n] Translate to Esperanto
bbccaad [i18n] Fix all dialogs in Esperanto
```

As you see, it is easier to scan the list of commits, and identify groups of commits belong to same feature.

As with dependencies analysis, these groups allow to more easily split the work to port to a new major release, as we can split the rebase work in different developers.

4.2.4 Upstream status tags

As part of the tags that can be added to commit summaries, we recommend using specific tags to explicitly state if a patch is backported from upstream. But also if it is a candidate for upstreaming.

The idea is that backported patches will go away in the future, so we need to check its status in further rebases.

The idea for upstreamable patches is not keeping them forever, but actively proposing them upstream, and in the end, eventually, replace them with the upstream version of the functionality.

4.2.5 Squashing a branch

For any Ecosia working branch, we'd create a new branch that would have the same name, but with the suffix `-squashed`. In this branch we do an interactive rebase on top of the same baseline of the original branch.

The goals of this interactive rebase are:

- Removing commit-revert pairs
- Joining the fixup commits with the original commit. It is important to update the commit message to reflect the resulting patch (and likely cleaning any information that is not relevant anymore). In case the fixups are trivial, then this step can be partially automated using `git rebase -i --autosquash`.
- Take a look to the history to see if there are other squashing opportunities. In special, after refactoring, it could be interesting to squash and apply only the final version of the refactoring, and not previous approaches.

Once the squashing branch is ready, this is the one that should be used to rebase onto newer releases.

4.3 Reducing the modifying delta

The general strategy will be keeping Ecosia specific files in ecosia specific folders (or in case of a small number of files in an upstream folder, ecosia suffixed files).

4.3.1 Copy vs diff

It may be tempting to take upstream files, copying them as Ecosia version of a component, and work on top of that. While this reduces the modifying delta, there are some problems:

- Fixes in the upstream file will not be applied to the Ecosia flavor of the file.
- Upstream refactorings will pass undetected in the rebase, and only when building or running it we'll find problems.

When rebasing, a git conflict is annoying of course, and need to be resolved immediately. But it is also a way to detect that a change is going to affect our codebase. So, in general a copy of upstream file is not a good idea, unless the new file is a completely different version of it and we don't expect to be interested in the upstream changes in that file.

5 Implementation patterns

Some patterns can help maintaining out of an upstream file major changes done downstream.

5.1 Inherit from downstream

Apparently, a way to add behavior to upstream class would be just inheriting from it, to override existing methods or use pattern or abstract class template to add behavior in its child class.

But this requires changing all the instantiations of the class to use the downstream one, or use a factory to create the right instance.

A way to avoid this problem is actually making the upstream class inherit from the downstream class, and revert the delegation. This implies usually a slightly bigger delta in the downstream class, but on the other hand, all code creating the upstream object will get the downstream behavior.

In Brave, a variant of this was used, automatically copying and renaming the upstream file to add the downstream implementation with upstream class name, as part of the build process.

5.2 Mixins

Mixin pattern usually allows to aggregate different behaviors in an instance. So it could fit in this case to allow to add downstream behavior to an existing implementation.

Mixins can be implemented in C++ using multiple inheritance or templates (if no explicit inheritance relationship with parent is expected).

For Java, it is less practical, but possible, using a mix of interfaces and delegates.

5.3 Decorator

Very similar to Mixins in implementation for C++, and with same problems (it affects the instantiation calls).

In the cases it is simple to add behavior and isolate where the class is instantiated it could be also practical.

5.4 Factory

For mixins, decorators and regular inheritance, adding or updating factories to get the downstream behavior.

6 Testing infrastructure

6.1 Current infrastructure

The testing infrastructure is currently mostly ad-hoc, due to the lack of resources. So there is not a lot of automation in place.

It consists on:

- Some unit tests for logical units.
- Continuous integration: a CircleCI job that deploys a Release-Build to AppCenter after merges and pull requests for the m78 branch. It does not build or run any test.
- Manual testing: provided by project management, UX experts and other developers. There is also a Beta-Version deployed in house to gather feedback.

It does not provide:

- Integration tests.
- UI/functional tests.

6.2 Diagnosis

For fast and continuous upgrade, the problem with current infrastructure is the lack of ability to verify a build is working, and did not introduce regressions.

Lack of coverage reports is another weak point, as it is not clear how much of the introduced codebase or major browser use cases is covered by testing.

6.3 Proposal for continuous integration

To reach the goals of fast and continuous upgrade, automated testing should happen as part of the continuous integration.

Ideally, continuous integration should happen:

- On every commit.
- On every automatic/semiautomatic rebase.

6.3.1 Tests to run

Upstream regular unit tests

Chromium already provides extensive unit tests. So that should help, as they are well maintained, and ensure a high level of coverage in the codebase used by Ecosia browser.

The selection used for continuous integration should ideally be similar to the ones used for official Android builds. The upstream script input is in `testing/buildbot/test_suites.pyl`.

Ideally, first, the upstream smoke tests should be added to the continuous integration so it happens per commit. As the CI infrastructure improves, progressively more testing should be added on top of smoke test, following the `chromium_android_gtests` target.

The upstream test expectations will, in some cases, require changes because of the modifications introduced downstream.

Ecosia specific unit tests.

On top of the already existing tests, specific test sets should be added. They should be independent from upstream tests, even at continuous integration configuration.

Ideally, as these tests are in downstream control, they should be ran on any pull request.

Considerations

The continuous integration capacity is going to determine how much testing can be done, and when.

The recommendation is incrementally adding test sets to continuous integration so it does not become a bottleneck for development of new features or upgrade process.

6.3.2 Manual tests

These tests are expected to be executed manually. Though, any of the manual tests should be designed so it is possible to automate it in the future.

Several levels of tests need to be defined, depending on the goals of the test:

- Basic test battery: developer should manually test on every commit this basic test battery before proposing a commit. It should be runnable in less than 5-10 minutes. The idea is just making sure the app is not fundamentally broken (ideally replaced by smoke tests).
- Release test battery: QA team. Before tagging any release, a bigger battery of tests need to happen. This way, any tagged release is expected to be working for all the major use cases.
- Full test battery: after each rebase, and before announcing publicly the release is available, a full battery of tests should happen, covering all the use cases. In case continuous upgrade happens, this should at least be done before releasing a rebase on top of a new major release.

All bugs reported should point to a test in the full test battery. So any major release can be verified.

The number of tests will basically depend on the bandwidth of the testing team.

Considerations

Bigger tests batteries will ensure better coverage. Because of that, a policy towards automation (i.e. finding a way to do the basic test battery automatically as part of the commit checks) will improve productivity of the team.

Adapting upstream smoke tests to already provide some basic functionality coverage should be enough to replace the basic test battery.

6.3.3 Branding tests

Additionally, we recommend setting up specific tests for Ecosia branding.

Some examples:

- Build target that renames Chromium branding files, and check if build finishes. If our build still uses them, then it will fail.
- Look for certain forbidden strings in generated resource files.

6.4 Performance testing

Performance testing requires separate consideration as the other type of tests as it doesn't deal with correctness so that running the performance tests should not be necessary for every commit.

The main thing to consider is that, unless there's a specific focus on performance, you shouldn't pay much attention to specific performance changes between major releases, as those might be related to underlying changes in upstream Chromium.

Only when there's an obvious performance problem, it should be further investigated. For that, we recommend the Chromium tracer, that can be used to target both C++ and JavaScript.

6.5 Mitigations for restricted continuous integration infrastructure

As reported by Ecosia, continuous integration infrastructure expenses can be a serious handicap for the implementation of all the proposed tests in the continuous integration.

Even as we strongly recommend automation in continuous integration as the best way to validate the frequent upgrades, some procedures can be implemented to work around this limitation.

6.5.1 In house extra continuous integration infrastructure

Providing an in-house machine with enough power to provide a full Chromium build and test every 2 hours could be done with a single machine, and it does not need the full-time availability requirements of the external continuous integration.

Several alternate solutions are available for moving the continuous integration efforts to in-house machines.

We want to explicitly discard these:

- Deploy LUCI. This is the upstream continuous integration infrastructure nowadays. We do not recommend it as it is poorly documented, and it is not really targeted for our use case, that is not expected to require lots of builds per hour from a massive contributors team.
- Deploy a Buildbot master and one or more slaves. Before LUCI, it was the upstream backend. Though the documentation is better, and there are examples about how to integrate it with Chromium compilation, it is still too much overhead for the Ecosia project requirements.

We think the simplest way is providing one or more self-hosted runners that are not rated per minute, CPU usage or bandwidth:

- One way is just self-hosting a GitHub actions pipeline.
- It is also possible to use GitLab CI in the free plan, with a self-hosted pipeline.

For improved efficiency, a compilation cache would be useful to be able to run more build cycles per day, and reduce waits for obtaining the result of a compilation in the runners.

Once a self-hosted machine provides at least some availability that can be offloaded from CircleCI, an updated policy could be:

- Build any new commit in the working branches, once they are maintained (stable branches and the stabilization branch on next release once rebase is completed).
- Allow developers to request a test build (explicitly enabling or disabling certain tests, and the branch in GitHub to compile). Though, it is preferred that developers actively working on a project can do a big part of the testing in their development machines instead of systematically trying with the continuous integration infrastructure.

6.5.2 Scripts-backed checklists

Another way to reduce the requirements and cost of an external continuous integration infrastructure is providing certain checklists that should be run in developers side before sending a pull request.

Ideally this could be a form of checklist, that could be partially based on scripts to speed up the procedure.

An example of a checklist that could be done in the developer side before submitting a commit request:

- Run `git-clang-format`
- Compile target and run the basic test battery.
- Conditional:
  - For UI changes, run the Ecosia Android unit tests.
  - For resources changes, run the branding tests.

7 First upgrade

After the analysis of the current delta, and reviewing the possible repository structure, and discussing the testing requirements for the different upgrade models, it is time for a detailed proposal for the first upgrade.

The diagnosis points to a good start point already, where the preparation work done by Ecosia as part of the work for porting to Chromium 78 baseline will speed up the work for the next upgrade.

7.1 Version to target

Upstream Chromium release procedure is:

- New features development happens in `master` branch.
- Every 6 weeks a branch for a new release is done, at the end of Thursday (Pacific time). The work in the branch from this point is for stabilization and new features should not be landed.
- 6 weeks later, first stable version should happen in the branch.

Detailed release cycle description available at `https://chromium.googlesource.com/chromium/src/+/master/docs/process/release_cycle.md`. Calendar is available at `https://chromiumdash.appspot.com/schedule`.

7.1.1 Alternative 1: upstream branch day

For first upgrade with the new procedures, our proposal is using as baseline m86 branch that is expected to be created on August 20th. This means there is more than a week for preparation work.

7.1.2 Alternative 2: upstream first beta release

Though, waiting some days for the first official beta release could be useful as this way work can start on top of a more tested version. In that case, we could just wait for the first official beta release for Android, and take that version as the baseline for the upgrade. This happens usually 2 weeks after branch point.

The official releases are announced at `https://chromereleases.googleblog.com/`. In this case waiting for m86 release is possible (though it would happen in September). Or just taking the latest m85 official beta release, that is already available.

7.2 Preparation

Preparation required is adding the minimal testing infrastructure, and squashing the current repository, to reduce the rebase effort.

7.2.1 Manual testing infrastructure

We recommend at this stage to document just manual steps to test the result of the rebase, so the intermediate results of the rebase effort can be validated.

More details can be seen in the Testing infrastructure chapter about the different tests. We propose, for this stage, getting ready to run manually these:

- Basic test battery.
- The already developed Ecosia unit tests.
- The upstream smoke tests (GN target `android_smoke_tests`).

So, the preparation work in this case is writing the basic test battery.

7.2.2 Squashing

First, in the current repository, a baseline tag needs to be defined:

```
git tag upstream-78.0.3904.96 1956716893e9e91bb316d4fa59a991458b956ff0
```

In the working branch, a branch named `v78.0.3904.96-squashed` should be created:

```
git checkout v78.0.3904.96
git checkout -b v78.0.3904.96-squashed
```

Then, several iterations would happen in that branch, for squashing the commit history:

```
git rebase -i upstream-78.0.3904.96
```

In these iterations, pairs of commits and reverts should be reordered first to be continuous (having the commit, and immediately after that its revert). Some trivial conflicts may happen in the middle.

Then next iterations should be for grouping functionality areas, and possibly squash them in less commits.

I.e. if feature X is present, it could have:

- Several commits to implement it.
- Several commits for different bugfixes.

At least bugfixes on the original implementation should be squashed to the commits introducing them. But it could be interesting to squash all the commits introducing a feature in a single commit. Specially if the different commits represent iterations of the implementation of the feature.

⚠ On finishing each rebase pass, validation should be done checking the diff between original and squashed branch is empty.

Git rebase vs revise

A nice speed up in the procedure to rebase patches is using the tool `git revise` instead of the standard `git rebase`. It is essentially the same tool, but instead of changing the working directory and index, it does all the intermediate work in memory, dramatically speeding up the rebase process.

Beyond the speed benefits, there are two other features that are very useful for the kind of work described here:

- It's non destructive: because it works in memory, the repository won't be changed so that you cannot end in a mid-rebase state while using it and it won't trigger unnecessary rebuilds.
- It checks that the state of the tree after the rebase process hasn't changed, making it unnecessary to do any checks for unintended changes, as warned earlier.

Some other useful workflows that are made easier by this tool are:

- Implicit base commit: `git revise -i` will probably find the root of the branch for you.
- Adding fixes to older commits: `git add . && git revise $COMMIT`.
- Splitting commits: `git revise -c $COMMIT`.
- Reword several commits in one go: `git revise -ie`.

The tool webpage is at https://github.com/mystor/git-revise/. These workflows and others are described in https://mystor.github.io/git-revise.html. It is also included in the Ubuntu and Debian APT repositories, with the package name `git-revise`.

git rerere

Another tool that can speed rebases up is `git rerere`. This tool automates conflict resolution by recording previous conflicted automerge results and the corresponding hand resolve results.

To enable this feature, run `git config --global rerere.enabled true`. This allows both `git merge` and `git rebase` to automatically call the tool when a merge fails to both try to use past resolutions and also to record a new one, in case it didn't succeed.

7.3 Rebase work

7.3.1 New repository

Following the proposals in Repository structure chapter, a new repository would be created:

- An empty one in case we are using proposal 1.
- A fork of upstream repository in case we use proposal 2.

This will make current `chromium` tree obsolete. Some proposals for the new repository name could be `chromium-src`, `ecosia-chromium`...

7.3.2 Land baseline in repository

The goal of this step is obtaining a commit that represents the baseline we chose.

- For proposal 1, we will have to obtain an snapshot of upstream and name it with the proposed format of with `upstream-` prefix.
- For proposal 2, this step is not needed, as the tag is already provided by upstream.

7.3.3 Apply squashed branch

As, this time, the repository is going to be a new one, so transferring commit history from the previous repository is needed.

Using `git format-patch` a patchset in form of files will be created, from the current repository:

```
git format-patch -k upstream-78.0.3904.96..v78.0.3904.96-squashed \
  -o squashed_patches
```

Then, in the new repository, checkout the baseline. In the example, with an hypothetical `86.0.9999.20`:

```
git checkout upstream-86.0.9999.20
```

In case repository follows proposal 2, it would be directly `86.0.9999.20` tag.

Create the branch for applying the rebase:

```
git checkout -b ecosia-86.0.9999.20
```

And start applying all the patches, using `git am -k --reject`. If the previous `git format-patch` target directory was `PATH_TO_DIR` it would be:

```
git am -k --reject PATH_TO_DIR
```

As it is a lot of patches, it may be interesting to use the `--interactive` option of `git am`, or just do each commit manually:

```
git am -k --reject PATH_TO_PATCH.patch
```

This is likely the best option as we expect non trivial conflicts applying the patches from previous releases.

This incremental approach is specially interesting in this first upgrade, as it allows testing the intermediate results.

7.3.4 Stabilization

Once all the patches are landed, result is compilable and the proposed basic tests are OK, it should be tagged:

```
git checkout ecosia-86.0.9999.20
git tag -a -m "86.0.9999.20.ecosia.1" 86.0.9999.20.ecosia.1
```

From this point, any new commit should pass the basic test battery, smoke test and Ecosia unit tests. Same applies, then, to tagged versions.

7.4 Further work

Once all rebasing process has been done, further development work can start in the new branch. This branch should also get any patch that is applied later to the old branch while it is still the official stable version.

7.4.1 New contributions

Any new contribution will follow the same requirements of the first tagged version (smoke tests, Ecosia unit tests, upstream smoke tests). And any deliverable provided to customers should be tagged.

7.4.2 Rebase to next upstream stabilization releases

As new stabilization tagged releases happen upstream, it is recommended to upgrade to them, once tagged Ecosia releases are done.

The procedure will start with landing a snapshot of the new upstream version (i.e. `86.0.9999.40`), or having a copy of the upstream tag in the downstream repository.

Then, a rebase would be done:

```
git checkout ecosia-86.0.9999.20
git checkout -b ecosia-86.0.9999.40
git rebase --onto upstream-86.0.9999.40 upstream-86.0.9999.20 ecosia-
86.0.9999.40
```

In this case the expectation of conflicts will be lower as the upstream stabilization progresses. So this procedure is expected to be fast and done by a single developer.

After rebase, all the basic testing should happen, and then a new tag should be done. Let's assume for this example ecosia release is now 5 (so in previous branch last tag was 86.0.9999.20.ecosia.5):

```
git tag -a -m "86.0.9999.40.ecosia.5" 86.0.9999.40.ecosia.5
```

8 Continuous upgrade

After the first successful upgrade using the new structure, as described in the first upgrade chapter, the expected scenario would be:

- Chromium repository including the upstream baseline, and on top of it, all the downstream patches from Ecosia, in a linear history.
- Minimal testing infrastructure as proposed in the Testing infrastructure chapter.
- The functionality in the rebased branch is at least the same of the previous working branch.
- Active development of new features happen in the new branch.

At this point, continuous upgrade procedures can start to be deployed. There are two different procedures involved:

- For point releases (changing the 4th part of the upstream version number). This is the rebase we do as stabilization of the beta branch is happening, but also the one that happens for the upstream stable releases.
- For major releases (changing the 1st part of the upstream version number). This is going to happen after upstream branches for a new major release.

8.1 Point release upgrades

8.1.1 When to start

We recommend first completing all the rebases on a single upstream version before taking any point release upgrade. If, for some test cases, we need to backport a fix from upstream, then that would be backported instead of rebasing to a newer version.

This is because the rebases should happen as fast as possible. So it is preferred to avoid targeting a different baseline in the middle of the process.

Once the initial rebase process is completed, tested and validated, the recommendation is taking all the point releases from upstream that hit the release channel (beta while it is still not stable, then stable). The rebase would start the very same day the official release happens.

8.1.2 Procedure

This is a very simplified version of the upgrade procedure described in First upgrade chapter, as the expectation is that the rebase itself should have almost no conflicts (and progressively less as the upstream stabilization continues).

1. Obtain the baseline (upstream-X.X.X.Y branch or X.X.X.Y tag)
2. Checkout latest `ecosia` branch: `git checkout ecosia-X.X.X.X`
3. Create branch: `git checkout -b ecosia-X.X.X.Y`
4. Rebase on top of the new baseline:
   - `git rebase --onto upstream-X.X.X.Y upstream-X.X.X.X ecosia-X.X.X.Y`
   - Or `git rebase --onto X.X.X.Y upstream-X.X.X.X ecosia-X.X.X.Y`
5. As rebase progresses, resolve conflicts.
6. Validate (with testing, up to available capacity).
7. Tag and publish Ecosia release.

8.1.3 Conflicts and errors

This kind of rebases should happen very fast. If, on rebase, certain patches have conflicts, apply the techniques proposed to reduce initial rebase times (skipping or delaying conflicting patches, disabling parts).

The goal is, first, having something that builds, then something that runs, and last, something that is releasable. This is to allow to split the work for fixing the rebase result among the members of the team.

8.2 Major release upgrades

Most of this procedure is going to be essentially the same used for the first upgrade, though, as most of the pieces are in place from previous upgrades, it should be simpler.

8.2.1 When to start

Again, as suggested in the point release procedures, any major upstream rebase should have been finished when a new major release is going to be taken into consideration.

The recommendation is following the same considerations already done for first upgrade version to target section. In particular, we again recommend alternative 2: waiting for the first Beta release to hit `https://chromereleases.googleblog.com/` (and the upstream release channels).

8.2.2 Procedure

In this case, this is a not so simplified version of the upgrade procedure described in First upgrade chapter. This is because the chances to get major conflicts in a major release are higher (i.e. maybe new features have been developed for the major release, or refactorings that are never expected to happen in the stabilization stage).

So, for upgrading from major version `M` to `N`:

1. Follow the squashing procedure. As a result, a branch named `ecosia-M.X.X.Y-squashed` will contain the squashed version of the working tree of the previous version.

2. Obtain the baseline (`upstream-N.X.X.X` branch or `N.X.X.X` tag)
3. Checkout the squashed branch: `git checkout ecosia-M.X.X.Y.squashed`
4. Create the new branch: `git checkout -b ecosia-N.X.X.X`
5. Rebase on top of the new baseline. Apply the strategy that fits better (see the next sections).
6. Validate (with testing, up to available capacity).
7. Tag and publish a new Ecosia release.

8.2.3 Rebase policy and conflict resolution

The main difference comes in the steps for doing the actual rebases. Instead of just automatically rebasing, we propose several alternate procedures, that could be tried in a certain order, to allow splitting the rebase work among several members of the team.

One of these rebase procedures should be used:

- Automatic rebase. In case an automatic or semiautomatic rebase works, as proposed in the point release upgrade.
- Defer conflicts. Same as in the automatic rebase case but, for conflicts and expected problems, the patches order is modified to handle any problem at the end of the rebase sequence.
- Non sequential rebase. In case the expectation of conflicts is big, then this procedure is recommended, splitting the rebase work at the start.

The way to decide which procedure to take can be as simple as trying each one, and when the rebase starts to get too complex, fallback to the next procedure.

8.2.4 Automatic rebase strategy

Rebase on top of the new baseline:

- `git rebase --onto upstream-N.X.X.X upstream-M.X.X.Y ecosia-N.X.X.X`
- Or `git rebase --onto N.X.X.X upstream-M.X.X.Y ecosia-N.X.X.X`

8.2.5 Defer conflicts rebase strategy

Same as previous approach, rebase on top of the new baseline:

- `git rebase --onto upstream-N.X.X.X upstream-M.X.X.Y ecosia-N.X.X.X`
- Or `git rebase --onto N.X.X.X upstream-M.X.X.Y ecosia-N.X.X.X`

If a patch does not seem trivial to solve, then use `git rebase --skip` and note it down to apply manually later.

Once the rebase is completed, the work for the pending patches should be scoped and assigned. Ideally an issue tracker ticket should be created per failed commit (or for a group of commits). This work can then be parallelized, as different tickets can be assigned to different members of the team.

8.2.6 Non sequential rebase strategy

If we see the number of conflicts in previous approaches is very big, then the recommended strategy is creating a table of all the commits, grouped by area (that should be easy if the commit tags proposal is in place).

In this table, for each implementation area (or tag), an issue tracker ticket should be created, documenting the list of commits. Then, the work in each ticket should be done cherry-picking the commits in the new development branch.

It is important to identify and add dependencies among the issue tracker tickets, as some areas may depend on other areas. And so, the commits for the dependent area should be ported after the area it depends on.

8.2.7 Testing and validation

As part of the stabilization work, we need to make sure all the batteries of tests are passed, before completing the Beta process upstream. The idea is that once first stable release for the new major release, no regression passes undetected.

This implies the time to complete the rebase is important, as it needs to provide some time before upstream stable release for making the quality of the product good enough for the user base.

9 Appendix

9.1 Version history

Version	Release	Changes
`0.1`	August 11th, 2020	`Initial draft.`
`0.2`	August 18th, 2020	• Continuous upgrade: major upgrade testing. • Continuous upgrade: add major release procedure. • Repository structure: add mitigations for GitHub repository limits. • Add some hints to reduce download and repo size. • Continuous upgrade: introduction and point release upgrade procedure. • Continuous upgrade: write notes about upstream status tags. • Patches strategy: commit tags and git-deps. • Repository structure: split in different chapters. • Move and expand git-revise section.
`1.0`	August 20th, 2020	• Spell fixes. • Performance testing. • git rerere. • Introduction: add credits and general introduction. • Testing infrastructure: hints for reducing the needs for CI.