Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Αυτοματοποιημένος, Κατανεμημένος Συγχρονισμός αρχείων με Χρήση του Εργαλείου Git και Υποστήριξη Διαφορετικών Μηχανισμών Αποθήκευσης

Διπλωματική Εργασία

του

Αλεξίου Β. Τσιτσιμπή

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Νοέμβριος 2016

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Automated, Distributed File Synchronization with Git over Multiple Storage Backends

Diploma Thesis

of

Alexios V. Tsitsimpis

Supervisor: Nectarios Koziris
Professor, N.T.U.A.

Computing Systems Laboratory
Athens, November 2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Αυτοματοποιημένος, Κατανεμημένος Συγχρονισμός αρχείων με Χρήση του Εργαλείου Git και Υποστήριξη Διαφορετικών Μηχανισμών Αποθήκευσης

Διπλωματική Εργασία

του

Αλεξίου Β. Τσιτσιμπή

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2η Νοεμβρίου, 2016.

.....	.....	.....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2016

.....
Αλέξιος Β. Τσιτσιμπής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

στην οικογένειά μου
στους φίλους μου
σε όσους με κάνουν να χαμογελάω

Περίληψη

Σε αυτή τη διπλωματική υλοποιούμε το Gitsync, ένα εργαλείο που επιδεικνύει πώς μπορεί να χρησιμοποιηθεί το Git για την επίτευξη αυτοματοποιημένου, κατανεμημένου συγχρονισμού αρχείων με υποστήριξη διαφορετικών μηχανισμών αποθήκευσης. Για το σκοπό αυτό, αρχικά εξερευνούμε και αναλύουμε τον τρόπο με τον οποίο λειτουργεί εσωτερικά το Git, και πώς αυτό χειρίζεται διενέξεις που προκύπτουν κατά τη διαδικασία της συγχώνευσης. Έπειτα, παρουσιάζουμε τη διαπροσωπία που χρησιμοποιεί το Git για να επικοινωνήσει χρησιμοποιώντας πρωτόκολλα που του είναι άγνωστα, και την αξιοποιούμε για να επιτύχουμε συγχρονισμό αρχείων με τη χρήση δικού μας πρωτοκόλλου. Τέλος, πειραματιζόμαστε με δύο διαφορετικές μεθόδους για την αυτόματη επίλυση διενέξεων, που βασίζονται στις διαδικασίες της συγχώνευσης και της αναθεμελίωσης, έννοιες του εργαλείου Git.

Λέξεις Κλειδιά

Git, file synchronization, conflict resolution, Gitsync, Git remote helpers, merge, rebase

Abstract

In this thesis we implement Gitsync, a prototype tool that demonstrates how Git can be used to achieve automated, distributed file synchronization over multiple storage backends. In order to do so, we first explore and analyze the way Git internally works, and how it handles conflicts during a merge process. Then, we present the API Git uses to communicate over protocols natively unknown to it, and utilize it in order to synchronize using our own protocol. Finally, we experiment with two Git-based strategies to automatically resolve conflicts, one called merging and the other rebasing.

Keywords

Git, file synchronization, conflict resolution, Gitsync, Git remote helpers, merge, rebase

Contents

List of Figures

Κεφάλαιο 1

Εισαγωγή

1.1 Κατανεμημένος Συγχρονισμός Αρχείων

Με την παρούσα διπλωματική εργασία επιχειρούμε να αναλύσουμε το πρόβλημα του κατανεμημένου συγχρονισμού αρχείων, και να το επιλύσουμε χρησιμοποιώντας το εργαλείο Git με τέτοιο τρόπο ώστε να είναι εφικτή η υποστήριξη διαφορετικών μηχανισμών αποθήκευσης και η αυτοματοποίηση της διαδικασίας του συγχρονισμού.

Συγχρονισμός αρχείων είναι η διαδικασία της διασφάλισης ότι αρχεία σε δύο ή περισσότερες τοποθεσίες ενημερώνονται με τη χρήση συγκεκριμένων κανόνων. Τις περισσότερες φορές το επιθυμητό αποτέλεσμα είναι οι δύο (ή περισσότερες) τοποθεσίες να διατηρούνται πανομοιότυπες. Με την εξέλιξη της τεχνολογίας, και τις ολοένα και περισσότερες "έξυπνες" συσκευές, η ανάγκη μία τροποποίηση σε ένα αρχείο, για παράδειγμα αρχείο κειμένου, να διαδίδεται σε όλες τις συσκευές που μπορεί να έχει κάποιος στην κατοχή του, ολοένα και αυξάνεται. Κάτι τέτοιο είναι εφικτό χάρη στο συγχρονισμό αρχείων.

Υπάρχουν δύο τρόποι τα αρχεία δύο τοποθεσιών να παραμένουν συγχρονισμένα. Είτε οι δύο τοποθεσίες επικοινωνούν απευθείας μεταξύ τους (αποκεντρωμένος συγχρονισμός) ή χρησιμοποιούν μία τρίτη τοποθεσία ως μεσάζοντα, και κάθε τοποθεσία συγχρονίζεται με τον μεσάζοντα (κεντρικός συγχρονισμός). Στην δεύτερη περίπτωση οι τοποθεσίες οργανώνονται σε τοπολογία αστέρα με την τοποθεσία-μεσάζοντα να αποτελεί τον κεντρικό κόμβο της τοπολογίας.

Στον κατανεμημένο συγχρονισμό αρχείων, οι διάφορες τοποθεσίες που προσπαθούμε να συγχρονίσουμε μπορεί να έχουν διαφορετικές εκδόσεις του ίδιου αρχείου, οπότε και προκύπτουν κάποια προβλήματα κατά τη διαδικασία του συγχρονισμού. Αυτά τα προβλήματα τα αποκαλούμε *διενέξεις (conflicts)*. Η επίλυση διενέξεων δεν είναι εύκολο έργο.

Υπάρχουν ήδη υπηρεσίες και προγράμματα (όπως το Git) που απλοποιούν την ανίχνευση διενέξεων, αλλά απαιτούν αλληλεπίδραση με τον χρήστη για την επίλυση τους. Επεκτείνοντας την ιδέα αυτή, άλλα προγράμματα (όπως το Git annex [8]) παρέχουν αυτόματη επίλυση διενέξεων, αλλά ο συγχρονισμός των αρχείων είναι κάτι που πρέπει ο ίδιος ο χρήστης να κάνει χειροκίνητα, πολλές φορές χρησιμοποιώντας εντολές περίπλοκες για το μέσο χρήστη. Έπειτα υπάρχουν υλοποιήσεις που αυτοματοποιούν τόσο την ανίχνευση διενέξεων και την επίλυση τους, όσο και το συγχρονισμό των αρχείων. Ωστόσο οι υλοποιήσεις αυτές είναι τις περισσότερες φορές ιδιόκτητες, καθιστώντας έτσι αδύνατη τη μελέτη και τροποποίηση του πηγαίου κώδικα από τρίτους. Τέτοιες λύσεις στερούνται φορητότητας, και μπορεί να εγείρουν ανησυχίες σχετικές με την προστασία απορρήτου.

Στην προσπάθειά μας να επιλύσουμε το πρόβλημα του κατανεμημένου συγχρονισμού αρχείων, χρησιμοποιούμε το Git για την ανίχνευση διενέξεων, και αυτοματοποιούμε τη διαδικασία του συγχρονισμού χρησιμοποιώντας την προσέγγιση της τοπολογίας αστέρα. Επιπλέον, βρήκαμε τρόπο αυτόματης επίλυσης διενέξεων, με τις επιλύσεις αυτές να διαδίδονται σε όλες τις τοποθεσίες που συμμετέχουν στη διαδικασία του συγχρονισμού. Για την επίτευξη φορητότητας και ευελιξίας, στηριζόμαστε στους git-remote-helpers, σενάρια-βοηθοί που χρησιμοποιούνται ώστε το Git να μπορεί να επικοινωνήσει πάνω από πρωτόκολλα που δεν γνωρίζει. Ως απόδειξη ορθότητας της ιδέας, γράψαμε δύο τέτοια σενάρια βοηθούς, που παρέχουν τη δυνατότητα συγχρονισμού τοποθεσιών χρησιμοποιώντας διαφορετικούς μηχανισμούς αποθήκευσης ως μεσάζοντα, συγκεκριμένα έναν τοπικό κατάλογο ή την υπηρεσία Amazon S3 [1]. Επιπροσθέτως, για να επιδείξουμε ότι είναι δυνατή η εξάλειψη ανησυχιών σχετικών με το απόρρητο όταν χρησιμοποιούνται απομακρυσμένες τοποθεσίες ως μεσάζοντες (όπως συμβαίνει στην περίπτωση του Amazon S3), γράψαμε ένα σενάριο-βοηθό που χρησιμοποιεί το Amazon S3, αλλά κρυπτογραφεί τα δεδομένα που αποθηκεύονται εκεί.

Κεφάλαιο 2

Υπόβαθρο

2.1 Τι είναι το Git

Όπως αναφέρθηκε στην Εισαγωγή, η επίτευξη κατανεμημένου συγχρονισμού αρχείων προϋποθέτει την ανίχνευση διενέξεων μεταξύ των αρχείων. Αυτός είναι ένας από τους λόγους που επιλέξαμε να χρησιμοποιήσουμε το Git. Άλλοι λόγοι περιλαμβάνουν το γεγονός ότι είναι ελεύθερο λογισμικό, ευρέως γνωστό, πολύ καλά τεκμηριωμένο, με ενεργή κοινότητα και πολύ υλικό διαθέσιμο στο διαδίκτυο. Επιπλέον θα μας επιτρέψει τελικώς τη χρησιμοποίηση διαφορετικών μηχανισμών αποθήκευσης, με τη χρήση git-remote-helpers.

Το *Git* είναι ένα κατανεμημένο *Σύστημα Ελέγχου Εκδόσεων (ΣΕΕ)* (λέγεται και Σύστημα Ελέγχου Αναθεωρήσεων ή Σύστημα Ελέγχου Πηγαίου Κώδικα), που δημιουργήθηκε από τον Linus Torvalds το 2005, χρησιμοποιείται ευρέως στην ανάπτυξη λογισμικού και δίνει έμφαση στην ταχύτητα, στην ακεραιότητα των δεδομένων και στην υποστήριξη για κατανεμημένες μη γραμμικές ροές εργασίας.

Το γεγονός ότι είναι *κατανεμημένο*, σημαίνει ότι αντίθετα με τα περισσότερα συστήματα πελάτη-διακομιστή, κάθε κατάλογος εργασίας του Git είναι ένα ολοκληρωμένο αποθετήριο λογισμικού με πλήρες ιστορικό και δυνατότητες πλήρους παρακολούθησης της έκδοσης, ανεξάρτητα από την πρόσβαση δικτύου ή ενός κεντρικού διακομιστή.

Όπως τα περισσότερα ΣΕΕ, το Git μπορεί να ανιχνεύει διενέξεις αρχείων όταν αυτές συμβαίνουν, αλλά (συνήθως) απαιτείται ανθρώπινη παρέμβαση για την επίλυση τους.

2.1.1 Πώς λειτουργεί το Git

Το Git χειρίζεται τα δεδομένα του πιο πολύ σαν στιγμιότυπα ενός μικρού συστήματος αρχείων. Κάθε φορά που γίνεται *υποβολή (commit)*, δηλαδή που αποθηκεύεται η κατάσταση του έργου που ελέγχεται από το Git, στην ουσία "φωτογραφίζει" την κατάσταση στην οποία βρίσκονται τα αρχεία τη στιγμή εκείνη, και αποθηκεύει μια αναφορά στο στιγμιότυπο αυτό. Για λόγους αποδοτικότητας, αν κάποια αρχεία δεν έχουν τροποποιηθεί, το Git δεν θα αποθηκεύσει ξανά τα αρχεία, παρά μόνο έναν σύνδεσμο στο προηγούμενο πανομοιότυπο αρχείο που έχει ήδη αποθηκευμένο. Τελικώς, τα δεδομένα που κρατάει το Git είναι μια ροή στιγμιοτύπων.

Στην βάση του, το Git είναι ένα *σύστημα αρχείων διευθυνσιοδοτούμενο μέσω περιεχομένου (content-addressable filesystem)* με ένα ΣΕΕ ως διεπαφή χρήστη γραμμένο πάνω από αυτό. Αυτό σημαίνει ότι στον πυρήνα του το Git είναι μία αποθήκη απλών δεδομένων τύπου ζεύγους κλειδιού-τιμής.

Παρέχει τη δυνατότητα εκχώρησης οποιουδήποτε περιεχομένου σε αυτό, και επιστρέφει ένα κλειδί που μπορεί να χρησιμοποιηθεί για την ανάκτηση του περιεχομένου αυτού.

Με την εκτέλεση της εντολής `git init` σε έναν καινούριο ή υπάρχοντα κατάλογο, το Git δημιουργεί τον κατάλογο `.git/`, όπου και βρίσκονται σχεδόν όλα όσα το Git αποθηκεύει και χειρίζεται. Η δομή του καταλόγου `.git/` είναι η ακόλουθη:

```
.git/
 |- config
 |- description
 |- HEAD
 |- hooks/
 |- index
 |- info/
 |- objects/
 |- refs/
```

Listing 2.1: Δομή του καταλόγου `.git/`

- Το `description` είναι απλά ένα αρχείο κειμένου που εμφανίζεται ως περιγραφή του έργου σε κάποια ιστοσελίδα.
- Το αρχείο `config` περιέχει ρυθμίσεις σχετικές με το συγκεκριμένο έργο.
- Ο κατάλογος `info/` διατηρεί ένα γενικό αρχείο εξαιρέσεων για αρχεία των οποίων η παρακολούθηση μέσω του `.gitignore` αρχείου δεν είναι επιθυμητή.
- Ο κατάλογος `hooks/` περιλαμβάνει σενάρια που αφορούν τη μεριά είτε πελάτη είτε εξυπηρετητή, και εκτελούνται μετά από συγκεκριμένα γεγονότα.

Οι εναπομένουσες 4 καταχωρήσεις είναι οι πιο σημαντικές.

- Στον κατάλογο `objects/` αποθηκεύονται όλα τα περιεχόμενα της βάσης δεδομένων.
- Στον κατάλογο `refs/` αποθηκεύονται δείκτες σε συγκεκριμένα αντικείμενα της βάσης, αντικείμενα υποβολής.
- Το αρχείο `HEAD` είναι δείκτης στον κλάδο (branch) στον οποίο βρίσκεται ο χρήστης.
- Το αρχείο `index` χρησιμοποιείται από το Git για την αποθήκευση πληροφοριών για την περιοχή προς καταχώρηση (staging area)

Θα ασχοληθούμε τώρα με κάθε μία από τις 4 αυτές καταχωρίσεις λεπτομερώς, για να καταλάβουμε πως λειτουργεί το Git.

2.2 Αντικείμενα του Git

Έχει ήδη αναφερθεί ότι το Git αποθηκεύει περιεχόμενα και επιστρέφει ένα κλειδί ώστε να είναι δυνατή η αναφορά στο αποθηκευμένο περιεχόμενο στο μέλλον. Το περιεχόμενο αυτό αποθηκεύεται ως ένα αντικείμενο. Το κλειδί στην πραγματικότητα είναι ένας *κωδικός κατακερματισμού* που παράγεται με βάση τον αλγόριθμο SHA-1, και είναι ένα άθροισμα ελέγχου μιας κεφαλίδας (διαφορετικής

μορφής αναλόγως το είδος του αντικειμένου που αποθηκεύεται) ακολουθούμενης από το περιεχόμενο προς αποθήκευση. Υπάρχουν 4 είδη αντικειμένων που χρησιμοποιούνται στο Git: τα *blobs (δυαδικά μεγάλα αντικείμενα)*, τα *αντικείμενα δένδρου (trees)*, τα *αντικείμενα υποβολής (commits)* και τα *αντικείμενα ετικέτας (tags)*. Θα αναλύσουμε καθένα από αυτά τα αντικείμενα σε λίγο. Πριν την αποθήκευση του αντικειμένου, το Git συμπιέζει το περιεχόμενο και την κεφαλίδα με τη χρήση της βιβλιοθήκης zlib.

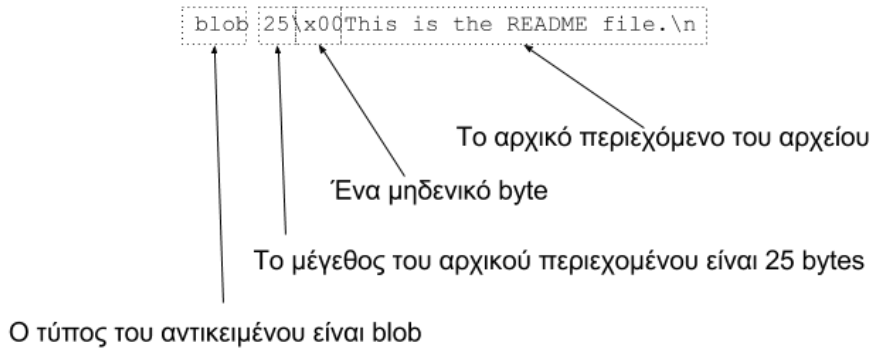
Τα αντικείμενα αποθηκεύονται στον κατάλογο `.git/objects/` στην ακόλουθη μορφή:

`<2 πρώτοι χαρακτήρες του κωδικού κατακερματισμού του αντικειμένου>/ <υπόλοιποι 38 χαρακτήρες του κωδικού κατακερματισμού του αντικειμένου>`

Γιατί να επιλεχθεί μία τέτοια μορφή; Το Git πρέπει να είναι φορητό, πρέπει δηλαδή να μπορεί να λειτουργεί σε διάφορα συστήματα αρχείων, όπως το FAT32, το NTFS και οι οικογένεια των EXT συστημάτων αρχείων (EXT2, EXT3, EXT4). Κάθε σύστημα αρχείων έχει διαφορετικό όριο στο πλήθος των αρχείων που μπορούν να αποθηκευτούν κάτω από έναν κατάλογο. Για παράδειγμα στο FAT32 το όριο είναι 65,535 αρχεία ανά κατάλογο. Επομένως είναι απαραίτητη η υποδιαίρεση ενός καταλόγου σε υποκαταλόγους, ώστε να γίνει λιγότερο πιθανό το γεγονός της πλήρωσης του. Επιπλέον, υπάρχουν συστήματα αρχείων (ιδιαίτερα τα παλιά, όπως το FAT32 και το EXT2) που οι δομές με τις οποίες αναπαρίστανται από το σύστημα οι κατάλογοι επιβάλουν γραμμική σάρωση των τμημάτων ενός καταλόγου για την αναζήτηση ενός συγκεκριμένου αρχείου. Η επίδοση της αναζήτησης και της ενημέρωσης σε τέτοιες μη δεικτοδοτημένες δομές υποβαθμίζεται γραμμικά με την αύξηση του μεγέθους του καταλόγου. Με τη διαίρεση σε υποκαταλόγους που χρησιμοποιεί το Git, διασφαλίζεται ότι η σάρωση του καταλόγου `.git/objects/` θα είναι γρήγορη ακόμη και σε παλιά συστήματα αρχείων.

2.2.1 Blobs

Το *blob* είναι η βασική μονάδα αποθήκευσης δεδομένων στο Git. Ο τύπος αυτός είναι απλά bytes που μπορεί να αντιπροσωπεύουν οποιοδήποτε περιεχόμενο (για παράδειγμα αρχείο κειμένου, πηγαίο κώδικα, εκτελέσιμο δυαδικό αρχείο, εικόνα κλπ). Η μορφή του περιεχομένου ενός blob φαίνεται στο Σχήμα 2.1.



Σχήμα 2.1: Η μορφή περιεχομένου ενός αντικειμένου blob

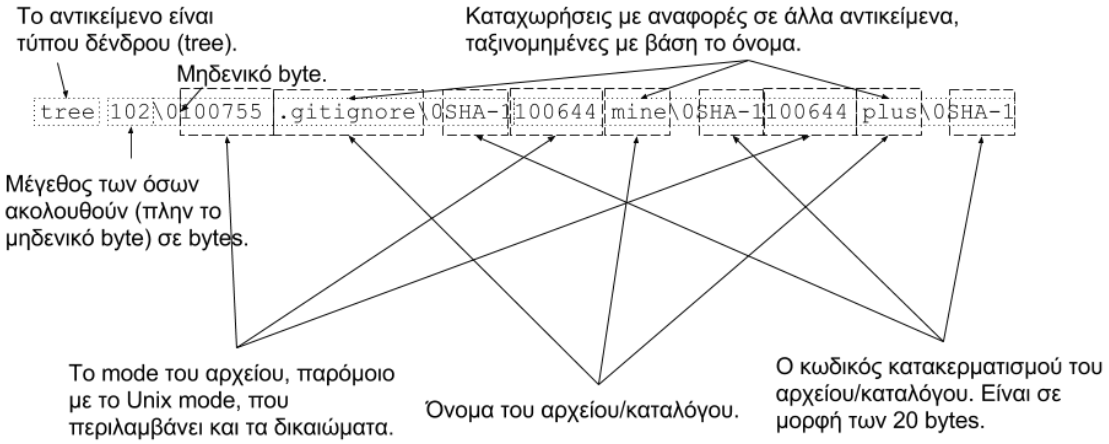
Είναι αξιοσημείωτο το γεγονός ότι το Git δεν αποθηκεύει το όνομα του αρχείου στα blobs. Αποθηκεύει τα αρχεία με βάση έναν κωδικό κατακερματισμού που απορρέει από το περιεχόμενο τους. Αυτή

η μέθοδος αποθήκευσης αρχείων με βάση το περιεχόμενό τους (αντί για το όνομά τους) ονομάζεται *αποθήκευση με διευθυνσιοδότηση μέσω περιεχομένου (content addressable storage)*.

Επιπλέον, το γεγονός ότι αποθηκεύεται μόνο το περιεχόμενο του αρχείου, καθιστά εφικτή την επαναχρησιμοποίηση των blobs από το Git για αρχεία με διαφορετικά ονόματα αλλά ίδιο περιεχόμενο. Στην περίπτωση αυτή, ο κωδικός κατακερματισμού θα είναι ίδιος, οπότε το Git δεν θα αποθηκεύσει επιπλέον αντίγραφο του αρχείου.

2.2.2 Αντικείμενα δένδρου (Trees)

Ένα *αντικείμενο δένδρου (tree)* του Git είναι αρκετά όμοιο με έναν κατάλογο σε ένα σύστημα αρχείων. Οι κατάλογοι σε ένα σύστημα αρχείων έχουν αναφορές σε άλλους καταλόγους και αρχεία, ενώ τα δένδρα του Git έχουν αναφορές σε άλλα δένδρα και σε blobs. Η μορφή του περιεχομένου ενός αντικειμένου δένδρου φαίνεται στο Σχήμα 2.2.



Σχήμα 2.2: Η μορφή περιεχομένου ενός αντικειμένου δένδρου

Το Git δημιουργεί ένα δένδρο χρησιμοποιώντας την κατάσταση της περιοχής προς υποβολή (staged area ή αλλιώς index) και κατασκευάζοντας μία σειρά από αντικείμενα δένδρου από αυτήν. Οπότε θα πρέπει να εξετάσουμε το αρχείο index.

2.2.3 Περιοχή προς καταχώρηση - Index

Το *index* είναι ουσιαστικά μία "περιοχή αναμονής" για αλλαγές που θα υποβληθούν την επόμενη φορά που θα γίνει υποβολή. Σε αντίθεση λοιπόν με άλλα ΣΕΕ, μία διαδικασία υποβολής στο Git δεν αποθηκεύει κατευθείαν την κατάσταση στην οποία βρίσκεται ο κατάλογος εργασίας. Το index επιτρέπει την επιλογή μερών του καταλόγου εργασίας που θα υποβληθούν με την δημιουργία του επόμενου αντικειμένου υποβολής.

Στην πραγματικότητα, το index λειτουργεί ως ένα "εικονικό δένδρο":

- Περιλαμβάνει αναφορές σε blobs
- Έχει μορφή κατάλληλη για καλύτερες επιδόσεις

- Το Git πρέπει να μπορεί γρήγορα και αποδοτικά να υπολογίζει αν η κατάσταση του καταλόγου εργασίας έχει αλλάξει από την προηγούμενη κατάσταση του index.
- Κάποια εργαλεία, όπως η προτροπή του φλοιού bash, πρέπει να μπορούν να καθορίζουν γρήγορα αν ο κατάλογος εργασίας είναι "καθαρός" ή όχι.

Στο index αποθηκεύονται όχι μόνο ονόματα αρχείων (σε μορφή σχετικού μονοπατιού), αλλά επίσης και ώρα τελευταίας τροποποίησης των αρχείων αυτών. Επιπλέον αποθηκεύονται οι κωδικοί κατακερματισμού κάθε blob. Επομένως το index μπορεί να ενημερώνεται όταν ένα αρχείο επιστρέφει σε μία προηγούμενη κατάσταση, αλλά με μεταγενέστερη χρονοσφραγίδα.

Μία ακριβής μορφή του περιεχομένου του αρχείου index μπορεί να βρεθεί στο εγχειρίδιο τεκμηρίωσης, στον πηγαίο κώδικα του Git [4].

Χρησιμοποιώντας την εντολή `git status` μπορεί κανείς να δει μονοπάτια που διαφέρουν μεταξύ του index και του αντικειμένου υποβολής στο οποίο δείχνει το αρχείο `HEAD`, μονοπάτια που διαφέρουν μεταξύ του καταλόγου εργασίας και του index, και μονοπάτια του καταλόγου εργασίας που το Git δεν παρακολουθεί (και που δεν περιέχονται στο αρχείο `.gitingore`, διαφορετικά, αυτά απλά αγνοούνται από το Git). Τα πρώτα μονοπάτια είναι αυτά που θα υποβάλλονταν αν εκτελούνταν η εντολή `git commit`. Τα δεύτερα και τρίτα μονοπάτια είναι αυτά που θα μπορούσαν να υποβληθούν, αν πριν την εντολή `git commit` εκτελεστεί η εντολή `git add`, ώστε τα μονοπάτια να καταχωρηθούν στο index προς υποβολή. Δηλαδή στο index μπορούν να προστεθούν αρχεία με την εντολή `git add`.

2.2.4 Αντικείμενα υποβολής (Commits)

Τώρα που εξηγήσαμε τι είναι τα blobs, τα δένδρα και το index, μπορούμε να μελετήσουμε τη βασική μονάδα του γράφου του Git, τα *αντικείμενα υποβολής (commits)*. Αυτά είναι σαν δείκτες σε "στιγμιότυπα" ενός συστήματος αρχείων.

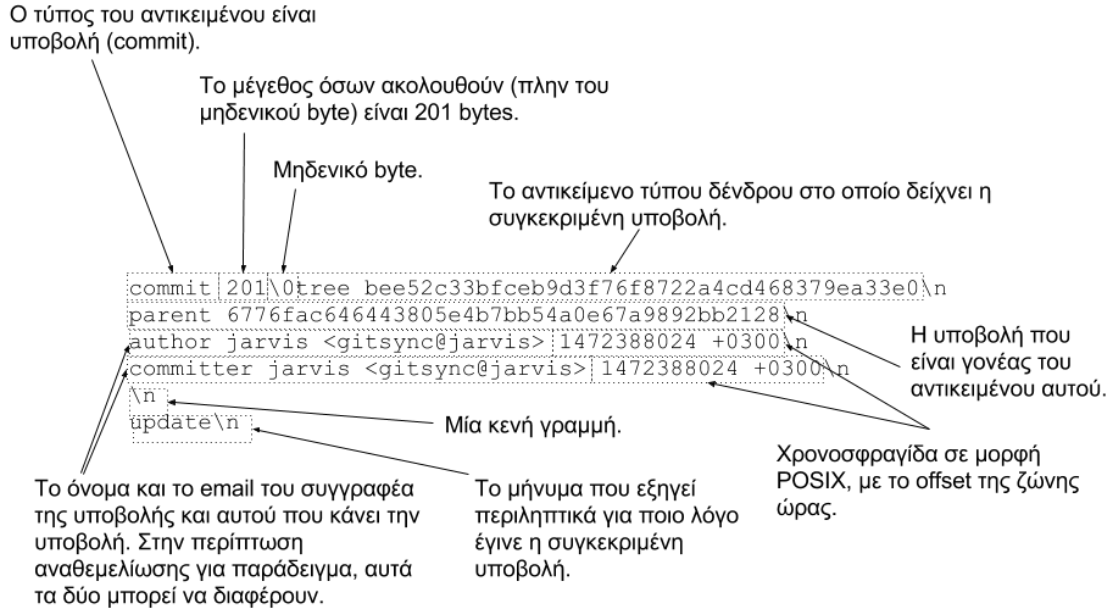
Όταν εκτελείται η εντολή `git commit` (ώστε να υποβληθούν οι τροποποιήσεις στο αποθετήριο του Git), το Git δημιουργεί ένα καινούριο αντικείμενο υποβολής, το οποίο και αποθηκεύεται στο αποθετήριο. Ένα τέτοιο αντικείμενο πρέπει τουλάχιστον να περιλαμβάνει:

- Τον κωδικό κατακερματισμού του αντικειμένου δένδρου που αντιπροσωπεύει την κατάσταση του index τη στιγμή που γίνεται η υποβολή.
- Το όνομα και το email αυτού που έγραψε τις αλλαγές που θα επιφέρει η υποβολή, μαζί με ημερομηνία/ώρα.
- Το όνομα και το email αυτού που έκανε την υποβολή, μαζί με ημερομηνία/ώρα.
- Ένα κείμενο - σχόλιο, που συνοψίζει τον λόγο για τον οποίο έγινε η υποβολή (για παράδειγμα "Υλοποίηση του χαρακτηριστικού με αριθμό #412")

Η μορφή του περιεχομένου ενός αντικειμένου υποβολής φαίνεται στο Σχήμα 2.3.

2.2.5 Αντικείμενα ετικέτας (Tags)

Τα *αντικείμενα ετικέτας (tags)* είναι σαν δείκτες σε υποβολές. Μπορούν να χρησιμοποιηθούν για να διευκολύνουν την αναφορά σε συγκεκριμένες υποβολές. Υπάρχουν δύο ήδη ετικετών που χρησιμοποιούνται από το Git. Οι *ελαφριές ετικέτες (lightweight tags)* και τα *annotated tags*.



Σχήμα 2.3: Η μορφή περιεχομένου ενός αντικειμένου υποβολής

Μία *ελαφριά ετικέτα* είναι σαν ένας κλάδος (branch) που δεν μεταβάλλεται - είναι απλά δείκτης σε μία συγκεκριμένη υποβολή. Δεν έχουμε μιλήσει ακόμα για κλάδους, οπότε δεν θα ασχοληθούμε εδώ με τέτοιου είδους ετικέτες.

Από την άλλη, τα *annotated tags* αποθηκεύονται σαν πλήρη αντικείμενα στην βάση δεδομένων του Git. Όπως όλα τα άλλα αντικείμενα, έχουν κωδικό κατακερματισμού, και περιλαμβάνουν τις ακόλουθες πληροφορίες:

- Το όνομα της ετικέτας (όπως 1.0beta)
- Τον κωδικό κατακερματισμού ενός αντικειμένου υποβολής στο οποίο αναφέρονται (πχ 126af20)
- Ένα μήνυμα σχετικό με την ετικέτα (πχ “Αυτή είναι η ετικέτα 1.0beta”)
- Το όνομα αυτού που δημιούργησε την ετικέτα, την διεύθυνση email του και την ώρα που δημιουργήθηκε η ετικέτα.

2.2.6 Packfiles

Ο τρόπος που έχουμε περιγράψει μέχρι στιγμής, σύμφωνα με τον οποίο το Git αποθηκεύει τα αντικείμενα έχει ένα μειονέκτημα: τη δέσμευση μεγάλης ποσότητας αποθηκευτικού χώρου για την αποθήκευση των blobs, παρόλο που χρησιμοποιείται η βιβλιοθήκη zlib για να γίνει συμπίεση.

Για να γίνει εμφανές το πρόβλημα, μπορούμε να δημιουργήσουμε ένα μεγάλο αρχείο `foo` και να χρησιμοποιήσουμε την εντολή `git add` για να το καταχωρήσουμε ως blob στη βάση δεδομένων του Git. Έπειτα, τροποποιούμε μία λέξη στο περιεχόμενο του αρχείου `foo` και εκτελούμε πάλι την εντολή `git add`. Στην περίπτωση αυτή το Git θα δημιουργήσει ένα καινούριο blob, παρόμοιο με το προηγούμενο, αλλά όχι ίδιο. Έτσι δεσμεύεται πολύς αποθηκευτικός χώρος για μία τόσο μικρή τροποποίηση.

Η αρχική μορφή με την οποία το Git αποθηκεύει τα αντικείμενα του στο δίσκο (και την οποία περιγράψαμε) καλείται *απακετάριστη* μορφή αντικειμένων. Παρόλα αυτά, σε κάποιες περιπτώσεις το Git ομαδοποιεί μερικά από αυτά τα αντικείμενα σε ένα ενιαίο δυαδικό αρχείο, το οποίο αποκαλείται *packfile (πακέτο αρχείων)*, έτσι ώστε να καταναλώνεται λιγότερος αποθηκευτικός χώρος και να είναι πιο αποδοτική η διαχείρισή τους. Μερικές από τις περιπτώσεις στις οποίες κάτι τέτοιο συμβαίνει είναι όταν υπάρχουν πάρα πολλά απακετάριστα αντικείμενα, όταν εκτελείται η εντολή `git gc`, ή όταν γίνεται push σε ένα απομακρυσμένο εξυπηρετητή.

Σύμφωνα με το εγχειρίδιο τεκμηρίωσης της εντολής `git-pack-object`, τα packfiles είναι ένας αποδοτικός τρόπος μεταφοράς ενός συνόλου αντικειμένων μεταξύ δύο αποθετηρίων, καθώς επίσης και μία αποδοτική μορφή αρχειοθέτησης. Σε ένα packfile, ένα αντικείμενο αποθηκεύεται είτε ολόκληρο συμπιεσμένο, είτε ως διαφορά με βάση ένα άλλο αντικείμενο. Η τελευταία μορφή συχνά αποκαλείται και *delta*. Η μορφή ενός packfile αρχείου (.pack) είναι τέτοια ώστε το packfile να είναι αυτάρκες αρχείο, και να μπορεί να ξεπακεταριστεί χωρίς να χρειάζονται επιπλέον πληροφορίες. Αυτό σημαίνει ότι κάθε ένα αντικείμενο το οποίο χρησιμοποιείται ως βάση για κάποιο delta πρέπει να περιέχεται στο packfile. Εξαίρεση αποτελεί το *λεπτό (thin) packfile*, στο οποίο περιέχονται deltas χωρίς να περιέχονται τα αντικείμενα που χρησιμοποιούνται ως βάση για την παραγωγή τους. Στην περίπτωση αυτή, όποιος ξεπακετάρει το packfile πρέπει να έχει ήδη σε κάποια μορφή τα αντικείμενα που χρησιμοποιούνται ως βάση και δεν συμπεριλαμβάνονται στο packfile. Ένα *αρχείο-ευρετήριο για packfile (.idx)* παράγεται για να είναι εφικτή η γρήγορη, τυχαία προσπέλαση στα αντικείμενα του packfile.

Τα packfiles αποθηκεύονται κάτω από τον κατάλογο `.git/objects/pack/`, και όταν δημιουργούνται, τα αντικείμενα που περιέχουν δεν χρειάζονται πια στην απακετάριστη μορφή τους και διαγράφονται από το σύστημα αρχείων. Το όνομα ενός packfile έχει τη μορφή `pack-<κωδικός κατακερματισμού>`, όπου σύμφωνα με τον πηγαίο κώδικα του Git [6], ο <κωδικός κατακερματισμού> προκύπτει από τα ονόματα των αντικειμένων που περιέχονται στο packfile, αφού πρώτα αυτά ταξινομηθούν.

Ωστόσο, υπάρχει όριο στο πλήθος των αρχείων που μπορεί να περιλαμβάνει ένα packfile. Όταν λοιπόν υπάρχουν πολλά απακετάριστα αντικείμενα, μπορεί να πρέπει να δημιουργηθούν περισσότερα από ένα packfiles. Στην περίπτωση αυτή, σύμφωνα με τους δημιουργούς του Git [5], το πόσο *πρόσφατο* είναι ένα αντικείμενο είναι το βασικό κριτήριο που χρησιμοποιεί το Git ώστε να πακετάρει τα αντικείμενα με αποδοτικό τρόπο. Όταν τα πιο πρόσφατα αντικείμενα βρίσκονται στην κορυφή του packfile, τότε τα μοτίβα Εισόδου/Εξόδου είναι καλύτερα.

Συγκεκριμένα, ο ακόλουθος ευριστικός αλγόριθμος χρησιμοποιείται για την ταξινόμηση των αντικειμένων στο packfile:

1. Αρχικά γίνεται ταξινόμηση των αντικειμένων ως προς το *είδος*.
2. Έπειτα γίνεται ταξινόμηση ως προς το *όνομα αρχείου/καταλόγου*.
3. Αν το packfile που παράγεται είναι λεπτό, τότε τα αντικείμενα που *δεν* θα συμπεριληφθούν στο packfile ταξινομούνται νωρίτερα από τα άλλα αντικείμενα.
4. Τέλος γίνεται ταξινόμηση με βάση το *μέγεθος*, κατά φθίνουσα σειρά.

Η λογική με την οποία επιλέγεται η παραπάνω ταξινόμηση είναι η εξής:

1. Δεν παράγεται delta μεταξύ αντικειμένων διαφορετικού τύπου.
2. Προτιμάται τα deltas να προκύπτουν από αντικείμενα με ίδιο μονοπάτι, ωστόσο είναι δυνατόν να δημιουργηθεί delta μεταξύ δύο αρχείων με το ίδιο όνομα, αλλά που βρίσκονται σε διαφορετικούς καταλόγους.

3. Προτιμάται πάντα να παράγονται deltas με βάση αντικείμενα τα οποία δεν θα αποσταλούν, αν υπάρχουν τέτοια.

4. Προτιμάται τα deltas να έχουν ως βάση αντικείμενα με μεγάλο μέγεθος, ώστε να προκύπτουν πολλές αφαιρέσεις.

Το ξεκίνημα γίνεται με μία λίστα από αντικείμενα προς πακετάρισμα, και έπειτα γίνεται ταξινόμηση με βάση τον παραπάνω ευριστικό αλγόριθμο. Στην ουσία είναι μία λεξικογραφική ταξινόμηση πάνω στην τούπλα (τύπος, όνομα, μέγεθος). Έπειτα, διασχίζεται η λίστα, και υπολογίζεται το delta κάθε αντικειμένου χρησιμοποιώντας ως βάση τα n τελευταία αντικείμενα. Το n είναι παραμετροποιήσιμο. Από αυτά τα deltas επιλέγεται το μικρότερο. Τέλος, αφού έχει αποφασιστεί για κάθε αντικείμενο αν θα αναπαρίσταται ως delta ή πλήρες, γίνεται ταξινόμηση με βάση το πόσο πρόσφατο είναι ένα αντικείμενο, με τα πιο πρόσφατα αντικείμενα να προηγούνται των υπολοίπων. Αυτή είναι και η σειρά με την οποία γράφονται τα αντικείμενα στο packfile.

Στο Σχήμα 2.4 φαίνεται η μορφή περιεχομένου ενός αντικειμένου packfile.

2.2.7 Ευρετήρια για packfiles

Όπως αναφέρθηκε προηγουμένως, τα αρχεία `.idx` χρησιμοποιούνται για την επίτευξη γρήγορης, τυχαίας προσπέλασης στα αντικείμενα που περιέχονται σε ένα packfile. Δεν θα ασχοληθούμε περαιτέρω με το πώς αυτό επιτυγχάνεται, δεδομένου ότι για το σκοπό μας δεν θα χρειαστεί να χρησιμοποιήσουμε packfiles. Ωστόσο, στο Σχήμα 2.5 παρατίθεται η δομή ενός αρχείου `.idx`.

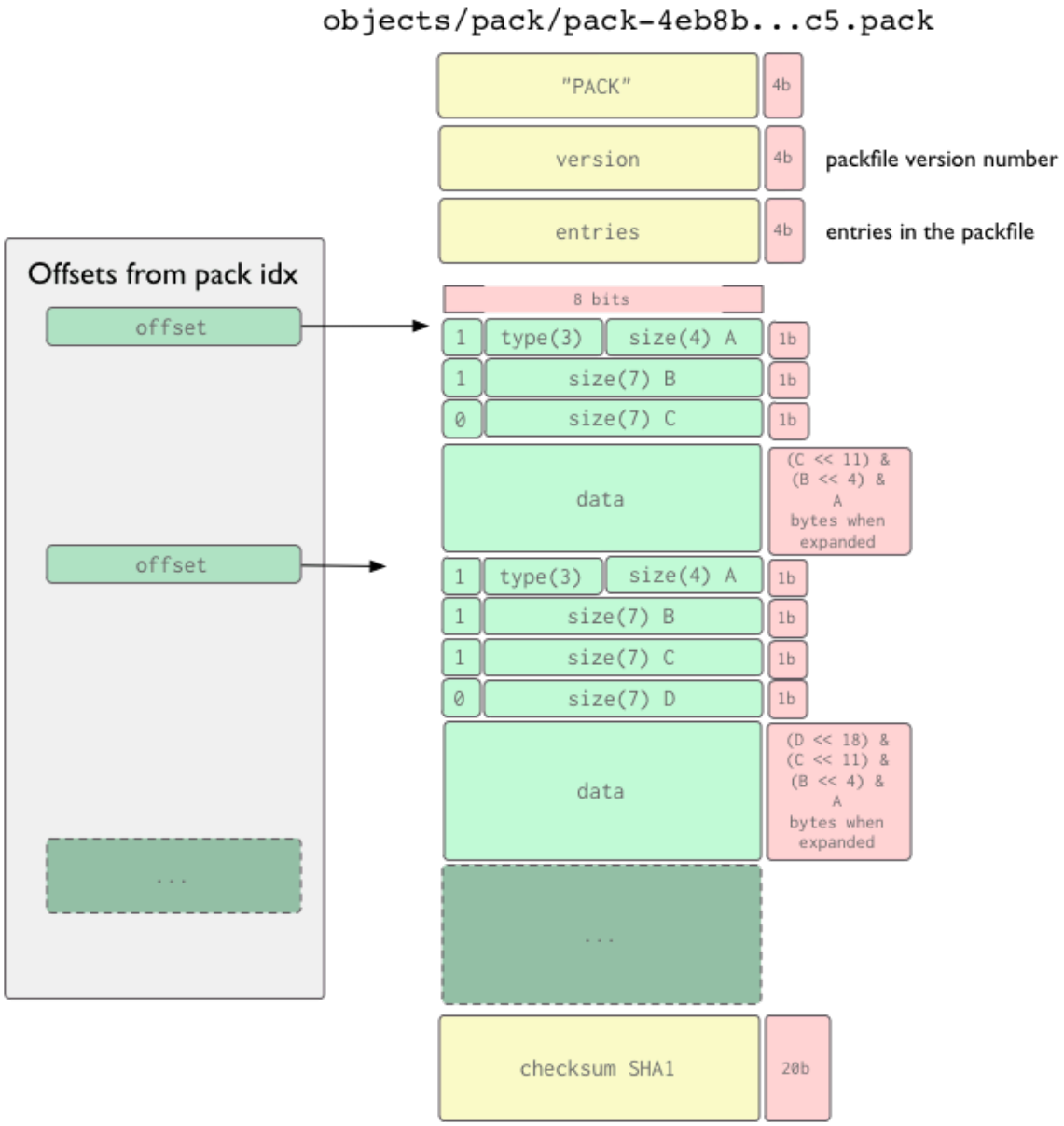
2.3 Αναφορές του Git

Οι *αναφορές (references)* του Git είναι δείκτες σε υποβολές. Είναι αρχεία που αποθηκεύονται στον κατάλογο `.git/refs/` και περιέχουν τον κωδικό κατακερματισμού του αντικειμένου υποβολής στο οποίο δείχνουν. Όπως έχει ήδη αναφερθεί, οι ελαφριές ετικέτες είναι αναφορές. Οι κλάδοι (branches) που χρησιμοποιούνται τόσο συχνά στη ροή εργασίας του Git, είναι επίσης αναφορές. Η διαφορά μεταξύ των κλάδων και των ετικετών αυτού του τύπου είναι ότι οι δεύτερες δεν μετακινούνται ποτέ - δείχνουν συνέχεια στην ίδια υποβολή, και παρέχουν ένα πιο φιλικό τρόπο αναφοράς σε αυτήν.

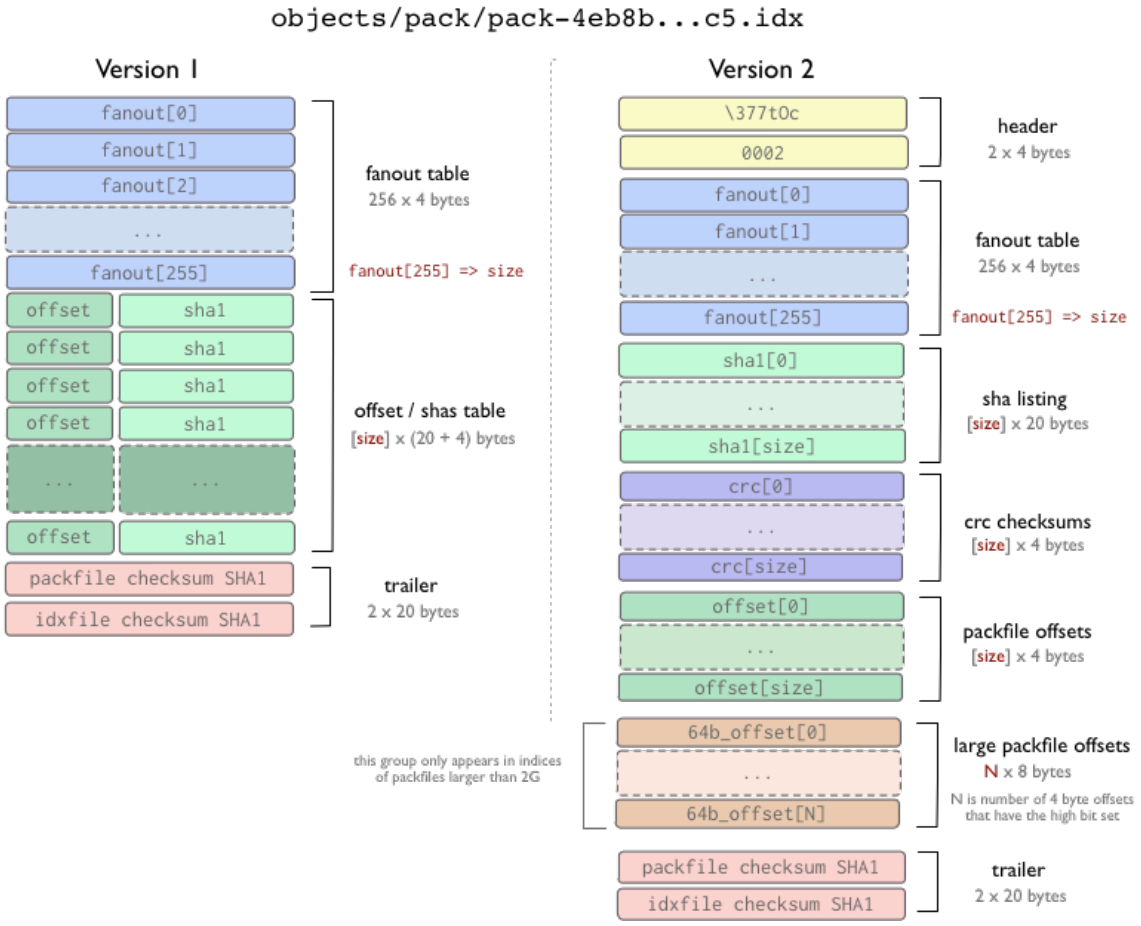
2.3.1 Τοπικές αναφορές

Οι τοπικοί κλάδοι αποθηκεύονται ως απλά αρχεία στον κατάλογο `.git/refs/heads/`. Το όνομα του κλάδου χρησιμοποιείται ως όνομα για το αρχείο που χρησιμοποιείται για την αποθήκευσή του, και το περιεχόμενο του αρχείου είναι απλά ο κωδικός κατακερματισμού της υποβολής στην οποία η αναφορά δείχνει.

Με την εντολή `git branch <όνομα κλάδου>`, το Git χρησιμοποιεί στο παρασκήνιο την εντολή `git update-ref` για τη δημιουργία ενός αρχείου στον κατάλογο `.git/refs/heads/` με το όνομα <όνομα κλάδου>. Ωστόσο πρέπει να υπάρχει ένας τρόπος ώστε το Git να γνωρίζει τον κωδικό κατακερματισμού της τελευταίας υποβολής, ώστε να μπορεί να τον χρησιμοποιήσει ως περιεχόμενο του αρχείου αναφοράς που δημιούργησε. Γιαυτό το σκοπό υπάρχει το αρχείο `HEAD`.



Σχήμα 2.4: Η μορφή περιεχομένου ενός αντικειμένου packfile



Σχήμα 2.5: Η μορφή περιεχομένου ενός ευρετηρίου για packfile (`.idx` αρχείου)

2.3.2 Το αρχείο HEAD

Το *αρχείο HEAD* αποθηκεύεται στον κατάλογο `.git/` και είναι μία *συμβολική αναφορά (symbolic reference)* στον κλάδο στον οποίο βρίσκεται ο χρήστης εκείνη τη στιγμή. Με τον όρο συμβολική αναφορά εννοούμε ότι σε αντίθεση με μία συνηθισμένη αναφορά, στη γενική περίπτωση δεν περιέχει έναν κωδικό κατακερματισμού, αλλά έναν δείκτη σε κάποια άλλη αναφορά. Με μια ματιά στο αρχείο `HEAD`, στη γενική περίπτωση το περιεχόμενο θα μοιάζει όπως στο Listing 2.2.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Listing 2.2: Παράδειγμα περιεχομένων του αρχείου `.git/HEAD`

Ωστόσο το αρχείο `HEAD` μπορεί να είναι επίσης ένας άμεσος δείκτης σε μία υποβολή. Το Git ονομάζει μία τέτοια περίπτωση ως κατάσταση *αποκομμένου HEAD (detached HEAD state)*. Κάτι τέτοιο μπορεί να συμβεί όταν χρησιμοποιείται η εντολή `git checkout <κωδικός κατακερματισμού μιας υποβολής>`.

2.3.3 Απομακρυσμένες αναφορές

Αντίστοιχα με τους τοπικούς κλάδους (heads), υπάρχουν και απομακρυσμένοι κλάδοι, οι οποίοι αποκαλούνται *απομακρυσμένες αναφορές (remote references)*, και αποθηκεύονται στον κατάλογο `.git/refs/remotes/`. Χρησιμοποιούνται ώστε να είναι γνωστό που έδειχνε ο κλάδος ενός απομακρυσμένου αποθετηρίου την τελευταία φορά που το Git επικοινώνησε με αυτό. Για παράδειγμα οι εντολές `git fetch` και `git push` ενημερώνουν τις απομακρυσμένες αναφορές με βάση τις *προδιαγραφές αναφορών (refspec)*. Θα μιλήσουμε για το refspec στην επόμενη ενότητα.

Οι απομακρυσμένες αναφορές διαφέρουν από τις τοπικές στο γεγονός ότι οι πρώτες θεωρούνται προσπελάσιμες μόνο για ανάγνωση. Η μετάβαση στην υποβολή που αυτές δείχνουν είναι εφικτή, αλλά το αρχείο `HEAD` δεν θα δείχνει ποτέ σε μία από αυτές, οπότε δεν είναι δυνατή η ενημέρωση μιας απομακρυσμένης αναφοράς με μία εντολή υποβολής. Το Git τις διαχειρίζεται περισσότερο ως σελιδοδείκτες στην τελευταία γνωστή κατάσταση στην οποία βρίσκονταν οι κλάδοι αυτοί στα απομακρυσμένα αποθετήρια.

2.3.4 Refspec

Το *refspec (προδιαγραφές αναφορών)* έχει τη μορφή `<πηγή>:<προορισμός>` και είναι ουσιαστικά μία αντιστοίχιση από αναφορές. Στην περίπτωση του *fetch*, ως `<πηγή>` αναφέρεται το μονοπάτι όπου αποθηκεύονται οι τοπικές αναφορές στο *απομακρυσμένο αποθετήριο*, και ως `<προορισμός>` το μονοπάτι που χρησιμοποιείται από το *τοπικό αποθετήριο* για την αποθήκευση των αναφορών αυτών. Στην περίπτωση του *push*, ο όρος `<πηγή>` αναφέρεται σε μονοπάτι του *τοπικού αποθετηρίου*, και ο όρος `<προορισμός>` σε μονοπάτι του *απομακρυσμένου αποθετηρίου*. Είναι ένας τρόπος να γνωρίζει το Git ποιες αναφορές πρέπει να ενημερώσει κατά τη διάρκεια ενός push/fetch. Στο αρχείο `.git/config` υπάρχει ένας τομέας παρόμοιος με αυτόν του Listing 2.3.

Το περιεχόμενο του Listing 2.3 δηλώνει ότι ως προεπιλογή, αν γίνει fetch από το απομακρυσμένο αποθετήριο με το όνομα `origin`, το Git θα ενημερώσει τοπικά όλες τις απομακρυσμένες αναφορές που βρίσκονται στον κατάλογο `.git/refs/remotes/origin/`, χρησιμοποιώντας ως πηγή πλη-

```
[remote ”origin”]
    url = ...
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Listing 2.3: Παράδειγμα ρυθμίσεων απομακρυσμένου αποθετηρίου στο αρχείο `.git/config`

ροφορίας τις αναφορές που το απομακρυσμένο αποθετήριο έχει αποθηκευμένες στον κατάλογο `.git/refs/heads/`. Αντίστοιχα, αν γίνει push στο απομακρυσμένο αποθετήριο με το όνομα origin, το Git θα χρησιμοποιήσει την αναφορά `.git/refs/heads/master` ως πηγή πληροφορίας για να ενημερώσει την αναφορά `.git/refs/heads/qa/master` του απομακρυσμένου αποθετηρίου.

2.4 Δημιουργία αποθετηρίου του Git

Το Git είναι ένα κατανεμημένο ΣΕΕ. Το γεγονός ότι παρέχει σε προχωρημένους χρήστες πολλές δυνατότητες, δεν σημαίνει ότι και χρήστες με λιγότερες γνώσεις σχετικά με την εσωτερική οργάνωση του Git δεν μπορούν να το χρησιμοποιήσουν. Θα εξηγήσουμε ορισμένες πολύ συνηθισμένες εντολές του Git, που χρησιμοποιούνται συχνά στη ροή εργασία με αυτό. Το αποτέλεσμα μιας τέτοιας ροής εργασίας είναι η δημιουργία ενός γράφου, που αποτελείται κυρίως από αντικείμενα υποβολής, και από αναφορές σε τέτοια αντικείμενα. Θα εξετάσουμε επίσης πώς το Git αντιμετωπίζει τις διενέξεις που μπορεί να προκύψουν κατά την απόπειρα συγχώνευσης (merging).

Αρχικά, πρέπει να δημιουργηθεί ένα αποθετήριο του Git. Υπάρχουν δύο τρόποι για να γίνει κάτι τέτοιο. Ο ένας είναι να δημιουργηθεί πρώτα ένα κενό αποθετήριο και μετά να ενημερωθεί αντλώντας πληροφορίες από ένα άλλο απομακρυσμένο αποθετήριο. Ο δεύτερος είναι να δημιουργηθεί το καινούριο αποθετήριο κατευθείαν ως *κλώνος* ενός άλλου απομακρυσμένου αποθετηρίου.

Στον πρώτο τρόπο, αρχικά χρησιμοποιείται η εντολή `git init <local_path>`, όπου `<local_path>` είναι ένα μονοπάτι που οδηγεί στον κατάλογο που δημιουργείται το αποθετήριο. Έπειτα, με την εντολή `git remote add origin <remote_url>`, ρυθμίζεται και προστίθεται ένα απομακρυσμένο αποθετήριο. Αυτό σημαίνει ότι αντί να χρησιμοποιείται η διεύθυνση του απομακρυσμένου αποθετηρίου κάθε φορά που γίνεται αναφορά σε αυτό, μπορεί να χρησιμοποιηθεί αντ᾽ αυτού το αυθαίρετο όνομα `origin`. Τέλος, με τη χρήση της εντολής `git pull origin`, το αποθετήριο που δημιουργήθηκε ενημερώνεται με τις τελευταίες αλλαγές που έχουν γίνει στο απομακρυσμένο αποθετήριο με το όνομα `origin`. Θα εξηγηθεί αργότερα πώς λειτουργεί η εντολή `git pull`.

Η μέθοδος της κλωνοποίησης βασίζεται στην εντολή `git clone <remote_url>`. Στην περίπτωση αυτή, το Git θα δημιουργήσει ένα αποθετήριο σε έναν τοπικό κατάλογο του οποίου το όνομα εξαρτάται από το όνομα του απομακρυσμένου αποθετηρίου που κλωνοποιείται. Επιπλέον, το Git θα ρυθμίσει το απομακρυσμένο αποθετήριο και θα το προσθέσει με το όνομα `origin`, και έπειτα θα χρησιμοποιήσει την εντολή `git pull origin` για να λάβει αυτόματα τις τελευταίες ενημερώσεις.

Με όποιον τρόπο και να δημιουργηθεί το αποθετήριο, το Git χρειάζεται ορισμένες επιπλέον πληροφορίες σχετικά με το χρήστη, όπως το όνομα και το email του. Αυτά θα χρησιμοποιηθούν σε περιπτώσεις όπως η δημιουργία αντικειμένων υποβολής και ετικέτας, και γιαυτό είναι απαραίτητα. Για να γνωστοποιηθούν οι πληροφορίες αυτές στο Git χρησιμοποιούνται οι ακόλουθες εντολές:
`git config user.name “<ονοματεπώνυμο χρήστη>”`
`git config user.email “<email χρήστη>”`

2.4.1 Βασική ροή εργασίας με το Git

Έχοντας ένα τοπικό αποθετήριο με ρυθμισμένο ένα απομακρυσμένο αποθετήριο ως `origin` (είτε αυτό δημιουργήθηκε με τη μέθοδο της κλωνοποίησης είτε όχι), διαισθητικά μία βασική ροή εργασίας είναι η παρακάτω (προς το παρόν θα αγνοήσουμε τους κλάδους και θα υποθέτουμε ότι εργαζόμαστε στον κλάδο με το όνομα `master`):

1. Γίνονται τροποποιήσεις σε αρχεία του τοπικού αποθετηρίου.
2. Οι τροποποιήσεις αυτές αποθηκεύονται στη βάση δεδομένων του Git.
3. Ενημερώνεται το τοπικό αποθετήριο σχετικά με τις τελευταίες τροποποιήσεις που έχουν γίνει στο απομακρυσμένο αποθετήριο `origin`
4. Ενημερώνεται το απομακρυσμένο αποθετήριο `origin` σχετικά με τις τροποποιήσεις που έγιναν στο τοπικό αποθετήριο.

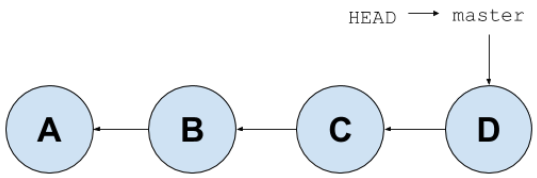
Χρησιμοποιώντας εντολές του Git, τα παραπάνω βήματα επιτυγχάνονται ως εξής:

1. Γίνονται τροποποιήσεις σε αρχεία του τοπικού αποθετηρίου, έστω στα αρχεία `foo` και `bar`. Εδώ δεν χρειάζεται να εμπλακεί το Git.
2. ```
   git add foo bar
   git commit
   ```
3. ```
   git pull origin
   ```
   ή
   ```
   git fetch origin
   git rebase origin/master
   ```
   Η διαφορά στις δύο παραπάνω προσεγγίσεις είναι ότι η πρώτη στηρίζεται στη *συγχώνευση* κλάδων του Git και στην εντολή `git merge`, ενώ η δεύτερη στηρίζεται στην *αναθεμελίωση* κλάδων και στην εντολή `git rebase`. Θα αναλύσουμε περισσότερο τις δύο αυτές προσεγγίσεις, τη συγχώνευση και την αναθεμελίωση, παρακάτω.
4. ```
   git push origin master
   ```

2.4.2 Ο γράφος του Git

Είναι αξιοσημείωτο ότι στο δεύτερο από τα παραπάνω βήματα, το Git ενημερώθηκε για τις αλλαγές που είχαν γίνει τοπικά, με τη χρήση των εντολών `git add` και `git commit`. Η εντολή `git add` θα δημιουργήσει ένα blob στην βάση δεδομένων του Git και θα ενημερώσει το index, δηλώνοντας ότι έχουν γίνει τροποποιήσεις σε αυτά τα αρχεία. Έπειτα, η εντολή `git commit` θα χρησιμοποιήσει το index για να δημιουργήσει ένα αντικείμενο δένδρου (και ενδεχομένως κι άλλα αντικείμενα δένδρου στα οποία αυτό θα δείχνει). Αυτό θα αντιπροσωπεύει το ριζικό κατάλογο του αποθετηρίου. Έπειτα, κατασκευάζεται ένα αντικείμενο υποβολής, το οποίο δείχνει στο δένδρο που δημιουργήθηκε από το index, και το οποίο θα έχει ως γονέα την υποβολή στην οποία δείχνει τη στιγμή της υποβολής το αρχείο `HEAD`. Τέλος, η αναφορά με όνομα `master` θα ενημερωθεί ώστε να δείχνει στο καινούριο αντικείμενο υποβολής. Το αποτέλεσμα πολλών επαναλήψεων του βήματος 2 είναι η δημιουργία ενός γράφου, που αποτελείται από αντικείμενα υποβολής, που δείχνουν σε προηγούμενα αντικείμενα υποβολής (με εξαίρεση την πρώτη υποβολή, η οποία είναι η ρίζα του γράφου και δεν έχει γονέα). Ένας τέτοιος γράφος απεικονίζεται στο Σχήμα 2.6.
   ```

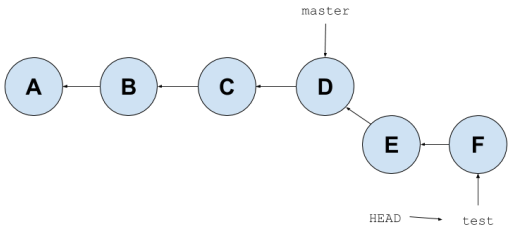
Στο Σχήμα 2.6 φαίνεται ένας γράφος με μόνο έναν κλάδο, με το όνομα `master`. Η δημιουργία περισσότερων κλάδων είναι εύκολη και βασική λειτουργία στη ροή εργασίας με το Git, και επιτυγχάνεται με την εντολή `git branch <όνομα νέου κλάδου>`. Θα εξηγήσουμε πώς λειτουργεί αυτή η εντολή.



Σχήμα 2.6: Ένας απλός γράφος του Git

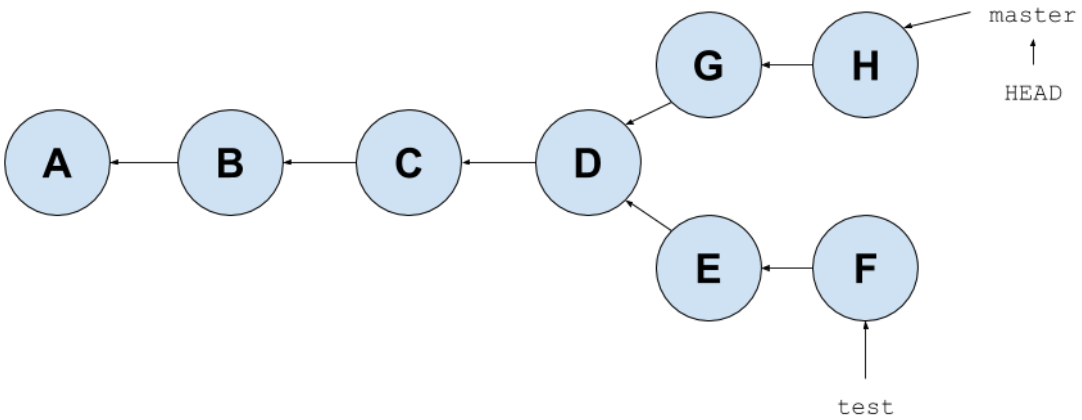
Ας υποθέσουμε ότι βρισκόμαστε στο κλάδο `master` του Σχήματος 2.6. Θα θέλαμε να κάνουμε κάποιες πειραματικές τροποποιήσεις σε αρχεία, χωρίς όμως να επηρεάσουμε τον κλάδο που βρισκόμαστε, εκτός και αν αποφασίσουμε να το κάνουμε αργότερα. Έτσι, δημιουργούμε έναν κλάδο στον οποίο θα εργαστούμε, με το όνομα `test`, χρησιμοποιώντας την εντολή `git branch test`. Δημιουργείται με αυτόν τον τρόπο μία καινούρια αναφορά, που δείχνει στην ίδια υποβολή που έδειχνε και η αναφορά `master`. Έπειτα, για να αλλάξουμε και να εργαστούμε στον καινούριο κλάδο με το όνομα `test`, εκτελούμε την εντολή `git checkout test`.

Πλέον, οι υποβολές θα ενημερώνουν την αναφορά `test`, και η αναφορά `master` θα συνεχίσει να δείχνει στην ίδια υποβολή που έδειχνε και πριν. Επομένως, έπειτα από μερικές υποβολές, ο γράφος δείχνει όπως στο Σχήμα 2.7.



Σχήμα 2.7: Δημιουργία υποβολών στον καινούριο κλάδο `test`

Μπορούμε ακόμη να κάνουμε αλλαγές στον κλάδο `master`, χωρίς να χάσουμε τις αλλαγές που έχουν υποβληθεί όσο εργαζόμασταν στον κλάδο `test`. Για να το επιτύχουμε αυτό, μεταβαίνουμε στον κλάδο `master` με την εντολή `git checkout master`, και κάνουμε ότι αλλαγές και υποβολές θέλουμε. Πλέον ο γράφος είναι όπως στο Σχήμα 2.8.

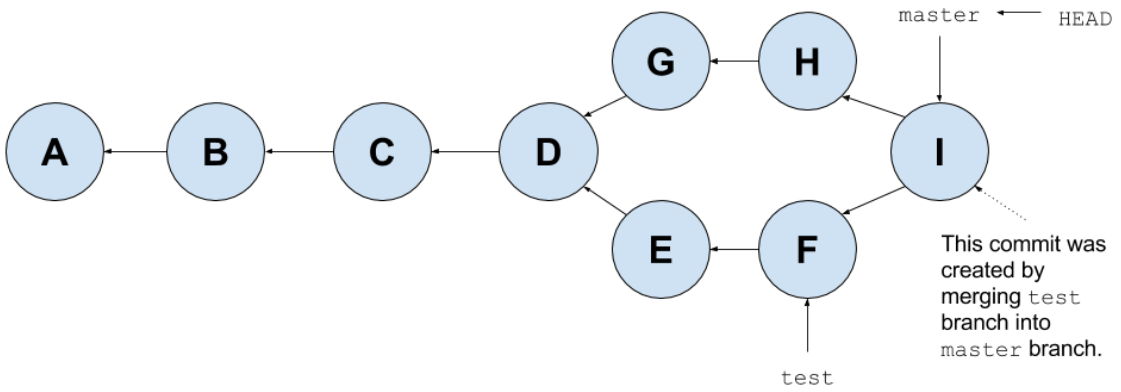


Σχήμα 2.8: Δημιουργία υποβολών στον κλάδο `master`

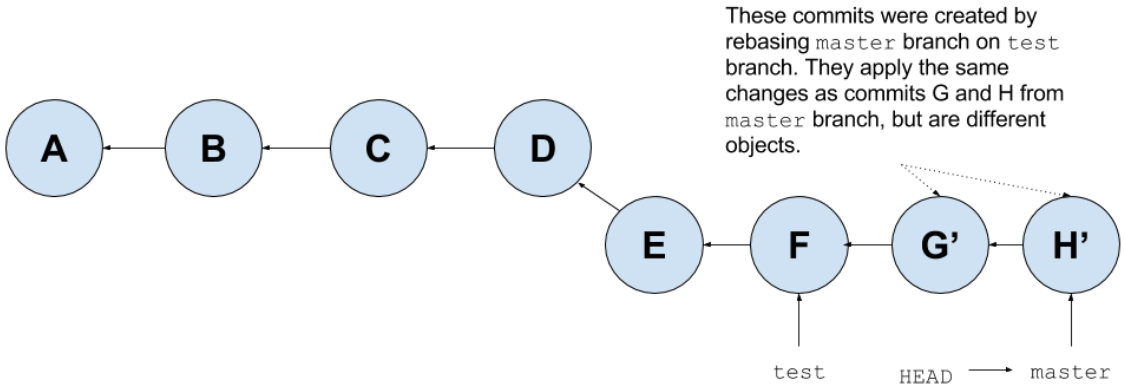
Τώρα αποφασίζουμε ότι οι αλλαγές που έχουμε κάνει στον κλάδο `test` είναι καλές, και τις θέλουμε και στον κλάδο `master`. Σε αυτό το σημείο θα πρέπει να μιλήσουμε για τη *συγχώνευση* και την αναθεμελίωση κλάδων.

2.5 Συγχώνευση και αναθεμελίωση στο Git

Η συγχώνευση στο Git (`git merge`) κάνει ότι ακριβώς το όνομα υποδηλώνει: συγχωνεύει δυο κλάδους. Η αναθεμελίωση (`git rebase`) από την άλλη πλευρά, είναι πιο προχωρημένη έννοια, και έχει πολλές εφαρμογές. Μπορεί να χρησιμοποιηθεί και αντί της συγχώνευσης. Κατά την αναθεμελίωση ο γράφος του git αναδομείται από ένα σημείο κι έπειτα, με την επανυποβολή υποβολών από το σημείο αυτό κι έπειτα. Στο Σχήμα 2.9 και στο Σχήμα 2.10 φαίνεται η διαφορά της συγχώνευσης και της αναθεμελίωσης.



Σχήμα 2.9: Ο γράφος του Σχήματος 2.8 μετά από συγχώνευση του κλάδου `test` στον κλάδο `master`



Σχήμα 2.10: Ο γράφος του Σχήματος 2.8 μετά από αναθεμελίωση του κλάδου `master` στον κλάδο `test`

Όπως φαίνεται στο Σχήμα 2.9, η συγχώνευση δημιουργεί αντικείμενα υποβολής με δύο (ή και περισσότερους) γονείς. Αυτό είναι το αποτέλεσμα μιας *τριμερούς συγχώνευσης (3-way merge)*. Αυτό το είδος συγχώνευσης χρησιμοποιεί και συγκρίνει 3 αντικείμενα υποβολής μεταξύ τους ώστε να παραχθεί μία τέταρτη υποβολή που θα συγχωνεύει τα δύο από αυτά. Επειδή αυτό ακούγεται περίπλοκο, θα περιγράψουμε τη συγχώνευση αρχείων. Η συγχώνευση κλάδων, εν τέλει καταλήγει να είναι συγχώνευση αρχείων και λειτουργεί με τον ίδιο τρόπο.

Ας υποθέσουμε ότι έχουμε δύο διαφορετικές εκδοχές ενός αρχείου `foo`, και θέλουμε να τις συγχωνεύσουμε σε μία μόνο εκδοχή. Με το να συγκριθούν απλά οι δύο εκδοχές μεταξύ τους, γίνονται

εμφανείς οι διαφορές τους. Δεν αντλούμε όμως καμία πληροφορία για το τι έχει αλλάξει σε κάθε εκδοχή. Επομένως αν επιχειρούσαμε μία συγχώνευση με αυτόν τον τρόπο (διμερής συγχώνευση), θα ήταν απαραίτητη η ανθρώπινη παρέμβαση. Ωστόσο, αν υπήρχε άλλη μία εκδοχή του αρχείου `foo`, από την οποία, έπειτα από τροποποιήσεις, απορρέουν οι άλλες δύο εκδοχές, τότε θα μπορούσε αυτή η τρίτη εκδοχή να χρησιμοποιηθεί ως *βάση συγχώνευσης* των άλλων δύο εκδοχών. Δηλαδή θα μπορούσαμε να αντλήσουμε πληροφορία για το τι αλλαγές έκανε η κάθε μία από τις 2 εκδοχές στη βάση συγχώνευσης, και τελικά η ανθρώπινη παρέμβαση μπορεί να αποφευχθεί. Έτσι ακριβώς λειτουργεί η τριμερής συγχώνευση.

Το Git χρησιμοποιεί τον κοινό πρόγονο δύο κλάδων που είναι προς συγχώνευση ως βάση συγχώνευσης. Ο κοινός πρόγονος είναι απλά η υποβολή στην οποία οι δύο κλάδοι στο γράφο διαχωρίζονται. Αν υπάρχουν πολλοί κοινοί πρόγονοι, τους χρησιμοποιεί αναδρομικά για να κατασκευάσει μία καινούρια προσωρινή υποβολή, την οποία και θα χρησιμοποιήσει ως βάση συγχώνευσης. Αυτή είναι η μέθοδος της αναδρομικής συγχώνευσης. Αξίζει να σημειωθεί ότι μετά τη συγχώνευση, η υποβολή που έχει δημιουργηθεί και που την αντιπροσωπεύει είναι ιδιαίτερη, αφού έχει 2 ή και περισσότερους γονείς. Αυτό έχει ως αποτέλεσμα τη δημιουργία πολύπλοκων γράφων, και γιαυτό το λόγο ορισμένοι προτιμούν τη μέθοδο της αναθεμελίωσης.

Κατά την αναδόμηση, όλες οι αλλαγές που βρίσκονται στον τωρινό κλάδο, και όχι στον κλάδο που χρησιμοποιείται ως βάση για την αναδόμηση (θα αποκαλούμε αυτόν τον κλάδο <*θεμέλιο*>) αποθηκεύονται σε έναν προσωρινό χώρο. Έπειτα, ο τωρινός κλάδος επαναφέρεται ώστε να δείχνει στην υποβολή που δείχνει κλάδος <θεμέλιο> (με την εντολή `git reset --hard <θεμέλιο>`), και οι υποβολές που είχαν αποθηκευτεί στον προσωρινό χώρο επανυποβάλλονται στον τωρινό κλάδο, με τη σειρά. Το αποτέλεσμα είναι ότι ο τωρινός κλάδος πλέον έχει και τις υποβολές του κλάδου <θεμέλιο>, όπως φαίνεται και στο Σχήμα 2.10.

Όπως υπονοήθηκε νωρίτερα, με την αναθεμελίωση το ιστορικό των υποβολών (και εν τέλει ο γράφος) είναι πιο καθαρό, αφού δεν υπάρχουν υποβολές με πολλαπλούς γονείς.

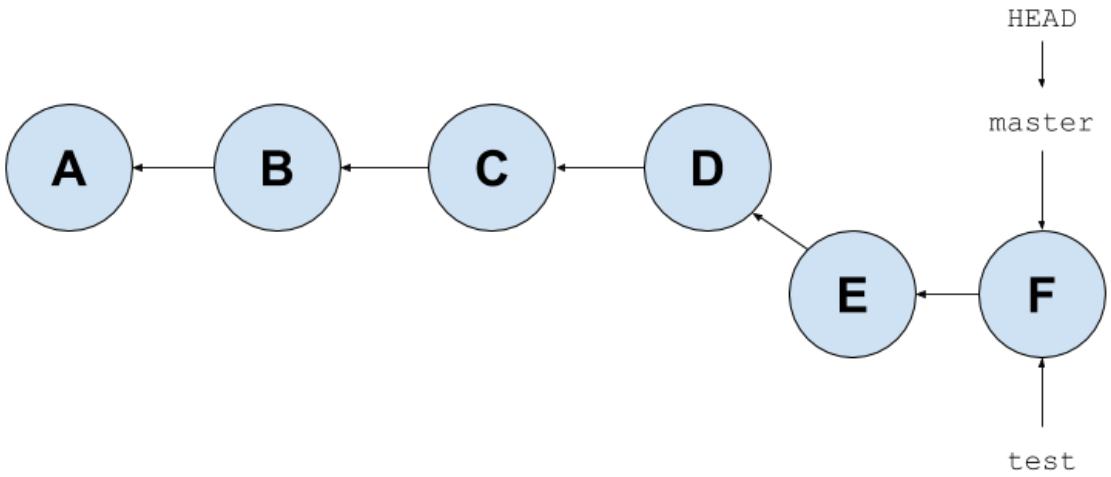
2.5.1 Συγχώνευση γρήγορης προώθησης (fast-forward merge)

Έχουμε αναφέρει ότι το αποτέλεσμα της συγχώνευσης είναι μία υποβολή με πολλαπλούς γονείς. Ωστόσο αυτό δεν συμβαίνει πάντοτε.

Αν θέλουμε να συγχωνεύσουμε δύο κλάδους, και η υποβολή στην οποία δείχνει ο κλάδος στον οποίο βρισκόμαστε μπορεί να προσπελαστεί ακολουθώντας μόνο γονείς της υποβολής του κλάδου που θέλουμε να συγχωνεύσουμε, τότε το Git απλοποιεί την κατάσταση και απλά μετακινεί τον κλάδο που βρισκόμαστε ώστε να δείχνει όπου δείχνει ο κλάδος προς συγχώνευση, δεδομένου ότι οι δύο κλάδοι δεν διαχωρίζονται. Αυτό ονομάζεται *συγχώνευση γρήγορης προώθησης (fast-forward merge)*. Ένα τέτοιο παράδειγμα είναι η απόπειρα συγχώνευσης του κλάδου `test` στον κλάδο `master` όταν ο γράφος έχει τη μορφή του Σχήματος 2.7. Στην περίπτωση αυτή, το αποτέλεσμα της συγχώνευσης φαίνεται στο Σχήμα 2.11.

2.5.2 Διενέξεις στο Git (Git conflicts)

Κατά τη διάρκεια της αναθεμελίωσης ή της συγχώνευσης, μπορεί να προκύψουν διενέξεις αρχείων. Και στις δύο περιπτώσεις, ο τρόπος επίλυσης είναι παρόμοιος, αφού η αναθεμελίωση ανάγεται κατά μία έννοια σε πολλές συγχωνεύσεις μεμονωμένων υποβολών. Θα εξηγήσουμε λοιπόν την ανίχνευση και επίλυση διενέξεων κατά τη διάρκεια της συγχώνευσης, χρησιμοποιώντας παράδειγμα, το οποίο είναι μάλιστα σύνηθες σενάριο κατά τη ροή εργασίας με το Git.



Σχήμα 2.11: Συγχώνευση γρήγορης προώθησης του κλάδου `test` στον κλάδο `master` του Σχήματος 2.7

Δημιουργούμε αρχικά 3 τοπικούς καταλόγους `test1/`, `test2/` και `remote/`. Οι κατάλογοι `test1/` και `test2/` θα χρησιμοποιηθούν ως κατάλογοι εργασίας, δύο διαφορετικές τοποθεσίες που θέλουμε όμως να παραμένουν συγχρονισμένες. Ο κατάλογος `remote/` θα διαδραματίζει το ρόλο ενός απομακρυσμένου κεντρικού αποθετηρίου, με το οποίο θα επικοινωνούν τα αποθετήρια `test1/` και `test2/`, ώστε αυτά να μην χρειάζεται να επικοινωνούν απευθείας μεταξύ τους. Για να το επιτύχουμε αυτό, το αποθετήριο `remote/` θα πρέπει να αρχικοποιηθεί ως *γυμνό* αποθετήριο, χρησιμοποιώντας την εντολή `git init --bare remote/`. Αυτό συμβαίνει γιατί από προεπιλογή το Git δεν επιτρέπει δεν την προώθηση ενημερώσεων απευθείας σε μη-γυμνά αποθετήρια (οπότε το αποθετήριο `test1/` δεν θα μπορούσε να προωθήσει ενημερώσεις απευθείας στο αποθετήριο `test2/`).

Δημιουργούμε λοιπόν το κενό αρχείο `foo` στο `test1/`, και ενημερώνουμε το αποθετήριο `remote/` με τη γνωστή σειρά εντολών `git add, git commit, git push`. Έπειτα, αλλάζουμε στο αποθετήριο `test2/` και το ενημερώνουμε και αυτό με πληροφορία που αντλούμε από το `remote/` με την εντολή `git pull`. Τώρα και τα δύο αποθετήρια που μας ενδιαφέρουν έχουν ένα κενό αρχείο `foo`.

Με σκοπό τη δημιουργία διένεξης, τροποποιούμε το `foo` στο `test1/` (προσθέτοντας τη γραμμή "Αλλαγή από το test1") και ενημερώνουμε το `remote/`. Έπειτα, αλλάζουμε στο `test2/` και χωρίς να το ενημερώσουμε, τροποποιούμε κι εκεί το (κενό ακόμα) `foo` (προσθέτοντας τη γραμμή "Αλλαγή από το test2"). Αφού καταχωρήσουμε την αλλαγή στο αποθετήριο (`git add` και `git commit`), κάνουμε απόπειρα να ενημερώσουμε το `remote/` για τις καινούριες αλλαγές μας. Ωστόσο η εντολή `git push` αποτυγχάνει με το μήνυμα που φαίνεται στο Listing 2.4.

Με απλούς όρους, το πρόβλημα είναι ότι ο κλάδος που προσπαθούμε να προωθήσουμε (`master`) δείχνει σε υποβολή που δεν είναι άμεσος απόγονος της υποβολής στην οποία δείχνει ο κλάδος του απομακρυσμένου αποθετηρίου (`master`).

Για την επίλυση του προβλήματος, το Git προτείνει την εκτέλεση της εντολής `git pull`. Η εντολή αυτή είναι στην ουσία μία σύντμηση των εντολών `git fetch` και `git merge FETCH_HEAD`.

Σύμφωνα με το εγχειρίδιο τεκμηρίωσης την εντολής `git fetch`, η εντολή αυτή θα φέρει κλάδους και ετικέτες (δηλαδή αναφορές) από ένα ή περισσότερα αποθετήρια, μαζί με οποιαδήποτε άλλα αντι-

```
To /remote/
 ! [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.
```

Listing 2.4: Αποτυχία εντολής `git push`

κείμενα είναι απαραίτητα ώστε να ολοκληρωθεί το ιστορικό των αναφορών αυτών. Επιπλέον ενημερώνονται τοπικά οι απομακρυσμένες αναφορές. Επομένως το Git απλά θα φέρει τοπικά κάποια αντικείμενα και κάποιες αναφορές, δεν θα επηρεάσει όμως τον κατάλογο εργασίας. Έπειτα με την εντολή `git merge FETCH_HEAD` γίνεται προσπάθεια να συγχωνευθούν οι τροποποιήσεις που έχουν γίνει στα απομακρυσμένα αποθετήρια (και που πλέον υπάρχουν και τοπικά στη βάση αντικειμένων του Git ως υποβολές) με τον τοπικό κλάδο στον οποίο βρίσκεται ο χρήστης (στην περίπτωσή μας τον κλάδο `master`). Το `FETCH_HEAD` είναι μία προσωρινή αναφορά, που χρησιμοποιείται για την παρακολούθηση όσων η εντολή `git fetch` έφερε στο τοπικό αποθετήριο. Στη συνηθισμένη περίπτωση, ένας κλάδος μόνο έρχεται από το απομακρυσμένο αποθετήριο, οπότε το `FETCH_HEAD` λειτουργεί ως συμβολική αναφορά στον κλάδο αυτό. Τώρα που πραγματικά γίνεται η απόπειρα συγχώνευσης περιμένουμε να προκύψει διένεξη λόγω του αρχείου `foo`. Πράγματι, με την εντολή `git pull` εμφανίζονται όσα μπορεί να δει κανείς στο Listing 2.5.

```
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /remote/
    e8b6c10..8b5f25f  master    -> origin/master
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

Listing 2.5: Διένεξη κατά την απόπειρα συγχώνευσης με το Git

Αυτό σημαίνει ότι η προσπάθεια του Git να συγχωνεύσει αυτόματα το αρχείο `foo` απέτυχε, όπως αναμέναμε. Με τη μέθοδο της τριμερούς συγχώνευσης το Git δεν μπορεί να αποφασίσει ποιες αλλαγές θα πρέπει να διατηρηθούν στην τελική εκδοχή του αρχείου `foo`, αφού και οι δύο εκδοχές έχουν την ίδια γραμμή τροποποιημένη, σε σχέση με την κενή εκδοχή του `foo` που χρησιμοποιείται ως βάση συγχώνευσης. Με ανθρώπινη παρέμβαση λοιπόν, μπορούμε να τροποποιήσουμε το αρχείο `foo`, και αφού το φέρουμε στην επιθυμητή τελική συγχωνευμένη κατάσταση, χρησιμοποιούμε την εντολή `git add foo` για να δηλώσουμε ότι η διένεξη λύθηκε, και την `git commit` για να υποβάλουμε στο αποθετήριο τις αλλαγές.

Είδαμε βέβαια ότι το Git όντως προσπάθησε να επιλύσει τη διένεξη αυτόματα, κάνοντας αυτόματη συγχώνευση. Αν οι αλλαγές δεν είχαν γίνει στην ίδια γραμμή του `foo`, τότε όλες οι τροποποιήσεις από όλες τις εκδοχές θα μπορούσαν να συγχωνευθούν και να δώσουν την τελική εκδοχή του `foo`, χωρίς να είναι απαραίτητη η ανθρώπινη παρέμβαση.

Κεφάλαιο 3

Σχεδίαση μηχανισμού αυτοματοποιημένου συγχρονισμού αρχείων

Σε πιο αφαιρετικό επίπεδο, αυτό που θέλουμε να σχεδιάσουμε είναι ένα εργαλείο που θα επιτυγχάνει αυτοματοποιημένο συγχρονισμό αρχείων που βρίσκονται σε καταλόγους που ο χρήστης θα επιλέγει. Με βάση τα όσα έχουμε δει, κάτι τέτοιο θα μπορούσε να γίνει με τη χρήση του Git.

Πιο συγκεκριμένα, το εργαλείο θα μπορεί να χρησιμοποιεί έναν κατάλογο που ο χρήστης επιθυμεί, και να τον μετατρέπει σε ένα αποθετήριο του Git. Όταν όλοι οι κατάλογοι προς συγχρονισμό μετατραπούν σε αποθετήρια του Git, βάση των όσων έχουμε δει, θα μπορούσαν με χρήση τοπολογίας αστέρα οι κατάλογοι να συγχρονίζονται, αν ακολουθηθεί συγκεκριμένη ροή εντολών του Git. Ο χρήστης θα μπορεί ελεύθερα να προσθαφαιρεί και να τροποποιεί αρχεία και υποκαταλόγους στους καταλόγους αυτούς, και το εργαλείο θα φροντίζει να γίνονται σε τακτά χρονικά διαστήματα υποβολές, τις οποίες θα προωθεί στον κεντρικό κόμβο ώστε να ενημερωθούν όλα τα αποθετήρια για τις αλλαγές. Θα πρέπει ωστόσο να βρεθεί τρόπος αυτόματης επίλυσης διενέξεων, και διάδοσης των επιλύσεων αυτών, δεδομένης της υπόθεσης ότι ο χρήστης εργάζεται σε κατανεμημένο περιβάλλον, και μπορεί να γίνονται αντικρουόμενες τροποποιήσεις. Στο κεφάλαιο αυτό λοιπόν θα σχεδιάσουμε έναν μηχανισμό που θα επιτυγχάνει την αυτόματη επίλυση διενέξεων.

Επιπλέον, θα ήταν πολύ βολικό να μπορεί να χρησιμοποιηθεί οποιοσδήποτε μηχανισμός αποθήκευσης ως κοινός κεντρικός κόμβος της τοπολογίας. Υπάρχουν περιπτώσεις που ο κεντρικός αυτός κόμβος δεν μπορεί να είναι αποθετήριο του Git, ή θα θέλαμε να προσδώσουμε επιπλέον χαρακτηριστικά στις πληροφορίες που αποθηκεύονται εκεί, όπως είναι η κρυπτογράφηση δεδομένων. Για να γίνει αυτό είναι απαραίτητη η χρήση δικού μας πρωτοκόλλου, το οποίο όμως το Git δεν γνωρίζει. Θα δούμε λοιπόν σε αυτό το κεφάλαιο πώς με τη χρήση συγκεκριμένης διεπαφής οι βοηθοί επικοινωνίας του Git με απομακρυσμένα αποθετήρια επιτρέπουν στο Git τη χρήση πρωτοκόλλων άγνωστων σε αυτό.

3.1 Βοηθοί επικοινωνίας του Git με απομακρυσμένα αποθετήρια (Git Remote Helpers)

Το Git, από προεπιλογή μπορεί να χρησιμοποιήσει τέσσερα πρωτόκολλα για τη μεταφορά δεδομένων: Local (όπου το απομακρυσμένο αποθετήριο είναι απλά ένας κατάλογος στο δίσκο), HTTP, Secure Shell (SSH) και Git. Τι γίνεται όμως αν θέλουμε να επικοινωνήσουμε με κάποιο απομακρυσμένο αποθετήριο χρησιμοποιώντας κάποιο άλλο πρωτόκολλο, άγνωστο για το Git; Κάτι τέτοιο επιτυγχάνεται με την υποστήριξη που παρέχει το Git για *βοηθούς επικοινωνίας με απομακρυσμένα αποθετήρια (Git Remote Helpers)*.

Σύμφωνα με το εγχειρίδιο τεκμηρίωσης [7], οι *βοηθοί επικοινωνίας του Git με απομακρυσμένα αποθετήρια (Git Remote Helpers)* είναι προγράμματα που βοηθούν στην αλληλεπίδραση του Git με απομακρυσμένα αποθετήρια. Συνήθως δεν χρησιμοποιούνται άμεσα από τελικούς χρήστες, αλλά τα καλεί το Git όταν χρειάζεται να αλληλεπίδραση με ένα απομακρυσμένο αποθετήριο μέσω ενός πρωτοκόλλου που το Git δεν γνωρίζει. Στην περίπτωση αυτή, το Git εκτελεί τον βοηθό ως μια ξεχωριστή διαδικασία, στέλνει εντολές στην κανονική είσοδο του βοηθού, και περιμένει αποτελέσματα από την κανονική έξοδο του βοηθού. Για το σκοπό αυτό, οι βοηθοί έχουν συγκεκριμένη μορφή ονόματος, και πρέπει να βρίσκονται σε μονοπάτι όπου το Git θα αναζητήσει εκτελέσιμα αρχεία. Η μορφή του ονόματος είναι `git-remote-<όνομα πρωτοκόλλου>`. Υπάρχει συγκεκριμένη Διεπαφή Προγραμματισμού Εφαρμογής (ΔΠΕ — API στα αγγλικά) που χρησιμοποιείται ώστε το Git να μπορεί να επικοινωνήσει με το βοηθό, και θα εξετάσουμε αυτή την διεπαφή. Αργότερα, θα χρησιμοποιήσουμε τέτοιους βοηθούς ώστε να παρέχουμε στο συγχρονισμό των αρχείων υποστήριξη για διαφορετικούς μηχανισμούς αποθήκευσης.

3.1.1 Εκτέλεση

Ας υποθέσουμε ότι επιθυμούμε την επικοινωνία (λήψη και προώθηση ενημερώσεων) του Git με ένα απομακρυσμένο αποθετήριο, χρησιμοποιώντας ένα πρωτόκολλο που ονομάζεται `myprotocol`. Σε ένα τοπικό λοιπόν αποθετήριο, προσθέτουμε το απομακρυσμένο αποθετήριο με την εντολή:

```
git remote add <remote_name> <remote_url>
```

όπου

`<remote_name>` μπορεί να είναι ένα αυθαίρετο όνομα, και είναι το όνομα με το οποίο θα αναφέρεται πλέον το Git στο απομακρυσμένο αυτό αποθετήριο.

`<remote_url>` μπορεί να έχει τη μορφή `myprotocol::<remote_address>`, ή τη μορφή `myprotocol://<remote_address>`

Ας εξηγήσουμε λίγο καλύτερα τη μορφή του `<remote_url>`. Οι βοηθοί καλούνται προς εκτέλεση με ένα ή (προαιρετικά) δύο ορίσματα, και η μορφή του <remote_url> επηρεάζει τα ορίσματα αυτά.

- Στην περίπτωση της μορφής `myprotocol://<remote_address>` ολόκληρη η διεύθυνση `myprotocol://<remote_address>` παρέχεται ως δεύτερο όρισμα.
- Στην περίπτωση της μορφής `myprotocol::<remote_address>`, σαν δεύτερο όρισμα παρέχεται μόνο το <remote_address>.
- Όσον αφορά το πρώτο όρισμα, αν το Git γνωρίζει το απομακρυσμένο αποθετήριο με κάποιο όνομα (όπως `<remote_name>`), τότε αυτό περνάται ως πρώτο όρισμα. Αλλιώς, αν το `<remote_url>` χρησιμοποιηθεί κατευθείαν στη γραμμή εντολών, το πρώτο όρισμα είναι ίδιο με το δεύτερο.

3.1.2 Μορφή εισόδου και ΔΠΕ

Το Git στέλνει στο βοηθό μία λίστα από εντολές στην κανονική είσοδο, μία εντολή ανά γραμμή. Η πρώτη εντολή είναι πάντα η εντολή `capabilities`, στην οποία ο βοηθός πρέπει να τυπώσει ως απάντηση μία λίστα από δυνατότητες που υποστηρίζει, ακολουθούμενη από μία κενή γραμμή. Η απάντηση στην εντολή `capabilities` καθορίζει και τι εντολές θα στείλει στη συνέχεια το Git. Η ροή των εντολών που στέλνει το Git τερματίζεται από μία κενή γραμμή. Σε κάποιες περιπτώσεις (που δηλώνονται ρητά στο εγχειρίδιο τεκμηρίωσης των σχετικών εντολών), η κενή αυτή γραμμή ακολουθείται από φορτίο άλλου πρωτοκόλλου (όπως πχ το πρωτόκολλο που χρησιμοποιεί το Git για τη μεταφορά πακέτων), ενώ σε άλλες περιπτώσεις, η κενή γραμμή δηλώνει το τέλος της εισόδου.

3.1.3 Δυνατότητες

Μία πλήρης λίστα των δυνατοτήτων που μπορεί να υποστηρίζει ένας βοηθός επικοινωνίας με απομακρυσμένα αποθετήρια μπορεί να βρεθεί στο εγχειρίδιο τεκμηρίωσης [7].

Εδώ απλά εξηγούμε τις βασικές (και πιο συνηθισμένες) δυνατότητες, τις οποίες αργότερα επίσης θα χρησιμοποιήσουμε για να γράψουμε τα δικά μας σενάρια-βοηθούς.

`push`
: Ο βοηθός μπορεί να ανακαλύψει απομακρυσμένες αναφορές και να προωθήσει τοπικές υποβολές και το ιστορικό που οδηγεί σε αυτές σε καινούριες ή υπάρχουσες απομακρυσμένες αναφορές.

`fetch`
: Ο βοηθός μπορεί να ανακαλύψει απομακρυσμένες αναφορές και να μεταφέρει αντικείμενα προσβάσιμα μέσω αυτών στην τοπική αποθήκη αντικειμένων.

Υπάρχουν επίσης οι δυνατότητες `export` και `import`, που αντικαθιστούν τις δυνατότητες `push` και `fetch` αντίστοιχα. Και οι δύο στηρίζονται σε μία *ροή γρήγορης εισαγωγής (fast-import stream)* για την προώθηση και τη λήψη αντικειμένων.

Τέλος, υπάρχει και η δυνατότητα `connect`. Ένας βοηθός που υποστηρίζει την δυνατότητα αυτή, σημαίνει ότι μπορεί να προσπαθήσει να συνδεθεί με υπηρεσίες όπως η `git upload-pack` (για λήψη αντικειμένων), ή η `git receive-pack` (για προώθηση αντικειμένων), ώστε να υπάρχει επικοινωνία χρησιμοποιώντας το γνωστό στο Git πρωτόκολλο μεταφοράς packfile.

Το Git προτιμά την δυνατότητα `connect`, έπειτα τις δυνατότητες `push`/`fetch`, και τέλος τις δυνατότητες `export`/`import`.

3.1.4 Εντολές

Όπως και στις δυνατότητες, έτσι κι εδώ εξηγούμε μόνο τις βασικές εντολές που χρησιμοποιούνται, με σκοπό την επίτευξη προώθησης και λήψης αντικειμένων προς και από διάφορα απομακρυσμένα αποθετήρια. Όπως και πριν, μία πλήρης λίστα με τις εντολές μπορεί να βρεθεί στο εγχειρίδιο τεκμηρίωσης [7].

`capabilities`
: Τυπώνει μια λίστα με τις δυνατότητες του βοηθού, μία σε κάθε γραμμή, τερματισμένη με μία

κενή γραμμή. Πριν από κάθε δυνατότητα μπορεί να τυπώνεται ο χαρακτήρας "*", που υποδηλώνει ότι η εκάστοτε δυνατότητα είναι απαραίτητο να αναγνωρίζεται από την έκδοση του Git που χρησιμοποιεί τον βοηθό. Οποιαδήποτε άγνωστη δυνατότητα οδηγεί σε σφάλμα κατάρρευσης.
Η υποστήριξη αυτής της εντολής είναι υποχρεωτική.

`list`
Τυπώνει σε λίστα τις αναφορές, μία ανά γραμμή, στη μορφή `<value> <name>`. Η τιμή μπορεί να είναι ένας δεκαεξαδικός κωδικός κατακερματισμού, "`@<dest>`" για συμβολική αναφορά ή ο χαρακτήρας "?", που δηλώνει ότι ο βοηθός δεν μπόρεσε να αποσπάσει την τιμή της αναφοράς.
Η λίστα τερματίζεται με μία κενή γραμμή.
Η εντολή `list` υποστηρίζεται από τον βοηθό αν αυτός έχει τη δυνατότητα `fetch`.

`list for-push`
Παρόμοια με την εντολή `list`, με την εξαίρεση ότι χρησιμοποιείται αν και μόνο αν αυτός που κάλεσε τον βοηθό χρειάζεται τη λίστα από αναφορές που θα επιστραφεί ως αποτέλεσμα για την προετοιμασία μιας εντολής προώθησης. Ένας βοηθός που υποστηρίζει και τη δυνατότητα `fetch` και τη δυνατότητα `push`, μπορεί να ξεχωρίσει έτσι το σκοπό για τον οποίο θα χρησιμοποιηθεί η λίστα που θα τυπώσει, και ενδεχομένως να μειώσει την ποσότητα της δουλειάς που χρειάζεται για την προετοιμασία της.
Η εντολή `list for-push` υποστηρίζεται από τον βοηθό αν αυτός έχει τη δυνατότητα `push`.

`fetch <κωδικός κατακερματισμού> <όνομα>`
Λαμβάνει το συγκεκριμένο αντικείμενο, γράφοντας τα απαραίτητα αντικείμενα στη βάση δεδομένων. Οι εντολές `fetch` στέλνονται μία ανά γραμμή, σε ομάδες που τερματίζονται από μία κενή γραμμή. Ως απάντηση τυπώνεται μία κενή γραμμή ανά ομάδα, αφού έχουν ολοκληρωθεί όλες οι εντολές `fetch` της ομάδας. Μόνο αντικείμενα που αναφέρθηκαν στην έξοδο της εντολής `list` με κωδικό κατακερματισμού μπορούν να ληφθούν με αυτόν τον τρόπο.
Η εντολή αυτή υποστηρίζεται αν ο βοηθός έχει τη δυνατότητα `fetch`.

`push +<πηγή>:<προορισμός>`
Προωθεί την τοπική υποβολή ή τον τοπικό κλάδο με το όνομα `<πηγή>` στον απομακρυσμένο κλάδο που περιγράφεται από τον `<προορισμό>`. Οι εντολές `push` αποστέλλονται μία ανά γραμμή, σε ομάδες που τερματίζονται από μία κενή γραμμή. Όταν ολοκληρώνεται η προώθηση τυπώνεται ως απάντηση μία ή περισσότερες γραμμές με "`ok <προορισμός>`", ή "`error <προορισμός>`", ή "`<γιατί>?`", που δηλώνουν επιτυχία ή αποτυχία της προώθησης κάθε αναφοράς. Η απάντηση τερματίζεται με μία κενή γραμμή. Το πεδίο `<why>` μπορεί να είναι σε μορφή συμβολοσειράς C, αν περιέχει το χαρακτήρα LF.
Η εντολή αυτή υποστηρίζεται αν ο βοηθός έχει τη δυνατότητα `push`.

3.2 Συγχρονισμός και αυτόματη επίλυση διενέξεων

Σύμφωνα με όσα έχουν αναφερθεί ως τώρα, διαισθητικά, για να επιτευχθεί συγχρονισμός καταλόγων, θα πρέπει να γίνει:

1. Αποθήκευση τοπικών αλλαγών.
2. Λήψη αλλαγών που έχουν γίνει στο απομακρυσμένο αποθετήριο.
3. Συγχώνευση αλλαγών και επίλυση οποιασδήποτε διένεξης.

4. Ενημέρωση απομακρυσμένου κοινού, κεντρικού αποθετηρίου για τις τοπικές αλλαγές.

Όπως αναφέραμε, η υπόθεση ότι ο χρήστης εργάζεται σε κατανεμημένο περιβάλλον δημιουργεί προβλήματα, καθώς θα πρέπει να βρεθεί μηχανισμός αυτόματης επίλυσης των διενέξεων. Εμείς σχεδιάσαμε έναν τέτοιο μηχανισμό, ο οποίος βασίζεται στη μετονομασία (με μοναδικό τρόπο) των αρχείων που δημιουργούν τη διένεξη, ώστε όχι μόνο αυτή να επιλυθεί, αλλά ταυτόχρονα η επίλυσή της να διαδοθεί σε όλους τους καταλόγους προς συγχρονισμό, χωρίς να δημιουργούνται περαιτέρω διενέξεις από τη διάδοση αυτή.

Βάση του μηχανισμού μας, σε οποιαδήποτε διένεξη υπερτερεί η εκδοχή του βρίσκεται στον κοινό κεντρικό κόμβο, και η τοπική εκδοχή μετονομάζεται με μοναδικό τρόπο. Έτσι, δημιουργούμε τον Πίνακα 3.1 που δείχνει ποια θα πρέπει να είναι η τελική κατάσταση μετά τον συγχρονισμό, ανάλογα με την κατάσταση που βρίσκεται ένα αρχείο στο τοπικό αποθετήριο και στον κοινό κεντρικό κόμβο πριν την εκκίνηση της διαδικασίας του συγχρονισμού. Είναι σημαντικό να παρατηρήσουμε ότι στον πίνακα αυτό δεν παρουσιάζεται η μετονομασία ενός αρχείου ως κατάσταση. Αυτό είναι σκόπιμο, και γίνεται διότι στην πραγματικότητα η μετονομασία ενός αρχείου μπορεί να αναλυθεί σε διαγραφή του αρχείου με το παλιό όνομα και σε δημιουργία ενός αρχείου με το καινούριο όνομα.

Κατάσταση στο τοπικό αποθετήριο	**Κατάσταση στον κοινό κόμβο**	**Αποτέλεσμα συγχρονισμού**
Τροποποιημένο	Καμία τροποποίηση	Τροποποιημένο
Καμία τροποποίηση	Τροποποιημένο	Τροποποιημένο
Τροποποιημένο (Hash = H)	Τροποποιημένο (Hash = H)	Τροποποιημένο (Hash = H)
Τροποποιημένο (Hash = H)	Τροποποιημένο (Hash = H')	(διένεξη) Μετονομασία τοπικής εκδοχής, διατήρηση εκδοχής κοινού κόμβου ως έχει
Δημιουργημένο	Καμία τροποποίηση	Δημιουργημένο
Καμία τροποποίηση	Δημιουργημένο	Δημιουργημένο
Δημιουργημένο (Hash = H)	Δημιουργημένο (Hash = H)	Δημιουργημένο (Hash = H)
Δημιουργημένο (Hash = H)	Δημιουργημένο (Hash = H')	(διένεξη) Μετονομασία τοπικής εκδοχής, διατήρηση εκδοχής κοινού κόμβου ως έχει
Διαγραμμένο	Διαγραμμένο	Διαγραμμένο
Διαγραμμένο	Καμία τροποποίηση	Διαγραμμένο
Καμία τροποποίηση	Διαγραμμένο	Διαγραμμένο
Διαγραμμένο	Τροποποιημένο	(διένεξη) Διαγραφή τοπικής εκδοχής, διατήρηση εκδοχής κοινού κόμβου ως έχει
Τροποποιημένο	Διαγραμμένο	(διένεξη) Μετονομασία τοπικής εκδοχής, διαγραφή εκδοχής κοινού κόμβου
Καμία τροποποίηση	Καμία τροποποίηση	Καμία τροποποίηση

Πίνακας 3.1: Αποτέλεσμα της διαδικασίας του συγχρονισμού

Κεφάλαιο 4

Υλοποίηση μηχανισμού αυτοματοποίησης συγχρονισμού αρχείων

Στο κεφάλαιο αυτό θα υλοποιήσουμε το εργαλείο Gitsync, που θα συνδυάζει εντολές του Git, με στόχο την αυτοματοποίηση της διαδικασίας του συγχρονισμού αρχείων, βάση του σχεδιασμού που αναλύσαμε στο Κεφάλαιο 3. Στόχος μας είναι ακόμα και χρήστες που δεν γνωρίζουν τι είναι το Git ή πώς αυτό λειτουργεί, να μπορούν να χρησιμοποιήσουν το Gitsync.

Θα δούμε επίσης την υλοποίηση ενός δικού μας πρωτοκόλλου, του rawobjects, το οποίο θα χρησιμοποιήσουμε βάση της διεπαφής των βοηθών επικοινωνίας του Git με απομακρυσμένα αποθετήρια που αναλύσαμε το Κεφάλαιο 3.

4.1 Υλοποίηση πρωτοκόλλου rawobjects

Έχοντας μελετήσει και κατανοήσει τον τρόπο που λειτουργεί το Git, καθώς και πώς αυτό επικοινωνεί με σενάρια-βοηθούς επικοινωνίας, δημιουργήσαμε ένα δικό μας πρωτόκολλο, που θα χρησιμοποιείται από το Gitsync για επικοινωνία με τον κοινό κόμβο. Είναι απλά απόδειξη της ορθότητας της ιδέας, ωστόσο κρίθηκε απαραίτητο για την υποστήριξη διαφόρων μηχανισμών αποθήκευσης κατά το συγχρονισμό αρχείων, γιατί όπως έχουμε αναφέρει, ο κοινός κόμβος δεν μπορεί πάντοτε να είναι αποθετήριο του Git, και θέλουμε σε κάποιες περιπτώσεις επιπλέον χαρακτηριστικά, όπως κρυπτογράφηση των αντικειμένων που αποθηκεύονται εκεί.

Το πρωτόκολλό μας επομένως δεν προϋποθέτει ο κοινός κόμβος να είναι ένα αποθετήριο Git. Μπορεί να είναι ένας απλός κατάλογος. Ωστόσο, στο υπόλοιπο κείμενο μπορεί θα αναφερόμαστε σε αυτόν και ως "απομακρυσμένο αποθετήριο". Η παρακάτω ιεραρχία καταλόγων χρησιμοποιείται στο απομακρυσμένο αποθετήριο:

```
<remote repository>
 |- objects/
 |- rawobjects/
 |- refs/
```

Listing 4.1: Ιεραρχία καταλόγων του απομακρυσμένου αποθετηρίου στο πρωτόκολλο *rawobjects*

Όπως και σε ένα κανονικό αποθετήριο του Git, οι κατάλογοι `objects/` και `refs/` χρησιμοποιούνται για την αποθήκευση απακετάριστων αντικειμένων και αναφορών αντίστοιχα. Ο κατάλογος `rawobjects/` χρησιμοποιείται για την αποθήκευση αποσυμπιεσμένων αντικειμένων του Git, που σημαίνει ότι το πρωτόκολλό μας αποσυμπιέζει τα αντικείμενα (που έχουν συμπιεστεί όπως έχουμε πει με τη χρήση της βιβλιοθήκης zlib) και τα αποθηκεύει στον κατάλογο αυτό ώστε να μπορούμε να δούμε το περιεχόμενό τους. Αυτό μπορεί εύκολα να απενεργοποιηθεί, και έγινε για λόγους μελέτης και κατανόησης των αντικειμένων του Git, αφού έτσι μπορούμε να μελετήσουμε τη μορφή περιεχομένου των αντικειμένων blob, δένδρου, υποβολής και ετικέτας με έναν απλό επεξεργαστή κειμένου. Με αφορμή αυτόν τον κατάλογο, ονομάσαμε το πρωτόκολλο *rawobjects*.

Σύμφωνα με όσα έχουμε δει μέχρι τώρα, θα πρέπει να γράψουμε ένα σενάριο-βοηθό με το όνομα `git-remote-rawobjects`, να το κάνουμε εκτελέσιμο, και να το τοποθετήσουμε σε μονοπάτι όπου το Git θα το αναζητήσει. Έτσι γράψαμε ένα σενάριο σε γλώσσα προγραμματισμού Python και το προσθέσαμε στο PATH.

Θέλαμε ωστόσο αυτό το πρωτόκολλο να χρησιμοποιείται με διάφορους μηχανισμούς αποθήκευσης. Για παράδειγμα θέλαμε να μπορούμε να προωθήσουμε αντικείμενα τόσο σε αποθετήρια που βρίσκονται στον τοπικό δίσκο, όσο και σε έναν κουβά στο Amazon S3. Για να μπορεί να γίνει διάκριση μεταξύ των διαφόρων μηχανισμών αποθήκευσης, θεωρήσαμε ότι οι διευθύνσεις των αποθετηρίων αυτών έχουν συγκεκριμένη μορφή. Συγκεκριμένα, αν ένας τοπικός κατάλογος χρησιμοποιείται σαν αποθετήριο, υποθέτουμε ότι η διεύθυνσή του είναι της μορφής `rawobjects::file://<κατάλογος αποθετηρίου>` ή `rawobjects::<κατάλογος αποθετηρίου>`. Αν το απομακρυσμένο αποθετήριο είναι ένας κουβάς στο Amazon S3, τότε αναμένουμε η διεύθυνση του αποθετηρίου να είναι της μορφής `rawobjects::s3://<όνομα κουβά>/<κατάλογος αποθετηρίου>`. Τέλος, στην περίπτωση που θέλουμε τα αντικείμενα που αποθηκεύονται σε κουβά του Amazon S3 να κρυπτογραφούνται, η διεύθυνση που θα χρησιμοποιηθεί για το αποθετήριο πρέπει να έχει τη μορφή `rawobjects::ss3://<όνομα κουβά>/<κατάλογος αποθετηρίου>`.

Γράψαμε επιπλέον κλάσεις σε Python, κάθε μία υπεύθυνη για διαφορετικό μηχανισμό αποθήκευσης. Έτσι, το σενάριο-βοηθός μας `git-remote-rawobjects`, αναλόγως τη διεύθυνση που το Git του περνάει ως παράμετρο όταν το καλεί, μπορεί να αρχικοποιήσει αντικείμενο της κατάλληλης κλάσης.

Επειδή δεν θέλαμε να στηριχτούμε στο πρωτόκολλο μεταφοράς packfile του Git για τους μηχανισμούς αποθήκευσης μας, υλοποιήσαμε τις δυνατότητες `push` και `fetch` στα σενάρια-βοηθούς που γράψαμε.

4.2 Το εργαλείο Gitsync

Το *Gitsync* είναι ένα πρόγραμμα που γράψαμε σε Python, και που χρησιμοποιείται για την προετοιμασία καταλόγων προς συγχρονισμό και για τον έλεγχο του δαίμονα που συγχρονίζει περιοδικά τα αρχεία. Ο συγχρονισμός, βάση του σχεδιασμού, θα γίνεται με τοπολογία αστέρα, χρησιμοποιώντας έναν κοινό κεντρικό κόμβο. Χρησιμοποιεί το πρωτόκολλο rawobjects, και έχει ως βάση λειτουργίας του το εργαλείο Git. Επιπλέον το Gitsync υποστηρίζει ονοματοδοσία συσκευών, ώστε σε περίπτωση διένεξης, να γίνεται εύκολα αντιληπτό ποια εκδοχή προήλθε από ποια συσκευή.

Αρχικά θα εξηγήσουμε πώς χρησιμοποιείται το εργαλείο, αναλύοντας τον κατάλογο βοηθείας του. Ξεκινάμε με την εντολή `gitsync --help`, η έξοδος της οποίας φαίνεται στο Listing 4.2.

Σε περίπτωση λοιπόν που θέλουμε να προετοιμάσουμε έναν κατάλογο για συγχρονισμό αρχείων, χρησιμοποιούμε την εντολή `gitsync setup`. Η εντολή αυτή, σύμφωνα με την έξοδο που παίρνουμε

```
$ gitsync --help
usage: gitsync [-h] command ...

Control the sync service

positional arguments:
  command
    start     Start the daemon
    restart   Restart the daemon
    stop      Stop the daemon
    setup     Setup directory to sync

optional arguments:
  -h, --help  show this help message and exit
```

Listing 4.2: Έξοδος της εκτέλεσης της εντολής `gitsync --help`

με την εκτέλεση `gitsync setup --help` χρησιμοποιείται ως εξής:
`usage: gitsync setup [-h] [-l LOGFILE] [-n NAME] directory remote`

Χρειάζεται δηλαδή υποχρεωτικά δύο ορίσματα, `directory` και `remote`. Τα υπόλοιπα (που βρίσκονται μέσα σε αγκύλες "[…]") είναι προαιρετικές παράμετροι.

Υποχρεωτικά ορίσματα:

directory
: Ο κατάλογος προς προετοιμασία για συγχρονισμό.

remote
: Η διεύθυνση του απομακρυσμένου αποθετηρίου που χρησιμοποιείται ως μεσάζοντας για τον συγχρονισμό. Οι μορφές διευθύνσεων που υποστηρίζονται είναι οι εξής:
`s3://<όνομα κουβά>/[<μονοπάτι αποθετηρίου>]` για τη χρήση Amazon S3
`ss3://<όνομα κουβά>/[<μονοπάτι αποθετηρίου>]` για τη χρήση Amazon S3 με κρυπτογράφηση
`file://<μονοπάτι αποθετηρίου>` ή `<μονοπάτι αποθετηρίου>` για τη χρήση τοπικού καταλόγου.

Προαιρετικές παράμετροι:

-h, --help
: Τυπώνει έναν κατάλογο βοήθειας

-l LOGFILE, --logfile LOGFILE
: Χρησιμοποιείται το αρχείο LOGFILE ως αρχείο καταγραφής. Ως προεπιλογή είναι το μονοπάτι DEVNULL.

-n NAME, --name NAME
: Χρησιμοποιείται το NAME ως όνομα της συσκευής. Ως προεπιλογή είναι το hostname.

Αφού έχει γίνει προετοιμασία ενός καταλόγου για συγχρονισμό, πρέπει να ξεκινήσει ο δαίμονας, που φροντίζει ώστε να γίνεται συγχρονισμός ανά τακτά διαστήματα. Κάτι τέτοιο γίνεται με την εντολή

`gitsync start` και όπως βλέπουμε με την εκτέλεση `gitsync start --help`, η χρήση της είναι η εξής:

```
usage: gitsync start [-h] -d DIRECTORY [-p PIDFILE] [-l LOGFILE] [-i INTERVAL] [--rebase | --automerge]
```

Δηλαδή υποχρεωτική παράμετρος είναι μόνο η `-d DIRECTORY`. Οι υπόλοιπες είναι προαιρετικές.

Υποχρεωτικές παράμετροι:

-d DIRECTORY, --directory DIRECTORY
: Ο δαίμονας θα παρακολουθεί και θα συγχρονίζει τον κατάλογο DIRECTORY. Ο κατάλογος αυτός θα πρέπει είναι υπαρκτό μονοπάτι.

Προαιρετικές παράμετροι:

-h, --help
: Τυπώνει έναν κατάλογο βοήθειας

-p PIDFILE, --pidfile PIDFILE
: Χρησιμοποιείται το αρχείο PIDFILE για την αποθήκευση του αναγνωριστικού διεργασίας (pid) που θα έχει ο δαίμονας όταν ξεκινήσει. Ως προεπιλογή είναι το αρχείο ~/.gitsync.pid.

-l LOGFILE, --logfile LOGFILE
: Χρησιμοποιείται το αρχείο LOGFILE ως αρχείο καταγραφής. Ως προεπιλογή είναι το μονοπάτι DEVNULL.

-i INTERVAL, --interval INTERVAL
: Το διάστημα (σε δευτερόλεπτα) που μεσολαβεί μεταξύ των αποπειρών συγχρονισμού τίθεται στην τιμή INTERVAL. Αν η τιμή αυτή είναι 0, τότε ο συγχρονισμός μπορεί να γίνει χειροκίνητα, με την αποστολή σημάτων SIGUSR1 στη διεργασία του δαίμονα. Ως προεπιλογή η τιμή είναι 5.

--rebase
: Χρησιμοποιείται η στρατηγική της αναθεμελίωσης αντί για τη συγχώνευση κατά τη διάρκεια του συγχρονισμού

--automerge
: Σε περίπτωση διένεξης, γίνεται αυτόματη συγχώνευση αρχείων, όπως στο Git, αν αυτό είναι εφικτό. Η παράμετρος αυτή δεν μπορεί να συνδυαστεί με την παράμετρο `rebase`

Για να σταματήσει ο δαίμονας να εκτελείται, χρησιμοποιείται η εντολή `gitsync stop`, που σύμφωνα με την εκτέλεση `gitsync stop --help` χρησιμοποιείται ως εξής:

```
usage: gitsync stop [-h] [-p PIDFILE] [-l LOGFILE]
```

Βλέπουμε δηλαδή ότι δεν παίρνει κάποια υποχρεωτική παράμετρο, αλλά μόνο προαιρετικές, και η εξήγησή τους είναι όμοια με αυτή της εντολής `gitsync start`.

Τέλος υπάρχει και η εντολή `gitsync restart` που χρησιμοποιείται για την επανεκκίνηση του δαίμονα με τον ίδιο τρόπο όπως η εντολή `gitsync start`.

4.3 Η προετοιμασία

Η διαδικασία της προετοιμασίας ενός καταλόγου για συγχρονισμό απαιτεί μόνο το μονοπάτι του καταλόγου προς συγχρονισμό και τη διεύθυνση του απομακρυσμένου αποθετηρίου που θα χρησιμοποιηθεί ως κοινός κόμβος στο συγχρονισμό. Σε λογικό επίπεδο, η διαδικασία πρέπει να είναι περίπου όπως περιγράφουν τα παρακάτω βήματα:

1. Αρχικοποίηση του τοπικού καταλόγου ως αποθετήριο του Git.
2. Παροχή ονόματος και email χρήστη. Το όνομα μπορεί να (και θα) χρησιμοποιηθεί ως όνομα συσκευής.
3. Προσθήκη και ρύθμιση του απομακρυσμένου αποθετηρίου με το όνομα `origin` τοπικό αποθετήριο.
4. Ρύθμιση του τοπικού κλάδου `master` ώστε να παρακολουθεί αλλαγές του κλάδου `master` του απομακρυσμένου αποθετηρίου `origin` (δηλαδή του κλάδου `origin/master`).
5. Απόπειρα για λήψη αντικειμένων από το απομακρυσμένο αποθετήριο, αν αυτά υπάρχουν. Διαφορετικά, δημιουργία μιας υποβολής και προώθηση στο απομακρυσμένο αποθετήριο.

Το τελευταίο βήμα διαβεβαιώνει ότι όλοι οι κατάλογοι που συγχρονίζονται χρησιμοποιώντας τον ίδιο μεσάζοντα θα έχουν μία κοινή πρώτη υποβολή.Επομένως, κατά την επίλυση διενέξεων, θα είναι πάντα εφικτός ο υπολογισμός βάσης συγχώνευσης.

Σε εντολές του Git, το Gitsync επιτυγχάνει τα παραπάνω ως εξής:

1. Μετάβαση στον κατάλογο του τοπικού αποθετηρίου.
   `git init .`
2. `git config user.name <device_name>`
   `git config user.email gitsync@<device_name>`
3. `git remote add origin rawobjects::<remote_url>`
4. `git config branch.master.remote origin`
   `git config branch.master.merge refs/heads/master`
5. `git fetch`
   Έλεγχος αν υπάρχει το αρχείο `.git/refs/remote/origin/master`.
   Αν υπάρχει:
   `git pull`
   Διαφορετικά:
   Δημιουργία αρχείου `.gitignore`
   `git add .gitignore`
   `git commit -m “init”`
   `git push`

Για την υποστήριξη των πρωτοκόλλων `s3://` και `ss3://` είναι απαραίτητη η χειροκίνητη δημιουργία δύο επιπλέον αρχείων: `$HOME/.aws/config` και `$HOME/.aws/credentials`. Αυτά τα αρχεία χρειάζονται ώστε το *boto3* (το εργαλείο σε python που χρησιμοποιείται για την επικοινωνία με τους διακομιστές της υπηρεσίας S3 της Amazon) να μπορεί να εντοπίσει τα διαπιστευτήρια που χρειάζονται για την πρόσβαση στην υπηρεσία αποθήκευσης S3. Στα Listing 4.3 και Listing 4.4 φαίνεται ένα παράδειγμα των αρχείων `$HOME/.aws/config` και `$HOME/.aws/credentials`.

```
$ cat $HOME/.aws/config
[default]
region=us-east-1
```

Listing 4.3: Παράδειγμα περιεχομένων του αρχείου `$HOME/.aws/config`

```
$ cat $HOME/.aws/credentials
[default]
aws_access_key_id = ABCDEFGHIJ1234567890
aws_secret_access_key = D1ySkxHS7ZUBiBeJv1mcS1i/MMvoI7+hiJhfleyR
```

Listing 4.4: Παράδειγμα περιεχομένων του αρχείου `$HOME/.aws/credentials`

Τέλος, στην περίπτωση του πρωτοκόλλου `ss3://`, ο χρήστης θα κληθεί να εισάγει ένα συνθηματικό. Αυτό θα χρησιμοποιηθεί ώστε να παραχθεί ένα κλειδί, που θα αποθηκευτεί σε μορφή απλού κειμένου στο αρχείο `.git/encrypt_key`. Αυτό το κλειδί θα χρησιμοποιείται ως συνθηματικό για την παραγωγή κλειδιών για την κρυπτογράφηση και την αποκρυπτογράφηση αντικειμένων. Κάθε αντικείμενο θα έχει το δικό του μοναδικό κλειδί κρυπτογράφησης. Αυτός δεν είναι ο πιο ασφαλής τρόπος κρυπτογράφησης, ωστόσο θεωρούμε ότι για τους σκοπούς της απόδειξης της ιδέας είναι αρκετός. Επιπλέον, θέλουμε να μην καλείται ο χρήστης επανειλημμένα να δώσει το συνθηματικό του, οπότε επιλέξαμε αυτόν τον απλό τρόπο.

4.4 Ο δαίμονας

Τα βήματα που πρέπει να ακολουθηθούν ώστε να έχουμε επιτυχή συγχρονισμό αρχείων, είναι διαισθητικά, όπως αναφέραμε κατά το σχεδιασμό του μηχανισμού:

1. Αποθήκευση τοπικών αλλαγών.
2. Λήψη αλλαγών που έχουν γίνει στο απομακρυσμένο αποθετήριο.
3. Συγχώνευση αλλαγών και επίλυση οποιασδήποτε διένεξης.
4. Ενημέρωση απομακρυσμένου αποθετηρίου για τις τοπικές αλλαγές.

Το Git μπορεί να πραγματοποιήσει τα παραπάνω, με τις ακόλουθες εντολές. Θεωρώντας ότι στο τοπικό αποθετήριο έχει προστεθεί και ρυθμιστεί ένα απομακρυσμένο αποθετήριο με το όνομα `origin`:

1. `git add -A`
   `git commit -m update`
2. `git fetch origin`
3. `git merge origin/master`
4. `git push origin master`

Οι παραπάνω εντολές είναι αρκετές στην περίπτωση που προσπαθούμε να επιτύχουμε απλό συγχρονισμό αρχείων. Ωστόσο εμείς αποσκοπούμε σε *κατανεμημένο* συγχρονισμό αρχείων, και σε αυτή την περίπτωση υπάρχει ένα βασικό πρόβλημα: Το βήμα της συγχώνευσης. Σε ένα σενάριο απλού

συγχρονισμού αρχείων, οι αλλαγές γίνονται σε μία τοποθεσία κάθε φορά, οπότε δεν δημιουργούνται διενέξεις. Στην περίπτωσή μας, κατά το στάδιο της συγχώνευσης μπορεί να προκύψουν διενέξεις, και το Git θα σταματήσει τη διαδικασία και θα ζητήσει ανθρώπινη επέμβαση. Για χρήστη που δεν γνωρίζει τι είναι ή πώς λειτουργεί το Git, η επιτυχής συγχώνευση μπορεί να αποδειχτεί πολύ δύσκολη. Επομένως πρέπει να βρούμε έναν τρόπο να γίνεται η επίλυση διενέξεων αυτόματα.

Θα το επιτύχουμε αυτό, βήμα-βήμα. Είναι λογικό στην αρχή το Gitsync να επιχειρήσει μία συγχώνευση γρήγορης προώθησης, αφού στην περίπτωση που αυτή επιτύχει, το στάδιο της συγχώνευσης απλοποιείται σημαντικά. Στην περίπτωση που ο συγχρονισμός δεν είναι κατανεμημένος, η μέθοδος αυτή θα επιτυγχάνει πάντα, και γιαυτό και τα παραπάνω βήματα θα αρκούσαν. Στον κατανεμημένο συγχρονισμό ωστόσο, ένα αρχείο μπορεί να τροποποιηθεί σε δύο τοποθεσίες ταυτόχρονα. Στην περίπτωση αυτή μπορεί να προκύψουν διενέξεις, και η συγχώνευση γρήγορης προώθησης να αποτύχει. Τότε το Gitsync προσπαθεί να επιλύσει αυτόματα τις διενέξεις, ακολουθώντας τη στρατηγική της αναθεμελίωσης ή τη στρατηγική της συγχώνευσης, ανάλογα με το αν έχει εκτελεστεί με την παράμετρο `--rebase` ή όχι. Πριν όμως εξηγήσουμε πώς γίνεται η αυτόματη επίλυση διενέξεων, θα κάνουμε μία παρατήρηση.

Όπως θα πρέπει να έχει γίνει κατανοητό μέχρι τώρα, το Git παρακολουθεί περιεχόμενα αρχείων. Δεν παρακολουθεί ούτε αρχεία αυτά καθ' αυτά, ούτε καταλόγους. Το Git γνωρίζει ότι σε μία υποβολή A το αρχείο `foo` είχε κάποιο περιεχόμενο και σε μία άλλη υποβολή B το αρχείο `foo` είχε κάποιο άλλο περιεχόμενο. Δεν ενδιαφέρει πώς έγινε η μετάβαση του `foo` από την υποβολή A στην υποβολή B, αν δηλαδή έγινε με συγχώνευση, με απλή τροποποίηση, με μετονομασία κάποιου άλλου αρχείου κλπ. Αυτό έχει δύο συνέπειες.

Η πρώτη είναι ότι το Git δεν παρακολουθεί κενούς καταλόγους. Δεν μπορεί δηλαδή να προστεθεί στη βάση δεδομένων του ένας κενός κατάλογος. Θα θέλαμε όμως να μπορούμε να συγχρονίζουμε κενούς καταλόγους με το Gitsync. Για να το επιτύχει αυτό, το Gitsync διατηρεί μία λίστα με γνωστούς καταλόγους, και όταν ανιχνεύει καινούριο κατάλογο, δημιουργεί σε αυτόν ένα κενό αρχείο `.gitkeep`, και προσθέτει αυτό στην βάση δεδομένων του Git. Έτσι επιτυγχάνεται ο συγχρονισμός φαινομενικά κενών καταλόγων.

Μία δεύτερη συνέπεια των παραπάνω είναι ότι το Git δεν μπορεί να ανιχνεύσει με βεβαιότητα αν ένα αρχείο προέκυψε με μετονομασία κάποιου άλλου. Στην πραγματικότητα συγκρίνει περιεχόμενα, μεταξύ αρχείων που προστίθενται και αρχείων που διαγράφονται, και αν βρει ομοιότητα πάνω από ένα ποσοστό, το Git θεωρεί ότι το αρχείο που προστέθηκε είναι αποτέλεσμα μετονομασίας του αρχείου που διεγράφη. Ωστόσο στην βάση δεδομένων του, η μετονομασία καταλογίζεται απλά ως μία διαγραφή αρχείου και μία αντίστοιχη προσθήκη. Επειδή η αβέβαιη ανίχνευση μετονομασιών μπορεί να προκαλέσει προβλήματα κατά το συγχρονισμό αρχείων, το Gitsync σε όποιες εντολές του Git είναι εφικτό χρησιμοποιεί την παράμετρο `--no-renames`, για να απενεργοποιήσει τον μηχανισμό αυτόν.

4.5 Μέθοδος της συγχώνευσης

Θα εξηγήσουμε τώρα πώς γίνεται η ανίχνευση και η αυτόματη επίλυση των διενέξεων χρησιμοποιώντας τη μέθοδο της συγχώνευσης. Το Gitsync παρέχει τη δυνατότητα αυτόματης συγχώνευσης αρχείων από το Git όπου αυτό είναι εφικτό, με τη χρήση της παραμέτρου `--automerge`. Ωστόσο για λόγους που θα γίνουν αργότερα εμφανείς, αυτή η δυνατότητα δεν παρέχεται με τη μέθοδο της αναθεμελίωσης.

Κατά τη μέθοδο της συγχώνευσης, το Gitsync ανιχνεύει τις διενέξεις κάνοντας μία απόπειρα συγχώνευσης που δεν θα οδηγήσει ωστόσο στη δημιουργία αντικειμένου υποβολής. Μετά, με μία σειρά από εντολές ανιχνεύει ποια αρχεία βρίσκονται σε διένεξη, αλλά μπορούν να συγχωνευθούν αυτόματα από το Git (τα αρχεία αυτά ανήκουν σε ένα σύνολο που θα καλούμε *automerged*), ποια αρχεία είναι σε διένεξη και απαιτούν ανθρώπινη παρέμβαση ώστε να συγχωνευθούν (τα αρχεία αυτά ανήκουν σε ένα σύνολο που θα καλούμε *unmerged*) και ποια αρχεία δεν προκάλεσαν διένεξη (αυτά ανήκουν σε ένα σύνολο που θα καλούμε *updated*). Έπειτα, το Gitsync ακυρώνει την αποτυχημένη απόπειρα συγχώνευσης, και κάνει τροποποιήσεις στα αρχεία που ανήκουν στα σύνολα automerged και updated. Ο στόχος είναι, έπειτα από τις τροποποιήσεις, να γίνει μία επιτυχημένη απόπειρα συγχώνευσης, χωρίς να προκύψουν διενέξεις.

Μπορούμε εύκολα, χρησιμοποιώντας την εντολή `git ls-files -u`, να εντοπίσουμε ποια αρχεία ανήκουν στο σύνολο unmerged. Η εντολή `git ls-files` ενώνει τη λίστα αρχείων που βρίσκονται στο index, με τη λίστα αρχείων που βρίσκονται στον κατάλογο εργασίας, και προβάλει ως αποτέλεσμα διάφορους συνδυασμούς αυτών. Με την παράμετρο `-u` τυπώνονται μόνο τα αρχεία που δεν έχουν συγχωνευθεί.

Προκειμένου να υπολογίσουμε τα σύνολα automerged και updated, ακολουθείται η κάτωθι διαδικασία:

1. Υπολογισμός της υποβολής που χρησιμοποιείται ως βάση συγχώνευσης από το Git.
2. Ανίχνευση αρχείων που έχουν τροποποιηθεί στο απομακρυσμένο αποθετήριο (σύνολο *modified_remote*).
3. Ανίχνευση αρχείων που έχουν τροποποιηθεί στο τοπικό αποθετήριο (σύνολο *modified_local*).
4. Ανίχνευση των αρχείων που επηρεάζονται από τη συγχώνευση (σύνολο *to_merge*).
5. Ανίχνευση αρχείων που μπορούν να συγχωνευθούν χωρίς ανθρώπινη παρέμβαση. Το σύνολο αυτό (που θα το αποκαλούμε *merged*) είναι η ένωση των συνόλων automerged και updated. Δηλαδή $merged = automerged \cup updated = to\_merge - unmerged$
6. Υπολογισμός του συνόλου automerged.
7. Υπολογισμός του συνόλου updated.

Το Gitsync λοιπόν, υπολογίζει πρώτα την υποβολή που χρησιμοποιείται ως βάση συγχώνευσης χρησιμοποιώντας την εντολή `git merge-base`. Αυτή βρίσκει τους καλύτερους κοινούς προγόνους μεταξύ δύο υποβολών για να χρησιμοποιηθεί σε μία τριμερή συγχώνευση. Κατά τη διάρκεια της προετοιμασίας του καταλόγου για συγχρονισμού, το Gitsync έχει φροντίσει όλα τα αποθετήρια που είναι προς συγχρονισμό μεταξύ τους να μοιράζονται τουλάχιστον μία κοινή υποβολή, την αρχική υποβολή, επομένως η εντολή `git merge-base` δεν πρέπει να αποτύχει.

Έπειτα, με κατάλληλη χρήση της εντολής `git diff` υπολογίζονται τα αρχεία που ανήκουν στο σύνολο to_merge. Για να γίνει αυτό, υπολογίζονται πρώτα τα αρχεία που έχουν τροποποιηθεί μεταξύ της υποβολής που χρησιμοποιείται ως βάση συγχώνευσης και του απομακρυσμένου κλάδου που προσπαθούμε να συγχωνεύσουμε (σύνολο modified_remote). Ωστόσο στο σύνολο αυτό ανήκουν και αρχεία τα οποία έχουν τροποποιηθεί και τοπικά και στο απομακρυσμένο αποθετήριο, και έχουν καταλήξει να έχουν το ίδιο περιεχόμενο. Δεν θα θέλαμε τα αρχεία αυτά στο σύνολο to_merge. Επομένως, ανιχνεύουμε αυτά τα αρχεία με χρήση της εντολής `git diff` μεταξύ του απομακρυσμένου και του τοπικού κλάδου (σύνολο *remote_local_diff*) και έτσι το σύνολο to_merge υπολογίζεται ως εξής: $to\_merge = modified\_remote \cap remote\_local\_diff$. Έπειτα, με χρήση `git diff` μεταξύ του τοπικού κλάδου και της υποβολής που χρησιμοποιείται ως βάση συγχώνευσης, υπολογίζουμε το σύνολο αρχείων που τροποποιήθηκαν τοπικά (modified_local).

Έχοντας πλέον υπολογισμένο το σύνολο to_merge, εύκολα υπολογίζεται το σύνολο merged, αφού όπως είπαμε $merged = to\_merge - unmerged$.

Στο σύνολο automerged ανήκουν αρχεία που ανήκουν στο σύνολο to_merge, δεν ανήκουν στο σύνολο unmerged, και έχουν τροποποιηθεί τοπικά (δηλαδή ανήκουν στο σύνολο modified_local). Επομένως $automerged = (to\_merge - unmerged) \cap modified\_local = merged \cap modified\_local$

Τέλος, το σύνολο unmerged αποτελείται από αρχεία που ανήκουν στο σύνολο to_merge, δεν ανήκουν στο σύνολο unmerged, και δεν ανήκουν και στο σύνολο automerged. Επομένως $updated = to\_merge - unmerged - automerged = merged - automerged$

Με τα σύνολα updated, automerged και unmerged υπολογισμένα, το Gitsync μπορεί να προχωρήσει στις κατάλληλες τροποποιήσεις, ώστε η επόμενη προσπάθεια για συγχώνευση να είναι επιτυχής. Σαν γενικός κανόνας, το τελικό αποτέλεσμα της αυτόματης επίλυσης διένεξης είναι η εκδοχή που βρίσκεται στο απομακρυσμένο αποθετήριο να διατηρείται ως έχει, και η εκδοχή που βρίσκεται στο τοπικό αποθετήριο να μετονομάζεται με έναν μοναδικό τρόπο. Το καινούριο όνομα έχει τη μορφή <όνομα αρχείου>-<κωδικός κατακερματισμού του Git>-<όνομα συσκευής από την οποία προήλθε η εκδοχή αυτή>. Χρησιμοποιείται ο κωδικός κατακερματισμού του Git για να εξασφαλισθεί η μοναδικότητα του ονόματος, για λόγους επίδοσης (αφού αυτός είναι ήδη υπολογισμένος) και επειδή μπορεί να χρησιμοποιηθεί και για καταλόγους (αφού τα αντικείμενα τύπου δένδρου στο Git έχουν κωδικό κατακερματισμού).

Γιατί ωστόσο επιλέξαμε να διατηρείται η εκδοχή του απομακρυσμένου αποθετηρίου ως έχει και όχι αυτή του τοπικού; Όπως έχουμε εξηγήσει, ο στόχος μας είναι μετά τις τροποποιήσεις να είναι εφικτή μία συγχώνευση χωρίς διενέξεις. Κάτι τέτοιο δεν εξασφαλίζεται με τη διατήρηση της τοπικής εκδοχής ως έχει, και το είχαμε προβλέψει κατά το στάδιο του σχεδιασμού του μηχανισμού. Όσον αφορά τα αρχεία που ανήκουν στο σύνολο automerged, για αυτά μπορούμε να κρατάμε ως τελική εκδοχή την συγχώνευση που το Git κάνει αυτόματα, και να διατηρούμε και τις δύο εκδοχές (τοπικού και απομακρυσμένου αποθετηρίου) μετονομασμένες.

Μετά τις τροποποιήσεις, το Gitsync κάνει μία υποβολή, που σηματοδοτεί την επίλυση των διενέξεων. Στο σημείο αυτό, γίνεται μία ακόμη απόπειρα συγχώνευσης, η οποία αυτή τη φορά επιτυγχάνει. Τέλος, με την εντολή `git push`, το απομακρυσμένο αποθετήριο ενημερώνεται για όλες τις αλλαγές που έγιναν.

4.6 Μέθοδος της αναθεμελίωσης

Έχουμε ήδη περιγράψει πώς λειτουργεί η αναθεμελίωση στο Git. Το Gitsync δεν εκτελεί άμεσα την εντολή `git rebase`, διότι κάτι τέτοιο θα απαιτούσε ανθρώπινη παρέμβαση σε περίπτωση διένεξης. Αντ' αυτού, προσομοιώνει τη διαδικασία της αναθεμελίωσης, κάνοντας ότι η εντολή `git rebase` θα έκανε. Υπολογίζει πρώτα μία λίστα από υποβολές που πρέπει να επανυποβληθούν στον κλάδο `origin/master` χρησιμοποιώντας το εύρος `origin/master..HEAD` — αυτό σημαίνει "όλες οι υποβολές που είναι προσβάσιμες από το `HEAD` αλλά δεν είναι προσβάσιμες από τον κλάδο `origin/master`". Μία τέτοια λίστα τυπώνεται με κατάλληλη μορφή της εντολής `git rev-list`, η οποία τυπώνει υποβολές σε αντίστροφη χρονολογική σειρά.

Η εντολή `git cherry-pick` χρησιμοποιείται για την μεταφορά μιας υποβολής από μία γραμμή ανάπτυξης σε μία άλλη. Με μία ή περισσότερες υποβολές ως ορίσματα, εφαρμόζει τις αλλαγές που κάθε μια από αυτές εισάγει, δημιουργώντας μία καινούρια υποβολή κάθε φορά. Επομένως, μπορούμε να επαναφέρουμε τον τοπικό κλάδο `master` στην υποβολή που δείχνει ο κλάδος `origin/master` και

έπειτα να χρησιμοποιήσουμε την εντολή `git cherry-pick` με κάθε μία από τις υποβολές που πρέπει να επανυποβληθούν. Κάπως έτσι προσομοιώνεται η εντολή `git rebase`.

Κατά τη διάρκεια της διαδικασίας του cherry-picking, μπορεί να προκύψουν διενέξεις. Έχουμε ήδη δει πώς αυτές μπορούν να επιλυθούν με αυτόματο τρόπο, με τη δημιουργία μετονομασμένων αρχείων. Ωστόσο αυτή η προσέγγιση δεν είναι και τόσο αποτελεσματική στη μέθοδο της αναθεμελίωσης. Η διατήρηση της εκδοχής του απομακρυσμένου αποθετηρίου ως έχει και η μετονομασία της τοπικής εκδοχής μπορεί να οδηγήσει σε πολλές διενέξεις κατά τη διαδικασία της αναθεμελίωσης. Για παράδειγμα:

1. Το αρχείο `foo` έχει τροποποιηθεί στο απομακρυσμένο αποθετήριο.
2. Το αρχείο `foo` έχει τροποποιηθεί και στο τοπικό αποθετήριο, με τέτοιο τρόπο ώστε η απόπειρα συγχώνευσης θα δημιουργήσει διένεξη.
3. Η συσκευή είναι εκτός σύνδεσης, οπότε παρόλο που τοπικά θα γίνονται υποβολές με όλες τις τροποποιήσεις, το τοπικό αποθετήριο δεν ενημερώνεται για τις αλλαγές που γίνονται στο απομακρυσμένο αποθετήριο (και αντίστροφα).
4. Όσο η συσκευή είναι εκτός σύνδεσης, το αρχείο `foo` τροποποιείται συνεχώς τοπικά, οπότε δημιουργούνται πολλές υποβολές στις οποίες οι αλλαγές του `foo` βασίζονται σε προηγούμενες τοπικές αλλαγές.
5. Κάποια στιγμή η συσκευή συνδέεται, οπότε και γίνεται απόπειρα συγχρονισμού.
6. Κατά τη διάρκεια της αναθεμελίωσης θα προκύψουν προβλήματα. Η προσπάθεια να γίνει cherry-pick της πρώτης υποβολής θα οδηγήσει σε διένεξη (του αρχείου `foo`) και με βάση τη στρατηγική επίλυσης διενέξεων, η εκδοχή του απομακρυσμένου αποθετηρίου θα διατηρηθεί ως έχει και η τοπική θα μετονομαστεί. Οι επόμενες υποβολές λοιπόν, στις οποίες όπως είπαμε οι αλλαγές στο `foo` βασίζονται σε προηγούμενες τοπικές αλλαγές, θα δημιουργήσουν πάλι διένεξη, όταν με τη σειρά τους θα πρέπει να τις κάνουμε cherry-pick. Οπότε τελικά θα δημιουργηθούν πολλά μετονομασμένα αρχεία.

Επομένως όταν γίνονται τα cherry-picks και προκύπτουν διενέξεις, θα πρέπει να διατηρούμε την τοπική εκδοχή ως έχει, και ένα μετονομασμένο αντίγραφο της εκδοχής του απομακρυσμένου αποθετηρίου. Ωστόσο αυτό είναι ασυνεπές τόσο με τον σχεδιασμό μας, όσο και με τον τρόπο που επιλύονται οι διενέξεις κατά τη μέθοδο της συγχώνευσης, και οι χρήστες περιμένουν συνέπεια.

Επιπλέον, τα μετονομασμένα αρχεία που προκύπτουν είναι όλα αντίγραφα των ενδιάμεσων καταστάσεων τις οποίες το `foo` περνάει μέχρι να φτάσει στην τελική του μορφή. Από την οπτική γωνία του τελικού χρήστη όμως, οι ενδιάμεσες αυτές καταστάσεις δεν ενδιαφέρουν. Μόνο η εκδοχή του απομακρυσμένου αποθετηρίου και αυτή του τοπικού αποθετηρίου έχουν νόημα να διατηρούνται.

Επομένως ο στόχος μας κατά τη μέθοδο της αναθεμελίωσης γίνεται σαφής: Στο τέλος της διαδικασίας, αν ένα αρχείο βρέθηκε να ανήκει στο σύνολο unmerged, δύο εκδοχές του θα πρέπει να υπάρχουν. Μία μετονομασμένη, και μία διατηρημένη ως έχει. Για λόγους συνέπειας με τη μέθοδο της συγχώνευσης και το σχεδιασμό μας, η μετονομασμένη θα είναι η τοπική εκδοχή, και η εκδοχή του απομακρυσμένου αποθετηρίου θα διατηρηθεί ως έχει.

Υπάρχουν δύο τρόποι προσέγγισης αυτού του στόχου.

- Κατά τη διάρκεια των cherry-picks, αν ένα αρχείο δημιουργήσει διένεξη, μετονομάζουμε την εκδοχή του απομακρυσμένου αποθετηρίου, και διατηρούμε την εκδοχή του τοπικού αποθετηρίου ως έχει, διασφαλίζοντας έτσι ότι κατά την επανυποβολή των υπολοίπων υποβολών δεν θα

προκύψει διένεξη για αυτό το αρχείο. Ωστόσο, στο τέλος της διαδικασίας της αναθεμελίωσης, θα πρέπει να γίνει μετονομασία της τοπικής εκδοχής, και της εκδοχής του απομακρυσμένου αποθετηρίου (ώστε η τελευταία να έχει το κανονικό όνομα του αρχείου). Επομένως θα πρέπει να παρακολουθούμε τα ονόματα των αρχείων που δημιουργούν διενέξεις κατά τη διάρκεια της αναθεμελίωσης.

- Κατά τη διάρκεια των cherry-picks, αν ένα αρχείο δημιουργήσει διένεξη, μετονομάζουμε την τοπική εκδοχή, διατηρούμε την εκδοχή του απομακρυσμένου αποθετηρίου ως έχει, και ξαναγράφουμε τις υποβολές που δεν έχουν επανυποβληθεί ακόμα, έτσι ώστε σε αυτές το αρχείο αυτό να εμφανίζεται μετονομασμένο. Έτσι, κατά τη διάρκεια των επόμενων cherry-picks, η απομακρυσμένη εκδοχή που έχουμε διατηρήσει ως έχει δεν θα επηρεαστεί, και οποιαδήποτε αλλαγή θα γίνεται στην μετονομασμένη τοπική εκδοχή, διασφαλίζοντας έτσι ότι δεν θα προκύψει διένεξη για αυτό το αρχείο. Για τη μετονομασία ωστόσο, θα χρειαστούμε τον κωδικό κατακερματισμού που χρησιμοποιείται για το αρχείο αυτό στην τελευταία υποβολή.

Θέλαμε να εξερευνήσουμε τις δυνατότητες που παρέχει το Git, οπότε επιλέξαμε να υλοποιήσουμε τη δεύτερη προσέγγιση. Επιπλέον, η προσέγγιση αυτή έχει το πλεονέκτημα των λιγότερων μετονομασιών και αλλαγών απευθείας στον κατάλογο εργασίας.

Η εντολή `git filter-branch` μπορεί να χρησιμοποιηθεί για την επανεγγραφή κλάδων. Διάφορα φίλτρα μπορούν να εφαρμοστούν και να τροποποιήσουν κάθε δένδρο (πχ διαγραφή αρχείου ή εκτέλεση ενός σεναρίου Perl για την επανεγγραφή όλων των αρχείων) ή τις πληροφορίες κάθε υποβολής. Διαφορετικά, όλες οι πληροφορίες (συμπεριλαμβανομένων των αρχικών χρονοσφραγίδων υποβολής ή πληροφοριών σχετικών με συγχώνευση) διατηρούνται. Έπειτα από την ανάγνωση του εγχειριδίου τεκμηρίωσης της εντολής, γίνεται κατανοητό ότι υπάρχουν δύο τρόποι να μετονομασθεί ένα αρχείο σε μία σειρά από υποβολές χρησιμοποιώντας την εντολή `git filter-branch`. Ο ένας είναι με τη χρήση της παραμέτρου `--tree-filter` και ο άλλος με χρήση της παραμέτρου `--index-filter`. Μπορεί να φαίνεται πιο εύκολη η υλοποίηση του πρώτου τρόπου, αλλά ο τρόπος αυτός είναι πολύ πιο αργός από τον δεύτερο, γιατί με την παράμετρο `--index-filter` το Git δεν διαβάζει τα αντικείμενα τύπου δένδρου στον κατάλογο εργασίας, αφήνοντας τον έτσι ανεπηρέαστο. Δυστυχώς αυτό σημαίνει ότι απλές εντολές τύπου `mv` ή `git mv` δεν μπορούν να χρησιμοποιηθούν, αφού βασίζονται στον κατάλογο εργασίας. Επομένως πρέπει να κατασκευαστεί μία πιο πολύπλοκη εντολή που να βασίζεται στην εντολή `git update-index`. Στο εγχειρίδιο τεκμηρίωσης υπάρχει ένα πολύ χρήσιμο παράδειγμα μίας τέτοιας εντολής, για τη μετακίνηση ενός ολόκληρου δένδρου σε έναν υποκατάλογο, ή τη διαγραφή του από εκεί, όπως φαίνεται στο Listing 4.5.

```
git filter-branch --index-filter \
    'git ls-files -s | sed "s-\t\"*-&newsubdir/-" |
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update-index --index-info &&
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

Listing 4.5: Παράδειγμα εντολής filter-branch

Με βάση αυτή την εντολή, το Gitsync χρησιμοποιεί αυτή που φαίνεται στο Listing 4.6.

Η εντολή `git ls-files -s` χρησιμοποιείται για να τυπώσει μία λίστα από αρχεία που είναι προς υποβολή. Έπειτα, με τη χρήση του εργαλείου sed, η λίστα αυτή τροποποιείται, με την αντικατάσταση του παλιού ονόματος του αρχείου με το καινούριο όνομα, και έπειτα η λίστα αυτή τροφοδοτεί την είσοδο της εντολή `git update-index`, η οποία σε συνδυασμό με την παράμετρο `--index-info` μπο-

```
git filter-branch --index-filter \
    'git ls-files -s | sed "s|\<old_file_name>\(.*\)|\<new-file-name>\1|" | \
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update-index --index-info && \
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' -f <branch_to_filter>
```

Listing 4.6: Εντολή filter-branch που χρησιμοποιεί το Gitsync

ρεί να τη διαβάσει. Αναλυτική περιγραφή της εντολής `filter-branch` που χρησιμοποιεί το Gitsync μπορεί κανείς να βρει εδώ [12].

Για να μην επηρεαστεί ο κλάδος `master` από της εντολή `git filter-branch`, το Gitsync δημιουργεί έναν προσωρινό κλάδο, με το όνομα `cherry-pick-temp` αμέσως πριν επαναφέρει τον κλάδο `master` στην υποβολή που δείχνει ο κλάδος `origin/master`. Στο τέλος της διαδικασίας της αναθεμελίωσης, ο προσωρινός αυτός κλάδος `cherry-pick-temp` διαγράφεται. Επιπλέον, επειδή δεν είναι αποδοτικό να επαναγράφονται όλες οι υποβολές του κλάδου κάθε φορά, η εντολή `git filter-branch` περιορίζεται κάθε φορά στο κατάλληλο εύρος υποβολών.

Συνοψίζοντας, το Gitsync κάνει τα παρακάτω για την αυτόματη επίλυση διενέξεων κατά τη μέθοδο της αναθεμελίωσης:

1. Υπολογισμός της λίστας των υποβολών που πρέπει να επανυποβληθούν.
2. Υπολογισμός του κωδικού κατακερματισμού της υποβολής που δείχνει ο κλάδος `master`.
3. Δημιουργία κλάδου `cherry-pick-temp`.
4. Επαναφορά του κλάδου `master` ώστε να δείχνει στην υποβολή που δείχνει ο κλάδος `origin/master`.
5. Αρχή διαδικασία cherry-picking.
   (i) Απόπειρα για cherry-pick μιας υποβολής.
   (ii) Σε περίπτωση διένεξης, μετονομασία των αρχείων που δημιουργούν διένεξη σε όλες τις υποβολές που δεν έχουν ακόμη επανυποβληθεί. Για τη μετονομασία χρησιμοποιείται ο κωδικός κατακερματισμού που έχει το αρχείο στην υποβολή της οποίας ο κωδικός κατακερματισμού υπολογίστηκε στο βήμα 2.
   (iii) Διατήρηση της μετονομασμένης τοπικής εκδοχής.
   (iv) Σε περίπτωση που η διένεξη προέκυψε από το γεγονός ότι το αρχείο έχει διαγραφεί στην υποβολή που προσπαθήσαμε να κάνουμε cherry-pick, αλλά έχει τροποποιηθεί στο απομακρυσμένο αποθετήριο, πρέπει να διαγραφεί το αρχείο, ώστε η επόμενη απόπειρα cherry-pick να είναι επιτυχημένη. Επομένως το διαγράφουμε, αλλά τοποθετούμε το όνομα σε μία λίστα ώστε στο τέλος του cherry-pick να το επαναφέρουμε.
   (v) Αφού τελείωσαν οι απαραίτητες τροποποιήσεις, γίνεται υποβολή, που σηματοδοτεί το τέλος της πρώτης φάσης επίλυσης διενέξεων.
   (vi) Γίνεται cherry-pick της (επαναγεγραμμένης πλέον) υποβολής της οποίας η απόπειρα για cherry-pick απέτυχε προηγουμένως και δημιούργησε διενέξεις. Όπως και στην περίπτωση της συγχώνευσης, το cherry-pick πρέπει αυτή τη φορά να πετύχει.
   (vii) Επαναφορά των αρχείων που χρειάστηκε να διαγραφούν κατά το βήμα `5iv`. Η εκδοχή που επαναφέρεται είναι αυτή του απομακρυσμένου αποθετηρίου.

(viii) Δημιουργία καινούριας υποβολής που περιλαμβάνει τα αρχεία που μόλις επαναφέρθηκαν, και σηματοδοτεί το τέλος της δεύτερης φάσης της αυτόματης επίλυσης διενέξεων.

(ix) Συνέχεια της διαδικασία cherry-pick με τις υπόλοιπες υποβολές.

6. Διαγραφή του κλάδου `cherry-pick-temp`.

Υπάρχει ακόμη ένα πράγμα που δεν έχουμε εξηγήσει. Τι γίνεται με τα αρχεία που ανήκουν στο σύνολο automerged; Κατά την εφαρμογή της μεθόδου της αναθεμελίωσης, τα αρχεία αυτά το Gitsync τα μεταχειρίζεται όπως και τα αρχεία που ανήκουν στο σύνολο unmerged. Γιατί;

Ο τελικός χρήστης δεν ενδιαφέρεται για τις ενδιάμεσες συγχωνεύσεις αρχείων, οπότε η μέθοδος της συγχώνευσης φαίνεται να υπερτερεί έναντι της μεθόδου της αναθεμελίωσης. Ωστόσο, όπως έχουμε αναφέρει, με την συγχώνευση, δημιουργούνται υποβολές με περισσότερους από έναν γονείς, και έτσι το ιστορικό υποβολών που δημιουργείται είναι δύσκολο να αναγνωσθεί. Από την άλλη μεριά, με την αναθεμελίωση εξασφαλίζεται ότι το ιστορικό υποβολών είναι μία "ευθεία γραμμή". Αυτό σημαίνει ότι είναι πολύ πιο εύκολη μία μελλοντική υλοποίηση δυνατότητας ανάγνωσης και επαναφοράς παλαιότερων εκδόσεων αρχείων από τον χρήστη. Σε μία τέτοια περίπτωση όμως, τα αρχεία που θα έχουν προκύψει από την αυτόματη συγχώνευση του Git θα δημιουργούν σύγχυση.

Ας υποθέσουμε ότι υπάρχουν 4 υποβολές που πρέπει να επανυποβληθούν κατά την αναθεμελίωση, οι υποβολές `A`, `B`, `C` και `D`. Η υποβολή `A` δεν επηρεάζει το αρχείο `foo`. Στις υποβολές `B` και `C`, το αρχείο `foo` μπορεί να συγχωνευθεί αυτόματα από το Git. Επομένως το αποτέλεσμα της αυτόματης συγχώνευσης διατηρείται, με την πρόθεση στο τέλος της διαδικασίας της αναθεμελίωσης να επαναφέρουμε την τοπική και την απομακρυσμένου αποθετηρίου εκδοχές μετονομασμένες. Ωστόσο, κατά την επανυποβολή της υποβολής `D` προκύπτει διένεξη στο αρχείο `foo`, οπότε και αυτό αντιμετωπίζεται όπως έχουμε εξηγήσει ως αρχείου του συνόλου unmerged. Δηλαδή η τοπική εκδοχή μετονομάζεται και η εκδοχή του απομακρυσμένου αποθετηρίου διατηρείται ως έχει. Τώρα, κατά τη διάσχιση του ιστορικού υποβολών, στο αρχείο `foo` υπάρχει ασυνέπεια. Για να είμαστε ακριβής, υποθέτοντας ότι οι επανυποβολές των υποβολών `A`, `B`, `C` και `D` δημιουργούν στο ιστορικό τις υποβολές `A'`, `B'`, `C'` και `D'`:

- Η υποβολή `A'` έχει την εκδοχή του `foo` που υπήρχε στο απομακρυσμένο αποθετήριο.
- Η υποβολή `B'` έχει μία εκδοχή του `foo` που προέκυψε από αυτόματη συγχώνευση.
- Η υποβολή `C'` έχει μία εκδοχή του `foo` που προέκυψε από αυτόματη συγχώνευση.
- Η υποβολή `D'` έχει την εκδοχή του `foo` που υπήρχε στο απομακρυσμένο αποθετήριο, και μία μετονομασμένη τοπική εκδοχή του `foo`.

Για τον παραπάνω λόγο, το Gitsync δεν επιτρέπει το συνδυασμό των παραμέτρων `--automerge` και `--rebase`.

Στον Πίνακα 4.1 και στον Πίνακα 4.2 που ακολουθούν, συνοψίζουμε τον τρόπο με τον οποίον επιλύονται οι διενέξεις με τη χρήση της μεθόδου της συγχώνευσης και της αναθεμελίωσης αντίστοιχα. Χρησιμοποιούμε το συμβολισμό `<κατάσταση-στο-τοπικό>/<κατάσταση-στο-απομακρυσμένο>` για να εξηγήσουμε τον λόγο για τον οποίο προκύπτει μία διένεξη. Για παράδειγμα η έκφραση τροποποιημένο/διαγραμμένο σημαίνει ότι υπάρχει διένεξη διότι το αρχείο έχει τροποποιηθεί στο τοπικό αποθετήριο, αλλά έχει διαγραφεί στο απομακρυσμένο αποθετήριο. Πρέπει να επισημανθεί ότι κατά τη μέθοδο της αναθεμελίωσης, το Gitsync επαναφέρει τον κλάδο `master` εκεί που δείχνει ο κλάδος `origin/master`. Αυτό σημαίνει ότι μία μετονομασία ενός αρχείου στον κατάλογο εργασίας ισοδυναμεί ουσιαστικά με μετονομασία της εκδοχής του απομακρυσμένου αποθετηρίου.

		κλάδος `master`	
		Το αρχείο υπάρχει	*Το αρχείο δεν υπάρχει*
κλάδος `origin/master`	*Το αρχείο υπάρχει*	Διένεξη τύπου τροποποιημένο/τροποποιημένο. Διατήρηση της απομακρυσμένης εκδοχής ως έχει, μετονομασία τοπικής εκδοχής.	Διένεξη τύπου διαγραμμένο/τροποποιημένο. Διατήρηση απομακρυσμένης εκδοχής ως έχει.
	Το αρχείο δεν υπάρχει	Διένεξη τύπου τροποποιημένο/διαγραμμένο. Μετονομασία της τοπικής εκδοχής.	Διένεξη τύπου μετονομασμένο/μετονομασμένο. Χρήση της παραμέτρου `-Xno-renames`. Έτσι η διένεξη μετατρέπεται σε τύπου διαγραμμένο/διαγραμμένο, που ουσιαστικά δεν είναι διένεξη.

Πίνακας 4.1: Επίλυση διένεξης με τη μέθοδο της συγχώνευσης

		υποβολή για την οποία γίνεται απόπειρα cherry-pick	
		Το αρχείο υπάρχει	*Το αρχείο δεν υπάρχει*
κλάδος `origin/master`	*Το αρχείο υπάρχει*	Διένεξη τύπου τροποποιημένο/τροποποιημένο. Χρήση εντολής filter-branch για μετονομασία του αρχείου στις υποβολές που δεν έχουμε κάνει cherry-pick ακόμα (μετονομάζοντας έτσι ουσιαστικά την τοπική εκδοχή του αρχείου) και διατήρηση της απομακρυσμένης εκδοχής (του κλάδου `origin/master`) ως έχει.	Διένεξη τύπου διαγραμμένο/τροποποιημένο. Διαγραφή της απομακρυσμένης εκδοχής (του κλάδου `origin/master`), και τοποθέτηση του ονόματος του αρχείου σε μία λίστα, ώστε να γίνει επαναφορά της εκδοχής αυτής στο τέλος της διαδικασίας cherry-pick.
	Το αρχείο δεν υπάρχει	Διένεξη τύπου τροποποιημένο/διαγραμμένο. Χρήση εντολής filter-branch για μετονομασία του αρχείου στις υποβολές που δεν έχουμε κάνει cherry-pick ακόμα (μετονομάζοντας έτσι ουσιαστικά την τοπική εκδοχή του αρχείου).	Διένεξη τύπου μετονομασμένο/μετονομασμένο. Χρήση της παραμέτρου `-Xno-renames`. Έτσι η διένεξη μετατρέπεται σε τύπου διαγραμμένο/διαγραμμένο, που ουσιαστικά δεν είναι διένεξη.

Πίνακας 4.2: Επίλυση διένεξης με τη μέθοδο της αναθεμελίωσης

Κεφάλαιο 5

Επίλογος και μελλοντικές κατευθύνσεις

Σε αυτή τη διπλωματική εργασία μελετήσαμε τον τρόπο με τον οποίο το εργαλείο Git, ένα κατανεμημένο Σύστημα Ελέγχου Εκδόσεων, λειτουργεί και περιγράψαμε μία βασική ροή εργασίας με αυτό, καθώς και πώς επιλύονται τυχόν διενέξεις. Επιπλέον, εξερευνήσαμε τις δυνατότητες λειτουργίας του Git με πρωτόκολλα άγνωστα σε αυτό, με τη βοήθεια των βοηθών επικοινωνίας του Git με απομακρυσμένα αποθετήρια και υλοποιήσαμε το rawobjects, ένα δικό μας πρωτόκολλο. Έπειτα πειραματιστήκαμε με τον αυτόματο κατανεμημένο συγχρονισμό αρχείων, κατασκευάζοντας ένα δικό μας εργαλείο, το Gitsync, το οποίο στηρίζεται στο Git για τη διαχείριση και διανομή αρχείων, καθώς και για την ανίχνευση διενέξεων. Υλοποιήσαμε έναν δικό μας τρόπο αυτόματης επίλυσης διενέξεων, χωρίς την ανάγκη ανθρώπινης παρέμβασης, και δοκιμάσαμε δύο διαφορετικές προσεγγίσεις, αυτή της συγχώνευσης και αυτή της αναθεμελίωσης, ώστε να επιτύχουμε την επίλυση των διενέξεων. Τέλος, παρείχαμε στο εργαλείο μας τη δυνατότητα λειτουργίας με διαφορετικούς μηχανισμούς αποθήκευσης, με τη βοήθεια των βοηθών επικοινωνίας του Git με απομακρυσμένα αποθετήρια και του πρωτοκόλλου rawobjects.

Ωστόσο, το Gitsync δεν είναι ολοκληρωμένο, και έχει αρκετές προοπτικές βελτίωσης. Εργασίες που μπορεί να γίνουν στο μέλλον περιλαμβάνουν:

- **Καλύτερος χειρισμός μεγάλων αρχείων**
  Δεν ασχοληθήκαμε ιδιαίτερα με τον συγχρονισμό μεγάλων δυαδικών αρχείων. Ένα αποθετήριο του Git περιλαμβάνει κάθε εκδοχή κάθε αρχείου. Ωστόσο κάτι τέτοιο δεν είναι βολικό για ορισμένους τύπους αρχείων. Πολλαπλές εκδοχές μεγάλων αρχείων αυξάνουν τους χρόνους κλωνοποίησης και λήψης, μετατρέποντας έτσι πιθανώς την πρώτη απόπειρα συγχρονισμού ενός καταλόγου σε μία χρονοβόρα διαδικασία.

- **Εξερεύνηση του ιστορικού**
  Δεδομένου ότι το Git είναι ένα Σύστημα Ελέγχου Εκδόσεων, είναι λογικό να επιθυμούμε την εκμετάλλευση των δυνατοτήτων του, και την παροχή στους χρήστες της δυνατότητας για εξερεύνηση του ιστορικού. Αυτό σημαίνει ότι οι χρήστες θα μπορούν εύκολα να δουν και να επαναφέρουν παλαιές εκδοχές των αρχείων τους.

- **Υποστήριξη περισσότερων μηχανισμών αποθήκευσης**
  Μέχρι τώρα έχουμε υλοποιήσει την υποστήριξη ορισμένων μηχανισμών αποθήκευσης, αλλά υπάρχουν ακόμη πολλοί και διαφορετικοί μηχανισμοί προς υποστήριξη.

- **Υλοποίηση Γραφικού Περιβάλλοντος**
  Επειδή επιθυμούμε το εργαλείο μας να είναι εύχρηστο και από άτομα που δεν ασχολούνται

ιδιαίτερα με την τεχνολογία των υπολογιστών, η δημιουργία ενός απλού, πλήρως λειτουργικού, και καλαίσθητου Γραφικού Περιβάλλοντος είναι εργασία υψηλής προτεραιότητας.

Chapter 6

Introduction

6.1 Distributed File Synchronization

In this paper we try to analyze the problem of distributed file synchronization, and make an attempt to solve it using Git in such a way that enables us to use different types of storage backends, and automate the process of synchronization.

File synchronization is the process of ensuring that computer files in two or more locations are updated via certain rules. Most of the time the desired result is that of keeping the two (or more) locations identical to each other. As technology advances, and devices become "smarter", there is an ever-increasing need of being able to modify a file, e.g. a document, in one device and see that modification reflected in all the other devices one may own. File synchronization makes that possible.

There are two ways to synchronize files between two locations. Either let the locations communicate directly (decentralized synchronization) or use a third location as a middleman, synchronizing each location with this third location (centralized synchronization). In the latter case, the locations are organized in a star topology, with the location-middleman being the central node of the topology.

In distributed file synchronization, different source locations may have different versions of the same file differing by a number of edits, raising certain problems during the process of synchronization. This kind of problems are called *conflicts*, and the way these conflicts are resolved is called *conflict resolution*. In order to automate the process of synchronization, one must detect conflicts, and find a way to automate conflict resolutions, which is not an easy task.

There are services and programs (like Git) that make conflict detection effortless, but require human intervention to resolve conflicts. Expanding on that idea, other services (like Git annex [8]) provide automatic conflict resolution, but synchronization is something left for the user to do manually, sometimes using complex (for the average user) commands. Then there are implementations that automate both file synchronization and conflict resolution, but are proprietary, meaning one cannot see or modify the source code. Such solutions lack portability, and may raise privacy concerns.

We use Git to detect conflicts, and automate the file synchronization process, based on the centralized approach. We also found a way to automatically resolve conflicts, reflecting these resolutions to all synced locations. To make our implementation portable, we rely on git-remote-helpers, scripts that instruct Git how to operate over protocols natively unknown to it. As proof of concept, we wrote remote helpers capable of synchronizing locations using a directory on a local machine and Amazon S3 (Amazon Simple Storage Service) [1] as storage backends respectively. Moreover, to demonstrate

that it is possible to eliminate privacy concerns raised by storing files at remote locations (such as Amazon S3), we wrote a remote helper that uses Amazon S3 as storage backend, but encrypts any content stored there.

6.2 What's next?

The rest of the thesis is organized as follows. In Chapter 7 the Git tool is presented and the way it internally works is described. We also describe a basic Git workflow, and the way Git handles conflicts. Chapter 8 is about designing an automated file synchronization mechanism. We explore the way Git communicates with remote repositories over protocols natively unknown to it. For that purpose, the concept of Git remote helpers and their API are introduced. Furthermore we describe the way conflicts during synchronization should be automatically resolved. Then, in Chapter 9 we present our tool for automated, distributed file synchronization, Gitsync, and describe how it utilizes Git in order to achieve file synchronization. We also take advantage of the API presented in the previous chapter to implement our own protocol, that Gitsync uses to communicate with remote repositories. Finally, in Chapter 10 we present our concluding remarks and future directions.

Chapter 7

Background

7.1 What is Git

As already mentioned in the Introduction, in order to achieve Distributed File Synchronization, one should be able to detect conflicts between files. This is one of the reasons we chose to use Git. Other reasons include the fact that it is a free and open source project, widely known, very well documented, with active community and a lot of useful material available on the Internet. Finally it will enable us to use multiple storage backends, with the help of git-remote-helpers.

Git is a *Version Control System (VCS)* that is used for software development and other version control tasks. It was created by Linus Torvalds in 2005, and as a Distributed Revision Control System it is aimed at speed, data integrity, and support for distributed, non-linear workflows.

Distributed means that unlike most client-server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

As a VCS, Git is capable of detecting conflicts when they occur, but (usually) requires human intervention to solve them.

7.1.1 How Git Works

Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time something is commited (or the state of a project is saved in Git) Git basically takes a picture of what all the files look like at that moment and stores a reference to that snapshot. To be efficient, if a file has not changed, Git does not store it again, just a link to the previous identical file it has already stored. At the end of the day, the data Git keeps is like a stream of snapshots.

Git is fundamentally a *content-addressable filesystem* with a VCS user interface written on top of it. It means that at the core of Git is a simple key-value data store. One can insert any kind of content into it, and get back a key that can be used to retrieve the content again at any time.

When `git init` is executed in a new or existing directory, Git creates the `.git/` directory, which is where almost everything that Git stores and manipulates is located. The structure of the `.git/` directory can be seen in Listing 7.1.

To briefly describe the entries in Listing 7.1:

```
.git/
 |- config
 |- description
 |- HEAD
 |- hooks/
 |- index
 |- info/
 |- objects/
 |- refs/
```

Listing 7.1: The structure of a `.git/` directory

- `description` is just a text file that is shown as project description in web frontend.
- The `config` file contains project-specific configuration options.
- The `info/` directory keeps a global exclude file for ignored patterns whose track via a .gitignore file is not desired.
- The `hooks/` directory contains client- or server-side hook scripts, that run after specific events.

The remaining 4 entries are the most important.

- The `objects/` directory stores all the content for the database.
- The `refs/` directory stores pointers into commit objects in that data (branches).
- The `HEAD` file points to the currently checked out branch.
- The `index` file is where Git stores the staging area information.

We will now look at each of those 4 entries in detail, to see how Git operates.

7.2 Git objects

It has already been mentioned that Git stores objects and gives back a key, so as to be able to refer to those objects later. This key is actually a 40-character *checksum hash*, the SHA1 hash - a checksum of the content being stored plus a header, which differs according to the type of object. There are 4 types of objects: *blobs*, *trees*, *commits*, and *tags*. We will talk about those types in a while. Git also compresses the content (plus the header) with zlib before storing it.

Objects are stored under the `.git/objects/` directory, using the following format:

`<2 first chars of object SHA1>/<rest 38 chars of object SHA1>`

Why chose such a format? Git needs to be portable. This means it should work under many filesystems, including FAT32, NTFS, EXT2/3/4. There are different limits of files per directory for each filesystem. For example FAT32 can only store up to 65,535 files in a single directory. This means that it is necessary to subdivide the directory structure so that filling a single directory is less likely. In addition, there are filesystems (especially old ones like FAT and EXT2) that use directory structures which require a linear scan of the directory blocks in searching for a particular file. The lookup and

update performance of these unindexed formats degrades linearly with the size of the directory. Dividing objects in the way Git does, ensures that scanning of `.git/objects/` and its subdirectories is fast even for old filesystems.

7.2.1 Blobs

Git `blob` is the basic data storage unit. The blob type is just a bunch of bytes that could be anything (for example a text file, source code, executable binary file, or picture, etc.). The format of a blob object can be seen in Figure 7.1.

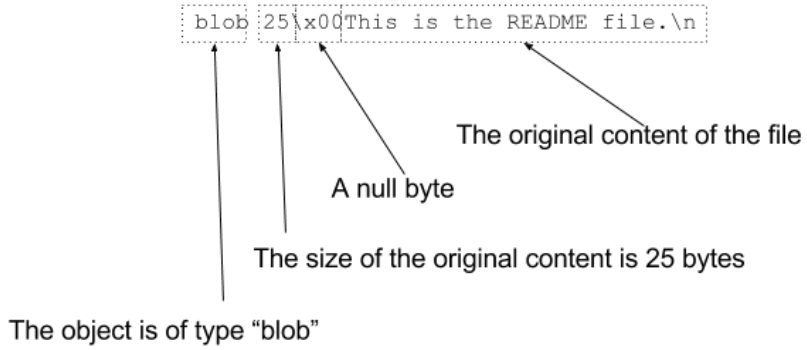


Figure 7.1: The format of a blob object

It is worth noting, that Git does not store the name of the file in a blob object. It stores files using a hash derived by their content. This method of storing files based on their content (instead of the original filename) is called *content addressable storage*.

Also, the fact that it is just the file's content that is stored, allows Git to reuse blobs for many files with different names but same content. If another file with the exact same content is added to Git, Git will notice that the hash is identical (identical file contents result in an identical hash) and will not need to store another copy of the file in its object store.

7.2.2 Trees

A Git *tree* object is very similar to a filesystem directory. Filesystem directories refer to other directories and files, while Git trees refer to other Git trees and Git blobs. A single tree object contains SHA1 pointers to blobs or subtrees, with associated mode, type, and filename. The format of a tree object can be seen in Figure 7.2.

Git normally creates a tree by taking the state of the staging area (or index) and writing a series of tree objects from it. So it is time to talk about the index.

7.2.3 Git index

Git index has many names. It may be referred to as Cache, Directory cache, Current directory cache, Staging area, Staged files. The *index* (or any of its other names) is essentially a “holding area” for

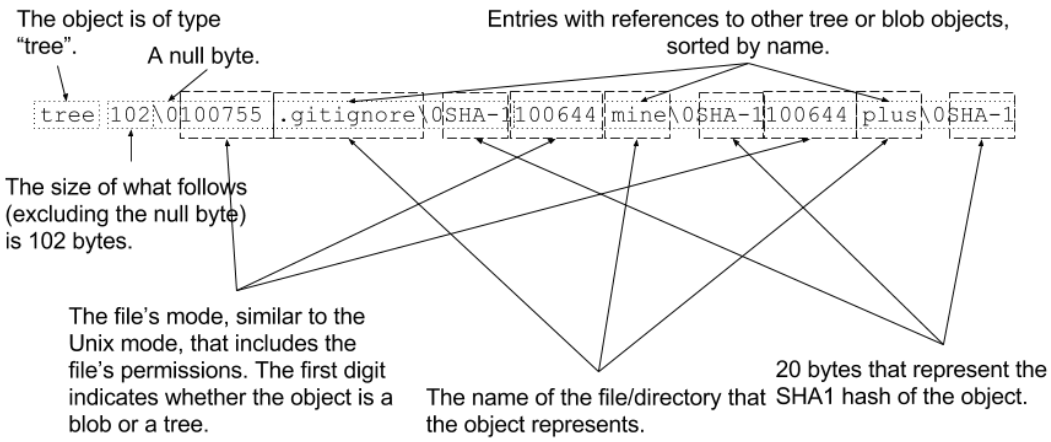


Figure 7.2: The format of a tree object

changes that will be committed during the next `git commit` command. That is, unlike other VCSs, a commit operation does not simply take the current working tree and check it as-is into the repository. The index provides control over what parts of the working tree will go into the repository on the next commit operation.

The index is actually a "virtual tree":

- It contains references to blobs
- It has a performance oriented format
  - Git needs to quickly and efficiently compute whether the state of the working tree has changed since the last index.
  - Some tools, including the bash shell prompt, need to be able to determine if the working tree is dirty or not quickly.

The index stores not only the files' names (in relative path format), but also their last modification time. SHA1 hashes of each blob are also stored. This allows the index to update itself, should the file be reverted to a previous state but with a later timestamp.

Exact format of index file can be found in the documentation, in Git's source code [4].

Using the common command `git status` one can see paths that have differences between the index file and the current `HEAD` commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git (and are not ignored by `.gitignore`). The first are what would be commited by running `git commit`; the second and third are what could be commited by running `git add` before running `git commit`. This practically means that files can be added to the index using the `git add` command.

7.2.4 Commits

Now that we explained what blobs, trees and the index are, we can explain the core part of a Git graph, the commit object. A *commit* object is like a pointer to a snapshot of a filesystem.

When a `git commit` command is executed (to "check in" the updates to the repository), Git creates a new commit object, which is also saved to the Git repository. The commit object includes, at a minimum:

- The hash of the Git tree that represents the state of the index at the time of the commit.
- Author: The name and email address of the one who originally wrote the changes to be applied by the commit, along with a date/time.
- Commiter: The name and email address of the one who applied the commit, along with a date/time.
- Comment: A text comment that summarizes the reason for the commit (for example "Feature #412 implemented!").

The format of a commit object can be seen in Figure 7.3.

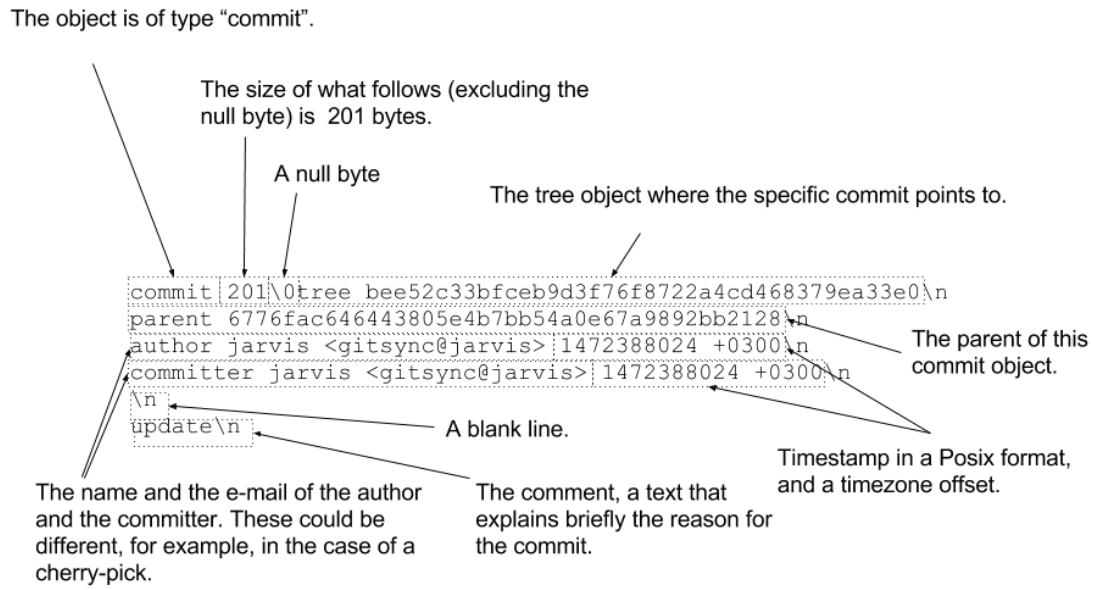


Figure 7.3: The format of a commit object

7.2.5 Tags

Tags are like pointers to commits. They can be used to easily refer to commits. Git uses two types of tags: lightweight and annotated.

A *lightweight tag* is very much like a branch that does not change - it is just a pointer to a specific commit. We will talk about branches later.

Annotated tags on the other hand are stored as full objects in the Git database. Like all other objects, they are checksumed, and they contain the following information:

- The name of the tag (such as 1.0beta).

- A commit that the tag refers to (such as 126af20).
- A tag message (such as "This is 1.0beta").
- The tagger's name, email address and the time the tag was added.

7.2.6 Packfiles

The way we have described that Git uses to store objects has a drawback: It uses up lots of space to store blobs, even though objects are compressed using zlib.

To demonstrate the problem, we can save a large file `foo`, and use `git add` to add it to Git's database as a blob. Then we modify one word from `foo`, and run `git add` again. Git creates a new blob object, similar to the first one, thus consuming a lot of space for such a little modification.

The initial format in which Git saves objects on disk is called a *loose object format*. However, occasionally Git packs up several of these objects into a single binary file called a *packfile* in order to save space and be more efficient. Git does this if there are too many loose objects around, when the `git gc` command is used manually, or when pushing to a remote server.

According to the "git-pack-object" command documentation:

> A packed archive is an efficient way to transfer a set of objects between two repositories as well as an access efficient archival format. In a packed archive, an object is either stored as a compressed whole or as a difference from some other object. The latter is often called a delta.
>
> The packed archive format (.pack) is designed to be self-contained so that it can be unpacked without any further information. Therefore, each object that a delta depends upon must be present within the pack.
>
> A pack index file (.idx) is generated for fast, random access to the objects in the pack.

Although packfiles are designed to be self-contained, there is an exception, called *thin packfile*. Thin packfiles contain deltas, without containing the objects that were used as a base for the delta production. In this case, those objects are expected to be present in some form in the place where such packfile gets uncompressed. Packfiles are stored under `.git/objects/pack/`, and when they are created, the objects they contain are no longer needed and are deleted from the filesystem. The name of a packfile is `pack-<sha1>`, where according to Git's source code [6] <sha1> is the SHA1 hash of sorted object names within the packfile.

However packfiles have limits as far as the number of the objects they contain is concerned. This means that if there are many loose objects, there might be a need for many packfiles to be created. In this case, according to the developers of Git [5] Git depends a lot on "recency" (which object is more recent) in order to pack objects efficiently. More recent things on top of packfile, results to better I/O patterns.

> -first sort by type. Different objects never delta with each other.
> -then sort by filename/dirname…
> -then if we are doing "thin" pack, the objects we are _not_ going to pack but we know about are sorted earlier than other objects. -and finally sort by size, larger to smaller.

> That's the sort order. What this means is:
> -we do not delta different object types.
> -we prefer to delta the objects with the same full path, but allow files with the same name from different directories.
> -we always prefer to delta against objects we are not going to send, if there are some.
> -we prefer to delta against larger objects, so that we have lots of removals.
>
> […]
>
> We start by getting a list of the objects we want to pack, we sort it by this heuristic (basically lexicographically on the tuple (type, basename, size)).
> Then we walk through this list, and calculate a delta of each object against the last n (tunable parameter) objects, and pick the smallest of these deltas.
> And then once we have picked a delta or fulltext to represent each object, we re-sort by recency, and write them out in that order.

The format of a packfile can be seen in Figure 7.4.

7.2.7 Packfile Index files

As stated before, `.idx` files are used in order to achieve fast, random access to the objects stored in a packfile. For the purposes of this thesis, there is no need to use packfiles, thus we will not analyze `.idx`. The format of an `.idx` file however can be seen in Figure 7.5.

7.3 Git references

Git references are pointers to commits. They are files stored under `.git/refs/` that contain the SHA1 of the commit they point to. As stated before, lightweight tags are references. Branches, that are so often used during the Git workflow, are also references. The difference between lightweight tags and branches is that the former never moves - it always points to the same commit but gives it a friendlier name.

7.3.1 Local references

Local branches are stored as files under `.git/refs/heads/`. The branch's name is also used as the file's name, and the content of the file is just the SHA1 of the commit the reference is supposed to point at.

When `git branch <branchname>` is executed, Git internally uses the `git update-ref` command to create a file under `.git/refs/heads/` with the name <branchname>. However there must be a way for Git to know the SHA1 of the last commit. This is why the `HEAD` file exists.

7.3.2 The HEAD file

The `HEAD` file is stored under `.git/` and is a *symbolic reference* to the branch the user is currently on. By symbolic reference, we mean that unlike a normal reference, it does not generally contain a SHA1

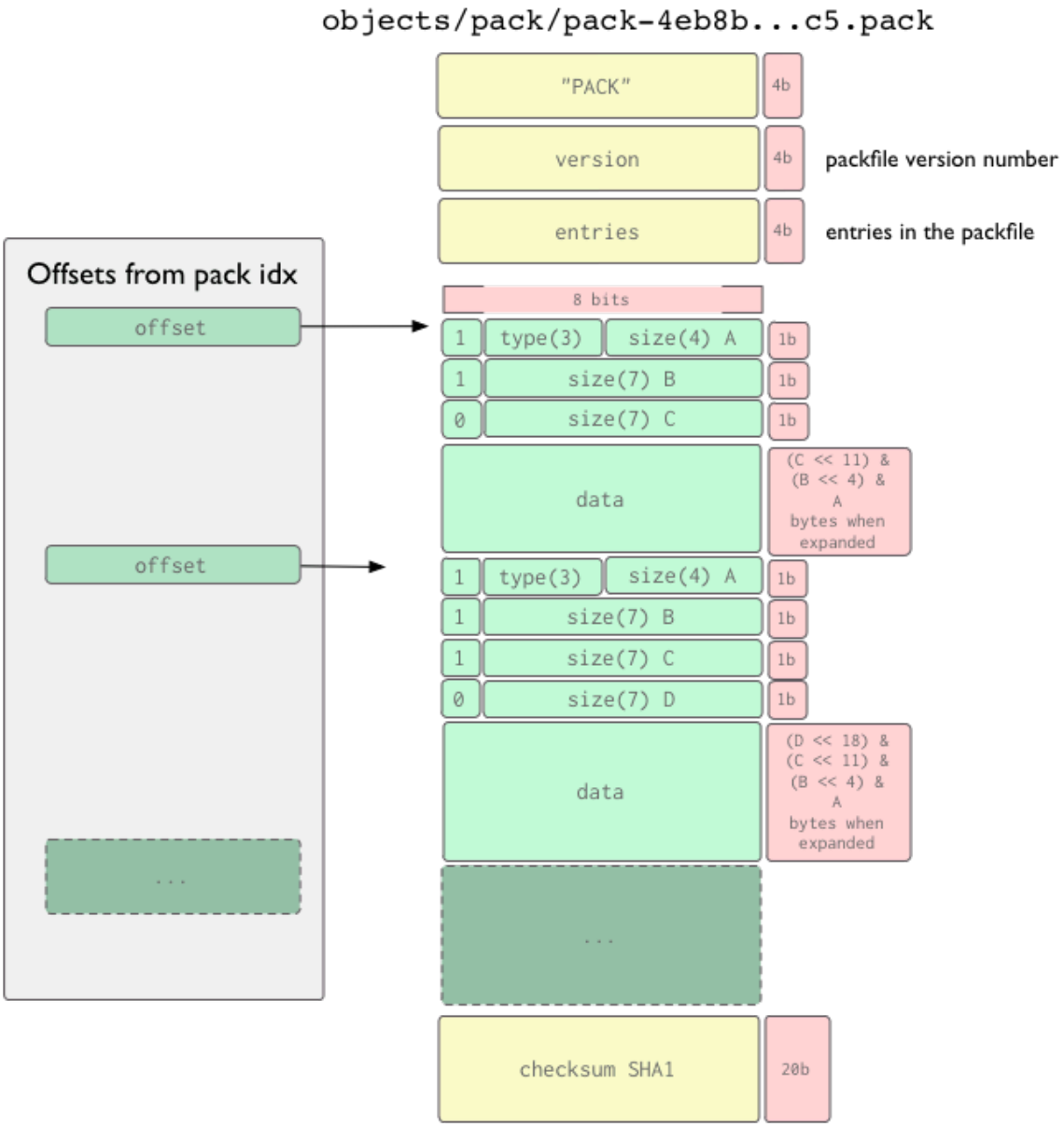


Figure 7.4: The format of a packfile

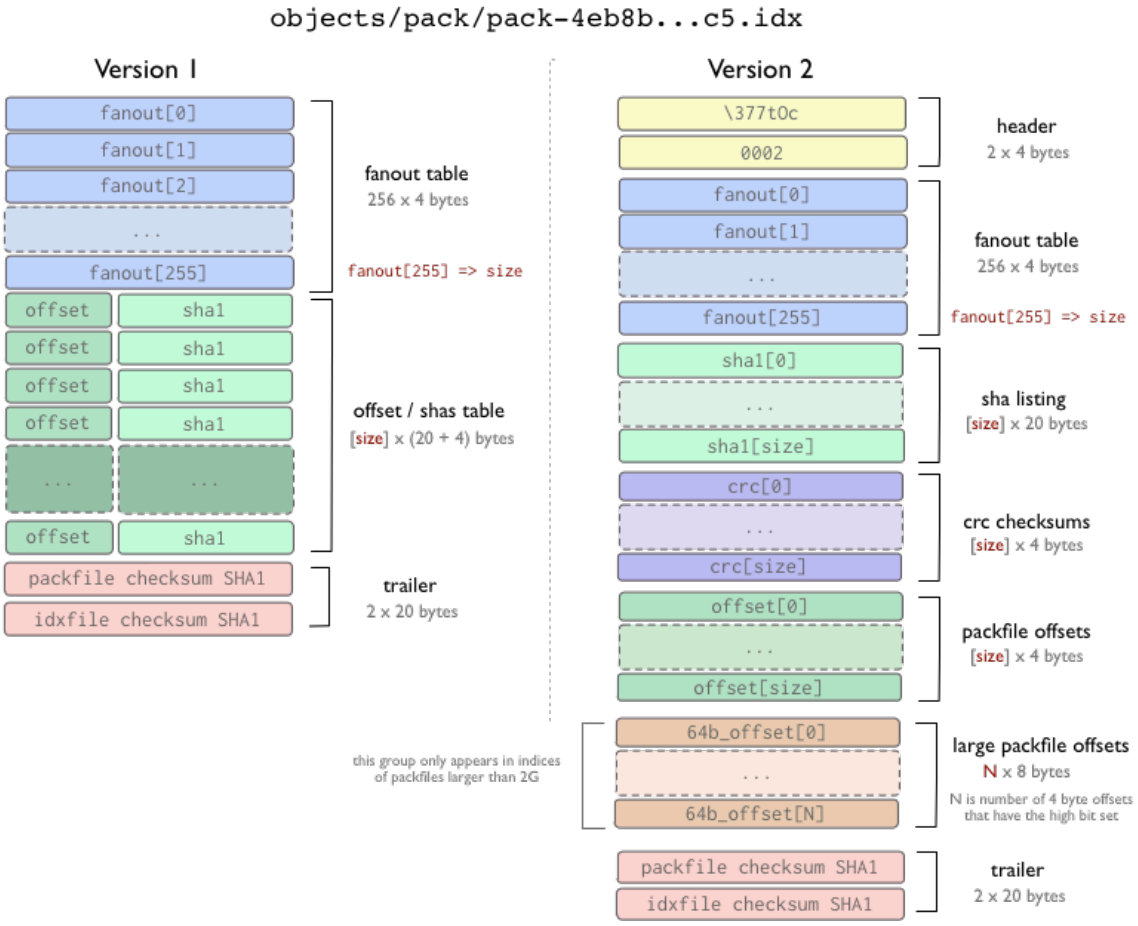


Figure 7.5: The format of a packfile index file

value but rather a pointer to another reference. The content of the HEAD file is generally like that of Listing 7.2.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Listing 7.2: Example content of the .git/HEAD file

However HEAD can also be a direct pointer to a commit. Git calls that *detached HEAD state*. This happens when git checkout <SHA1 of commit> command is used.

7.3.3 Remote references

In a way similar to local branches (heads), remote references are stored under .git/refs/remotes/. They are used to know where a remote's branch was pointing at the last time Git communicated with that specific remote. For instance, when git fetch or git push command is used, the appropriate remote references are updated, according to the *refspec* (we will explain what refspec is in the following section).

Remote references differ from local references mainly in that they are considered read-only. One can git checkout to one, but Git won't point HEAD at one, so it will never be updated with a commit command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

7.3.4 The Refscpec

The *refspec* is of the format <src>:<dst> and is a mapping of references. In case of a fetch, <src> is the pattern for references on the remote side and <dst> is where those references will be written locally. In case of a push, <src> is the pattern for references locally, and <dst> is the pattern for references on the remote side. It is a way to tell Git which references to update during pushing/fetching. In .git/config, a section similar to the one of Listing 7.3 can be found.

```
[remote ”origin”]
    url = ...
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Listing 7.3: Example of a remote repository configuration in the .git/config file

The configuration of Listing 7.3 means that if Git try to fetch from the remote repository named origin, it will update all the remote references that reside under the .git/refs/remotes/origin/ directory, using as source of information references that are stored under .git/refs/heads/ directory on the remote repository. In a similar way, if a push operation to the remote repository with the name origin is attempted, Git will use the reference .git/refs/heads/master as source of information, in order to update the reference .git/refs/heads/qa/master that reside at the remote repository.

7.4 Creating a Git repository

Git is a distributed version control system. Despite the overwhelming power it gives to an advanced user, it can also easily be used by less knowledgeable people, using basic commands and following a basic workflow. We will explain those basic commands, and how the basic workflow results in a graph consisting of Git commit objects and references to them. Then we will go a little further to explain how Git deals with conflicts during merging.

To begin with, a Git repository must be created. There are two ways to create a Git repository. One would be to first create an empty repository and then optionally update it with changes from a remote repository, and the other would be to directly clone a remote repository to a local machine.

The former can be done using `git init <local_path>`, where `<local_path>` is the path to the directory that is about to become a Git repository. Then using `git remote add origin <remote_url>`, a remote is configured. This means that instead of using the `<remote_url>` each time one needs to refer to the remote repository, they can just refer to it as `origin`. `origin` is an arbitrary name. Then the command `git pull origin` is used to get locally the latest changes made to the remote repository. The `git pull` command will later be explained.

The clone method is based on the `git clone <remote_url>` command. Git will then create a local directory named after the remote repository, and configure a remote called `origin` using the `<remote_url>` that was given as an argument. Then it will use `git pull origin` to get the remote changes.

Regardless of the way used to create the Git repository, Git will need to know a few things about the user, like their email and name. The fact that they will be used to create commit and tag objects makes them mandatory. In order to let Git know of the user's email and name, one can use the two following commands:

```
git config user.name "<user's first and last name>"
git config user.email "<user's email>"
```

7.4.1 Basic Git workflow

Having already a local Git repository with a configured remote as `origin` (whether it was created using `git init` or `git clone`), conceptually a basic workflow is the following (branching is for now ignored, and we assume we work on `master` branch):

1. Make changes to files in the local repository.
2. Make Git aware of the changes.
3. Update the local repository with the latest changes from the remote.
4. Update the remote with the changes made locally.

In Git commands, the workflow is as follows:

1. Make changes to files in the local repository, the files `foo` and `bar`. In this step there is no need for Git to be involved.
2.
```
git add foo bar
git commit
```

3. `git pull origin`
   or
   `git fetch origin`
   `git rebase origin/master`
   The difference between the two above approaches lies in that one relies on *merging* and the other on *rebasing*. We will better analyze the merging and rebasing functions of Git.

4. `git push origin master`

7.4.2 Git graph

Notice that in step 2 Git was made aware of the changes made locally using `git add` and `git commit` commands. `git add` will create a blob in Git's database and update the index file, indicating that changes have been made to these files. Then `git commit` will use the index file to create a tree object. The created tree object represents the root directory of the repository. Then a commit object will be created, pointing to the tree object that was created using the index file, and having the commit where `HEAD` is pointing at as parent. Finally, the `master` branch (reference) will be updated to point to the newly created commit object. The result of multiple repetitions of step 2 is a graph consisting of commit objects, each pointing to the previous commit (with an exception for the very first commit, which is the root of the graph). An example of such a graph is depicted by Figure 7.6.

In Figure 7.6 there is a graph with a single branch, called `master`. The creation of more branches is an easy task, achieved by using the `git branch <new_branch_name>` command.

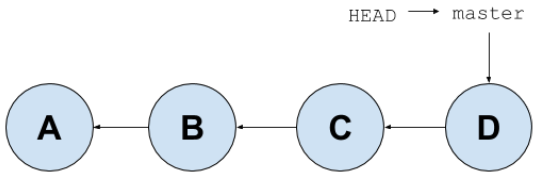


Figure 7.6: A simple Git graph

Suppose we are on `master` branch of the graph depicted by Figure 7.6. We want to make some experimental changes to the files, and we do not want `master` branch to be affected by them, until we decide so. Thus, we create a new branch to work on, using the command `git branch test`. Now a reference called `test` has been created, and it points to the same commit the `master` reference was pointing to, because when we ran the above command we were on the `master` branch. Then in order to actually change to the `test` branch we use the `git checkout test` command.

Now any commits made will update the `test` reference, leaving the `master` reference pointing to the same commit it was pointing to. Thus after a few commits our graph should look like the one in Figure 7.7.

We can still make changes to the `master` branch, without losing the changes commited while being on the `test` branch. To do so, we simply switch to the `master` branch using `git checkout master`, and make any changes and commits we desire. After some commits, our graph resembles the one depicted by Figure 7.8.

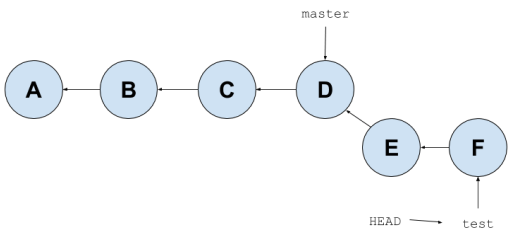


Figure 7.7: Commits made to the new `test` branch

We now decide that the changes we made on `test` branch are good, and we also want them on `master` branch. At this point we should explain merging and rebasing.

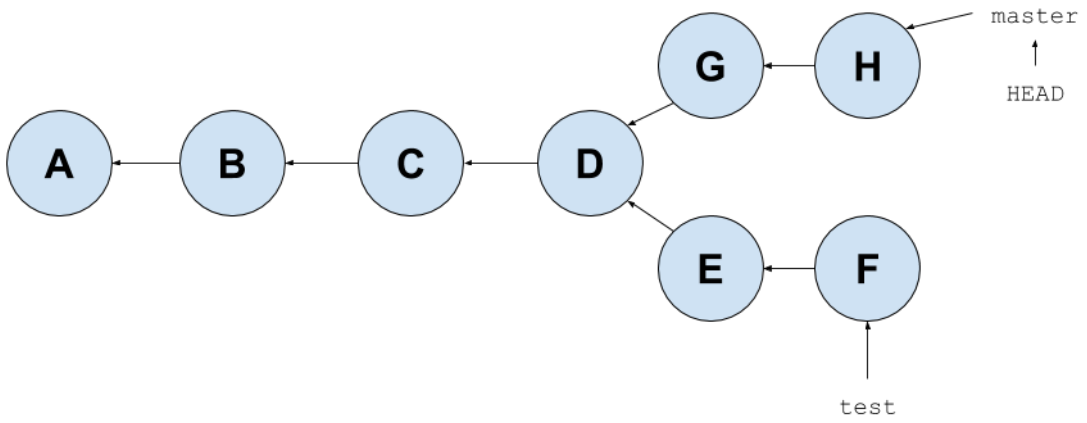


Figure 7.8: Commits made to the `master` branch

7.5 Git merge and Git rebase

`git merge` does exactly what the name suggests; it merges two branches. `git rebase` on the other hand is a more advanced command that has many uses. One of them would be to replace merge command. What it does, is reconstructing the graph starting from a point, reapplying commits on top of that specific point of the graph. In Figure 7.9 and Figure 7.10 the difference between rebase and merge is visualized.

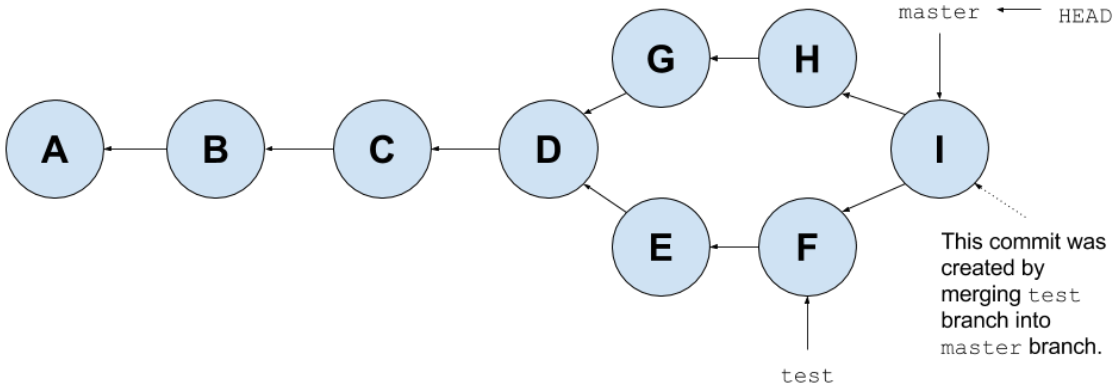


Figure 7.9: The graph of Figure 7.8 after merging `test` branch into `master` branch

As can be seen in Figure 7.9, merge will create one commit that has two parents. This commit is the result of a 3-way merge. This type of merge needs 3 commits in order to create a fourth one that is the merge of the 2 of them. Is that complicated enough? Let's clarify it, using file merging instead of branch merging. Branch merging is ultimately file merging, and works in the same way.

Suppose we have two different versions of a file named `foo`, and we want to merge them into a single version. Simply comparing the two versions does not give us enough information about what has been changed in each version, it just gives us the differences between those two versions. So if Git was trying to merge them into a single version (2-way merge) human intervention would be surely needed. However what if there was another version of `foo`, a version that both the other two version

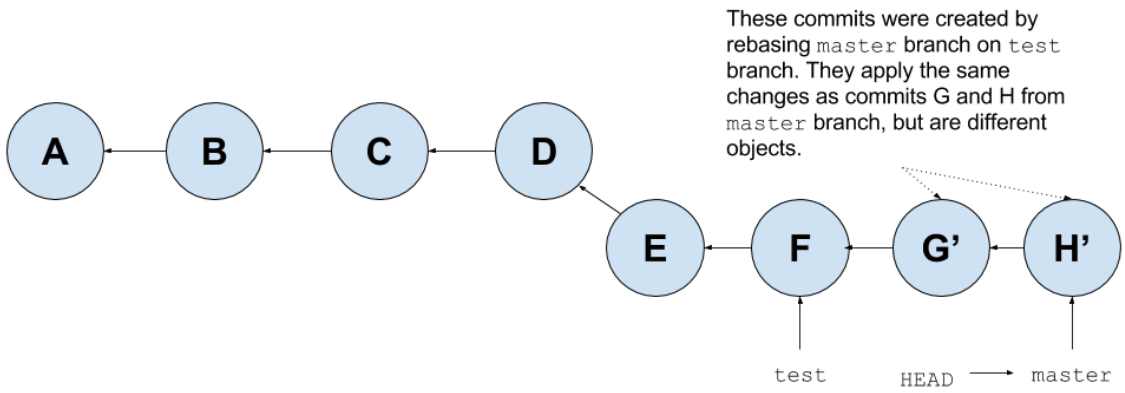


Figure 7.10: The graph of Figure 7.8 after rebasing `master` branch on `test` branch

used as a base in order to make modifications? Then, comparing the two versions, and taking into account the `base` version, could give us information about what changes each version of `foo` made over the base version, and ultimately human intervention could in some cases be avoided. This is what a 3-way merge is.

Git uses the common ancestor of the two branches we want to merge as a base version. The common ancestor is simply the commit where the branch fork was created. It then does a 3-way merge comparing the 2 commits pointed by the branches and using the common ancestor commit as base. It is worth pointing that the commit that is created by a merge is special in that it has 2 (or more) parents. This can make the graph a little complicated if many branches are used and makes rebase the preferred method in some cases.

On the other hand, during a rebase, all changes made by commits in the current branch but that are not in the branch we rebase our work on (we will refer to this branch as <*upstream*>) are saved to a temporary area. Then the current branch is reset to <upstream> (using the `git reset --hard <upstream>` command), meaning it now points to the commit <upstream> is pointing to, and the commits that were previously saved into the temporary area are now reapplied to the current branch, one by one, in order. The result is that the current branch now also has the changes of the <upstream> branch, as depicted by Figure 7.10.

As stated before, rebase creates a cleaner commit history (and graph) as there are no commits with multiple parents.

7.5.1 Fast-forward merge

We have mentioned that merge creates another commit with multiple parents. However this is not always the case.

If the commit pointed by the branch we are about to merge in is directly ahead of the commit we are on, Git simply moves the pointer forward. To phrase that another way, when we try to merge a branch pointing at commit F, into another one pointing at commit D, and commit D can be reached by following the F commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together - this is called a *fast-forward*. In this case, the result of a merge resembles what Figure 7.11 depicts.

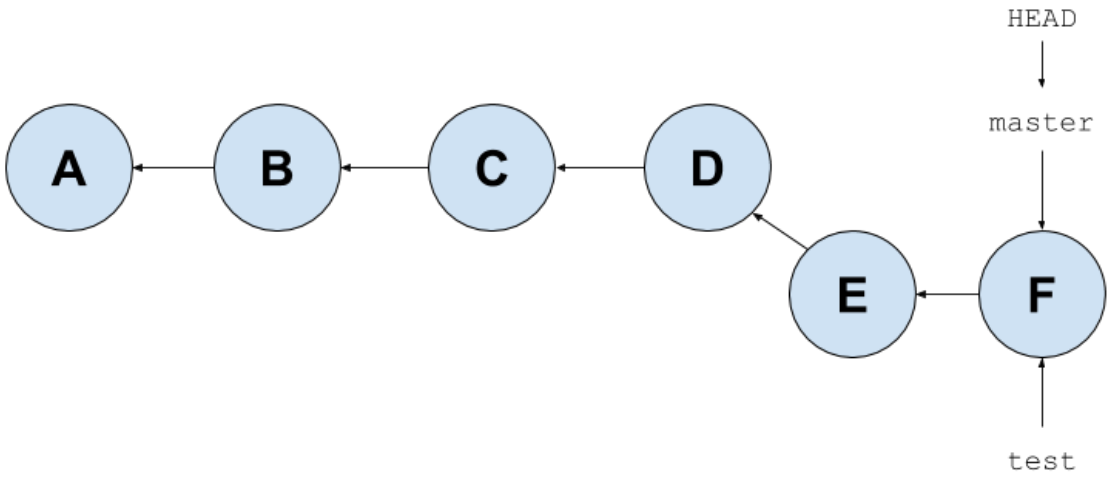


Figure 7.11: Fast-forward merge of `test` branch into `master` branch, based on the graph of Figure 7.7

7.5.2 Conflicts in Git

During rebase or merge, conflicts can arise. In both cases, the resolution concept is the same, as rebase consists in a way of multiple merges of single commits. Thus we will just explain conflict detection and resolution during merge, using an example, that is also a quite usual scenario of the Git workflow.

To begin with, we create 3 local directories `test1/`, `test2/` and `remote/`. `test1/` and `test2/` will be used as working directories, two locations we want to keep in sync. `remote/` will be used as a central remote repository to/from which both `test1/` and `test2/` push/fetch. This way, any direct communication between `test1/` and `test2/` is avoided. In order to achieve that however, the `remote/` repository should be initialized as a *bare repository*, using the command `git init --bare remote/`. The reason for that is that by default Git does not allow pushing to non-bare repositories. Bare repositories contain no working or checked out copy of files, and store Git revision history of the repository in its root directory instead of in a `.git/` subdirectory. Thus, we initialize `test1/` and `test2/` as regular Git repositories, and `remote/` as a bare repository. Then we add `remote/` as a remote to both `test1/` and `test2/`.

We then create the empty file `foo` under `test1/` and after the usual Git workflow (`git add`, `git commit`, `git push`) we change to the `test2/` repository and use `git pull`. Both repositories now have and empty file called `foo`. In order to create a conflict, we modify `foo` from `test1/`, adding a single line "This change was made from test1.", and use the common Git workflow again to push the changes to the remote repository `remote/`. Now we change into the repository `test2/`, and instead of updating it with the information we just pushed from `test1/` (using `git pull` like we did before) we modify the still empty file `foo`, adding a single line "This change was made from test2.". We now run `git add`, then `git commit` and finally try to execute the `git push` command. However `git push` fails with a message that can be seen in Listing 7.4.

In simple terms, `git push` fails because the branch we try to push (`master`) points to a commit that is not a direct descendant of the commit the `master` branch at the remote points to. In order to solve this, Git suggests that we first run the command `git pull`. This command is shorthand for `git`

```
To /remote/
 ! [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.
```

Listing 7.4: `git push` command fails

`fetch` followed by `git merge FETCH_HEAD`.

According to `git fetch` documentation:

> Git fetch will fetch branches and/or tags (collectively, "refs") from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are also updated.

So `git fetch` will just bring locally some objects and some references, but it will not affect the working directory. Then, using `git merge FETCH_HEAD` an attempt is made to merge in the changes made at the remote repositories (and that now exist locally in Git's database). `FETCH_HEAD` is a short-lived reference, used to keep track of what has just been fetched from the remote repository. `git fetch`, in normal cases fetches one branch from the remote; FETCH_HEAD points to the tip of this branch. Now that finally an attempt for merge is being made, we expect it to fail with a conflict. Indeed, using the `git pull` command, results in what can be seen in Listing 7.5.

```
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /remote/
    e8b6c10..8b5f25f  master    -> origin/master
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

Listing 7.5: Conflict during a `git merge` attempt

This means that automatically merging the file `foo` failed. This was to be expected. Using 3-way merge, Git cannot decide which change is to be kept, because the 2 versions of `foo` we try to merge have the same line modified when compared to the empty version of `foo`, that is used as base. We can manually resolve this conflict by editing `foo`, and when it is in the desired state, we can use `git add foo` to mark `foo` as resolved. Then the `git commit` command is used to store the changes in Git repository.

However we also see that Git tried "auto-merging" `foo`. This means that if the modifications were on different lines of `foo`, Git could automerge and keep all changes from all versions, thus creating a file containing all the modifications, without the need of human intervention.

Chapter 8

Design of an automated file synchronization mechanism

In an abstract level, we would like to design a tool that can achieve automated synchronization of files that are stored under user-defined directories. According to everything we have explained till now, it is clear that something like this can be done using Git as a backend.

To be specific, our tool should be able to convert a user-defined directory into a Git repository and set it up. Once all directories that are to be synced are set up, according to what we have presented in previous sections, they can stay synced, in a star topology, if Git commands are issued in the appropriate order. The user can freely add, remove or modify files and directories, while our tool periodically takes care of committing any changes, and then pushing them to the central node of the star topology, to make them available to all the other repositories. However, we need to design a way to automatically resolve any conflicts that might arise, considering that the user works in a distributed environment and conflicting changes can be made. So in this chapter, we will design a mechanism in order to automate the conflict resolution process.

Moreover, it would be very convenient, if any storage backend could be used as a central common node. There are cases where this central node cannot be a Git repository, or additional features, like data encryption, are desired. This makes the use of a custom protocol, unknown to Git, mandatory. Thus, in this chapter we will also explain how Git remote helpers can be used with a specific API, to make Git compatible with unknown protocols.

8.1 What are Git Remote Helpers

Natively Git can use four major protocols to transfer data: Local (where the remote repository is simply a directory at the local disk), HTTP, Secure Shell (SSH) and Git. What if we want to communicate with a remote repository using another protocol, unknown to Git? This can be achieved using Git Remote Helpers.

According to the documentation for git-remote-helpers [7]:

> *Git remote helpers* are helper programs to interact with remote repositories. They are not normally used directly from end users, but they are invoked by Git when it needs to interact with remote repositories Git does not support natively. When Git needs to

interact with a repository using a remote helper, it spawns the helper as an independent process, sends commands to the helper's standard input, and expects results from the helper's standard output.

In order for Git to be able to spawn remote helpers, their name format must be specific, and they should be in a path that Git looks up for executable files. The name format is `git-remote-<protocol-name>`. There is a specific API used in order for Git to communicate with the remote helper, and we will give it a thorough look in this section. We will later use remote helpers in order to achieve file synchronization over multiple storage backends.

8.1.1 Invocation

Let's assume that we want Git to communicate with (fetch from and push to) a remote repository using a protocol called `myprotocol`. In a local Git repository, we add the specified remote using the command:

```
git remote add <remote_name> <remote_url>
```

where

`<remote_name>` could be an arbitrary name, and is the name Git will use from now on to refer to this specific remote.

`<remote_url>` could be of the format `myprotocol::<remote_address>`, or of the format `myprotocol://<remote_address>`

Let us clarify the format of <remote_url>. Remote helper programs are invoked with one or (optionally) two arguments, and what those arguments will be depends on the format of <remote_url>.

- In case `<remote_url>` is of format `myprotocol://<remote_address>`, the full URL `myprotocol://<remote_address>` is passed as the second argument.
- In case `<remote_url>` is of the format `myprotocol::<remote_address>`, `<remote_address>` is passed as the second argument.
- As far as the first argument is concerned, if the remote Git communicates with is a configured remote (as in the case of using `git remote add`), then `<remote_name>` is passed as the first argument. Else, if `<remote_url>` is encountered directly on the command line, the first argument is the same as the second.

8.1.2 Input Format and API

Git sends the remote helper a list of commands on standard input, one per line. The first command is always the `capabilities` command, in response to which the remote helper must print a list of the capabilities it supports followed by a blank line. The response to the capabilities command determines what commands Git uses in the remainder of the command stream. The command stream is terminated by a blank line. In some cases (indicated in the documentation of the relevant commands), this blank line is followed by a payload in some other protocol (e.g., Git's pack protocol), while in others it indicates the end of input.

8.1.3 Capabilities

A full list of the capabilities a remote helper can support, can be found in the documentation [7].

Here we only explain basic (and most commonly used) capabilities, which we also later use to write our own remote helper scripts.

`push`
: The remote helper can discover remote refs and push local commits and the history leading up to them to new or existing remote refs.

`fetch`
: The remote helper can discover remote refs and transfer objects reachable from them to the local object store.

There are also the `export` and `import` capabilities, replacing `push` and `fetch` respectively. Both of them rely on a *fast-import stream* to push or fetch objects.

Finally there is the `connect` capability. A remote helper that advertises this capability can try to connect to services like `git upload-pack` (for fetching), `git receive-pack` (for pushing), for communication using the Git's native packfile protocol.

Git prefers the `connect` capability over `push` or `fetch` capabilities. `export` and `import` capabilities are the least preferred.

8.1.4 Commands

Just like we did for capabilities, here we only explain the basic commands used, in order to achieve push and fetch. Again, a full list of the commands can be found in the documentation [7].

`capabilities`
: Lists the capabilities of the helper, one per line, ending with a blank line. Each capability may be preceded with "*", which marks them mandatory for Git versions using the remote helper to understand. Any unknown mandatory capability is a fatal error.
Support for this command is mandatory.

`list`
: Lists the refs, one per line, in the format `<value> <name>`. The value may be a hex SHA1 hash, "`@<dest>`" for a symbolic reference, or "`?`" to indicate that the helper could not get the value of the ref. The list ends with a blank line.
`list` is supported by the remote helper if the helper has the `fetch` capability.

`list for-push`
: Similar to list, except that it is used if and only if the caller wants the resulting ref list to prepare push commands. A helper supporting both `push` and `fetch` capabilities can use this to distinguish for which operation the output of `list` is going to be used, possibly reducing the amount of work that needs to be performed.
`list for-push` is supported by the remote helper if the helper has the `push` capability.

`fetch <sha1> <name>`
: Fetches the given object, writing the necessary objects to the database. `fetch` commands are sent in a batch, one per line, terminated with a blank line. Outputs a single blank line when all `fetch` commands in the same batch are complete. Only objects which were reported in the output of `list` with a SHA1 may be fetched this way.
Supported if the helper has the `fetch` capability.

`push +<src>:<dst>`
: Pushes the given local `<src>` commit or branch to the remote branch described by `<dst>`. A batch sequence of one or more `push` commands is terminated with a blank line (if there is only one reference to push, a single `push` command is followed by a blank line). When the push is complete, outputs one or more `ok <dst>` or `error <dst> <why>?` lines to indicate success or failure of each pushed ref. The status report output is terminated by a blank line. The option field `<why>` may be quoted in a C style string if it contains an LF.
Supported if the helper has the `push` capability.

8.2 Synchronization and automatic conflict resolution

According to everything mentioned till now, conceptually, in order to achieve synchronization between directories the following steps should be followed:

1. Save local changes.
2. Get remote changes.
3. Attempt to merge changes / resolve any conflicts.
4. Keep remote updated about any changes made locally.

The assumption that the user works in a distributed environment, creates problems, as we must find a way to automatically resolve conflicts. We designed a mechanism to do so, that relies on file renames (in a unique way) in order to not only resolve conflicts, but also distribute those resolutions to all the directories that have to stay synced, in a non-conflicting way.

We decided that in every conflict, the version that resides at the central, common node of the topology, always prevails, while the local version gets uniquely renamed. Thus, we created the Table 8.1 that shows what the final state after the synchronization process should be, based on the state a file is locally and at the central common node before the beginning of the synchronization process. We should note that in that table, renames do not represent an acceptable state, as a file rename is actually equivalent to the removal of the file with the old name and the creation of a file with the new name.

State at local repository	State at common node	Result of synchronization
Modified	Not modified	Modified
Not modified	Modified	Modified
Modified (Hash = H)	Modified (Hash = H)	Modified (Hash = H)
Modified (Hash = H)	Modified (Hash = H')	(conflict) Rename local version, keep common node version as is
Created	Not modified	Created
Not modified	Created	Created
Created (Hash = H)	Created (Hash = H)	Created (Hash = H)
Created (Hash = H)	Created (Hash = H')	(conflict) Rename local version, keep common node version as is
Deleted	Deleted	Deleted
Deleted	Not modified	Deleted
Not modified	Deleted	Deleted
Deleted	Modified	(conflict) Delete local version, keep common node version as is
Modified	Deleted	(conflict) (conflict) Rename local version, delete common node version
Not modified	Not modified	Not modified

Table 8.1: Result of synchronization process

Chapter 9

Implementation of an automated file synchronization mechanism

In this chapter we will implement Gitsync, a tool that combines Git commands in order to automate the file synchronization process, in accordance with the design we analyzed in Chapter 8. Our goal is that even people that do not know what Git is or how it operates can still use Gitsync.

We will also implement our own protocol, rawobjects, which Gitsync will use, utilizing the Git remote helpers' API we described in Chapter 8.

9.1 Rawobjects protocol implementation

Having thoroughly studied and understood the way Git internally works, and how Git communicates with remote helper scripts, we decided to create our own protocol to be used during communication with remote repositories. It is just a proof of concept, but it was deemed necessary to support multiple storage backends during the synchronization process, because, as already stated, the common node cannot always be a Git repository, and sometimes more features are needed, like data encryption.

Our protocol does not require the common central node to be a Git repository. It can be just a simple directory. However, we will also be using the term "remote repository", or just "remote" to refer to it.

For our protocol, the following hierarchy is created at the remote:

```
<remote repository>
 |- objects/
 |- rawobjects/
 |- refs/
```

Listing 9.1: Directory hierarchy of a remote repository when *rawobjects* protocol is used.

Just like in a regular Git repository, `objects/` and `refs/` directories are used to store loose objects and references respectively. The `rawobjects/` directory is used to store uncompressed Git objects, meaning that our protocol also decompresses Git objects (that have been compressed using the zlib library) and stores them for us to see their content. This feature can easily be deactivated, and was

used for study purposes, as we can then inspect the format of blob, tree and commit objects with a text editor. Just because of this directory, we named our protocol *rawobjects*.

According to what we have seen till now, we should make a remote helper script named `git-remote-rawobjects`, make it executable and put it somewhere where Git can find it and execute it. Thus we wrote a Python script and added it to our PATH.

We wanted this protocol to be used with multiple storage backends. For example, we wanted to be able to push to a local directory, as well as push to an Amazon S3 bucket. In order to be able to distinguish the two of them, we assumed specific formats for the URLs used for the remotes. To be specific, if a local directory is to be used as remote, we expect the URL to be of the format `rawobjects::file://<repository_path>` or `rawobjects::<repository_path>`. If the remote is an Amazon S3 bucket, then we expect the URL to be of the format `rawobjects::s3://<bucket_name>/<repository_path>`. Finally, if the remote is an Amazon S3 bucket, and encryption of the objects stored there is desired, we expect the URL to be of the format `rawobjects::ss3://<bucket_name>/<repository_path>`.

We also wrote Python classes responsible for communication with each type of storage backend. Thus, our `git-remote-rawobjects` helper, depending on the URL Git passes as argument to it, can instantiate the appropriate class.

We did not want to rely on Git's native packfile protocol for our storage backends, and thus implemented `push` and `fetch` capabilities in our remote helper scripts.

9.2 Gitsync

Gitsync is a Python tool we developed, used to setup directories for syncing and control the daemon that syncs files periodically. For the synchronization process, in accordance with our design, a star topology is adopted, using a common central node. Gitsync uses the rawobjects protocol and is based on Git. Furthermore, it supports device names, to make distinguishing which version came from which device during a conflict an easy task.

We will explain how Gitsync is used, by analyzing its help menu. Starting with the command `gitsync --help`, the output can be seen in Listing 9.2.

```
$ gitsync --help
usage: gitsync [-h] command ...

Control the sync service

positional arguments:
  command
    start     Start the daemon
    restart   Restart the daemon
    stop      Stop the daemon
    setup     Setup directory to sync

optional arguments:
  -h, --help  show this help message and exit
```

Listing 9.2: Output of `gitsync --help` command

In order to setup a directory for syncing, gitsync setup should be used. The appropriate help menu (`gitsync setup --help`) can be seen in Listing 9.3.

```
$ gitsync setup --help
usage: gitsync setup [-h] [-l LOGFILE] [-n NAME] directory remote

positional arguments:
  directory             the directory to setup
  remote                the url of the remote to track.
                        Supported url formats are:
                          s3://<bucket-name>/[<path>] for amazon s3
                          ss3://<bucket-name>/[<path>] for encrypted amazon s3
                          file://<path> | <path> for local directory

optional arguments:
  -h, --help            show this help message and exit
  -l LOGFILE, --logfile LOGFILE
                        file to store logs. Default is DEVNULL path
  -n NAME, --name NAME  name of the device. Default is the hostname
```

Listing 9.3: Output of `gitsync setup --help` command

After setting up the directory, the daemon that periodically syncs the files should be started. The command to be used in order to do so is `gitsync start` and according to its help menu (`gitsync start --help`), it can be used in the way presented by Listing 9.4.

```
$ gitsync start --help
usage: gitsync start [-h] -d DIRECTORY [-p PIDFILE] [-l LOGFILE] [-i INTERVAL]
                     [--rebase | --automerge]

optional arguments:
  -h, --help            show this help message and exit
  -d DIRECTORY, --directory DIRECTORY
                        path to the directory to sync.
                        It should be an existing directory
                        Required for 'start' and 'restart' commands
  -p PIDFILE, --pidfile PIDFILE
                        pidfile to use. Default ~/.gitsync.pid
  -l LOGFILE, --logfile LOGFILE
                        file to store logs. Default is DEVNULL path
  -i INTERVAL, --interval INTERVAL
                        Set sync interval.
                        A value of 0 inticates manual syncing using SIGUSR1 signals.
                        Default is 5 seconds.
  --rebase              Use 'rebase' instead of 'merge' during syncing
  --automerge           When there is conflict, automerge if possible.
                        Cannot be combined with --rebase
```

Listing 9.4: Output of `gitsync start --help` command

In order to stop the daemon, `gitsync stop` can be used, and according to the output we get from `gitsync stop --help`, it can be used as shown by Listing 9.5.

Finally `gitsync restart` can be used to restart the daemon. It takes the same arguments as `gitsync start`.

```
$ gitsync stop --help
usage: gitsync stop [-h] [-p PIDFILE] [-l LOGFILE]

optional arguments:
  -h, --help            show this help message and exit
  -p PIDFILE, --pidfile PIDFILE
                        pidfile to use. Default ~/.gitsync.pid
  -l LOGFILE, --logfile LOGFILE
                        file to store logs. Default is DEVNULL path
```

Listing 9.5: Output of `gitsync stop --help` command

9.3 The setup

Setting up a directory to be synced with a remote location should require just the path to the local directory and the URL of the remote location that will be used as a middleman during the synchronization. Conceptually, the process should consist of the following steps:

1. Initialize local directory as a Git repository.
2. Provide name and email of user (name will be used as device name)
3. Add the remote URL as a remote with the name `origin`.
4. Set `master` branch to track `origin/master`.
5. Attempt to pull from the remote repository if there are objects to fetch. Otherwise make a commit and push it to the remote.

The final step ensures that every directory that is synced using the same remote as middleman will have a common first commit. Thus it is always possible to calculate a merge-base during conflict resolution.

In Git commands, Gitsync achieves the above in the following way:

1. Change to the directory of the local repository.
   `git init .`
2. `git config user.name <device_name>`
   `git config user.email gitsync@<device_name>`
3. `git remote add origin rawobjects::<remote_url>`
4. `git config branch.master.remote origin`
   `git config branch.master.merge refs/heads/master`
5. `git fetch`
   Check if file `.git/refs/remote/origin/master` exists.
   If it does:
   `git pull`
   Otherwise:
   Create `.gitignore` file.
   `git add .gitignore`

```
git commit -m “init”
git push
```

In order to support `s3://` and `ss3://` protocols, two extra files should be manually created: `$HOME/.aws/config` and `$HOME/.aws/credentials`. This is so that *boto3* (the Python tool used to communicate with Amazon S3 servers) can find the credentials to access the S3 storage service. Examples of `$HOME/.aws/config` and `$HOME/.aws/credentials` follow in Listing 9.6 and Listing 9.7 respectively.

```
$ cat $HOME/.aws/config
[default]
region=us-east-1
```

Listing 9.6: Example of content of `$HOME/.aws/config` file

```
$ cat $HOME/.aws/credentials
[default]
aws_access_key_id = ABCDEFGHIJ1234567890
aws_secret_access_key = D1ySkxHS7ZUBiBeJv1mcS1i/MMvoI7+hiJhfleyR
```

Listing 9.7: Example of content of `$HOME/.aws/credentials` file

Finally, in case of `ss3://` protocol, the user will be asked for a passphrase. This will be used to generate a key, stored under `.git/.encrypt_key` in plaintext. This key will then in turn be used as a passphrase to generate keys to encrypt and decrypt objects. Each object has a unique encryption key. This is not the most secure way to store a key, but as this is a proof of concept and we wanted to avoid repeatedly asking the user for a passphrase, we chose to implement it in a simple way.

9.4 The daemon

The steps that need to followed in order to achieve file synchronization are conceptually:

1. Save local changes.
2. Get remote changes.
3. Attempt to merge changes / resolve any conflicts.
4. Keep remote updated about any changes made locally.

Git can actually do each one of these tasks using specific commands, assuming that during setup a Git remote has been configured the name `origin`:

1. `git add -A`
   `git commit -m update`
2. `git fetch origin`
3. `git merge origin/master`

4. `git push origin master`

These commands should be enough if we were trying to achieve mere file synchronization. However what we are after is *distributed* file synchronization, and in this case there is a core problem: step 3. In a simple file synchronization scenario changes would only be made on one location, thus avoiding the appearance of any conflicts. In our case, during merge phase conflicts could arise, and Git will stop the process requiring human intervention. To someone with no knowledge of what Git is or how it works, successful manual merging could be a very tough task. Thus we have to figure a way to achieve automatic conflict resolution.

We will do so step by step. It is only logical that Gitsync first attempts a fast-forward merge, since that way the merge phase is greatly simplified. And in case of mere file synchronization, this would always work, and that is why the above commands would suffice. In distributed file synchronization though, a file can be modified in two different locations at the same time. In this case, conflict could arise and fast-forward merge could fail. It is then that Gitsync attempts to automatically resolve conflicts, following the merge or rebase strategy, depending on if it was launched with the `--rebase` argument or not. Before explaining how automatic conflict resolution works though, we should notice one thing.

From what we have seen till now, it should be clear that Git tracks file contents. It does not track files nor directories. What Git knows is that in revision A, file `foo` had this content, and in revision B, file `foo` had that other content. Git does not care how `foo` got from point A to point B, whether it was an edit, or a rename, or a conflict resolution. This has two consequences.

The first one is that Git cannot track empty directories. It cannot add an empty directory to its database. We would like to be able to sync empty directories though using Gitsync. In order to do so, Gitsync keeps a list of known directories, and when a new directory is detected, it creates an empty file with the name `.gitkeep` in it. Then Git tracks the contents of this empty file, and so "empty" directories get synced.

The second consequence is that Git cannot detect for sure when a file was renamed into something else. Instead, it compares the content of added files with that of removed files, and if the similarities between two of them pass a specific threshold, Git assumes that the added file is the result of renaming the removed file. Despite that, in Git's database, a rename will still be counted as one file deletion and one file addition. Because uncertain rename detection could cause problems during file synchronization, Gitsync passes the `--no-renames` option to Git commands, whenever that is possible, in order to disable this mechanism.

9.5 Using merge during conflict resolution

We will now explain how conflict detection and resolution happen using the merge method. We shall note that Gitsync offers the choice to automerge files whenever Git can automerge them, with the use of the `--automerge` option. This option however is not available during rebase, for reasons that will be later explained.

During the merge method, Gitsync detects conflicts by attempting a merge that will not result in a merge commit. Then, using a series of commands, it detects which files are in conflict but can be automatically merged by Git (those files belong in a set we will call *automerged* from now on), which files are in conflict and require human intervention in order to be merged (those files belong in a set we will call *unmerged* from now on) and which files are not in conflict (those files belong in a set we will call *updated* from now on). Then, Gitsync aborts the merge, and proceeds to make modifications

to files that belong to the unmerged and automerged sets. The goal is after the modifications, to make a successful merge attempt, without conflicts.

Finding which files are part of the unmerged set can easily be done using the `git ls-files -u` command. `git ls-files` merges the file listing in the directory cache index with the actual working directory list, and shows different combinations of the two. Using the `-u` option one can get only the unmerged files.

In order to find which files are automerged and which are updated, we follow the following procedure:

1. Calculate the SHA1 of the commit that is used as a merge-base by Git.
2. Detect files that have been modified at the remote repository (*modified_remote* set).
3. Detect files that have been modified at the local repository (*modified_local* set).
4. Detect which files are affected by the merge (*to_merge* set).
5. Detect which files can be merged without human intervention. The set that consists of those files (and to which we will refer to as *merged*) is in fact the union of automerged and updated sets. Thus $merged = automerged \cup updated = to\_merge - unmerged$
6. Calculate the automerged set.
7. Calculate the updated set.

So Gitsync first calculates the merge-base commit. It uses the `git merge-base` command, which does exactly that; it finds best common ancestor(s) between two commits to use in a three-way merge. During the setup procedure, Gitsync has ensured that all synced repositories will have at least on commit in common, the very first commit, so `git merge-base` should not fail.

Then, with the appropriate use of the `git diff` command, the files that make up the to_merge set are calculated. In order to do so, Gitsync first detects which files have been modified between the merge-base commit and the remote branch that is to be merged in (these files form the modified_remote set). However in this set, there are also files that have been modified both locally and remotely, but their final content at both locations is the same. We would not want those files to be part of the to_merge set, so we need to exclude them. Gitsync first detects those files using `git diff` between the remote and the local branch (those files form the *remote_local_diff* set), and then forms the to_merge set in the following way: $to\_merge = modified\_remote \cap remote\_local\_diff$. Afterwards the files that have been modified locally since the merge-base commit are detected, using `git diff` in an appropriate way between the merge-base commit and the local branch. In this way, the modified_local set gets calculated.

Having calculated the to_merge set, it is now easy to calculate the merged set, because as stated above $merged = to\_merge - unmerged$.

The automerged set consists of files that are part of the to_merge set, are not part of the unmerged set, and have been locally modified (meaning they are part of the modified_local set). Thus $automerged = (to\_merge - unmerged) \cap modified\_local = merged \cap modified\_local$

Finally, to the updated set belong files that are part of the to_merge set, are not part of the unmerged set, and are not in the automerged set either. So $updated = to\_merge - unmerged - automerged = merged - automerged$

Having calculated updated, automerged and unmerged sets, Gitsync can now move on to making appropriate modifications to the files they contain, in order to attempt a successful merge. As a

general rule, the final result of the automatic conflict resolution is that the remote version of the file should be kept as is, and the local version should be renamed in a unique way. The unique way files are renamed is `<file_name>-<git_object_sha1_hash>-<device_this_version_came_from>`. `<git_object_sha1_hash>` is used because it ensures uniqueness, efficiency (no need to calculate SHA1 hash of file contents) and can also be used for directories (Git tree objects have a SHA1 hash).

However why keep the remote version as is and not the local one? As we have already explained, our goal is after file modifications to be able to make a clean merge without conflicts. The fact that the remote branch we want to merge in (and that we cannot modify) has a file `foo` with content *foo file*, instructs us that in order to have a clean merge, our local branch cannot have a file named foo with different content. This was taken into consideration during the design stage, thus the local version gets renamed. As far as the files that belong to the automerged set are concerned, we could keep their resulted automerged version as the final result, and keep renamed both versions that were used to build the automerged one.

Gitsync uses the `git checkout` command to get each version of the file (local and remote), then calculates the required SHA1 hash. It also uses `git show --format=%an` to get the author name for the commit the file "belongs" to, which has been used as device name during setup. Finally, using the `git mv` command, the files get renamed and the changes are also added in index (meaning they will be included in the next commit).

After making modifications in both unmerged and automerged files, a commit is made that marks the conflict resolutions (all files were renamed using `git mv` so there is no need to `git add` them). From that point on, a merge attempt should be successful. Finally, a `git push` command ensures that the remote repository is updated with the latest changes made.

9.6 Using rebase during conflict resolution

We have already described how rebase works in Git. Gitsync does not make direct calls to `git rebase`, because this would require human intervention in case of conflicts. It does exactly what a `git rebase` command would do instead. It calculates a list of commits that need to be reapplied on top of the `origin/master` branch (the remote branch), using the range `origin/master..HEAD` — that means "all commits reachable by `HEAD` that are not reachable by `origin/master`". Git can list those commits with the appropriate use of `git rev-list` command, which lists commit objects in reverse chronological order.

`git cherry-pick` is used to bring a commit from one line of development to another. Given one or more existing commits, it applies the changes each one introduces, recording a new commit for each. This means we can reset the local `master` branch to `origin/master` (so that `master` reference points to the same commit `origin/master` reference points to) and then `git cherry-pick` each commit that needs to be reapplied. This way we can simulate `git rebase`.

While cherry-picking a commit, conflicts could arise. We have already seen how they can automatically be resolved, by creating renamed files. However this approach seems to not be very effective in the case of the rebase method. Keeping the remote version as is and renaming the local version can lead to many conflicts during a rebase. For example:

1. File `foo` has been modified at the remote repository.
2. File `foo` has also been modified at local repository, in such a way that an attempt to merge would lead to conflict.

3. The device is offline, so although locally commits will be made, the local repository will not get updated with the changes made at the remote repository (and vice versa).

4. While the device is offline, `foo` keeps getting modified locally, and many commits are made, all based on previous local modifications made to file `foo`.

5. At some point the device goes online, so an attempt for synchronization will be made.

6. During the rebase procedure there will be problems. Cherry-picking the first commit will lead to conflicts (caused by the `foo` file), and according to our strategy, the remote version will be kept as is. However we keep cherry-picking commits with local modifications. This means that all the following commits we will cherry-pick, will raise conflicts whose resolution results to many renames.

So during cherry-picks, when a conflict arises we have to keep the local version as is, and a renamed copy of the remote version. This however is not consistent with the way conflicts are resolved during merge, nor with our design, and users expect consistency.

Furthermore, the renamed files that are created, are due to intermediate stages the file `foo` goes through until its final version. From the perspective of the end user, these intermediate version do not matter. It is only the remote and the local versions that needs to be kept.

This sets our goal: at the end of the rebase, if a file was found to be part of the unmerged set, two versions of it should exist. One renamed and one as is. For consistency with our design and the merge strategy, the renamed one will be the local version and the remote version will be kept as is.

There are two ways to approach that goal.

- During cherry-picks, if a file conflicts, rename the remote version, and checkout the local version. This ensures that the following cherry-picks will not raise conflict for this file again. However, at the end of the whole rebase process, the names of the remote and the local versions should change, in order for the first one to have the original name of the file, and the second to be the renamed version. This means that we should keep track of the names of the files that conflict during rebase.

- During cherry-picks, if a file conflicts, keep the remote version as is, rename the local version, and re-write the commits that are yet to be cherry-picked, in order to rename that file in them too. Now the following cherry-picked commits will not modify the remote version (which is kept as is), but only the renamed local version of the file, ensuring that no conflict will arise. However, we need to know the SHA1 of the last commit to be cherry-picked, because the name we will assign to the local version depends on the SHA1 the blob for the final local version has in that last commit.

We wanted to explore the possibilities Git offers, so we chose to follow the second approach. This approach has also the advantage of less renames directly on the working directory.

`git filter-branch` is a command one can use in order to rewrite branches. Custom filters are applied and can modify each tree (e.g. removing a file or running a Perl rewrite on all files) or information about each commit. Otherwise, all information (including original commit times or merge information) will be preserved. After reading the documentation, it is understood that there are two ways to rename a file in a series of commits using filter-branch. One would be by using the `--tree-filter` argument and another by using the `--index-filter` argument. It might seem easier to implement the former, however it is very slow compared to the latter, because `--index-filter`

does not check out the tree, leaving the working directory unaffected. Unfortunately that is the reason files cannot simply be renamed using `mv` or `git mv` commands with `--index-filter`, since they rely on the working directory. Thus, a more complex filter command should be used, one that will rely on `git update-index`. The manpages for `git filter-branch` have a useful example of such a command, used to move the whole tree into a subdirectory, or remove it from there and it can be seen in Listing 9.8.

```
git filter-branch --index-filter \
    'git ls-files -s | sed "s-\t\"*-&newsubdir/-" |
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update-index --index-info &&
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

Listing 9.8: `filter-branch` command example

Relying on that command, Gitsync uses the one presented in Listing 9.9.

```
git filter-branch --index-filter \
    'git ls-files -s | sed "s|\<old_file_name>\(.*\)|\<new-file-name>\1|" | \
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update-index --index-info && \
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' -f <branch_to_filter>
```

Listing 9.9: `filter-branch` command that Gitsync uses

`git ls-files -s` is used to print a list of staged files. Then, using sed, this list is modified, replacing the old name of the file with the new one, and then this list is piped into `git update-index`, which combined with the `--index-info` option can interpret it. A detailed explanation of the command used for filtering by Gitsync can be found here [12].

In order to leave the `master` branch intact, Gitsync creates a new branch called `cherry-pick-temp` right before resetting `master` to `origin/master`. At the end of the rebase process `cherry-pick-temp` is removed. Furthermore, since it is not efficient to rewrite all the branch's commits, the effects of the `git filter-branch` command are restricted to the appropriate range of commits each time.

To sum up, Gitsync does the following things to autoresolve conflicts during the rebase method:

1. Calculate list of commits to be cherry-picked.
2. Calculate the SHA1 hash of the commit `master` branch points to.
3. Create `cherry-pick-temp` branch.
4. Hard reset `master` branch to `origin/master` branch.
5. Start the cherry-picking process.
   (a) Attempt to cherry-pick a commit.
   (b) In case of conflict, rename the file in conflict in all the commits that are yet to be cherry-picked. For the rename use the SHA1 the file has in the commit whose SHA1 was calculated in step 2.

(c) Checkout the renamed file from the rewritten version of the commit whose cherry-pick failed due to conflicts, and `git add` that renamed file.

(d) In case a conflict arises due to the fact that the file was locally deleted, but modified at the remote, in order to be able to cherry-pick successfully, this file must be deleted. Thus add it in a list, delete it, and restore it later, after successfully cherry-picking the commit.

(e) Now that all conflicts have been resolved, `git commit` the changes (phase 1 of resolution).

(f) Cherry-pick the new rewritten version of the commit that caused the conflict. Just like in the case of the merge, this cherry-pick should succeed because conflicts have been resolved.

(g) Restore any files deleted in step `5d`. Since the version to be kept intact is the remote one, checkout the file from `origin/master`.

(h) `git add` the restored files, and `git commit` the changes (phase 2 of resolution).

(i) Continue cherry-picking process with the rest of the commits.

6. Delete `cherry-pick-temp` branch.

There is still something that we have not explained however. What about files that belong to the automerged set? During rebase, automerged files are treated like the files that are part of the unmerged set by Gitsync, so no automerging is taking place. Why is that?

Since the end user cares only for the local and the remote versions and not the intermediate results, it seems as if merge is the perfect fit for our situation. However, with merge, commits with multiple parents are created, and this results in a difficult-to-traverse commit history. Rebase on the other hand ensures that the commit graph is a single line of commits. This makes it much easier to implement in the future a way for the end user to traverse the history and see past versions of a file. In such a scenario however, automatically merged files would cause confusion.

Let's assume there are 4 commits to be cherry-picked during rebase, commits `A`, `B`, `C` and `D`. Commit `A` does not affect file `foo`. In commits `B` and `C`, file `foo` can be automerged, so the automerged version is kept (with the intention of checking-out renamed local and remote versions of `foo` after the rebase). However, cherry-picking commit `D` raises a conflict for file `foo`, so now foo becomes part of the unmerged set. This means the local version will be kept renamed and the remote will remain as is. Now while traversing the history in reverse order, the versions of `foo` will be inconsistent. To be specific, assuming cherry-picking `A`, `B`, `C` and `D` resulted in commits `A'`, `B'`, `C'` and `D'`:

- Commit `A'` will have the remote version of `foo`.
- Commit `B'` will have an automerged version of `foo`.
- Commit `C'` will have an automerged version of `foo`.
- Commit `D'` will have the remote version of `foo` and a renamed local version of `foo`.

The above situation is why Gitsync does not allow combining `--automerge` and `--rebase` options.

In the Table 9.1 and the Table 9.2 that follow, we summarize the way file conflicts are handled using the merge or rebase method respectively. We use the <local-state>/<remote-state> notation to explain the reason a conflict arises. For example modified/deleted means that there is a conflict due to the file being modified locally but deleted at the remote repository. Note that during the rebase method, Gitsync resets master branch to origin/master. This means that a simple rename in the working directory is actually like renaming the remote version of the file.

		`master` branch	
		File exists	*File does not exist*
`origin/master` branch	*File exists*	modified/modified conflict. Keep remote version as is, rename local version.	deleted/modified conflict. Keep remote version as is.
	File does not exist	modified/deleted conflict. Rename local version.	renamed/renamed conflict. Use `-Xno-renames` parameter. Thus, this becomes a deleted/deleted conflict, which is actually not a conflict.

Table 9.1: Conflict handling during merge

		commit to cherry-pick	
		File exists	*File does not exist*
`origin/master` branch	*File exists*	modified/modified conflict. Use filter-branch command to rename the file in all commits left to be cherry-picked (thus renaming local version). Keep remote version (origin/master) as is.	deleted/modified conflict. Remove remote version (origin/master) and add it to a list, to restore it at the end of cherry-pick process.
	File does not exist	modified/deleted conflict. Use filter-branch command to rename the file in all commits left to be cherry-picked (thus renaming local version).	renamed/renamed conflict. Use `-Xno-renames` parameter. Thus, this becomes a deleted/deleted conflict, which is actually not a conflict.

Table 9.2: Conflict handling during rebase

Chapter 10

Conclusions and future directions

In this thesis we have studied the way Git, a distributed Version Control System works, and described a basic workflow using it and how conflicts are resolved. We also went further, exploring how Git can work over protocols natively unknown to it, with the help of Git remote helpers, and implemented our own protocol, rawobjects. Then, we experimented with automated distributed file synchronization, building our own tool, called Gitsync, that relies on Git to manage files, distribute them and detect conflicts. We found our own way of automatically resolving conflicts without the need of human intervention, and went as far as testing two different strategies in order to do so, merge and rebase. Finally, we extended our tool to work with multiple storage backends, with the help of Git remote helper scripts an our rawobjects protocol

Gitsync is far from complete however, and has a lot of room for improvements. Tasks for the future include:

- **Better handling of large files**
  We did not explore synchronization of large binary files. A Git repository contains every version of every file. But for some file types, this is not practical. Multiple revisions of large files increase the clone and fetch times, perhaps making the very first synchronization of a new directory a time consuming process.

- **History traversing**
  Since Git is a version control system, it seems logical to try to take advantage of its capabilities, and provide users the capability of traversing the history. This means that users, in a user-friendly way, could easily view and even restore past versions of their files.

- **Asynchronous synchronization attempt**
  Our current implementation simply makes commit attempts periodically, even if there is not any modified file. A better impementation would be to make this asynchronous, and only attempt to make commits and push if there was any modification.

- **Add more storage backends**
  We implemented support for some storage backends, but the list is far from over.

- **Provide GUI**
  Since we want the tool to be as user-friendly as possible, a simple, fully-featured, and pleasing to the eye Graphical User Interface is a task of top-priority.

Bibliography

[1] *Amazon S3 Website*. accessed September 14, 2016. URL: https://aws.amazon.com/s3/.

[2] S. Balasubramaniam and Benjamin C. Pierce. "What is a File Synchronizer?" In: *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '98. Dallas, Texas, USA: ACM, 1998, pp. 98–108. ISBN: 1-58113-035-X. DOI: 10.1145/288235.288261. URL: http://doi.acm.org/10.1145/288235.288261.

[3] S. Chacon and B. Straub. *Pro Git*. Apress, 2014. ISBN: 1484200772.

[4] *Git Documentation, index-format.txt*. accessed September 14, 2016. URL: https://github.com/git/git/blob/master/Documentation/technical/index-format.txt.

[5] *Git Documentation, pack-heuristics.txt*. accessed September 14, 2016. URL: https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt.

[6] *Git Source Code, pack-write.c*. accessed September 14, 2016. URL: https://github.com/git/git/blob/master/pack-write.c.

[7] *Git Website, Documentation, gitremote-helpers*. accessed September 14, 2016. URL: https://git-scm.com/docs/gitremote-helpers.

[8] *Git-annex Website*. accessed September 14, 2016. URL: https://git-annex.branchable.com/.

[9] Jeffrey K. Hollingsworth and Ethan L. Miller. "Using Content-derived Names for Configuration Management". In: *Proceedings of the 1997 Symposium on Software Reusability*. SSR '97. Boston, Massachusetts, USA: ACM, 1997, pp. 104–109. ISBN: 0-89791-945-9. DOI: 10.1145/258366.258399. URL: http://doi.acm.org/10.1145/258366.258399.

[10] Puneet Kumar and M. Satyanarayanan. "Flexible and Safe Resolution of File Conflicts". In: *Proceedings of the USENIX 1995 Technical Conference Proceedings*. TCON'95. New Orleans, Louisiana: USENIX Association, 1995, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=1267411.1267419.

[11] *ownCloud Website*. accessed October 14, 2016. URL: https://owncloud.org/.

[12] A. Parkins. *Renaming The Past With Git*. Oct 2013 (accessed September 14, 2016). URL: https://www.fussylogic.co.uk/blog/?p=1250.

[13] Santiago Perez De Rosso and Daniel Jackson. "What's Wrong with Git?: A Conceptual Design Analysis". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 37–52. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509584. URL: http://doi.acm.org/10.1145/2509578.2509584.

[14] Sean Quinlan and Sean Dorward. "Venti: A New Approach to Archival Storage". In: *Proceedings of the Conference on File and Storage Technologies*. FAST '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 89–101. ISBN: 1-880446-03-0. URL: http://dl.acm.org/citation.cfm?id=645371.651321.

[15] Norman Ramsey and El''od Csirmaz. "An Algebraic Approach to File Synchronization". In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria: ACM, 2001, pp. 175–185. ISBN: 1-58113-390-1. DOI: 10.1145/503209.503233. URL: http://doi.acm.org/10.1145/503209.503233.

Appendix A

Table of Git commands

Command	Brief description (manpages, Git version 2.10.0)
`git add`	Add file contents to the index.
`git branch`	List, create, or delete branches.
`git checkout`	Switch branches or restore working tree files.
`git cherry-pick`	Apply the changes introduced by some existing commits.
`git clone`	Clone a repository into a new directory.
`git commit`	Record changes to the repository.
`git config`	Get and set repository or global options.
`git diff`	Show changes between commits, commit and working tree, etc.
`git fetch`	Download objects and refs from another repository.
`git filter-branch`	Rewrite branches.
`git gc`	Cleanup unnecessary files and optimize the local repository.
`git init`	Create an empty Git repository or reinitialize an existing one.
`git ls-files`	Show information about files in the index and the working tree.
`git merge`	Join two or more development histories together.
`git merge-base`	Find as good common ancestors as possible for a merge.
`git mv`	Move or rename a file, a directory, or a symlink.
`git pull`	Fetch from and integrate with another repository or a local branch.
`git push`	Update remote refs along with associated objects.
`git rebase`	Reapply commits on top of another base tip.
`git remote`	Manage set of tracked repositories.
`git rev-list`	Lists commit objects in reverse chronological order.
`git show`	Show various types of objects.
`git status`	Show the working tree status.
`git update-index`	Register file contents in the working tree to the index.
`git update-ref`	Update the object name stored in a ref safely.