Volume 2 - Annexes - TagFS

Steven Liatti

Projet de bachelor - Prof. Florent Glück

Hepia ITI 3ème année

8 juillet 2018

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Table des matières

A Code source de Tag Manager

A.1 src/lib.rs

```rust
//! # Tag Manager API
//! Here are the public functions for getting, setting and deleting tags on files giv
//! The tags are stored in an extended attribute called "user.tags" and separated by

use std::fs;
use std::collections::HashSet;

extern crate xattr;
extern crate clap;

const ATTR_NAME : &str = "user.tags";
const SEPARATOR : u8 = ',' as u8;

enum Operation { Set, Delete }
use Operation::*;

/// Return the tags (if there is at least one) associated with the file given. Print
/// on stderr if there is an error.
pub fn get_tags(file: &str) -> Option<HashSet<String>> {
    match check_existent_tags(file) {
        Ok(res) => res,
        Err(_) => None
    }
}

/// Set given tags to given file. If a tag is already present, he's not added. Preser
/// tags. The recursion in subtree is activated with `recursive` to true.
/// Print to stdout the new tags added to file.
pub fn set_tags(file: &str, new_tags: &HashSet<String>, recursive: bool)
{
    recursion(file, recursive, Set, new_tags);
    match check_existent_tags(file) {
        Ok(res) => match res {
            Some(mut tags) => {
                for tag in new_tags { tags.insert(tag.clone()); }
                xattr::set(file, ATTR_NAME, &hash_set_to_vec_u8(&tags))
```

```
36                      .expect("Error when setting tag(s)");
37              },
38              None => xattr::set(file, ATTR_NAME,
&hash_set_to_vec_u8(new_tags))
39                  .expect("Error when setting tag(s)")
40          },
41          Err(err) => {
42              eprintln!("Error for file \"{}\" : {}", file, err);
43              return;
44          }
45      }
46      println!("Tag(s) {:?} for file {:?} have been setted", new_tags,
file);
47  }
48
49  /// Delete given tags of given file. Preserve other existent tags.
50  /// The recursion in subtree is activated with ‘recursive‘ to true.
51  /// Print to stdout the deleted tags.
52  pub fn del_tags(file: &str, tags_to_del: &HashSet<String>, recursive:
bool) {
53      recursion(file, recursive, Delete, tags_to_del);
54      match check_existent_tags(file) {
55          Ok(res) => match res {
56              Some(mut tags) => {
57                  // Delete only the given tags
58                  for tag in tags_to_del {
59                      tags.retain(|ref e| e != &tag);
60                  }
61                  // To avoid to let an empty array of tags
62                  if tags.is_empty() {
63                      match xattr::remove(file, ATTR_NAME) { _ => () }
64                  }
65                  else {
66                      xattr::set(file, ATTR_NAME,
67                          &hash_set_to_vec_u8(&tags))
68                          .expect("Error when (re)setting tag(s)");
69                  }
70              }, _ => ()
71          },
```

```
72          Err(err) => {
73              eprintln!("Error for file \"{}\" : {}", file, err);
74              return;
75          }
76      }
77      println!("Tag(s) {:?} for file {:?} have been deleted",
78          tags_to_del, file);
79  }
80  // TODO: doc
81  pub fn rename_tag(file: &str, old : String, new : String) {
82      match check_existent_tags(file) {
83          Ok(res) => match res {
84              Some(mut tags) => {
85                  if tags.remove(&old) {
86                      tags.insert(new.clone());
87                      xattr::set(file, ATTR_NAME,
    &hash_set_to_vec_u8(&tags))
88                          .expect("Error when setting tag(s)");
89                  }
90              },
91              None => ()
92          },
93          Err(err) => {
94              eprintln!("Error for file \"{}\" : {}", file, err);
95              return;
96          }
97      }
98  }
99
100 fn recursion(file: &str, recursive: bool, operation: Operation, tags:
    &HashSet<String>) {
101     if fs::metadata(file).unwrap().file_type().is_dir() && recursive {
102         for entry in fs::read_dir(file).unwrap() {
103             let sub_file =
    entry.unwrap().path().to_str().unwrap().to_string();
104             match operation {
105                 Set => set_tags(&sub_file, tags, recursive),
106                 Delete => del_tags(&sub_file, tags, recursive)
107             }
```

```
108         }
109     }
110 }
111 
112 fn check_existent_tags(file: &str) -> Result<Option<HashSet<String>>,
    std::io::Error> {
113     match xattr::get(file, ATTR_NAME) {
114         Ok(res) => match res {
115             Some(tags) => Ok(Some(vec_u8_to_hash_set(tags))),
116             None => Ok(None)
117         },
118         Err(err) => Err(err)
119     }
120 }
121 
122 fn hash_set_to_vec_u8(tags_set: &HashSet<String>) -> Vec<u8> {
123     let mut tags_u8 : Vec<u8> = Vec::new();
124     if !tags_set.is_empty() {
125         for tag in tags_set {
126             for u in tag.bytes() { tags_u8.push(u); }
127             tags_u8.push(SEPARATOR);
128         }
129         // remove last separator
130         tags_u8.pop();
131     }
132     tags_u8
133 }
134 
135 fn vec_u8_to_hash_set(tags_u8: Vec<u8>) -> HashSet<String> {
136     let mut s = String::new();
137     let mut tags_set = HashSet::new();
138     if !tags_u8.is_empty() {
139         for u in tags_u8 {
140             if u == SEPARATOR {
141                 tags_set.insert(s.to_string());
142                 s = String::new();
143             }
144             else { s.push(u as char); }
145         }
```

```
146         tags_set.insert(s.to_string());
147     }
148     tags_set
149 }
150 
151 // ------------------------------------------- TESTS
    ---------------------------------------------
152 
153 #[cfg(test)]
154 mod tests {
155     use std::fs::File;
156     use std::fs;
157     use std::collections::HashSet;
158 
159     #[test]
160     fn vec_u8_to_set_string_empty() {
161         let empty_u8 : Vec<u8> = Vec::new();
162         let empty_set : HashSet<String> = HashSet::new();
163         assert_eq!(empty_set, super::vec_u8_to_hash_set(empty_u8));
164     }
165 
166     #[test]
167     fn vec_u8_to_set_string_one() {
168         // ["ACDC"]
169         let vec_u8 : Vec<u8> = vec![65, 67, 68, 67];
170         let mut set_string = HashSet::new();
171         set_string.insert("ACDC".to_string());
172         assert_eq!(set_string, super::vec_u8_to_hash_set(vec_u8));
173     }
174 
175     #[test]
176     fn vec_u8_to_set_string_two() {
177         // ["ACDC", "BOB"]
178         let vec_u8 : Vec<u8> = vec![65, 67, 68, 67, super::SEPARATOR,
    66, 79, 66];
179         let mut set_string = HashSet::new();
180         set_string.insert("ACDC".to_string());
181         set_string.insert("BOB".to_string());
182         assert_eq!(set_string, super::vec_u8_to_hash_set(vec_u8));
```

```
183     }
184
185     #[test]
186     fn set_string_to_vec_u8_empty() {
187         let empty_u8 : Vec<u8> = Vec::new();
188         let empty_set : HashSet<String> = HashSet::new();
189         assert_eq!(empty_u8, super::hash_set_to_vec_u8(&empty_set));
190     }
191
192     #[test]
193     fn set_string_to_vec_u8_one() {
194         // ["ACDC"]
195         let vec_u8 : Vec<u8> = vec![65, 67, 68, 67];
196         let mut set_string = HashSet::new();
197         set_string.insert("ACDC".to_string());
198         assert_eq!(vec_u8, super::hash_set_to_vec_u8(&set_string));
199     }
200
201     #[test]
202     #[should_panic]
203     fn check_existent_tags_no_file() {
204         let path = "/tmp/check_existent_tags_no_tags";
205
206         // Test with file inexistent
207         let result = super::check_existent_tags(&path);
208         panic!(result);
209     }
210
211     #[test]
212     fn check_existent_tags_no_tags() {
213         let path = "/tmp/check_existent_tags_no_tags";
214         File::create(path).expect("Error when creating file");
215
216         // Test with file with no tags
217         let option = super::check_existent_tags(&path).unwrap();
218         assert_eq!(option, None);
219
220         fs::remove_file(path).expect("Error when removing file");
221     }
```

```
222
223     #[test]
224     fn check_existent_tags_tags() {
225         let path = "/tmp/check_existent_tags_tags";
226         File::create(path).expect("Error when creating file");
227
228         // Test with file with tags
229         let vec_u8 = &[65, 67, 68, 67, super::SEPARATOR, 66, 79, 66];
230         let mut tags = HashSet::new();
231         tags.insert("ACDC".to_string());
232         tags.insert("BOB".to_string());
233         super::xattr::set(path, super::ATTR_NAME, vec_u8).unwrap();
234         let option =
super::check_existent_tags(&path).unwrap().unwrap();
235         assert_eq!(option, tags);
236
237         fs::remove_file(path).expect("Error when removing file");
238     }
239
240     #[test]
241     fn set_tag() {
242         let path = "/tmp/set_tags";
243         File::create(path).expect("Error when creating file");
244
245         let mut tags = HashSet::new();
246         tags.insert("bob".to_string());
247         let tags_u8 = vec![98, 111, 98];
248         super::set_tags(path, &tags, false);
249         if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
250             if let Some(tags) = res {
251                 assert_eq!(tags, tags_u8);
252             }
253         }
254
255         // Reset the same tag
256         super::set_tags(path, &tags, false);
257         if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
258             if let Some(tags) = res {
259                 assert_eq!(tags, tags_u8);
```

```
260             }
261         }
262
263         fs::remove_file(path).expect("Error when removing file");
264     }
265
266     #[test]
267     fn del_tags() {
268         let path = "/tmp/del_tags";
269         File::create(path).expect("Error when creating file");
270
271         let mut tags = HashSet::new();
272         tags.insert("bob".to_string());
273         tags.insert("max".to_string());;
274         super::set_tags(path, &tags, false);
275
276         // Delete "bob"
277         let mut bob = HashSet::new();
278         bob.insert("bob".to_string());
279         super::del_tags(path, &bob, false);
280         let tags_u8 = vec![109, 97, 120];
281         if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
282             if let Some(tags) = res {
283                 assert_eq!(tags, tags_u8);
284             }
285         }
286
287         fs::remove_file(path).expect("Error when removing file");
288     }
289 }
```

A.2 src/main.rs

```
//! # Tag Manager
//! Little CLI tool for getting, setting and deleting tags for files and folders.
//! The tags are stored in an extended attribute called "user.tags" and separated by
//! Run `tag_manager -h` to see help.

extern crate tag_manager;
extern crate clap;
use clap::{App, Arg, ArgGroup};
use std::fs;
use std::collections::HashSet;

use std::os::unix::net::UnixStream;
use std::io::prelude::*;

const SOCKET_ADDRESS : &str = "/tmp/tag_engine";
const CODE_ENTRIES : &str = "0x0";
const CODE_TAGS : &str = "0x1";
const CODE_RENAME_TAG : &str = "0x2";

fn main() {
    // TODO: update help
    let help = "\
        tag_manager v0.1.0\nManage your tags\nBy default, this tool store your tags \
        in the extended attribute\n\"user.tags\" and separe them by a comma (\",\").\
        Usage:\n    tag_manager [Options] <files> [[--set|--del] <tags>]\n\n\
        Options:\n    \
        -h    Display this message\n    \
        -r    Recursive option. Get, set or delete tags for each folder and file in f
        Arguments:\n    \
        -s, --set <tags>       Set the given tags\n    \
        -d, --del <tags>       Delete the given tags\n\n\
        Examples:\n    \
        tag_manager myfile                  => Show the actual tags of file \"myfile\
        tag_manager myfile -s work          => Set the tag \"work\" to the file \"myf
        tag_manager myfile -d work          => Delete the tag \"work\" to the file \"
        tag_manager myfolder -r -s geneva   => Set the tag \"geneva\" to the folder \
    let matches = App::new("tag_manager")
```

```
38          .help(help)
39          .group(ArgGroup::with_name("ops").args(&["set", "del"]))
40          .group(ArgGroup::with_name("queries")
41              .args(&["list", "query", "rename"]))
42          .arg(Arg::with_name("set").short("s").long("set")
43              .takes_value(true).multiple(true))
44          .arg(Arg::with_name("del").short("d").long("del")
45              .takes_value(true).multiple(true))
46          .arg(Arg::with_name("files").short("-f").long("--files")
47              .takes_value(true).multiple(true).required(false))
48          .arg(Arg::with_name("recursive").short("-r")
49              .long("--recursive"))
50          .arg(Arg::with_name("query").short("-q").long("--query")
51              .takes_value(true).multiple(true))
52          .arg(Arg::with_name("list").short("-l").long("--list")
53              .takes_value(false))
54          .arg(Arg::with_name("rename").short("-R").long("--rename")
55              .number_of_values(2))
56          .get_matches();
57
58      if matches.is_present("files") {
59          let files: Vec<&str> =
matches.values_of("files").unwrap().collect();
60          let recursive = matches.is_present("recursive");
61
62          if !matches.is_present("set") && !matches.is_present("del") {
63              for file in &files { show_tags(file, recursive); }
64          }
65          else if matches.is_present("set") {
66              let tags: HashSet<&str> =
matches.values_of("set").unwrap().collect();
67              let tags = &hash_set_str_to_hash_set_string(&tags);
68              for file in &files { tag_manager::set_tags(file, tags,
recursive); }
69          }
70          else if matches.is_present("del") {
71              let tags : HashSet<&str> =
matches.values_of("del").unwrap().collect();
72              let tags = &hash_set_str_to_hash_set_string(&tags);
```

```
73              for file in &files { tag_manager::del_tags(file, tags,
    recursive); }
74          }
75      }
76      else if matches.is_present("list") || matches.is_present("query") ||
    matches.is_present("rename") {
77          let mut request = String::new();
78          if matches.is_present("query") {
79              let query : Vec<&str> =
    matches.values_of("query").unwrap().collect();
80              request = String::from(CODE_ENTRIES);
81              for q in query {
82                  request.push_str(q);
83                  request.push(' ');
84              }
85          }
86          if matches.is_present("list") {
87              request = String::from(CODE_TAGS);
88          }
89          if matches.is_present("rename") {
90              let query : Vec<&str> =
    matches.values_of("rename").unwrap().collect();
91              request = String::from(CODE_RENAME_TAG);
92              request.push_str(query[0]);
93              request.push(' ');
94              request.push_str(query[1]);
95          }
96          let mut stream = UnixStream::connect(SOCKET_ADDRESS).unwrap();
97          stream.write_all(request.as_str().as_bytes()).unwrap();
98          let mut response = String::new();
99          stream.read_to_string(&mut response).unwrap();
100         print!("{}", response);
101     }
102     else {
103         println!("{}", help);
104     }
105 }
106
107 fn show_tags(file: &str, recursive: bool) {
```

```
108     match tag_manager::get_tags(file) {
109         Some(tags) => {
110             let mut tags : Vec<String> = tags.into_iter().collect();
111             tags.sort();
112             println!("Tag(s) {:?} for file \"{}\"", tags, file);
113         },
114         None => println!("File \"{}\" has no tags", file)
115     }
116     match fs::metadata(file) {
117         Ok(result) => {
118             if result.file_type().is_dir() && recursive {
119                 for entry in fs::read_dir(file).unwrap() {
120                     let sub_file =
entry.unwrap().path().to_str().unwrap().to_string();
121                     show_tags(&sub_file, recursive);
122                 }
123             }
124         },
125         Err(err) => eprintln!("Error for file \"{}\" : {}", file, err)
126     }
127 }
128 
129 fn hash_set_str_to_hash_set_string(files: &HashSet<&str>) ->
HashSet<String> {
130     let mut new_files : HashSet<String> = HashSet::new();
131     for f in files { new_files.insert(f.to_string()); }
132     new_files
133 }
```

A.3 Cargo.toml

```
[package]
name = "tag_manager"
version = "0.1.0"
authors = ["steven.liatti <steven.liatti@etu.hesge.ch>"]

[dependencies]
clap = "2"
xattr = "0.2"
```

B Code source de Tag Engine

B.1 src/graph.rs

```rust
use std::collections::{HashMap, HashSet};
use std::collections::hash_map::Entry::{Occupied, Vacant};
use std::collections::hash_set::Difference;
use std::collections::hash_map::RandomState;
use std::fs::metadata;
use std::fmt::{Debug, Formatter, Result};

use walkdir::WalkDir;

use petgraph::stable_graph::StableGraph;
use petgraph::graph::NodeIndex;
use petgraph::Direction;

extern crate tag_manager;

#[derive(Debug, Clone)]
pub struct Nil;

#[derive(Debug, Clone)]
pub enum NodeKind {
    Tag,
    File,
    Directory
}

#[derive(Clone)]
pub struct Node {
    pub name : String,
    pub kind : NodeKind
}

pub type MyGraph = StableGraph<Node, Nil>;

impl Nil {
    fn new() -> Self { Self {} }
```

```
}

impl Node {
    fn new(name : String, kind : NodeKind) -> Self {
        Self { name, kind }
    }

    fn set_name(&mut self, name : String) {
        self.name = name;
    }
}

impl Debug for Node {
    fn fmt(&self, f: &mut Formatter) -> Result {
        write!(f, "{:?} {:?}", self.kind, self.name)
    }
}

// TODO: check every call to expect()

pub fn make_subgraph(root_index : NodeIndex, tags_index : &mut
HashMap<String, NodeIndex>,
    graph : &mut MyGraph, local_path : String, base_path : String) {
    let mut path_vec : Vec<&str> = local_path.split('/').collect();
    let mut parent_index = root_index;
    let mut found = false;
    let mut build_path : String = base_path;
    build_path.push_str(path_vec[0]);
    if !path_vec.is_empty() {
        // remove path_root
        path_vec.remove(0);
        for entry in path_vec {
            build_path.push('/');
            build_path.push_str(entry);
            parent_index = find_parent(&graph, parent_index, entry, &mut
found);
            if !found {
                let new_node = if metadata(build_path.clone())
```

```
72
.expect("make_subgraph, new_node, metadata").file_type().is_dir() {
73                      Node::new(String::from(entry), NodeKind::Directory)
74                  }
75                  else { Node::new(String::from(entry), NodeKind::File) };
76                  let new_node = graph.add_node(new_node);
77                  graph.add_edge(parent_index, new_node, Nil::new());
78                  update_tags(build_path.clone(), tags_index, graph,
new_node);
79                  parent_index = new_node;
80              }
81          }
82      }
83  }
84
85  pub fn make_graph(path_root : String, base_path : String)
86      -> (MyGraph, HashMap<String, NodeIndex>, NodeIndex) {
87      let mut graph : MyGraph = StableGraph::new();
88      let mut tags_index = HashMap::new();
89      let local_root = local_path(&mut path_root.clone(),
90          base_path.clone());
91      let root_index = graph.add_node(
92          Node::new(local_root, NodeKind::Directory)
93      );
94      update_tags(path_root.clone(), &mut tags_index,
95          &mut graph, root_index);
96      let mut is_root = true;
97
98      for entry in WalkDir::new(path_root).into_iter()
99          .filter_map(|e| e.ok()) {
100         if is_root {
101             is_root = false;
102             continue;
103         }
104         let mut path = entry.path().display().to_string();
105         let path = local_path(&mut path, base_path.clone());
106         make_subgraph(root_index, &mut tags_index, &mut graph,
107             path, base_path.clone());
108     }
```

```
109     (graph, tags_index, root_index)
110 }
111 
112 pub fn local_path(absolute_path : &mut String, base_path : String) ->
    String {
113     absolute_path.split_off(base_path.len())
114 }
115 
116 pub fn get_node_index(root_index : NodeIndex, graph : &MyGraph, path :
    String) -> NodeIndex {
117     let mut path_vec : Vec<&str> = path.split('/').collect();
118     let mut parent_index = root_index;
119     let mut found = false;
120     if !path_vec.is_empty() {
121         // remove path_root
122         path_vec.remove(0);
123         for entry in path_vec {
124             parent_index = find_parent(&graph, parent_index, entry, &mut
    found);
125         }
126     }
127     parent_index
128 }
129 
130 pub fn move_entry(root_index : NodeIndex, entry_index : NodeIndex, graph
    : &mut MyGraph, new_path : String) {
131     let mut parent_index = entry_index;
132     for neighbor_index in graph.neighbors_directed(entry_index,
    Direction::Incoming) {
133         match graph.node_weight(neighbor_index) {
134             Some(data) => {
135                 match data.kind {
136                     NodeKind::Directory => {
137                         parent_index = neighbor_index;
138                         break;
139                     },
140                     _ => ()
141                 }
142             },
```

```
143             None => ()
144         }
145     }
146     let new_parent_index = get_node_index(root_index, graph,
    new_path.clone());
147     if parent_index == new_parent_index {
148         let mut path_vec : Vec<&str> = new_path.split('/').collect();
149         let new_name =
    path_vec.pop().expect("move_entry, path_vec.pop()").to_string();
150         let node =
    graph.node_weight_mut(entry_index).expect("move_entry, graph.node_weight_mut");
151         node.set_name(new_name);
152     }
153     else {
154         let edge = graph.find_edge(parent_index, entry_index);
155         match edge {
156             Some(edge_index) => { graph.remove_edge(edge_index); },
157             None => ()
158         }
159         graph.add_edge(new_parent_index, entry_index, Nil::new());
160     }
161 }
162
163 // TODO: bug with orphaned tags
164 pub fn remove_entries(entry_index : NodeIndex, graph : &mut MyGraph,
    tags_index : &mut HashMap<String, NodeIndex>) {
165     let mut entries_index = Vec::new();
166     let mut check_tags_index = Vec::new();
167     entries_to_remove(entry_index, graph, &mut entries_index, &mut
    check_tags_index);
168     for index in entries_index.into_iter().rev() {
169         graph.remove_node(index);
170     }
171     for tag_index in check_tags_index {
172         if graph.edges(tag_index).count() == 0 {
173
    tags_index.remove(&graph.node_weight(tag_index).unwrap().name);
174             graph.remove_node(tag_index);
175         }
```

```
176     }
177 }
178 
179 fn find_parent(graph : &MyGraph, index : NodeIndex, entry : &str, found
    : &mut bool) -> NodeIndex {
180     for neighbor_index in graph.neighbors(index) {
181         match graph.node_weight(neighbor_index) {
182             Some(data) => {
183                 match data.kind {
184                     NodeKind::File | NodeKind::Directory => {
185                         if String::from(entry) == data.name {
186                             *found = true;
187                             return neighbor_index;
188                         }
189                     },
190                     _ => ()
191                 }
192             },
193             None => ()
194         }
195     }
196     *found = false;
197     index
198 }
199 
200 fn entries_to_remove(entry_index : NodeIndex, graph : &MyGraph,
201     entries_index : &mut Vec<NodeIndex>, check_tags_index : &mut
    Vec<NodeIndex>) {
202     entries_index.push(entry_index);
203     for neighbor_index in graph.neighbors_directed(entry_index,
    Direction::Outgoing) {
204         match graph.node_weight(neighbor_index) {
205             Some(data) => {
206                 match data.kind {
207                     NodeKind::Directory =>
208                         entries_to_remove(neighbor_index, graph,
    entries_index, check_tags_index),
209                     NodeKind::File =>
    entries_index.push(neighbor_index),
```

```
210                     NodeKind::Tag =>
check_tags_index.push(neighbor_index)
211                 }
212             },
213             None => ()
214         }
215     }
216 }
217
218 // ---------------------------------------- TAGS
----------------------------------------
219
220 pub fn update_tags(path : String,
221     tags_index : &mut HashMap<String, NodeIndex>,
222     graph : &mut MyGraph, entry_index : NodeIndex) {
223     let existent_tags = get_tags(graph, entry_index);
224     let fresh_tags = match tag_manager::get_tags(&path) {
225         Some(tags) => tags,
226         None => HashSet::new()
227     };
228     remove_tags(existent_tags.difference(&fresh_tags),
229         tags_index, graph, entry_index);
230     add_tags(fresh_tags.difference(&existent_tags),
231         tags_index, graph, entry_index);
232 }
233
234 fn get_tags(graph : &MyGraph, tag_index : NodeIndex) -> HashSet<String>
{
235     let mut tags = HashSet::new();
236     for neighbor_index in graph.neighbors_directed(tag_index,
Direction::Incoming) {
237         match graph.node_weight(neighbor_index) {
238             Some(data) => {
239                 match data.kind {
240                     NodeKind::Tag => { tags.insert(data.name.clone());
},
241                     _ => ()
242                 }
243             },
```

```
244             None => ()
245         }
246     }
247     tags
248 }
249 
250 fn add_tags(tags_to_add : Difference<String, RandomState>, tags_index :
    &mut HashMap<String, NodeIndex>,
251     graph : &mut MyGraph, entry_index : NodeIndex) {
252     for tag in tags_to_add {
253         match tags_index.entry(tag.clone()) {
254             Vacant(entry) => {
255                 let new_node_tag = graph.add_node(Node::new(tag.clone(),
    NodeKind::Tag));
256                 entry.insert(new_node_tag);
257                 graph.add_edge(new_node_tag, entry_index, Nil::new());
258             },
259             Occupied(entry) => {
260                 let &tag_index = entry.get();
261                 graph.add_edge(tag_index, entry_index, Nil::new());
262             }
263         }
264     }
265 }
266 
267 fn remove_tags(tags_to_remove : Difference<String, RandomState>,
    tags_index : &mut HashMap<String, NodeIndex>,
268     graph : &mut MyGraph, entry_index : NodeIndex) {
269     for tag in tags_to_remove {
270         match tags_index.entry(tag.clone()) {
271             Occupied(entry) => {
272                 let &tag_index = entry.get();
273                     match graph.find_edge(tag_index, entry_index) {
274                         Some(edge) => { graph.remove_edge(edge); },
275                         None => ()
276                     }
277                     if graph.edges(tag_index).count() == 0 {
278                         entry.remove();
279                         graph.remove_node(tag_index);
```

```
280                         }
281                 },
282                 Vacant(_) => ()
283             }
284         }
285 }
```

B.2 src/lib.rs

```
use std::collections::HashMap;

extern crate walkdir;

extern crate petgraph;
use petgraph::graph::NodeIndex;

extern crate notify;
use notify::DebouncedEvent;
use notify::DebouncedEvent::{Create, Chmod, Remove, Rename};

extern crate tag_manager;

pub mod graph;
use graph::{MyGraph, local_path, make_subgraph, get_node_index,
update_tags, move_entry, remove_entries};

pub mod server;
pub mod parse;

// TODO: if create dir by mv, scan subgraph
pub fn dispatcher(event : DebouncedEvent, tags_index : &mut
HashMap<String, NodeIndex>,
    graph : &mut MyGraph, root_index : NodeIndex, base : String) {
    match event {
        Create(path) => {
            let mut path =
path.as_path().to_str().expect("dispatcher, create, path").to_string();
            let local = local_path(&mut path, base.clone());
            println!("create : {:?}", local);
            make_subgraph(root_index, tags_index, graph, local,
base.clone());
        },
        Chmod(path) => {
            let mut path =
path.as_path().to_str().expect("dispatcher, chmod, path").to_string();
            let local = local_path(&mut path.clone(), base);
```

```
33              println!("chmod : {:?}", local);
34              let entry_index = get_node_index(root_index, graph, local);
35              update_tags(path, tags_index, graph, entry_index);
36          },
37          Remove(path) => {
38              let mut path =
path.as_path().to_str().expect("dispatcher, remove, path").to_string();
39              let local = local_path(&mut path.clone(), base);
40              println!("remove : {:?}", local);
41              let entry_index = get_node_index(root_index, graph, local);
42              remove_entries(entry_index, graph, tags_index);
43          },
44          Rename(old_path, new_path) => {
45              let mut old_path =
old_path.as_path().to_str().expect("dispatcher, rename, old_path").to_string();
46              let new_path =
new_path.as_path().to_str().expect("dispatcher, rename, new_path").to_string();
47              let old_local = local_path(&mut old_path.clone(),
base.clone());
48              let new_local = local_path(&mut new_path.clone(),
base.clone());
49              println!("rename, old_path : {:?}, new_path : {:?}",
old_local, new_local);
50              let entry_index = get_node_index(root_index, graph,
old_local);
51              move_entry(root_index, entry_index, graph, new_local);
52          }
53          _ => ()
54      }
55  }
```

B.3 src/main.rs

```
use std::io::prelude::*;
use std::fs::File;
use std::process::Command;
use std::thread;
use std::sync::{Mutex, Arc};
use std::sync::mpsc::channel;
use std::time::{Duration, Instant};

extern crate petgraph;
use petgraph::dot::{Dot, Config};

extern crate notify;
use notify::{Watcher, RecursiveMode, watcher};
use notify::DebouncedEvent::{Create, Chmod, Remove, Rename};

extern crate tag_manager;

extern crate tag_engine;
use tag_engine::graph::MyGraph;

use std::path::Path;
use std::process::exit;

extern crate clap;
use clap::{App, Arg};

// TODO: check every call to expect()

fn split_root_path(absolute_path : &mut String) -> (String, String) {
    let clone = absolute_path.clone();
    let mut path_vec : Vec<&str> = clone.split('/').collect();
    let local_path =
path_vec.pop().expect("split_root, local_path").to_string();
    absolute_path.truncate(clone.len() - local_path.len());
    (absolute_path.clone(), local_path)
}

```

```
37 fn write_dot_image(graph : &MyGraph, dot_name : &str, image_name : &str)
   {
38     let mut file = File::create(dot_name).expect("file create");
39     let graph_dot = format!("{:?}", Dot::with_config(graph,
   &[Config::EdgeNoLabel]));
40     file.write(graph_dot.as_bytes()).expect("file write");
41     let mut output = String::from("-o");
42     output.push_str(image_name);
43     let _exec_dot = Command::new("dot").args(&["-Tpng", output.as_str(),
   dot_name]).output().expect("exec");
44 }
45
46 fn main() {
47     let matches =
   App::new("Tag Engine").version("0.1.0").author("Steven Liatti")
48
   .arg(Arg::with_name("path").takes_value(true).required(true).multiple(false))
49
   .arg(Arg::with_name("debug").short("-d").long("--debug").required(false).multiple(fal
50         .get_matches();
51
52     let absolute_path_root = matches.value_of("path").unwrap();
53     let path = Path::new(absolute_path_root);
54     if !path.exists()  {
55         eprintln!("The path doesn't exist");
56         exit(1);
57     }
58     if path.is_relative()  {
59         eprintln!("The path must be absolute");
60         exit(1);
61     }
62     if !path.is_dir()  {
63         eprintln!("The path must point to a directory");
64         exit(1);
65     }
66
67     let (base_path, _) = split_root_path(&mut
   absolute_path_root.to_string());
68     let now = Instant::now();
```

```
69      let (graph, tags_index, root_index) =
tag_engine::graph::make_graph(String::from(absolute_path_root),
base_path.clone());
70      let new_now = Instant::now();
71      let elapsed = new_now.duration_since(now);
72
73      let dot_name = "graph.dot";
74      let image_name = "graph.png";
75      let debug = matches.is_present("debug");
76      if debug {
77          println!("{}", elapsed.as_secs() as f64 + elapsed.subsec_nanos()
as f64 * 1e-9);
78          println!("graph {:#?}, tags_index {:#?}", graph, tags_index);
79          write_dot_image(&graph, dot_name, image_name);
80      }
81
82      let graph = Arc::new(Mutex::new(graph));
83      let tags_index = Arc::new(Mutex::new(tags_index));
84      let main_graph = Arc::clone(&graph);
85      let main_tags_index = Arc::clone(&tags_index);
86
87      let base_clone = base_path.clone();
88      thread::spawn(move || {
89          tag_engine::server::server(base_clone, &graph, &tags_index);
90      });
91
92      let (tx, rx) = channel();
93      let mut watcher = watcher(tx,
Duration::from_secs(1)).expect("watcher");
94      watcher.watch(absolute_path_root,
RecursiveMode::Recursive).expect("watcher watch");
95
96      loop {
97          match rx.recv() {
98              Ok(event) => {
99                  match event {
100                     Create(_) | Chmod(_) | Remove(_) | Rename(_, _) => {
101                         let mut ref_graph = main_graph.lock().unwrap();
```

```
102                         let mut ref_tags_index =
main_tags_index.lock().unwrap();
103                         tag_engine::dispatcher(event, &mut
ref_tags_index, &mut ref_graph, root_index, base_path.clone());
104                         if debug {
105                             println!("graph {:#?}, tags_index {:#?}",
*ref_graph, *ref_tags_index);
106                             write_dot_image(&ref_graph, dot_name,
image_name);
107                         }
108                     }
109                     _ => ()
110                 }
111             },
112             Err(e) => println!("watch error: {:?}", e)
113         }
114     }
115 }
```

B.4 src/parse.rs

```rust
const AND_OPERATOR_STR : &str = "AND";
const OR_OPERATOR_STR : &str = "OR";

use self::Operator::*;
#[derive(Debug, Clone, PartialEq)]
pub enum Operator { AND, OR }
impl Operator {
    fn compare(&self, other : &Operator) -> i8 {
        match (self, other) {
            (&AND, &OR) => 1,
            (&OR, &AND) => -1,
            _ => 0
        }
    }
}

#[derive(Debug, Clone, PartialEq)]
pub enum Arg {
    Operand(String),
    Operator(Operator)
}

fn str_to_operator(op_str : &str) -> Option<Operator> {
    if op_str == AND_OPERATOR_STR {
        Some(AND)
    }
    else if op_str == OR_OPERATOR_STR {
        Some(OR)
    }
    else {
        None
    }
}

pub fn infix_to_postfix(infix : String) -> Vec<Arg> {
    let infix : Vec<&str> = infix.split(' ').collect();
    let mut stack = Vec::new();
```

```
38      let mut postfix : Vec<Arg> = Vec::new();
39      for arg in infix {
40          if arg == AND_OPERATOR_STR || arg == OR_OPERATOR_STR {
41              let arg = str_to_operator(arg).unwrap();
42              if stack.is_empty() {
43                  stack.push(arg);
44              }
45              else {
46                  while !stack.is_empty() {
47                      let mut top_stack = stack.get(stack.len() -
    1).unwrap().clone();
48                      let mut compare = arg.compare(&top_stack);
49                      if compare > 0 {
50                          break;
51                      }
52                      else {
53
    postfix.push(Arg::Operator(stack.pop().unwrap()));
54                      }
55                  }
56                  stack.push(arg);
57              }
58          }
59          else {
60              postfix.push(Arg::Operand(arg.to_string()));
61          }
62      }
63      for op in stack.into_iter().rev() {
64          postfix.push(Arg::Operator(op));
65      }
66      postfix
67  }
68
69  #[cfg(test)]
70  mod tests {
71      use super::*;
72
73      #[test]
74      fn test_infix_to_postfix_1() {
```

```
75          let infix = String::from("bob AND fred");
76          let postfix = vec![
77              Arg::Operand(String::from("bob")),
78              Arg::Operand(String::from("fred")),
79              Arg::Operator(Operator::AND)
80          ];
81          assert_eq!(infix_to_postfix(infix), postfix);
82      }
83
84      #[test]
85      fn test_infix_to_postfix_2() {
86          let infix = String::from("bob OR fred");
87          let postfix = vec![
88              Arg::Operand(String::from("bob")),
89              Arg::Operand(String::from("fred")),
90              Arg::Operator(Operator::OR)
91          ];
92          assert_eq!(infix_to_postfix(infix), postfix);
93      }
94
95      #[test]
96      fn test_infix_to_postfix_3() {
97          let infix = String::from("bob AND fred OR max");
98          let postfix = vec![
99              Arg::Operand(String::from("bob")),
100             Arg::Operand(String::from("fred")),
101             Arg::Operator(Operator::AND),
102             Arg::Operand(String::from("max")),
103             Arg::Operator(Operator::OR)
104         ];
105         assert_eq!(infix_to_postfix(infix), postfix);
106     }
107
108     #[test]
109     fn test_infix_to_postfix_4() {
110         let infix = String::from("bob OR fred AND max");
111         let postfix = vec![
112             Arg::Operand(String::from("bob")),
113             Arg::Operand(String::from("fred")),
```

```
            Arg::Operand(String::from("max")),
            Arg::Operator(Operator::AND),
            Arg::Operator(Operator::OR)
        ];
        assert_eq!(infix_to_postfix(infix), postfix);
    }

    #[test]
    fn test_infix_to_postfix_5() {
        let infix = String::from("bob AND fred AND max");
        let postfix = vec![
            Arg::Operand(String::from("bob")),
            Arg::Operand(String::from("fred")),
            Arg::Operator(Operator::AND),
            Arg::Operand(String::from("max")),
            Arg::Operator(Operator::AND)
        ];
        assert_eq!(infix_to_postfix(infix), postfix);
    }

    #[test]
    fn test_infix_to_postfix_6() {
        let infix = String::from("bob AND fred OR max AND paul");
        let postfix = vec![
            Arg::Operand(String::from("bob")),
            Arg::Operand(String::from("fred")),
            Arg::Operator(Operator::AND),
            Arg::Operand(String::from("max")),
            Arg::Operand(String::from("paul")),
            Arg::Operator(Operator::AND),
            Arg::Operator(Operator::OR)
        ];
        assert_eq!(infix_to_postfix(infix), postfix);
    }
}
```

B.5 src/server.rs

```
use std::collections::{HashMap, HashSet};
use std::io::prelude::*;
use std::sync::{Mutex, Arc};
use std::os::unix::net::{UnixListener, UnixStream};
use std::fs::remove_file;

extern crate petgraph;
use petgraph::graph::NodeIndex;
use petgraph::Direction;

extern crate tag_manager;

use graph::{MyGraph, NodeKind};
use parse::{Arg, Operator};
use parse::infix_to_postfix;

const BUFFER_SIZE : usize = 4096;
const CODE_SIZE : usize = 3;
const BIND_ADDRESS : &str = "/tmp/tag_engine";

#[derive(Debug, Clone)]
enum RequestKind {
    Entries(String),
    Tags,
    RenameTag(String)
}

fn parse_request(stream : &mut UnixStream) -> Option<RequestKind> {
    let mut buffer = [0; BUFFER_SIZE];
    let size = stream.read(&mut buffer).unwrap();
    if size >= CODE_SIZE {
        let mut request = String::new();
        for i in CODE_SIZE..size {
            request.push(buffer[i] as char);
        }
        let mut kind = String::new();
        for i in 0..CODE_SIZE {
```

```
38              kind.push(buffer[i] as char);
39          }
40          if kind == String::from("0x0") {
41              Some(RequestKind::Entries(request.trim().to_string()))
42          }
43          else if kind == String::from("0x1") {
44              Some(RequestKind::Tags)
45          }
46          else if kind == String::from("0x2") {
47              Some(RequestKind::RenameTag(request.trim().to_string()))
48          }
49          else { None }
50      }
51      else { None }
52  }
53
54  fn make_path_vec(graph : &MyGraph, entry : NodeIndex, path_vec : &mut
    Vec<String>) {
55      path_vec.push(graph.node_weight(entry).unwrap().name.clone());
56      for neighbor in graph.neighbors_directed(entry, Direction::Incoming)
    {
57          match graph.node_weight(neighbor).unwrap().kind {
58              NodeKind::Directory => {
59                  make_path_vec(graph, neighbor, path_vec);
60              },
61              _ => ()
62          }
63      }
64  }
65
66  fn make_path(graph : &MyGraph, entry : NodeIndex, base_path : String) ->
    String {
67      let mut path_vec = Vec::new();
68      make_path_vec(&graph, entry, &mut path_vec);
69      let mut path = base_path.clone();
70      for entry in path_vec.into_iter().rev() {
71          path.push_str(&entry);
72          path.push_str("/");
73      }
```

```
74      path.pop();
75      path
76  }
77
78  fn entries(graph : &MyGraph, tag_index : NodeIndex, base_path : String)
    -> Vec<String> {
79      let mut nodes_names = Vec::new();
80      for entry in graph.neighbors(tag_index) {
81          nodes_names.push(make_path(graph, entry, base_path.clone()));
82      }
83      nodes_names.sort();
84      nodes_names
85  }
86
87  fn expression_to_entries(infix_request : String, graph : &MyGraph,
    tags_index : &HashMap<String,
88      NodeIndex>, base_path : String) -> Vec<String> {
89      let postfix = infix_to_postfix(infix_request.clone());
90      let mut stack = Vec::new();
91      for arg in postfix {
92          match arg {
93              Arg::Operand(tag) => {
94                  if tags_index.contains_key(&tag) {
95                      let tag_index = tags_index.get(&tag).unwrap();
96                      let tags_set : HashSet<NodeIndex> =
    graph.neighbors(*tag_index).collect();
97                      stack.push(tags_set);
98                  }
99                  else { stack.push(HashSet::new()); }
100             },
101             Arg::Operator(op) => {
102                 if stack.len() >= 2 {
103                     let operand_two = stack.pop().unwrap();
104                     let operand_one = stack.pop().unwrap();
105                     match op {
106                         Operator::AND =>
    stack.push(operand_one.intersection(&operand_two).map(|e|
    *e).collect()),
```

```
107                         Operator::OR =>
stack.push(operand_one.union(&operand_two).map(|e| *e).collect())
108                     }
109                 }
110             }
111         }
112     }
113     let mut nodes_names = Vec::new();
114     if stack.len() == 1 {
115         for entry in stack.pop().unwrap() {
116             nodes_names.push(make_path(graph, entry,
base_path.clone()));
117         }
118         nodes_names.sort();
119     }
120     nodes_names
121 }
122
123 fn write_response(entries : Vec<String>, stream : &mut UnixStream) {
124     let mut response : Vec<u8> = Vec::new();
125     for name in entries {
126         for byte in name.as_bytes() {
127             response.push(*byte);
128         }
129         response.push('\n' as u8);
130     }
131     stream.write(response.as_slice()).unwrap();
132     stream.flush().unwrap();
133 }
134
135 fn request_entries(request : String, graph_thread :
&Arc<Mutex<MyGraph>>,
136     tags_index_thread : &Arc<Mutex<HashMap<String, NodeIndex>>>,
base_path : String,
137     stream : &mut UnixStream) {
138     println!("Request for Entries {:?}", request);
139     let graph = graph_thread.lock().unwrap();
140     let tags_index = tags_index_thread.lock().unwrap();
```

```
141     let entries = expression_to_entries(request, &graph, &tags_index,
    base_path);
142     if entries.is_empty() {
143         stream.write("No files\n".as_bytes()).unwrap();
144         stream.flush().unwrap();
145     }
146     else {
147         write_response(entries, stream);
148     }
149 }
150 
151 fn request_tags(tags_index_thread : &Arc<Mutex<HashMap<String,
    NodeIndex>>>, stream : &mut UnixStream) {
152     println!("Request for Tags");
153     let tags_index = tags_index_thread.lock().unwrap();
154     let mut entries : Vec<String> = tags_index.keys().map(|key|
    key.clone()).collect();
155     entries.sort();
156     write_response(entries, stream);
157 }
158 
159 fn request_rename_tag(request : String, graph_thread :
    &Arc<Mutex<MyGraph>>,
160     tags_index_thread : &Arc<Mutex<HashMap<String, NodeIndex>>>,
    base_path : String,
161     stream : &mut UnixStream) {
162     println!("Request for RenameTag {:?}", request);
163     let v : Vec<&str> = request.split(' ').collect();
164     if v.len() == 2 {
165         let old_name = v[0];
166         let new_name = v[1];
167         let mut graph = graph_thread.lock().unwrap();
168         let mut tags_index = tags_index_thread.lock().unwrap();
169         match tags_index.remove(old_name) {
170             Some(index) => {
171                 tags_index.insert(new_name.to_string(), index);
172                 graph.node_weight_mut(index).unwrap().name =
    new_name.to_string();
```

```
173                 let mut entries = entries(&graph, index,
    base_path.clone());
174                 for e in &entries {
175                     tag_manager::rename_tag(e, old_name.to_string(),
    new_name.to_string());
176                 }
177                 entries.insert(0,
    format!("Rename {:?} to {:?} for files :", old_name, new_name));
178                 write_response(entries, stream);
179             },
180             None => {
181 
    write_response(vec![String::from("No tag with this old name")], stream);
182             }
183         }
184     }
185     else {
186         write_response(vec![String::from("Bad request")], stream);
187     }
188 }
189 
190 pub fn server(base_path : String, graph : &Arc<Mutex<MyGraph>>,
    tags_index : &Arc<Mutex<HashMap<String, NodeIndex>>>) {
191     match remove_file(BIND_ADDRESS) {
192         _ => ()
193     }
194     let listener = UnixListener::bind(BIND_ADDRESS).unwrap();
195     let graph_thread = Arc::clone(graph);
196     let tags_index_thread = Arc::clone(tags_index);
197 
198     for stream in listener.incoming() {
199         let mut stream = stream.unwrap();
200         match parse_request(&mut stream) {
201             Some(kind) => match kind {
202                 RequestKind::Entries(request) =>
    request_entries(request, &graph_thread,
203                     &tags_index_thread, base_path.clone(), &mut stream),
204                 RequestKind::Tags => request_tags(&tags_index_thread,
    &mut stream),
```

```
205                 RequestKind::RenameTag(request) =>
request_rename_tag(request, &graph_thread,
206                     &tags_index_thread, base_path.clone(), &mut stream)
207             },
208             None => {
209                 stream.write("Invalid request\n".as_bytes()).unwrap();
210                 stream.flush().unwrap();
211             }
212         }
213     }
214 }
```

B.6 Cargo.toml

```
[package]
name = "tag_engine"
version = "0.1.0"
authors = ["steven.liatti <steven.liatti@etu.hesge.ch>"]

[dependencies]
tag_manager = { path = "../tag_manager" }
walkdir = "2"
petgraph = "0.4.12"
notify = "4.0.0"
clap = "2"
```