# Volume 2 - Annexes - TagFS
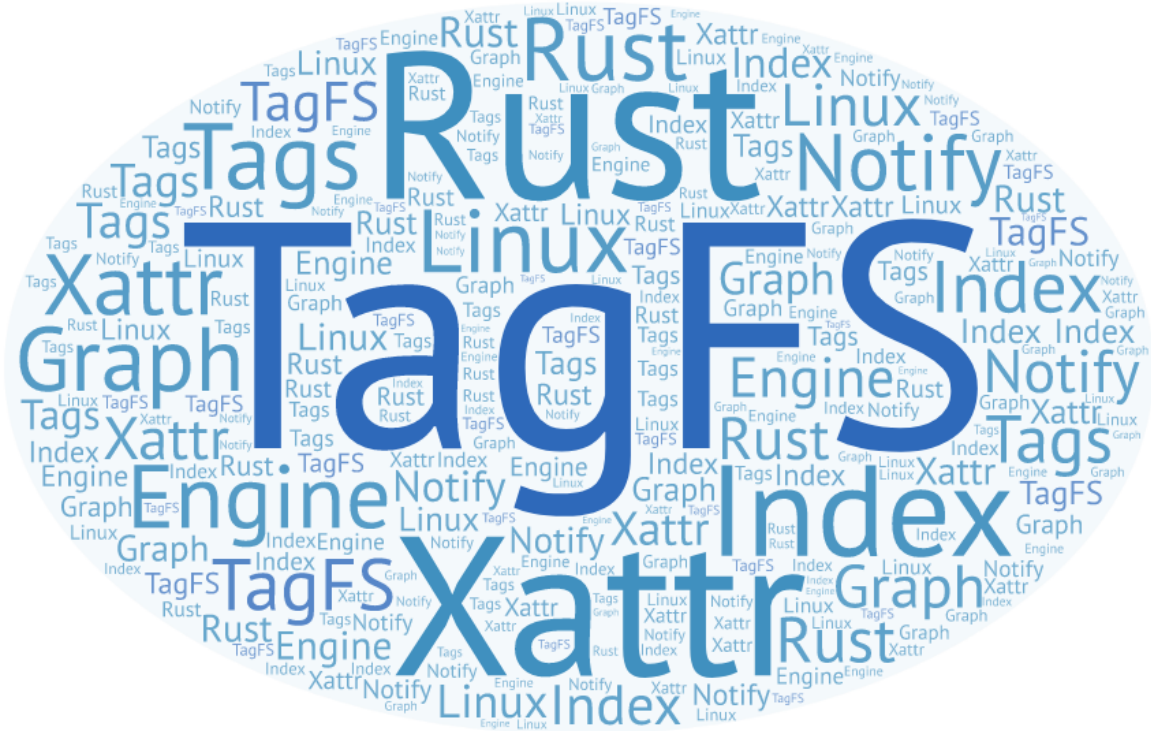
Steven Liatti

Projet de bachelor - Prof. Florent Glück

Hepia ITI 3ème année

Septembre 2018

# Table des matières

# A  Compiler et exécuter le code

Le répertoire racine du projet contient deux répertoires, `tag_engine` et `tag_manager` dans lesquels se trouvent le code source de chaque programme. Pour compiler et exécuter les programmes, il faut tout d'abord avoir installé Rust et Cargo comme décrit dans le listing 1 pour Linux.

```
1  $ curl https://sh.rustup.rs -sSf | sh
2  $ source $HOME/.cargo/env
```

Listing 1 – Installation de Rust sur Linux ou macOS

En se plaçant dans le répertoire racine avec un shell, il faut en premier lieu exécuter la commande `cargo build`. Elle crée deux exécutables `tag_engine` et `tag_manager` dans le répertoire `target/debug`. L'utilisateur peut soit utiliser ces deux exécutables, soit passer par Cargo avec la syntaxe suivante :

— `cargo run -p tag_engine /path/to/files` pour Tag Engine.

— `cargo run -p tag_manager` pour Tag Manager.

Le code source du projet est disponible sur https://github.com/stevenliatti/tagfs.

# B  Code source de Tag Manager

## B.1  `src/lib.rs`

```rust
1   //! # Tag Manager API
2   //! Here are the public functions for getting, setting and deleting tags
3   //! on files given.
4   //! The tags are stored in an extended attribute called "user.tags" and
5   //! separated by comma.
6
7   use std::fs;
8   use std::collections::HashSet;
9
10  extern crate xattr;
11  extern crate clap;
12
13  const ATTR_NAME : &str = "user.tags";
14  const SEPARATOR : u8 = ',' as u8;
15
16  enum Operation { Set, Delete }
```

```rust
17  use Operation::*;
18
19  /// Return the tags (if there is at least one) associated with the file
20  /// given. Print error on stderr if there is an error.
21  pub fn get_tags(file: &str) -> Option<HashSet<String>> {
22      match check_existent_tags(file) {
23          Ok(res) => res,
24          Err(_) => None
25      }
26  }
27
28  /// Set given tags to given file. If a tag is already present, he's not
29  /// added. Preserve existent tags. The recursion in subtree is
30  /// activated with 'recursive' to true.
31  /// Print to stdout the new tags added to file.
32  pub fn set_tags(file: &str, new_tags: &HashSet<String>, recursive: bool)
    {
33      recursion(file, recursive, Set, new_tags);
34      match check_existent_tags(file) {
35          Ok(res) => match res {
36              Some(mut tags) => {
37                  for tag in new_tags { tags.insert(tag.clone()); }
38                  xattr::set(file, ATTR_NAME, &hash_set_to_vec_u8(&tags))
39                      .expect("Error when setting tag(s)");
40              },
41              None => xattr::set(file, ATTR_NAME,
    &hash_set_to_vec_u8(new_tags))
42                  .expect("Error when setting tag(s)")
43          },
44          Err(err) => {
45              eprintln!("Error for file \"{}\" : {}", file, err);
46              return;
47          }
48      }
49      println!("Tag(s) {:?} for file {:?} have been setted", new_tags,
    file);
50  }
51
52  /// Delete given tags of given file. Preserve other existent tags.
```

```rust
/// The recursion in subtree is activated with 'recursive' to true.
/// Print to stdout the deleted tags.
pub fn del_tags(file: &str, tags_to_del: &HashSet<String>, recursive:
bool) {
    recursion(file, recursive, Delete, tags_to_del);
    match check_existent_tags(file) {
        Ok(res) => match res {
            Some(mut tags) => {
                // Delete only the given tags
                for tag in tags_to_del {
                    tags.retain(|ref e| e != &tag);
                }
                // To avoid to let an empty array of tags
                if tags.is_empty() {
                    match xattr::remove(file, ATTR_NAME) { _ => () }
                }
                else {
                    xattr::set(file, ATTR_NAME,
                        &hash_set_to_vec_u8(&tags))
                        .expect("Error when (re)setting tag(s)");
                }
            }, _ => ()
        },
        Err(err) => {
            eprintln!("Error for file \"{}\" : {}", file, err);
            return;
        }
    }
    println!("Tag(s) {:?} for file {:?} have been deleted",
        tags_to_del, file);
}

/// Rename old tag with new tag of given file.
/// Preserve other existent tags.
pub fn rename_tag(file: &str, old : String, new : String) {
    match check_existent_tags(file) {
        Ok(res) => match res {
            Some(mut tags) => {
                if tags.remove(&old) {
```

```
 91                     tags.insert(new.clone());
 92                     xattr::set(file, ATTR_NAME,
    &hash_set_to_vec_u8(&tags))
 93                         .expect("Error when setting tag(s)");
 94                 }
 95             },
 96             None => ()
 97         },
 98         Err(err) => {
 99             eprintln!("Error for file \"{}\" : {}", file, err);
100             return;
101         }
102     }
103 }
104
105 fn recursion(file: &str, recursive: bool, operation: Operation, tags:
    &HashSet<String>) {
106     if fs::metadata(file).unwrap().file_type().is_dir() && recursive {
107         for entry in fs::read_dir(file).unwrap() {
108             let sub_file =
    entry.unwrap().path().to_str().unwrap().to_string();
109             match operation {
110                 Set => set_tags(&sub_file, tags, recursive),
111                 Delete => del_tags(&sub_file, tags, recursive)
112             }
113         }
114     }
115 }
116
117 fn check_existent_tags(file: &str) -> Result<Option<HashSet<String>>,
    std::io::Error> {
118     match xattr::get(file, ATTR_NAME) {
119         Ok(res) => match res {
120             Some(tags) => Ok(Some(vec_u8_to_hash_set(tags))),
121             None => Ok(None)
122         },
123         Err(err) => Err(err)
124     }
125 }
```

```rust
126
127  fn hash_set_to_vec_u8(tags_set: &HashSet<String>) -> Vec<u8> {
128      let mut tags_u8 : Vec<u8> = Vec::new();
129      if !tags_set.is_empty() {
130          for tag in tags_set {
131              for u in tag.bytes() { tags_u8.push(u); }
132              tags_u8.push(SEPARATOR);
133          }
134          // remove last separator
135          tags_u8.pop();
136      }
137      tags_u8
138  }
139
140  fn vec_u8_to_hash_set(tags_u8: Vec<u8>) -> HashSet<String> {
141      let mut s = String::new();
142      let mut tags_set = HashSet::new();
143      if !tags_u8.is_empty() {
144          for u in tags_u8 {
145              if u == SEPARATOR {
146                  tags_set.insert(s.to_string());
147                  s = String::new();
148              }
149              else { s.push(u as char); }
150          }
151          tags_set.insert(s.to_string());
152      }
153      tags_set
154  }
155
156  // --------------------------- TESTS ---------------------------
157
158  #[cfg(test)]
159  mod tests {
160      use std::fs::File;
161      use std::fs;
162      use std::collections::HashSet;
163
164      #[test]
```

```rust
fn vec_u8_to_set_string_empty() {
    let empty_u8 : Vec<u8> = Vec::new();
    let empty_set : HashSet<String> = HashSet::new();
    assert_eq!(empty_set, super::vec_u8_to_hash_set(empty_u8));
}

#[test]
fn vec_u8_to_set_string_one() {
    // ["ACDC"]
    let vec_u8 : Vec<u8> = vec![65, 67, 68, 67];
    let mut set_string = HashSet::new();
    set_string.insert("ACDC".to_string());
    assert_eq!(set_string, super::vec_u8_to_hash_set(vec_u8));
}

#[test]
fn vec_u8_to_set_string_two() {
    // ["ACDC", "BOB"]
    let vec_u8 : Vec<u8> = vec![65, 67, 68, 67, super::SEPARATOR,
66, 79, 66];
    let mut set_string = HashSet::new();
    set_string.insert("ACDC".to_string());
    set_string.insert("BOB".to_string());
    assert_eq!(set_string, super::vec_u8_to_hash_set(vec_u8));
}

#[test]
fn set_string_to_vec_u8_empty() {
    let empty_u8 : Vec<u8> = Vec::new();
    let empty_set : HashSet<String> = HashSet::new();
    assert_eq!(empty_u8, super::hash_set_to_vec_u8(&empty_set));
}

#[test]
fn set_string_to_vec_u8_one() {
    // ["ACDC"]
    let vec_u8 : Vec<u8> = vec![65, 67, 68, 67];
    let mut set_string = HashSet::new();
    set_string.insert("ACDC".to_string());
```

```
203        assert_eq!(vec_u8, super::hash_set_to_vec_u8(&set_string));
204    }
205
206    #[test]
207    #[should_panic]
208    fn check_existent_tags_no_file() {
209        let path = "/tmp/check_existent_tags_no_tags";
210
211        // Test with file inexistent
212        let result = super::check_existent_tags(&path);
213        panic!(result);
214    }
215
216    #[test]
217    fn check_existent_tags_no_tags() {
218        let path = "/tmp/check_existent_tags_no_tags";
219        File::create(path).expect("Error when creating file");
220
221        // Test with file with no tags
222        let option = super::check_existent_tags(&path).unwrap();
223        assert_eq!(option, None);
224
225        fs::remove_file(path).expect("Error when removing file");
226    }
227
228    #[test]
229    fn check_existent_tags_tags() {
230        let path = "/tmp/check_existent_tags_tags";
231        File::create(path).expect("Error when creating file");
232
233        // Test with file with tags
234        let vec_u8 = &[65, 67, 68, 67, super::SEPARATOR, 66, 79, 66];
235        let mut tags = HashSet::new();
236        tags.insert("ACDC".to_string());
237        tags.insert("BOB".to_string());
238        super::xattr::set(path, super::ATTR_NAME, vec_u8).unwrap();
239        let option =
    super::check_existent_tags(&path).unwrap().unwrap();
240        assert_eq!(option, tags);
```

```
241
242        fs::remove_file(path).expect("Error when removing file");
243    }
244
245    #[test]
246    fn set_tag() {
247        let path = "/tmp/set_tags";
248        File::create(path).expect("Error when creating file");
249
250        let mut tags = HashSet::new();
251        tags.insert("bob".to_string());
252        let tags_u8 = vec![98, 111, 98];
253        super::set_tags(path, &tags, false);
254        if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
255            if let Some(tags) = res {
256                assert_eq!(tags, tags_u8);
257            }
258        }
259
260        // Reset the same tag
261        super::set_tags(path, &tags, false);
262        if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
263            if let Some(tags) = res {
264                assert_eq!(tags, tags_u8);
265            }
266        }
267
268        fs::remove_file(path).expect("Error when removing file");
269    }
270
271    #[test]
272    fn del_tags() {
273        let path = "/tmp/del_tags";
274        File::create(path).expect("Error when creating file");
275
276        let mut tags = HashSet::new();
277        tags.insert("bob".to_string());
278        tags.insert("max".to_string());;
279        super::set_tags(path, &tags, false);
```

```
280
281          // Delete "bob"
282          let mut bob = HashSet::new();
283          bob.insert("bob".to_string());
284          super::del_tags(path, &bob, false);
285          let tags_u8 = vec![109, 97, 120];
286          if let Ok(res) = super::xattr::get(path, super::ATTR_NAME) {
287              if let Some(tags) = res {
288                  assert_eq!(tags, tags_u8);
289              }
290          }
291
292          fs::remove_file(path).expect("Error when removing file");
293      }
294 }
```

## B.2   src/main.rs

```rust
//! # 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 comma. 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() {
    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 (\",\").\n\n\
        Usage:\n      \
        tag_manager [Options] [[--recursive] --files <files> [[--set|--del] <tags>]]\
        | [--query <query> | --list | --rename <old> <new>]\n\n\
        Options:\n      \
        -h, --help          Display this message\n      \
        -r, --recursive     Recursive option. Get, set or delete tags for each folder \
        and file in folder subtree\n\n\
        Arguments:\n      \
        -f, --files <files>         List of files\n      \
        -s, --set <tags>            Set the given tags\n        \
        -d, --del <tags>            Delete the given tags\n      \
        -q, --query <query>         A logical query to get files\n      \
        -l, --list                  List of existent tags\n      \
```

```
38          -R, --rename <old> <new>   Rename tag, from <old> name to <new> name\n\n\
39          Examples:\n     \
40          tag_manager -f myfile                => Show the actual tags of \
41          file \"myfile\"\n     \
42          tag_manager -f myfile -s work        => Set the tag \"work\" to \
43          the file \"myfile\"\n     \
44          tag_manager -f myfile -d work        => Delete the tag \"work\" \
45          to the file \"myfile\"\n     \
46          tag_manager -f myfolder -r -s geneva   => Set the tag \"geneva\" \
47          to the folder \"myfolder\" and his subtree\n     \
48          tag_manager -q bob AND fred OR max     => Show files corresponding to query\n
49          tag_manager -l                       => Show the list of existent tags\n
50          tag_manager -R old_name new_name     => Rename the tag \"old_name\" to \"ne
51      let matches = App::new("tag_manager")
52          .help(help)
53          .group(ArgGroup::with_name("ops").args(&["set", "del"]))
54          .group(ArgGroup::with_name("queries")
55              .args(&["list", "query", "rename"]))
56          .arg(Arg::with_name("set").short("s").long("set")
57              .takes_value(true).multiple(true))
58          .arg(Arg::with_name("del").short("d").long("del")
59              .takes_value(true).multiple(true))
60          .arg(Arg::with_name("files").short("-f").long("--files")
61              .takes_value(true).multiple(true).required(false))
62          .arg(Arg::with_name("recursive").short("-r")
63              .long("--recursive"))
64          .arg(Arg::with_name("query").short("-q").long("--query")
65              .takes_value(true).multiple(true))
66          .arg(Arg::with_name("list").short("-l").long("--list")
67              .takes_value(false))
68          .arg(Arg::with_name("rename").short("-R").long("--rename")
69              .number_of_values(2))
70          .get_matches();
71
72      if matches.is_present("files") {
73          let files: Vec<&str> =
    matches.values_of("files").unwrap().collect();
74          let recursive = matches.is_present("recursive");
75
```

```
76          if !matches.is_present("set") && !matches.is_present("del") {
77              for file in &files { show_tags(file, recursive); }
78          }
79          else if matches.is_present("set") {
80              let tags: HashSet<&str> =
    matches.values_of("set").unwrap().collect();
81              let tags = &hash_set_str_to_hash_set_string(&tags);
82              for file in &files { tag_manager::set_tags(file, tags,
    recursive); }
83          }
84          else if matches.is_present("del") {
85              let tags : HashSet<&str> =
    matches.values_of("del").unwrap().collect();
86              let tags = &hash_set_str_to_hash_set_string(&tags);
87              for file in &files { tag_manager::del_tags(file, tags,
    recursive); }
88          }
89      }
90      else if matches.is_present("list") || matches.is_present("query") ||
    matches.is_present("rename") {
91          let mut request = String::new();
92          if matches.is_present("query") {
93              let query : Vec<&str> =
    matches.values_of("query").unwrap().collect();
94              request = String::from(CODE_ENTRIES);
95              for q in query {
96                  request.push_str(q);
97                  request.push(' ');
98              }
99          }
100         if matches.is_present("list") {
101             request = String::from(CODE_TAGS);
102         }
103         if matches.is_present("rename") {
104             let query : Vec<&str> =
    matches.values_of("rename").unwrap().collect();
105             request = String::from(CODE_RENAME_TAG);
106             request.push_str(query[0]);
107             request.push(' ');
```

```rust
108            request.push_str(query[1]);
109        }
110        let mut stream = UnixStream::connect(SOCKET_ADDRESS).unwrap();
111        stream.write_all(request.as_str().as_bytes()).unwrap();
112        let mut response = String::new();
113        stream.read_to_string(&mut response).unwrap();
114        print!("{}", response);
115    }
116    else {
117        println!("{}", help);
118    }
119 }
120
121 fn show_tags(file: &str, recursive: bool) {
122    match tag_manager::get_tags(file) {
123        Some(tags) => {
124            let mut tags : Vec<String> = tags.into_iter().collect();
125            tags.sort();
126            println!("Tag(s) {:?} for file \"{}\"", tags, file);
127        },
128        None => println!("File \"{}\" has no tags", file)
129    }
130    match fs::metadata(file) {
131        Ok(result) => {
132            if result.file_type().is_dir() && recursive {
133                for entry in fs::read_dir(file).unwrap() {
134                    let sub_file =
    entry.unwrap().path().to_str().unwrap().to_string();
135                    show_tags(&sub_file, recursive);
136                }
137            }
138        },
139        Err(err) => eprintln!("Error for file \"{}\" : {}", file, err)
140    }
141 }
142
143 fn hash_set_str_to_hash_set_string(files: &HashSet<&str>) ->
    HashSet<String> {
144    let mut new_files : HashSet<String> = HashSet::new();
```

```
145        for f in files { new_files.insert(f.to_string()); }
146        new_files
147 }
```

## B.3 `Cargo.toml`

```
1  [package]
2  name = "tag_manager"
3  version = "0.1.0"
4  authors = ["steven.liatti <steven.liatti@etu.hesge.ch>"]
5
6  [dependencies]
7  clap = "2"
8  xattr = "0.2"
```

# C Code source de Tag Engine

## C.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 {} }
```

```rust
36  }
37
38  impl Node {
39      fn new(name : String, kind : NodeKind) -> Self {
40          Self { name, kind }
41      }
42
43      fn set_name(&mut self, name : String) {
44          self.name = name;
45      }
46  }
47
48  impl Debug for Node {
49      fn fmt(&self, f: &mut Formatter) -> Result {
50          write!(f, "{:?} {:?}", self.kind, self.name)
51      }
52  }
53
54  pub fn make_subgraph(root_index : NodeIndex, tags_index : &mut
    HashMap<String, NodeIndex>,
55      graph : &mut MyGraph, local_path : String, base_path : String) {
56      let mut path_vec : Vec<&str> = local_path.split('/').collect();
57      let mut parent_index = root_index;
58      let mut found = false;
59      let mut build_path : String = base_path;
60      build_path.push_str(path_vec[0]);
61      if !path_vec.is_empty() {
62          // remove path_root
63          path_vec.remove(0);
64          for entry in path_vec {
65              build_path.push('/');
66              build_path.push_str(entry);
67              parent_index = find_parent(&graph, parent_index, entry, &mut
    found);
68              if !found {
69                  let new_node = if metadata(build_path.clone())
70
    .expect("make_subgraph, new_node, metadata").file_type().is_dir() {
71                      Node::new(String::from(entry), NodeKind::Directory)
```

```
72                    }
73                    else { Node::new(String::from(entry), NodeKind::File) };
74                    let new_node = graph.add_node(new_node);
75                    graph.add_edge(parent_index, new_node, Nil::new());
76                    update_tags(build_path.clone(), tags_index, graph,
   new_node);
77                    parent_index = new_node;
78                }
79            }
80        }
81  }
82
83  pub fn make_graph(path_root : String, base_path : String)
84      -> (MyGraph, HashMap<String, NodeIndex>, NodeIndex) {
85      let mut graph : MyGraph = StableGraph::new();
86      let mut tags_index = HashMap::new();
87      let local_root = local_path(&mut path_root.clone(),
88          base_path.clone());
89      let root_index = graph.add_node(
90          Node::new(local_root, NodeKind::Directory)
91      );
92      update_tags(path_root.clone(), &mut tags_index,
93          &mut graph, root_index);
94      let mut is_root = true;
95
96      for entry in WalkDir::new(path_root).into_iter()
97          .filter_map(|e| e.ok()) {
98          if is_root {
99              is_root = false;
100             continue;
101         }
102         let mut path = entry.path().display().to_string();
103         let path = local_path(&mut path, base_path.clone());
104         make_subgraph(root_index, &mut tags_index, &mut graph,
105             path, base_path.clone());
106     }
107     (graph, tags_index, root_index)
108 }
109
```

```
110  pub fn local_path(absolute_path : &mut String, base_path : String) ->
     String {
111      absolute_path.split_off(base_path.len())
112  }
113
114  pub fn get_node_index(root_index : NodeIndex, graph : &MyGraph, path :
     String) -> NodeIndex {
115      let mut path_vec : Vec<&str> = path.split('/').collect();
116      let mut parent_index = root_index;
117      let mut found = false;
118      if !path_vec.is_empty() {
119          // remove path_root
120          path_vec.remove(0);
121          for entry in path_vec {
122              parent_index = find_parent(&graph, parent_index, entry, &mut
     found);
123          }
124      }
125      parent_index
126  }
127
128  pub fn move_entry(root_index : NodeIndex, entry_index : NodeIndex, graph
     : &mut MyGraph, new_path : String) {
129      let mut parent_index = entry_index;
130      for neighbor_index in graph.neighbors_directed(entry_index,
     Direction::Incoming) {
131          match graph.node_weight(neighbor_index) {
132              Some(data) => {
133                  match data.kind {
134                      NodeKind::Directory => {
135                          parent_index = neighbor_index;
136                          break;
137                      },
138                      _ => ()
139                  }
140              },
141              None => ()
142          }
143      }
```

```
144    let new_parent_index = get_node_index(root_index, graph,
new_path.clone());
145    if parent_index == new_parent_index {
146        let mut path_vec : Vec<&str> = new_path.split('/').collect();
147        let new_name =
path_vec.pop().expect("move_entry, path_vec.pop()").to_string();
148        let node = graph.node_weight_mut(entry_index)
149            .expect("move_entry, graph.node_weight_mut");
150        node.set_name(new_name);
151    }
152    else {
153        let edge = graph.find_edge(parent_index, entry_index);
154        match edge {
155            Some(edge_index) => { graph.remove_edge(edge_index); },
156            None => ()
157        }
158        graph.add_edge(new_parent_index, entry_index, Nil::new());
159    }
160 }
161
162 pub fn remove_entries(entry_index : NodeIndex, graph : &mut MyGraph,
tags_index : &mut HashMap<String, NodeIndex>) {
163    let mut entries_index = Vec::new();
164    let mut check_tags_index = Vec::new();
165    entries_to_remove(entry_index, graph, &mut entries_index, &mut
check_tags_index);
166    for index in entries_index.into_iter().rev() {
167        graph.remove_node(index);
168    }
169    for tag_index in check_tags_index {
170        if graph.edges(tag_index).count() == 0 {
171
tags_index.remove(&graph.node_weight(tag_index).unwrap().name);
172            graph.remove_node(tag_index);
173        }
174    }
175 }
176
```

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

```rust
210            },
211            None => ()
212        }
213     }
214 }
215
216 // ----------------------- TAGS ------------------------
217
218 pub fn update_tags(path : String,
219     tags_index : &mut HashMap<String, NodeIndex>,
220     graph : &mut MyGraph, entry_index : NodeIndex) {
221     let existent_tags = get_tags(graph, entry_index);
222     let fresh_tags = match tag_manager::get_tags(&path) {
223         Some(tags) => tags,
224         None => HashSet::new()
225     };
226     remove_tags(existent_tags.difference(&fresh_tags),
227         tags_index, graph, entry_index);
228     add_tags(fresh_tags.difference(&existent_tags),
229         tags_index, graph, entry_index);
230 }
231
232 fn get_tags(graph : &MyGraph, tag_index : NodeIndex) -> HashSet<String>
    {
233     let mut tags = HashSet::new();
234     for neighbor_index in graph.neighbors_directed(tag_index,
    Direction::Incoming) {
235         match graph.node_weight(neighbor_index) {
236             Some(data) => {
237                 match data.kind {
238                     NodeKind::Tag => { tags.insert(data.name.clone());
    },
239                     _ => ()
240                 }
241             },
242             None => ()
243         }
244     }
245     tags
```

```
246  }
247
248  fn add_tags(tags_to_add : Difference<String, RandomState>, tags_index :
     &mut HashMap<String, NodeIndex>,
249      graph : &mut MyGraph, entry_index : NodeIndex) {
250      for tag in tags_to_add {
251          match tags_index.entry(tag.clone()) {
252              Vacant(entry) => {
253                  let new_node_tag = graph.add_node(Node::new(tag.clone(),
     NodeKind::Tag));
254                  entry.insert(new_node_tag);
255                  graph.add_edge(new_node_tag, entry_index, Nil::new());
256              },
257              Occupied(entry) => {
258                  let &tag_index = entry.get();
259                  graph.add_edge(tag_index, entry_index, Nil::new());
260              }
261          }
262      }
263  }
264
265  fn remove_tags(tags_to_remove : Difference<String, RandomState>,
     tags_index : &mut HashMap<String, NodeIndex>,
266      graph : &mut MyGraph, entry_index : NodeIndex) {
267      for tag in tags_to_remove {
268          match tags_index.entry(tag.clone()) {
269              Occupied(entry) => {
270                  let &tag_index = entry.get();
271                  match graph.find_edge(tag_index, entry_index) {
272                      Some(edge) => { graph.remove_edge(edge); },
273                      None => ()
274                  }
275                  if graph.edges(tag_index).count() == 0 {
276                      entry.remove();
277                      graph.remove_node(tag_index);
278                  }
279              },
280              Vacant(_) => ()
281          }
```

```
282          }
283    }
```

## C.2 src/lib.rs

```rust
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;

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);
            println!("========== CHMOD : {:?} ==========", local);
```

```
33          let entry_index = get_node_index(root_index, graph, local);
34          update_tags(path, tags_index, graph, entry_index);
35      },
36      Remove(path) => {
37          let mut path =
path.as_path().to_str().expect("dispatcher, remove, path").to_string();
38          let local = local_path(&mut path.clone(), base);
39          println!("========== REMOVE : {:?} ==========", local);
40          let entry_index = get_node_index(root_index, graph, local);
41          remove_entries(entry_index, graph, tags_index);
42      },
43      Rename(old_path, new_path) => {
44          let mut old_path = old_path.as_path().to_str()
45              .expect("dispatcher, rename, old_path").to_string();
46          let new_path = new_path.as_path().to_str()
47              .expect("dispatcher, rename, new_path").to_string();
48          let old_local = local_path(&mut old_path.clone(),
base.clone());
49          let new_local = local_path(&mut new_path.clone(),
base.clone());
50
println!("========== RENAME, old_path : {:?}, new_path : {:?} ==========",
old_local, new_local);
51          let entry_index = get_node_index(root_index, graph,
old_local);
52          move_entry(root_index, entry_index, graph, new_local);
53      }
54      _ => ()
55  }
56 }
```

## C.3  `src/main.rs`

```rust
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};

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)
}

fn write_dot_image(graph : &MyGraph, dot_name : &str, image_name : &str)
{
```

```rust
36      let mut file = File::create(dot_name).expect("file create");
37      let graph_dot = format!("{:?}", Dot::with_config(graph,
   &[Config::EdgeNoLabel]));
38      file.write(graph_dot.as_bytes()).expect("file write");
39      let mut output = String::from("-o");
40      output.push_str(image_name);
41      let _exec_dot = Command::new("dot").args(&["-Tpng", output.as_str(),
   dot_name]).output().expect("exec");
42  }
43
44  fn main() {
45      let matches =
   App::new("Tag Engine").version("0.1.0").author("Steven Liatti")
46          .arg(Arg::with_name("path")
47              .takes_value(true).required(true).multiple(false))
48          .arg(Arg::with_name("debug")
49
   .short("-d").long("--debug").required(false).multiple(false))
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();
```

```rust
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!();
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 }
```

## C.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() {
```

```rust
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")),
```

```rust
114                Arg::Operand(String::from("max")),
115                Arg::Operator(Operator::AND),
116                Arg::Operator(Operator::OR)
117            ];
118            assert_eq!(infix_to_postfix(infix), postfix);
119        }
120
121        #[test]
122        fn test_infix_to_postfix_5() {
123            let infix = String::from("bob AND fred AND max");
124            let postfix = vec![
125                Arg::Operand(String::from("bob")),
126                Arg::Operand(String::from("fred")),
127                Arg::Operator(Operator::AND),
128                Arg::Operand(String::from("max")),
129                Arg::Operator(Operator::AND)
130            ];
131            assert_eq!(infix_to_postfix(infix), postfix);
132        }
133
134        #[test]
135        fn test_infix_to_postfix_6() {
136            let infix = String::from("bob AND fred OR max AND paul");
137            let postfix = vec![
138                Arg::Operand(String::from("bob")),
139                Arg::Operand(String::from("fred")),
140                Arg::Operator(Operator::AND),
141                Arg::Operand(String::from("max")),
142                Arg::Operand(String::from("paul")),
143                Arg::Operator(Operator::AND),
144                Arg::Operator(Operator::OR)
145            ];
146            assert_eq!(infix_to_postfix(infix), postfix);
147        }
148    }
```

## C.5   src/server.rs

```rust
1   use std::collections::{HashMap, HashSet};
2   use std::io::prelude::*;
3   use std::sync::{Mutex, Arc};
4   use std::os::unix::net::{UnixListener, UnixStream};
5   use std::fs::remove_file;
6
7   extern crate petgraph;
8   use petgraph::graph::NodeIndex;
9   use petgraph::Direction;
10
11  extern crate tag_manager;
12
13  use graph::{MyGraph, NodeKind};
14  use parse::{Arg, Operator};
15  use parse::infix_to_postfix;
16
17  const BUFFER_SIZE : usize = 4096;
18  const CODE_SIZE : usize = 3;
19  const BIND_ADDRESS : &str = "/tmp/tag_engine";
20
21  #[derive(Debug, Clone)]
22  enum RequestKind {
23      Entries(String),
24      Tags,
25      RenameTag(String)
26  }
27
28  fn parse_request(stream : &mut UnixStream) -> Option<RequestKind> {
29      let mut buffer = [0; BUFFER_SIZE];
30      let size = stream.read(&mut buffer).unwrap();
31      if size >= CODE_SIZE {
32          let mut request = String::new();
33          for i in CODE_SIZE..size {
34              request.push(buffer[i] as char);
35          }
36          let mut kind = String::new();
37          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()),
```

```rust
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 }
```

## C.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"
```