



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

DESARROLLO DE UN SISTEMAS DISTRIBUIDOS

Optimizacion

Grupo: 4CM1

Integrantes:

Angel Lopez Manriquez

Aldo Suárez cruz

Mónica Nataly Toxtli Calderón

Irving Arturo Aguiar Hernández

Profesor:

Coronilla Contreras Ukranio

Fecha de realización: 7 de mayo de 2020

Reporte de practica

4CM1

7 de mayo de 2020

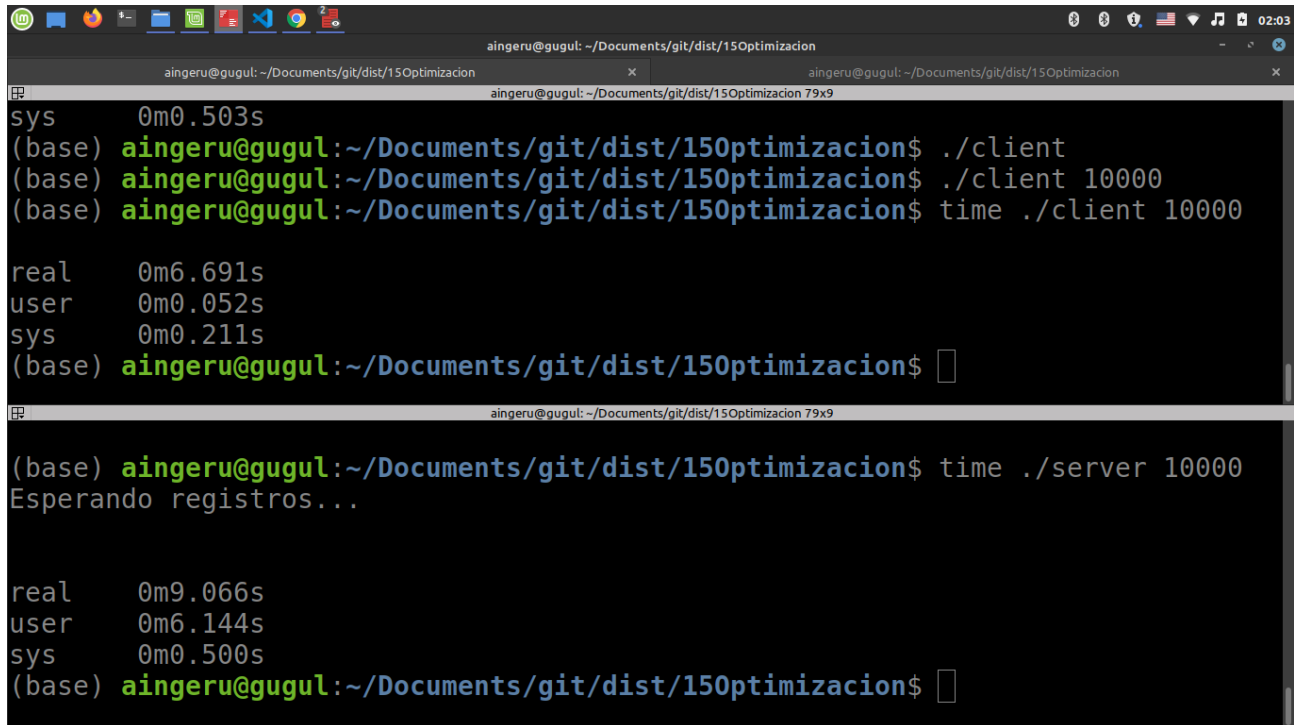
Índice

1. Parte	2
1.1. Tiempo usado para 10k votantes	2
1.2. Tiempo usado para 10k votantes	2
1.3. Preguntas	3
2. Parte 2	3

1. Parte

1.1. Tiempo usado para 10k votantes

El servidor no se preocupa si hay votos repetidos.



```
sys      0m0.503s
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ ./client
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ ./client 10000
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ time ./client 10000

real      0m6.691s
user      0m0.052s
sys       0m0.211s
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$

(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ time ./server 10000
Esperando registros...

real      0m9.066s
user      0m6.144s
sys       0m0.500s
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$
```

1.2. Tiempo usado para 10k votantes

El servidor se preocupa si hay votos repetidos.



```
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ time ./client 10000

real      0m6.346s
user      0m0.183s
sys       0m0.539s
(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$

(base) aingeru@gugul:~/Documents/git/dist/150optimizacion$ time ./server 10000
Datos cargados en la estructura trie.
Esperando registros...

real      0m11.567s
user      0m5.900s
sys       0m0.049s
```

Mas sin embargo no hay que confiarse mucho de los tiempos ya que hay un ligero retraso

en lo que se ordena que un cliente empiece a enviar solicitudes.

1.3. Preguntas

Para saber si es posible atender 70 millones podríamos hacer uso de la teoría de análisis de algoritmos o bien de algoritmos de aprendizaje automático y obtener puntos (n_i, t_i) . Para validar si la solicitud es repetida se usó un **set** el cual hace inserciones en $O(\lg n)$ mientras que **Trie** se demora $O(n)$ donde n es la longitud de la palabra. Claro está que el orden asintótico de la operación de búsqueda del set es mayor a partir de un N , aunque no consideramos otros aspectos como el tiempo de resolución de los routers, por lo que un análisis un poco más preciso sería usar algún algoritmo de aprendizaje automático. Por ejemplo, usando los *Polinomios de Lagrange* y ajustándolo para $n = 5k, 10k, 20k$ tenemos que $f(x) \rightarrow 0.0555333x^2 + 0.077x + 0.266667$, así $f(70000k) = 272118560.26666 \text{ secs} = 3149.5203734567126 \text{ días} = 8.6288 \text{ años}$. Aunque habría que recolectar más datos para una mejor aproximación.

2. Parte 2

Se hizo la implementación de la estructura usando árboles n-arios.

```

1  /** Trie implementation by using KTrees, it maps string -> bool.
2      *
3      */
4
5  #ifndef __TRIE_H__
6  #define __TRIE_H__
7
8  #include <iostream>
9  #include <algorithm>
10 #include <iterator>
11 #include <vector>
12 #include <memory>
13 #include <string>
14
15
16 using namespace std;
17
18 template<typename T>
19 struct Trie {
20
21     struct Node {
22         char character;
23         T data;
24         std::vector<std::unique_ptr<struct Node > > children;
25
26         Node() {}

```

```

27     Node(char &_char): character(_char) { }
28     Node(char &_char, T _data): character(_char), data(_data) { }
29 };
30
31
32 using u_ptr_node = std::unique_ptr<Node >;
33 using u_ptr_nodes = std::vector<u_ptr_node >;
34
35 std::unique_ptr<Node > root;
36
37
38 Trie() {
39     root = std::make_unique<Node >();
40 }
41
42 void __put(std::string &key, unsigned i, T &value, u_ptr_node &current_ptr)
43 {
44     auto &children = current_ptr.get()->children;
45     // Insert the leaf with the value associated
46     // If the element was inserted previously it'll be inserted again
47     if (i == key.size() - 1) {
48         children.push_back(make_unique<Node> (key[i], value));
49         return;
50     }
51     auto it = std::find_if(children.begin(), children.end(),
52         [&](u_ptr_node &child) -> bool {
53         return (child.get()->character == key[i]);
54     });
55     if (it != children.end()) { // found
56         __put(key, i + 1, value, *it);
57         return;
58     }
59     children.push_back(make_unique<Node> (key[i]));
60     __put(key, i + 1, value, children[children.size() - 1]);
61 }
62
63 void put(std::string &key, T value) {
64     __put(key, 0, value, root);
65 }
66
67 Node * __get_node(u_ptr_node &self, std::string &key, unsigned i) {
68     if (i >= key.size())
69         return nullptr; // index out of bounds
70     auto &children = self.get()->children;
71     auto it = std::find_if(children.begin(), children.end(),
72         [&](auto &node_uptr) -> bool {
73         return node_uptr.get()->character == key[i];
74     });
75     if (it == children.end())

```

```

76         return nullptr; // No such value
77     if (i == key.size() - 1)
78         return it->get();
79     return __get_node(*it, key, i + 1);
80 }
81
82 T get(std::string &key) {
83     Node *node_ptr = __get_node(root, key, 0);
84     if (node_ptr == nullptr)
85         throw "Trie:get: There's no such value\n";
86     return node_ptr->data;
87 }
88
89 T get(std::string &key, T &default_value) {
90     Node *node_ptr = __get_node(root, key, 0);
91     if (node_ptr == nullptr)
92         return default_value;
93     return node_ptr->data;
94 }
95
96 bool has(std::string key) {
97     return __get_node(root, key, 0) != nullptr;
98 }
99
100 };
101
102
103 #endif

```

Las pruebas descritas anteriormente fueron hechas usando esta estructura.

Cada entrada a la estructura Trie ocupa al menos 11 bytes como clave (`char [11] celular`) y 16 como valor (`2 * sizeof(long)`). Así tenemos que la función de espacio sería aproximadamente $\Theta(n) = 16n$ por lo que con 10 000 votos tenemos 160 KB y con 70 M votos tenemos 1.12 Gb en ram.