Kademlia DHT implementation :

Hello,

For this homework, I wrote my own implementation of Kademlia DHT in Rust language. I used the **kademlia-dht** library available at this address: https://crates.io/crates/kademlia-dht

How to install?

If you use Windows on a 64 bits platform, you can just download the executable here: https://github.com/thomasarmel/simple_kademlia_implementation/releases/tag/v1

Otherwise, run the following commands:

git clone https://github.com/thomasarmel/simple_kademlia_implementation.git

cd simple_kademlia_implementation

cargo build –release

How to use it?

At first, run the first node:

.\target\release\kademlia.exe -l 127.0.0.1:8080

In case you want to listen on local interface on port 8080.

Then another node can join using the first as entry point:

.\target\release\kademlia.exe -l 127.0.0.1:8081 -r 127.0.0.1:8080

It will listen on local interface, on port 8081

Finally, a third node can join using the second one as entry point:

`.\target\release\kademlia.exe -l 127.0.0.1:8082 -r 127.0.0.1:8081`

It will listen on local interface, on port 8082

Commands:

I my_key my_value => store **my_value** using key **my_key**.

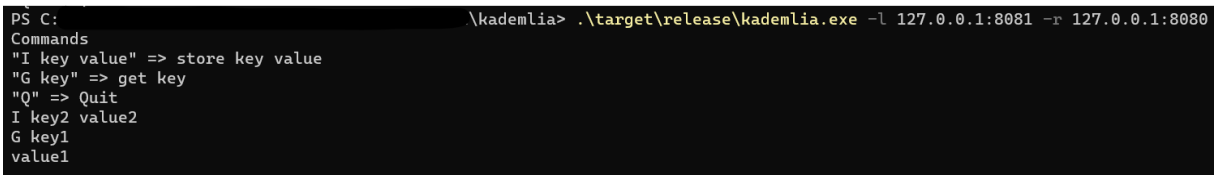
G my_key => Read value stored with key **my_key**.

Q => Quit.

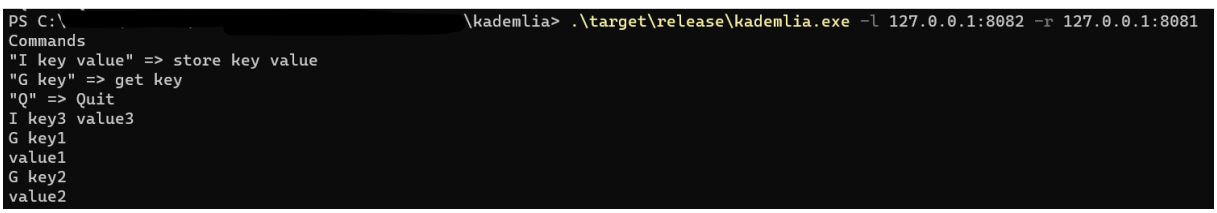
Example:

Create first node and insert **key1 => value1**:

Create second node, insert **key2 => value2** and read **key1**:



Create third node, insert **key3 => value3** and read **key1** and **key2**:



On first node, read **key2** and **key3**:

```
PS C:\                                    \kademlia> .\target\release\kademlia.exe -l 127.0.0.1:8080
Commands
"I key value" => store key value
"G key" => get key
"Q" => Quit
I key1 value1
G key1
value1
G unexisting_key
Not found
G key2
value2
G key3
value3
```

If we modify a value on a node:

```
I key1 value1bis
```

Then it will be modified for all nodes:

```
G key1
value1bis
```

How it works?

Hashing

Each node will have an unique identifier, **sha256(ip:port):**

```
let remote_addr_hash_vec : Vec<u8> = sha3::Sha3_256::digest( data: remote_str.as_bytes()).as_slice().to_vec();
```

When a node wants to join an existing DHT, it needs to know the entry point and the entry point ID:

```
Node::new( ip: listen[0], port: listen[1], bootstrap: Some(NodeData {
    addr: remote_str,
    id: Key::new( data: <[u8; 32]>::try_from(remote_addr_hash_vec).unwrap()),
}))
```

The same hashing algorithm is using for the keys.

Here the insertion:

```
let key_hash_vec : Vec<u8> = Sha3_256::digest( data: command[1].as_bytes()).as_slice().to_vec();
let key : Key = Key::new( data: <[u8; 32]>::try_from(key_hash_vec).unwrap());
node.insert(key, value: command[2]);
```

And the reading:

```
let key_hash_vec : Vec<u8> = Sha3_256::digest( data: command[1].as_bytes()).as_slice().to_vec();
let key : Key = Key::new( data: <[u8; 32]>::try_from(key_hash_vec).unwrap());
match node.get(&key) {
    Some(value : String ) => println!("{}", value),
    None => println!("Not found"),
}
```

XOR distance

Here you can see the closest node is calculated using the XOR distance:

```
pub fn get_closest_nodes(&self, key: &Key, count: usize) -> Vec<NodeData> {
    let index = cmp::min(
        self.node_data.id.xor(key).leading_zeros(),
        self.buckets.len() - 1,
    );


ret.sort_by_key(|node| node.id.xor(key));
ret.truncate(count);
ret
```

Buckets:

We can see buckets work using the node hash as identifier.

```rust
pub fn update_node(&mut self, node_data: NodeData) -> bool {
    let distance = self.node_data.id.xor(&node_data.id).leading_zeros();
    let mut target_bucket = cmp::min(distance, self.buckets.len() - 1);

    if self.buckets[target_bucket].contains(&node_data) {
        self.buckets[target_bucket].update_node(node_data);
        return true;
    }

    loop {
        // bucket is not full
        if self.buckets[target_bucket].size() < REPLICATION_PARAM {
            self.buckets[target_bucket].update_node(node_data);
            return true;
        }

        let is_last_bucket = target_bucket == self.buckets.len() - 1;
        let is_full = self.buckets.len() == ROUTING_TABLE_SIZE;

        // bucket cannot be split
        if !is_last_bucket || is_full {
            return false;
        }

        // split bucket
        let new_bucket = self.buckets[target_bucket].split(&self.node_data.id, target_bucket);
        self.buckets.push(new_bucket);
```

Find a value:

If the value is stored on local node, return it. Otherwise, find the closest node for the value in the routing table.

```
   RequestPayload::FindValue(key) => {
       if let Some(value) = self.storage.lock().unwrap().get(&key) {
           ResponsePayload::Value(value.clone())
       } else {
           ResponsePayload::Nodes(
               self.routing_table
                   .lock()
                   .unwrap()
                   .get_closest_nodes(&key, REPLICATION_PARAM),
           )
       }
   }
:
```

And then ask this node for the value.