# Stressing my Kademlia implementation and diving into the code

## Stressing the Kademlia implementation:

First, we will have a look on how the Kademlia implementation manage stressful situations.

Because I don't have a large IT infrastructure at my disposal, I will have to launch all the Kademlia instances on my laptop (11th Gen Intel Core i7-11390H @ 3.4 GHz, 16 GB RAM). Furthermore I will use Ubuntu on WSL 2 instead of Windows for obvious practical reasons.

Let's write the following **stress.sh** bash script:

```
#!/bin/bash

for i in {1..10000}
do
  listening_port=$(($i + 8081))
  echo "I key$i value$i" | target/release/kademlia -l 127.0.0.1:$listening_port -r 127.0.0.1:8080 2>&1
> /dev/nul &l
done
```

It will launch **10 000 instances** of Kademlia, with each an unique id **i**. Each instance will connect to the **entry point 127.0.0.1:8080** and listen on 127.0.0.1:{8081 + i}.

**Note:** On a real network, it's preferable that each instance use a different entry point to join the network, but it doesn't change anything here because the CPU will be used anyway.

Then the implementations will insert in the DHT **key{i} => value{i}**.

So let's have a look on what happened.

First of all, we **launch the first entry point** on a separated terminal:

```
thomas@DESKTOP-4DF2HCG:~/simple_kademlia_implementation$ target/release/kademlia -l 127.0.0.1:8080
Commands
"I key value" => store key value
"G key" => get key
"Q" => Quit
```

Then we launch the script:

```
thomas@DESKTOP-4DF2HCG:~/simple_kademlia_implementation$ ./stress.sh
```

Obviously I rapidly ran out of resources (even if the program is written in Rust):

| Nom | Statut | 96% Processeur | 88% Mémoire | 0% Disque | 0% Réseau |
|---|---|---|---|---|---|
| VmmemWSL | | 57,5% | 6719,3 Mo | 0 Mo/s | 0 Mbits/s |

.

Around last 80% of nodes just crashed because of the resources lack:

```
thread '<unnamed>' panicked at 'failed to spawn thread: Os { code: 11, kind: WouldBlock, message: "Resource temporarily
unavailable" }', /rustc/4b91a6ea7258a947e59c6522cd5898e7c0a6a88f/library/std/src/thread/mod.rs:656:29
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

The 20% (2000) first nodes took a bunch of minutes to start. Because the Kademlia process runs in a separate thread, in means the **time bottleneck is the process and thread creation**, and not the connection to the Kademlia network.

**Let's drive back to the first entry point**, and try to read values injected by the alive nodes.

If we try to read a value that hasn't been injected into the DHT, the response "Not found" will take a while to appear:

```
G key5000
Not found
```

However, if we try to read a key that has actually been injected, the response comes quasi instantly, no matter it's number:

```
G key100
value100
```

```
G key2000
value2000
```

It means that finding an existing key is a lot quicker than looking for an nonexistent key, because the number of nodes to interrogate is a lot shorter.

Finally, let's perform a lot of cleanup:

```
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
kademlia: no process found
thomas@DESKTOP-4DF2HCG:~$ killall kademlia
```

**Conclusion:**

The performances test demonstrated that the Kademlia DHT network is really performant. Indeed, the performant bottleneck on a local machine was the thread and process creation, and node the Kademlia tasks themselves (joining the network and inserting / reading keys).

## Dive into the code:

**Find node:**

```rust
/// The maximum number of entries in a k-bucket.
const REPLICATION_PARAM: usize = 20;
```

```rust
RequestPayload::FindNode(key) => ResponsePayload::Nodes(
    self.routing_table
        .lock()
        .unwrap()
        .get_closest_nodes(&key, REPLICATION_PARAM),
),
```

```rust
/// Returns the closest `count` nodes to `key`.
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,
    );
    let mut ret = Vec::new();

    // the closest keys are guaranteed to be in bucket which the key would reside
    ret.extend_from_slice(self.buckets[index].get_nodes());

    if ret.len() < count {
        // the distance between target key and keys is not necessarily monotonic
        // in range (key.leading_zeros(), self.buckets.len()], so we must iterate
        for i in (index + 1)..self.buckets.len() {
            ret.extend_from_slice(self.buckets[i].get_nodes());
        }
    }

    if ret.len() < count {
        // the distance between target key and keys in [0, key.leading_zeros())
        // is monotonicly decreasing by bucket
        for i in (0..index).rev() {
            ret.extend_from_slice(self.buckets[i].get_nodes());
            if ret.len() >= count {
                break;
            }
        }
    }

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

So as we can see, the replication param is set to 20: *FindNode(key)* will return **a list of 20 nodes**, sorted by the XOR distance to the key. The buckets attribute is a growable list of "routing buckets". So the algorithm will find the closest nodes in the nodes tree, with a logarithmic complexity. It's a search-tree based on XOR distance.

## k-bucket structure:

```rust
/// A k-bucket in a node's routing table that has a maximum capacity of `REPLICATION_PARAM`.
///
/// The nodes in the k-bucket are sorted by the time of the most recent communication with those
/// which have been most recently communicated at the end of the list.
#[derive(Clone, Debug)]
struct RoutingBucket {
    nodes: Vec<NodeData>,
    last_update_time: SteadyTime,
}
```

As explained in the comment, the buckets are automatically refreshed.

## Reading and writing message:

```rust
impl Protocol {
    pub fn new(socket: UdpSocket, tx: Sender<Message>) -> Protocol {
        let protocol = Protocol {
            socket: Arc::new(socket),
        };
        let ret = protocol.clone();
        thread::spawn(move || {
            let mut buffer = [0u8; MESSAGE_LENGTH];
            loop {
                let (len, _src_addr) = protocol.socket.recv_from(&mut buffer).unwrap();
                let message = bincode::deserialize(&buffer[..len]).unwrap();

                if tx.send(message).is_err() {
                    warn!("Protocol: Connection closed.");
                    break;
                }
            }
        });
        ret
    }

    pub fn send_message(&self, message: &Message, node_data: &NodeData) {
        let size_limit = bincode::Bounded(MESSAGE_LENGTH as u64);
        let buffer_string = bincode::serialize(&message, size_limit).unwrap();
        let NodeData { ref addr, .. } = node_data;
        if self.socket.send_to(&buffer_string, addr).is_err() {
            warn!("Protocol: Could not send data.");
        }
    }
}
```

It’s simply socket transmission over UDP, on the port specified in the command line argument.

## Ping:

For pinging a node, the implementation just send a message with a special payload:

```rust
/// Sends a `PING` RPC.
fn rpc_ping(&mut self, dest: &NodeData) -> Option<Response> {
    self.send_request(dest, RequestPayload::Ping)
}
```

And the other node responds as the same way:

```
/// Handles a request RPC.
fn handle_request(&mut self, request: &Request) {
    info!(
        "{} - Receiving request from {} {:#?}",
        self.node_data.addr, request.sender.addr, request.payload,
    );
    self.clone().update_routing_table(request.sender.clone());
    let receiver = (*self.node_data).clone();
    let payload = match request.payload.clone() {
        RequestPayload::Ping => ResponsePayload::Pong,
        RequestPayload::Store(key, value) => {
            self.storage.lock().unwrap().insert(key, value);
```

**Leave:**

There is no implementation of the leaving process, the node will just be removed after the refreshment period. However, there is a method to “kill” a node, meaning ask it to disconnect.