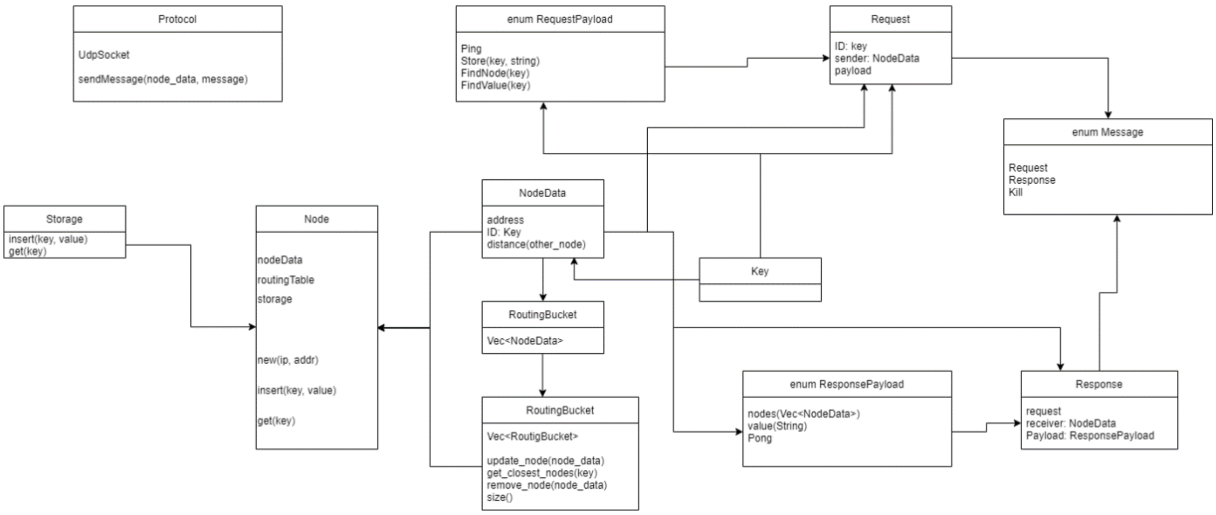
Anne Honyme

Kademlia code review

https://gitlab.com/jeffrey-xiao/kademlia-dht-rs

Architecture :



Messages are serialized with **serde crate** before being sent to the other nodes via UDP.

Comments :

The implementation uses the UDP protocol.

Bucket table :

src/routing.rs

```rust
/// A node's routing table tree.
///
/// `RoutingTable` is implemented using a growable vector of `RoutingBucket`. The relaxation of
/// k-bucket splitting proposed in Section 4.2 is not implemented.
#[derive(Clone, Debug)]
pub struct RoutingTable {
    buckets: Vec<RoutingBucket>,
    node_data: Arc<NodeData>,
}
```

```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,
}
```

The bucket table is a list of list of NodeData. NodeData contains the node ID and address, and allows to sort nodes and get distance between them.

Update bucket table :

All constants are defined in src/lib.rs.

REPLICATION_PARAM is the maximum number of entries in a k-bucket, here 20.

ROUTING_TABLE_SIZE is the maximum number of k-buckets in the routing table, here 256.

```rust
    /// Upserts a node into the routing table. It will continue to split the routing table until the
    /// routing table is full or until the node can be upserted.
    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);

            target_bucket = cmp::min(distance, self.buckets.len() - 1);
        }
    }
```

Find_node :

Src/node/mod.rs

```rust
    /// Sends a `FIND_NODE` RPC.
    fn rpc_find_node(&mut self, dest: &NodeData, key: &Key) -> Option<Response> {
        self.send_request(dest, RequestPayload::FindNode(*key))
    }
```

It will send a request of type FindNode to the destination, with the requested node ID as parameter.

As a response to a SendNode request, the other node will return the closest nodes to the targeted one:

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

(It can be done because we are able to calculate the inter-nodes distance).

Find_value :

Same as find_node, a request of type FindValue is sent:

```
    /// Sends a `FIND_VALUE` RPC.
    fn rpc_find_value(&mut self, dest: &NodeData, key: &Key) -> Option<Response> {
        self.send_request(dest, RequestPayload::FindValue(*key))
    }
```

Then, when a FindValue request is received, 2 options:

- We can return the value
- We return the closest nodes able to contain the value

```
            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),
                    )
                }
```

Ping :

We send a request to the target of type ping:

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

The target answers PONG:

```
            RequestPayload::Ping => ResponsePayload::Pong,
```

Join :

The join procedure is done directly when creating a new node. The entry point is the bootstrap parameter:

```
    /// Constructs a new `Node` on a specific ip and port, and bootstraps the node with an existing
    /// node if `bootstrap` is not `None`.
    pub fn new(ip: &str, port: &str, bootstrap: Option<NodeData>) -> Self {
        let addr = format!("{}:{}", ip, port);
        let socket = UdpSocket::bind(addr).expect("Error: could not bind to address.");
        let node_data = Arc::new(NodeData {
            addr: socket.local_addr().unwrap().to_string(),
            id: Key::rand(),
        });
        let mut routing_table = RoutingTable::new(Arc::clone(&node_data));
        let (message_tx, message_rx) = channel();
        let protocol = Protocol::new(socket, message_tx);

        // directly use update_node as update_routing_table is async
        if let Some(bootstrap_data) = bootstrap {
            routing_table.update_node(bootstrap_data);
        }

        let mut ret = Node {
            node_data,
            routing_table: Arc::new(Mutex::new(routing_table)),
            storage: Arc::new(Mutex::new(Storage::new())),
            pending_requests: Arc::new(Mutex::new(HashMap::new())),
            protocol: Arc::new(protocol),
            is_active: Arc::new(AtomicBool::new(true)),
        };

        ret.start_message_handler(message_rx);
        ret.start_bucket_refresher();
        ret.bootstrap_routing_table();
        ret
    }
```

Line 48, the entry point is added to the buket table.

Then let's have a look to the *bootstrap_routing_table()* function:

```
    /// Bootstraps the routing table using an existing node. The node first looks up its id to
    /// identify the closest nodes to it. Then it refreshes all routing buckets by looking up a
    /// random key in the buckets' range.
    fn bootstrap_routing_table(&mut self) {
        let target_key = self.node_data.id;
        self.lookup_nodes(&target_key, true);

        let bucket_size = { self.routing_table.lock().unwrap().size() };

        for i in 0..bucket_size {
            self.lookup_nodes(&Key::rand_in_range(i), true);
        }
    }
```

First of all, the node will try to find itself to fill its bucket table. Then it will search for random nodes in order to fulfill the table.

This part could be improved: it could be done concurrently in order to avoid loosing time.

Leave:

A **kill** message is sent to the node itself:

```
    /// Kills the current node and all active threads.
    pub fn kill(&self) {
        self.protocol.send_message(&Message::Kill, &self.node_data);
    }
```

When received, the corresponding node will be marked as inactive by itself, the other nodes are not informed.

```
                match request {
                    Message::Request(request) => node.handle_request(&request),
                    Message::Response(response) => node.handle_response(&response),
                    Message::Kill => {
                        node.is_active.store(false, Ordering::Release);
                        info!("{} - Killed message handler", node.node_data.addr);
                        break;
                    }
                }
            }
```

This part could be improved, in order to inform the other nodes that we are leaving.

Anne Honyme

Benchmarks :

It looks like the project isn't maintained anymore, the last commit has been pushed 3 years ago. The merge request hasn't been considered.

There are 8 files, sliced into structures and enum (Rust is a functional paradigm language).

There is a total of 1245 lines of an excellent Rust code, including the unit tests; which is quite small.

The nodes communicate through UDP, and the messages are serialized using **serde crate**, in bincode format (https://docs.rs/bincode/latest/bincode/), as you can see in the file **src/protocol.rs**.

Criteria for Software Self-Assessment :

kademlia-dht-rs: Family=vehicle; Audience=partners; Evolution=nofuture; Duration=2; Contribution=none; Url=https://gitlab.com/jeffrey-xiao/kademlia-dht-rs

It's a library aiming to create a small implementation of the kademlia protocol in Rust, for educational purposes.