Marius Bosler, Jörn Fischbach

Peer to Peer Systems and Security

Documentation for the VoidPhone Project

Team “The Hashies”

1. Latest version of our architecture / structure as introduced in the midterm report

Our DHT-module is now split up into three main parts:

- dht_server.cpp
- routing.cpp
- ssl.cpp

In the following, we explain the dht_server.cpp file. This file is compiled to the final executable. It contains major server logic and is dependent on the other two source files. Important remark / disclaimer for grading: The ssl.cpp was almost exclusively auto generated with LLMs like ChatGPT a. o. We do not want to take credit for this file. Nonetheless, the integration, correct usage, and rewriting of several methods by hand easily summed up to 30 hrs. of effort for Jörn.

In the main function of dht_server, we use boost to parse command line arguments. We pass the following arguments to start a server.

Command line (Input) Arguments:

• host-address: The IP address by which the host will be reachable for peers. This can be an IPv4 or IPv6 address.

• module-port: The port on which we listen for API request from other modules (DHT PUT, DHT GET)

• p2p-port: The port on which we listen for RtiPCs by other peers.

Only passing these arguments will start the DHT server as a new network. To connect to an existing network, we need to pass two additional arguments to reach a node in the existing network:

• peer-address: IP of one peer to join their network (again, IPv4 or IPv6)

• peer-port: Corresponding p2p-port of the peer

If we know multiple addresses of potential nodes, we can try to start the server with any node (for this, several executions with modified command line arguments may be needed). As soon as one is responsive, the server joins the network.

Additional flags such as –l for logging level and –v for verification of certification can be read from the help page.

We can also pass --help to see examples of usage and description for the available arguments.

Threading scheme:

Our server currently uses two threads;
One main thread does all logical decisions, all state-keeping of connections and answering requests and even implements an asynchronous task queue, in which we can emplace work needed to be done in the future.
One additional background thread is used that periodically purges the local_storage of our node, which contains the data previously DHT_PUT by users. It is used to comply with the TimeToLive requirement, which each DHT_PUT contains and was specified. The use of this extra thread was a convenience-driven decision, as the local_storage is rarely accessed for most work like routing and session management. Using a mutex that guarantees resource exclusivity during access times to local_storage suffices and does not create any noticeable delays.

Routing, Routing Table, Kademlia:

Next, we are generating a RoutingTable object, which will handle all our needs related to our K_Buckets and node handeling. This will also randomly generate our node ID. More on that later, same as epoll, which we set up following that.

After the RoutingTable, if a peer was passed, we try to connect to its network. To do that, we are sending a "find_node" RPC to the given peer with our own network. The peer will then look in its own network to find peers close to our node ID and return them to us with the Node struct, made up of IP, Port and node ID of the peer. We will receive these node IDs with an epoll_wait call. Now, to expand our network further than up to 20 nodes which we got from the peer, we are then doing a so called "bucket refresh": For each of our buckets we look for random keys in the range of each bucket. We do this until we don't add any more nodes than before. In a very large network, this could mean that we filled all of our buckets, but for our purposes we usually find the few peers we started, for example with a network of about 30 peers we should get the contact details and node ID of every peer in the network.

After gathering the Node struct of our peers, our main loop starts. We have set up two ports, one for the module API and one for p2p traffic. For both, we create one socket that is listened on with epoll. Every time we receive a new request on either socket, an epoll event is triggered.

When a new node is contacting us, it's on the module API socket or on the P2P socket. To distinguish between them we pass a "ConnectionType" to the function that set the new connection up. The ConnectionType for “accept_new_connection“ is either MODULE_API or P2P.

accept_new_connection accepts and builds up the TCP connection on a new socket, sets up an SSL tunnel and adds a new entry to the most important structure of our networking: the connection_map. More on TLS/SSL later. The map saves a ConnectionInfo object for every socket we create. The ConnectionInfo Object's main purpose is therefore session management. It contains the ConnectionType, client address and port, RPC ID (explained later), TLS data, and two buffers: one for incoming and one for outgoing traffic.

When we now get a new request on that newly registered socket, another EPOLL event is triggered. We then read the incoming traffic and pass it to the try_processing function, which parses the header. We then look whether the traffic came in on a module API socket or a p2p socket (indicated by the ConnectionType in the ConnectionInfo) and check the validity of the header.

Now comes the main part of the networking: For every request type we have a handle and send function. These are sending and receiving nonblockingly, so in a way that enables other traffic to flow even when we have larger requests. This is implemented by never calling read() or write() on sockets directly, even though marked nonblocking.

Epoll-in events flush the kernel-side socket buffer into our session-management's receive-buffer. Handling of the content is postponed to the next Epoll-in event if the request is not fully present in our receive buffer (handle-functions detect this incompleteness of messages). Epoll-out events flushe our session-management's send-buffer into the kernel-side socket buffer. If we can't flush everything, we flush only partially and leave the rest for the next Epoll-out event (the forged message can fully or partially reside in the send-buffer at any time). The flushing operations are the only ones which call read() or write() on the sockets. All our defined functions only directly interact with the session-management buffers, never with the kernel-side socket buffers.

If we are receiving a request on a socket of ConnectionType Module API, we continue with the parse_API_request function, otherwise with the parse_P2P_request function.

For the former, we decide between a PUT request and GET request, for the latter, we have 9 RPC functions: PING, STORE, FIND_NODE, FIND_VALUE, a reply call for each and ERROR, which tells the sender of an RPC if the call failed because of a malformed request (e.g. forgot key with a FIND_NODE request) or an internal server error.

Every RPC sends the normal header (4 bytes total: size and type) followed by the RPC header (66 bytes total: node ID, port, and RPC ID). The RPC ID is randomly generated ID that should prevent address forgery, which is tested for when the call is received. This is a change to the protocol outlined in the midterm report, as we initially forgot to pass the node ID and port (the port because we send requests from a new socket instead of the p2p port). The security implication here is that in production, where malicious nodes could join, we assume that a stranger's peer's nodes are started on a different IP address. //TODO: How in testing

For every RPC we have 4 functions (coming one after another in the code):

- forge_DHT_RPC_<method>

Generate an RPC ID and fill in the appropriate fields.

- handle_DHT_RPC_<method>

Read the RPC ID and save it into the connection_map, also read node ID, Port and IP. Test if we already know the peer. If not, we add him to our RoutingTable.

- forge_DHT_RPC_<method>_reply

We look up the previously received RPC ID and fill in the fields for replying.

- handle_DHT_RPC_<method>_reply

We first check whether the RPC ID was the one which was previously sent, then we process the response.

In each of these functions we either use the build_DHT_header, build_RPC_header and write_body functions (in the forge functions) or the read_body (in the handle functions). The read_body and write_body functions move data between the message buffer and the given data structures and are called for every field in the message (key, value etc.). For the message itself, we use a vector of unsigned chars, abstracted away in the "Message"-type. Unsigned, because they display a more appropriate decimal number when debugging and are therefore easier to handle during development.

Now, if we receive a PUT request on the API socket, we call the method "crawl_blocking_and_store". This function, as implied by its name, is blocking, which is why it is called in a new thread. Inside, we call the function "blocking_node_lookup", the most important function for the DHT. This function sends FIND_NODE requests to nodes that are ever closer to a given key/node ID until it can't get any closer to the targeted ID, which is when we return the closest nodes in the network. The number of these is determined by the "replication" factor, provided in the API call. After that, we iterate over the returned nodes and send a store request to each.

The equivalent to this function with a GET request is "crawl_blocking_and_get". The main difference here is that instead of sending a STORE request, we send a FIND_VALUE request to the nodes closest to the key.

To ensure that the nodes we send to and receive from in our routing table are still valid, we call the perform_maintenance function. This is blocking for about 5 seconds, so we don't want to perform it too often. The current default value between these calls is 10 minutes. The function goes through all our nodes and sends a PING RPC. If we don't receive an answer, the node is considered unresponsive and therefore discarded.

Local storage and retrieval of values

Locally, each Node stores the values for which it is responsible in a simple hash map. This map uses the hashes of the values maps it to the value to store. The hashes are in the 256-bit space, as well as the Kademlia-IDs, so we store the values only on the k closest (->XOR-metric) Nodes, where k is the replication factor. The specification also required the Nodes to respect the time to live of a value in the storage that is provided in the TTL field of the PUT request.
For our implementation, this is done via an extra thread that periodically purges the storage and all its outdated values. The thread is not actively waiting, but elegantly waits on a condition variable to not consume resources. The condition variable was necessary as we needed to be able to join the purging thread at any point of time in our application. This is essential, as the user can send an interrupt or kill

signal to the server at any time, so a signal handler was setup to perform trailing clean up logic. As this logic also joins the purging thread, we use a condition variable to notify the purging thread that it's time to return has come.

How does the server work?

After this DHT-related introduction to our structure, we want to introduce you to some design decisions regarding the pure server functionality of our DHT-module.

One small introduction to this was the previously mentioned session management via connection_map and its contained ConnectionInfo objects, each one representing a singular session. Let us start with the big picture: TLS, a custom handshake protocol enhancing security and epoll:

How are sessions initiated?

As our server implements TLS, we need a TLS client and a TLS server role in each connection, even though we treat everyone equally in the Peer-to-Peer network setting. For clarity: A client is typically a user or other server which has a request for a serving server. The client connect()s to the server. The server is the peer in the connection which actively listen()s for new connections and accept()s them on a different port.

For implementing TLS, a slightly custom session instantiation protocol was used. On the start of a server , the server generates an SSL certificate which contains the server IP and the node ID from Kademlia. Both values are constant for the lifetime of the server. Therefore, the certificate can be generated before sessions are accepted, especially since the certificate will be provided on every new session, this keeps computational effort and memory usage negligible.

Maybe a question arises, namely, how does one generate a certificate for oneself? Principles like "root of trust" hinder. After reflecting for quite some time, for our P2P-Network setting, a centralized trusted instance like a Certification Authority (CA) or ticket server were out of question. PKI's as used for e.g. the internet are also hard to realize, as for this to work, certain peers have substantially more power (trust) than others. We wanted to keep the network a network of equals. Web-of-Trust based designs like gathering trust from known peers were also thought of, but hard to realize. Finally, we stuck to X509 certificates but decided on self-signing them. This, of course, does not impact the trustworthiness of the certificates in any way, but with the "Trust On First Use" (TOFU) principle, we can initially assume the credibility of the certificates and their proof of authenticity of their creators. More on this certificate usage below.

To dive deeper into the session instantiation protocol, we consider three distinguishable parts of the protocol: TCP Handshake, Server Provides Certificate to Client, TLS Handshake.

1.1. TCP Handshake:

Typical socket-layer TCP handshake: Server and client create sockets with socket(), the server bind()s and listen()s on dedicated port, the client connect()s to the dedicated port and the server finally accept()s the connection (and answers via a different port than his dedicated listen() port). This entire TCP handshake is also performed fully non-blocking by keeping additional state and waiting on epoll for the 1.5 round-trip to successfully complete after initiating it on the client-side.

1.2. Server Provides Certificate to Client:

After the TCP-Handshake succeeded, the first custom operation is initialized. The server provides its pre-generated, self-signed certificate which contains two major X509 v.3 custom extensions: The IPv6 field and the Kademlia-ID in a field. The port is not contained (the three would make a true identifier of any server), as the self-signing effort for each new port would be infeasible.
Before transmission, the certificate is preceded by 4 bytes of unsigned 32-bit integer in network byte order which indicate the length of the succeeding certificate. A typical length-prefixed send, as certificates could vary arbitrarily in length. During debugging, we realized that almost all our certificates are 1245 bytes long.

As soon as the full certificate is present on the client's side, the client does the heavy lifting part. It takes the certificate and compares it against all previously received certificates (which it bookkeeps for its entire lifetime). If the Kademlia-ID is recognized (contained in another certificate), the client gets suspicious. Theoretically, the provided certificate could be valid if the old, stored certificate’s subject (and issuer, as self-signed) churned out of the network. Then, another peer with a different IP could replace the churned peer under the same Kademlia-ID.

Due to this special case, the client performs an RPC_ping to the owner of the old server certificate. If the old owner is still reachable, the new connection is aborted, as the new certificate could be maliciously generated by choosing a Kademlia-ID instead of randomly generating it, and therefore trying to spoof the identity of a possibly long-term established P2P-network server. If the old owner is unreachable, the old certificate is considered deprecated and new “Trust On First Use” (TOFU) can be established in the freshly received certificate.
Finally, after the certificate was sent by the server and read (& verified) by the client, both parties can start with the typical SSL-Handshake.

1.3. Typical TLS Handshake:

The last part of our custom protocol is a typical TLS handshake. As we use OpenSSL, this is done with SSL_accept() and SSL_connect(). The TLS version our server uses is TLS 1.3, state of the art. We use 2048-bit private key-length and RSA key-generation.
Additionally, as a necessity of our nonblocking sending and receiving, even this TLS handshake is implemented nonblocking via epoll event notification and handle_custom_ssl_protocol() for automaton-based SSLState transition. Once CONNECTED or ACCEPTED, depending on whether client or server succeeded in the handshake, we automatically flush out all future send_buffer contents via nonblocking SSL_write and flush all future incoming data in our recv_buffer via nonblocking SSL_read. For data being written to or from our ConnectionInfo Struct (emplaced in the connection_info map), the use of SSL encryption or not is fully transparent. This implementation design improved development drastically.

In the folder "tests" we have storage and routing unit tests

2. Software Documentation

• Building / running dependencies

1. If you want to run the program on your local machine, the following steps are necessary. For docker-usage, refer to 2.

To build, we use CMAKE, make sure you have a version > 3.13.

Apart from that, we need a compiler for C++, e.g. g++ or clang, that can compile C++23.

Our dependencies are Catch2, Boost, OpenSSL and sdplog, of which only Catch2 is installed automatically with CMAKE. The other two can be installed via your package manager, e.g. on ubuntu with:

```
sudo apt-get install libboost-all-dev
```

```
sudo apt-get install libssl-dev
```

```
sudo apt-get install libspdlog-dev
```

First, we clone the repository:

https://gitlab.lrz.de/netintum/teaching/p2psec_projects_2024/DHT-16

We then need to create a build folder:

```
        mkdir build && cd ./build
```

configure cmake in the build folder:

```
cmake ..
```

and build:

```
cmake --build . --target dht_server
```

This should download catch2, then build the project and generate a "dht_server" executable.

To run, we first need to create a network on at least one node. Let's assume we create a network with IPv4 local addresses for now, one peer on 192.168.0.42 and one peer on 192.168.0.43. The arguments -a -m and -p are the host address, module API port and P2P port of our new node, respectively. Therefore, to start a new network on a node with IP 192.168.0.42, call in the build folder:

```
        ./dht_server -a 192.168.0.42 -m 7401 -p 7402
```

Now, to join the network on a node with IP 192.168.0.43, we pass the IP and p2p-port 7402 of the just started peer with -A and -P, respectively.

```
        ./dht_server -a 192.168.0.43 -m 7401 -p 7402 -A 192.168.0.42 -P 7402
```

To test the server on your local machine you can create test nodes with:

```
./dht_server -a ::1 -m 7401 -p 7402

./dht_server -a ::1 -m 7403 -p 7404 -A ::1 -P 7402

./dht_server -a ::1 -m 7405 -p 7406 -A ::1 -P 7402

./dht_server -a ::1 -m 7407 -p 7408 -A ::1 -P 7402
```

For further testing, we also created the folder tests/integration. This tests file tests/integration/test_multiple_peers.py that can be used to start and communicate between several nodes.

2. Docker-based testing /starting the DHT module(s):

If you want to use Docker for testing, please view the Dockerfile in our projects root directory. It uses Ubuntu 22.04 as a base image for avoiding the "works-on-my-machine" effect. With it, you can start peers themselves in separate containers. First, view our testing recommendation: Use our python script tests/integration/start_docker_containers.py to instantiate and run predefined tests and investigate results as well as logging. Second, not recommended but easily possible for extra testing effort, build the docker network and run the peer containers yourself by following this guide:

Setup custom docker network for the application...

```
docker network inspect dht_network
```

Is the network up? If not, run...

```
docker network create --subnet=172.20. 0.0/16 dht_network
```

Starting 172.20.0.2:< P2P port> to create a network...

```
docker run
    -p <Expos. ModuleAPI port>:<Expos. ModuleAPI port>
    -p <Expos. P2P port>: <Expos. P2P port>
    -t
    --name dht_swarm_0
    --network dht_network
    --ip 172.20.0.2
    dht_swarm
    /server/build/dht_server -a 172.20.0.2 -m <Expos. ModuleAPI port> -p <Expos. P2P port> -l info
```

Both –p options epose the port of the container to the host machine. The –t option shows you the terminal output of the program inside the container. The --name simply sets a name for the container, --network specifies the name of the docker network that shall be used. Network specifies the network, ip the IP. dht_swarm the image, and the rest is the application execution call with the according arguments. Refer the --help for questions about our executable.

To join a second container to the peer, view this example command:

```
docker run
        -t
        --name dht_swarm_1
        --network dht_network
        --ip 172.20.0.3 dht_swarm /server/build/dht_server -a 172.20.0.3 -m <Module API port> -p
<P2P port>  -A 172.20.0.2 -P <P2P port> -l info
```

The ports of this new peer are not exposed to the host machine. You could do this to send new module API requests to a second peer but be aware of not using the same host-ports when exposing peer ports via the –p <host - port>:<container - port>. For any further reading regarding the use-case, view the start_docker_containers.py script in the tests/integration/ folder. The print mode of this script will print example commands that can directly be copy pasted.

During manual testing, we recommend setting all Module-API ports to 7401, and all P2P-Ports to 7402 and only exposing one or two containers' ports to the host machine, as exposed ports and unexposed are easily mixed and confused. By doing this, the containers will only differ by IP in the docker network, not by ports.

• Known issues:

When TLS is set to SSL_VERIFY_PEER for the client of our TLS connection, sometimes an error occurs. We tried to debug it for hours of hours, this bug even motivated us to radically rewrite the entire server side on the last week before the updated submission deadline and implement detailed logging, mainly to avoid these errors where our deviation from protocol could cause fatal errors during connection setup phase.
The main challenge was writing everything non-blocking, so always waiting for callback, also maintaining excessive state for each session. Regarding the already addressed bug, we currently have only one suspicion which we think we already ruled out:
The server side unintendedly send some user-data, maybe also the certificate which is sent by the server in between the TCP and SSL handshake, which is in turn interpreted as part of the SSL handshake on the client side of our session. This leads the internal SSL library to parse the certificate or the other user data as SSL-handshake related data that is otherwise expected from an SSL_accept() call of the server-side.

We had similar misbehavior of our protocol some time ago and fixed it, but the error back then was "error:0A00010B:SSL routines::wrong version number". This made sense, as the Client-Hello requests the server certificate during the handshake, expects the very first bytes of the handshake-certificate to be the version number of TLS, but instead receives user-data. Currently, the client side error is "error:0A000086:SSL routines::certificate verify failed", indicating that only the certificate validation fails. The server error is "error:0A000418:SSL routines::tlsv1 alert unknown ca ", indicating that the certification authority is unknown, even though we have the identical certificate contained in our trust-store. We wrote methods to dump the current certificate store of our client and also to dump the initially generated certificate of our server. They match, byte for byte, formatted identically. During debug-inspections, the transmitted byte count is identical, everything seems perfect. We invested over 10 hours debugging only to fix this singular bug, back then before we had rewritten the server drastically and after we rewrote it. The part that is truly hard to debug is that we cannot inspect the data that relies on the kernel-side buffer of our socket. If we flush it into our receive buffer (fetch the data from the socket into our user-data buffer), we cannot continue with the SSL protocol as this data is needed by SSL_accept/connect to reside on the file-descriptor's kernel-side buffer so that the SSL library can consume the data for internal decision logic. Meanwhile, if it was user data, we could not inspect it as it must reside on the kernel-side buffer of our socket…

Unfortunately, our attempts were without success. Therefore, we needed to set verification of the server certificate to SSL_VERIFY_NONE on the client side for most tests. We still a test to showcase this bug. For this, the SSL_VERIFY_<PEER/NONE> is passed as a command line option which defaults to SSL_VERIFY_NONE. The entire traffic itself is still TLS encrypted; we only cannot ensure that the provided certificates are signed correctly / contained in our application's X509 store and therefore trusted.

The second issue is that our logger normally prints colorful to the console, but when running the docker container and redirecting the console output via –t, the logging looks like "[[32minfo[m]" instead of a grey color that says [info]. There is currently no workaround for this. If the program is executed directly, without docker abstraction, the colorful fonts work.

3. Future Work

In the future, the epoll-event-loop based server can be multithreaded. Achieving this should be realistic, as epoll provides really good threading-support, such that individual events can be worked on by a worker-thread pool. Until now, we have not sided on implementing multithreading as this makes debugging even more difficult and currently, we still have the abovementioned bug of SSL certificate validation which we could not fix yet.

An additional goal would be to detect malicious nodes in the network. During our concurrent lookup, if a singular node repeatedly replies with DHT values which differ from the majority of answers, we can grow suspicion on these nodes and remove them if the suspicion is above a certain threshold. Also, if a peer replies repeatedly with syntax-errored reply data-frames, we could blacklist this node as it might seem to send random traffic just to exhaust bandwidth and computation power. These are future considerations whose implementation would make the nodes, and thereby the network more robust.

Lastly, a nice future work would be to fix the bug regarding the SSL certificate validation, as the SSL certificate generation and verification works in some cases but seems to fail arbitrarily in others. We still provided a test in the python script that enforces one-sided certificate verification of the server-certificate on the client side. Fixing this bug is a future goal for employing authentication during messaging. The general message passing logic of the server (write on user buffer->flush to socket, received on socket buffer-> flush to user buffer) works flawlessly, excluding the already mentioned SSL certificate verification step.

4. Workload distribution — Who did what

In the initial workload separation Marius planned to handle the RoutingTable part, while Jörn wanted to focus on the networking in the dht_server. After the main server implementation with epoll and the epoll event loop was finished by Jörn, which was a lot of effort, Marius focused on implementing K Buckets and the Bucket List that contains the K Buckets. Later, Jörn decided to deal with the implementation of all security related aspects, including creating and self-signing certificates, initiating TLS connections, and sending and receiving secure messages, while Marius implemented the rest of the server, e.g. handle and forge functions, the RPC's and crawl logic. In the final week of the project, Jörn contributed to major rewrites of the server structure while Marius completely rewrote the processing of requests. Both tasks were very time-intensive and can be compared in workload intensity.

5. Effort spent for the project (individually)

Jörn's main contributions, the networking logic and TLS were both an immense effort as there was no clear-cut guide on how this was to be implemented. Apart from that, we were both not very familiar with signing certificates and implementing TLS, so he used the internet for help where needed and spent a lot of time iterating over the functionality. This resulted in about 1700 additional lines of code just for implementing the TLS logic (ssl.cpp file, 800 of the 1700 LoC, for which we want no credit, as it is, as previously mentioned, LLM generated), which took him in total about 65 hours. Then, during the last week before the submission, Jörn dedicated his life into rewriting the server functionality entirely, making structural changes such as propagating errors up instead of handling them deeply nested, simplifying the epoll-structure, adding more state for connection-status bookkeeping, setting up docker, and implementing extensive logging with sdplog for the entire project, setting up an async task queue etc. This easily summed up to 800+ LoC, and amounted to at least 60 hours of work time, averaging more than 8 hours work per day. This is only an estimate, we do not want to exaggerate, please view the commit history. Adding to this came about 15 hours of work on other things.

Meanwhile, Marius' part was also time-consuming but initially more predictable, as it was relatively clear from the specification and the kademlia paper what needed to be implemented. His initial work on the routing and kademlia protocol took about 55 hours. The rewrite of the entire server that was mentioned earlier was done mainly within the last 5 days, where he worked over 12 hours per day on average, sometimes working till the morning and sleeping little, until the features were finished. A lot of the past work was completely redone so the main server logic can be completely single threaded and still operate concurrently. The rewrite was still necessary, as it greatly improved our code.

Considering all of the work over the last ~2 months, Jörn had a time effort on average of about 17.5 hours per week while Marius an effort of about 13 hours per week. Both of us did vastly more work towards the end of the working period, again, view the commit history.