Jörn Fischbach, Marius Bosler

Peer to Peer Systems and Security

Midterm Report for the VoidPhone Project

Team “The Hashies”

This report will start with advances of-/ changes to the initial assumptions of the first report. We are still using the programming language C++. Also, CMake stays our build tool of choice. Catch2 is still used for bottom-up testing, Valgrind for dynamic code analysis.

A substantial change to our implementation was made by not using boost for socket programming anymore. It is solely used for initial program argument parsing. This decision was made after countless hours of trying to grind through boost’s poor documentation and layers of layers of abstraction for concepts that are less complicated in original C-style socket programming. This high degree of abstraction lead to written code that we ourselves only partially understood. Therefore, it was hard to guarantee correctness of the code, i.e., compliance of code lines to intended control flow. Boost really made working with the library feel like a boost in the opposite direction of achieving our milestones.

Jörn Fischbach therefore made a huge effort in purging boost socket programming and rewriting the server side as event-based waiting for sockets to be ready for reading or writing. This was done using epoll, a Linux system event notification facility. This happened a few days before the deadline of this document, so the Catch2 testcases still need to be rewritten and expanded on. This is still work in progress, as the drastic changes in method signatures, overall control flow, and control logic need to be respected.

Next, focus is put on the logical class structure. We use several different classes and enumerations to strive for preemptive bug avoidance by, e.g., using more detailed method return values via C++ Enumerations for more fine-granular control-flow in specific error cases. For the time to live of values, we also provide default values if the requested time to live is out of a sensible range.

The core of our implementation is written in the dht_server.cpp file, with some general configuration dispatched to the dht_server.h file. The two most important variables are following maps:

1. The local_storage, our internal storage that handles key to value mapping with the extension of maintaining the value’s time to live, in other words an expiration date of the value.

2. The connectionMap, our internal session management. It maps sockets (file descriptors) of previously accepted connections to ConnectionInfo structs, which contain buffers for send and receive and information on how a message will be relayed (via k-buckets routing of Kademlia).

The most important control flow, the event-awaiting epoll-based infinite loop of our server, is contained in the main function of dht_server.cpp. The entire rest of the document refers to servers and clients uniformly as peers. They are rarely distinguishable and almost never need explicit treatment. The coarse control flow can be described in the following steps:

1. Our server with the correct port is added into the epoll event queue as a socket. It is only listening for incoming connections.
2. The **infinite loop is** (re-)**run,** the **epoll_wait**(...) call **blocks and waits** for events on the previously enqueued sockets.
3. The **epoll_wait returns** with the number of wake-up events as well as a standard vector of sockets on which the wake-up event took place (e.g., the server accepted a new peer connection, a peer sent a request (put/get) to us, a peer answered our request (success/failure))
4. All **events** of the returned vector **are traversed** by our server and further processed. This includes opening up a new session and therefore inserting the socket as a key and a ConnectionInfo struct as value into the connectionMap. It also includes exhausting all bytes that came in on a socket and transferring them into the corresponding ConnectionInfo struct of the session.
5. After all received bytes on kernel-buffer side are exhausted and transferred to our application, the **bytes are tried to be processed**. As the used TCP-protocol is stream-based, we do not know if the entire application-layer message is present, so we need to perform a deeper inspection of the *size field of our DHT message headers*. If and only if the size field corresponds to the bytes present in our ever appended receive buffer of our connectionInfo struct can we work on a full message and do the heaviest lifting part of our server-side. **Else, restart at control-flow point 2.**
6. The heavy lifting part: Message parsing, answering, or relaying, depending on our locally stored key-scope.
    a. If the modify- or read- DHT operation can be performed locally, it is processed and answered immediately. The key was in our scope of responsibility.
    b. If it cannot be performed locally, it is relayed via the k-bucket routing of Kademlia. This opens a second session (a relaying session), which is enqueued into the epoll events to listen for the request relay answer. The key was not in our scope of responsibility.
7. At last, after having sent the answer to the original requesting peer (with preceded relaying and event-based answer awaiting or without), the connection (and relay connection) is erased from the connectionMap and the socket(s) are closed and removed from the epoll event queue. **Restart at control-flow point 2.**

This sums up the server activity for accepting and answering peers, performing local operations, and relaying connections if necessary.

Regarding the process architecture, the server currently runs single-threaded, in a single process. This is a design decision that was previously made to facilitate debugging, correct control flow and estimated CPU workload of the server in more work-heavy networks. Depending on our progress in the security measures that all still need to be implemented, multithreading may be considered and tested.

In terms of security measures, we plan to look up values concurrently over different paths to avoid a skewed answer of only one malicious peer. The multiple request sampling process will be implemented as soon as the k-bucket routing works. Additionally, TLS/SSL measures proved to be a hard concept to work with, as the decentralized, unstructured nature of the peer-to-peer network makes key-exchanges for session keys immensely difficult. Considerations about trust-on-first-use (TOFU) policies were also made, but peers' churning and joining makes the network unviable for this approach. Peers could also collectively act as a Certification Authority and sign another peer for trust with threshold cryptography, but the number of peers that are needed to trust a new peer would need to dynamically grow/shrink based on the NSE modules estimated network size, which is hard to realize. A Web of Trust (WoT) approach with mutual identity validation seems to be the most viable option and we will investigate the implementation of this security measure in the future.

For the key length, we decided to use 256 bits, as it provides advantages in computation and storage. First, it is a standard size that fits into 32 bytes evenly and is easy to make calculations with and provides enough range that would even work in large-scale networks. Apart from that the 256 bits aligns with the SHA-256 hash function, which could be convenient later.

As mentioned previously, our peer-to-peer protocol of choice is Kademlia. For messaging, we use the methods outlined in the specification, DHT PUT, DHT GET, DHT SUCCESS and DHT FAILURE. Kademlia also provides methods for protocol communication, namely FIND_NODE, PING, STORE, and FIND_VALUE. It first seemed to us as though we could somehow combine the given methods, like for example DHT PUT with the RPC method STORE, but we later decided to use the additional space reserved for the DHT module in message types to have distinct RPC messages as outlined in the Kademlia specification. Apart from those, we also have message types for the echo a recipient should return in the Kademlia protocol and for general errors and a reply type. In the header of RPC messages, we include an RPC-ID that is randomly generated on send and will be echoed by the recipient, which is an additional security feature that avoids malicious responses.

We therefore currently want to use the following additional message types:

660 DHT RPC PING

The purpose of this is to find out whether a node is still active. With a Routingtable, we must ensure that the nodes we have in our list are not all churned, otherwise we will lose contact with the network. Also, when all nodes are online it is much faster to communicate in the network. The request does not have a body.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

661 DHT RPC STORE

This is the protocol message to store a key, value-pair. It takes a 256-bit key and a value of arbitrary length.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

|| TTL (16b) | Replication (8b) | Reserved(8b) ||

|| Key (256b) ||

|| Value (?b) ||

662 DHT RPC FIND_NODE

This is the most important RPC for routing. It takes a 256-bit Node-ID and the recipient returns triples of IP, TCP-Port and Node-ID of the nodes that are closest to the Node that is looked for.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

|| NodeID (256b) ||

663 DHT RPC FIND_VALUE

This request works the same as FIND_NODE, except that we search for a value. So, when a Node has a value in its storage, it returns the value instead of the aforementioned triple.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

|| Key (256b) ||

670 DHT RPC PING REPLY

This request is to be sent after receiving an RPC PING message.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

671 DHT RPC STORE REPLY

This is the protocol reply for a store request. This only gets sent when the key was in the range.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

|| Key (256b) ||

|| Value (?b) ||

672 DHT RPC FIND_NODE REPLY

This will return the mentioned (up to) 20 triples. This reply will also be used when a FIND_VALUE request was received, but the value was not in range.

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

k times: || IP Address (32b) | TCP Port (16b) | NodeID (256b) ||

673 DHT RPC FIND_VALUE REPLY

This returns the value found under key. This only returns when the actual value was found, not when we

|| size (16b) | type (16b) ||

|| RPC ID (256b) ||

|| Key (256b) ||

|| Value (?b) ||

680 DHT ERROR

The purpose of this message is to signal that some kind of error occurred. This could be either an internal server error or a malformed message.

As for error types, we currently have only the following error codes taken from the HTML Standard:

400 – Bad Request

404 – Not Found

406 – Not acceptable (e.g. when a store is sent to the wrong node)

500 – Server Error

|| size (16b) | type (16b) ||

|| Error_Type(16b) ||

We have now set up the fundamental structure of communicating in and out of the server, so storing and receiving values based on keys. Next, we plan to implement more of the peer-to-peer architecture given by Kademlia, so routing based on k-buckets and Node ID's. Another big part of our future work is then extending our software with more security features. Apart from this, we plan to set up a better CI pipeline and also better testing, so we can simulate running many Nodes at the same time, ideally also on different machines.

About workload distribution, we initially mentioned that we will distribute our work dynamically, which is exactly what we have done. We also met on a weekly basis to discuss what we did and what we plan on doing in the following week.

The distribution of work can be approximately summarized in the following way.

- Setting up build system + testing: Marius
- Setting up the asynchronous server structure with socket programming + request parsing and command line parsing with Boost: Jörn
- Replacing everything in relation to the server with the Linux epoll API as Boost was found to be hugely cumbersome and annoying to work with: Jörn
- Implementing routing with k-buckets: Marius

Concluding, Jörn had a very large effort, as the fundamental server architecture was hard to implement in Boost, given that we both knew next to nothing about the library, and in the end a lot of work had to be done again when reimplementing it with epoll.
Estimated average weekly hours spent on this project: ~7h/week

Meanwhile, Marius spent a more acceptable amount of effort, as the used libraries stayed the same and were much easier to set up and work with and his workload was more manageable.
Estimated average weekly hours spent on this project: ~3h/week