Bachelor Project Proposals 2025

Logic and Semantics & Programming Languages Groups

General information

The projects in this specialization aim at reasoning about *programming* in a scientific manner. The underlying motivation is to increase the reliability of software systems, in particular for parallel and distributed systems. This is achieved by using better programming languages, writing better code, and proving its properties (using logic or algorithms).

The topics are often related (but not exclusively) to the following BSc courses that you know:

- Programming Café (Magnus Madsen)
- Computability and Logic (Jaco van de Pol)
- Programming Languages (Anders Møller)
- Compilers (Aslan Askarov, Amin Timany)

Some projects are more theoretical in nature (logic, semantics of programming languages, correctness proofs, foundations), while other projects are quite practical (programming language design, compiler technology, tool support, analysis algorithms, test strategies, applications, computer security).
The teachers are experts in Logic and Semantics and Programming Languages, and are all committed to dedicated and intensive supervision to their project groups:

- Aslan Askarov (language-based security, compilation)
- Lars Birkedal (program semantics, program verification, concurrency)
- Magnus Madsen (programming languages, compilers, type and effect systems)
- Anders Møller (programming languages, program analysis)
- Andreas Pavlogiannis (algorithms in program verification, concurrency)
- Jean Pichon-Pharabod (concurrency, semantics)
- Jaco van de Pol (model checking, automata, verification of algorithms)
- Bas Spitters (type theory, blockchain, computer aided cryptography)
- Amin Timany (program verification, type theory)

Please contact the supervisor directly, and as soon as possible, when you are interested in one of the projects on the following pages.
For general questions, or in case of doubt, you can always contact pavlogiannis@cs.au.dk. He can also help you to find a supervisor if you have other ideas for a bachelor project subject.

Aslan Askarov (aslan@cs.au.dk)

http://askarov.net

Property-based testing in Troupe

Property-based testing is a software testing methodology in which test inputs are automatically generated from high-level specifications. This approach originates from Haskell's QuickCheck library; by now, many programming languages have comprehensive property-based testing libraries.

In this project, you will design and implement a property-based testing library for Troupe – a research programming language for secure distributed and concurrent programming. A characteristic feature of Troupe is its runtime security monitoring, which allows Troupe programs to execute untrusted code securely. However, security monitoring introduces new failure paths corresponding to potential security violations that are not exhibited in traditional languages. Because of this extensive surface failure, Troupe programs must be well-tested. Property-based testing will significantly boost programmer productivity.

This project will include the following steps:

- learning foundations of property-based testing
- learning foundations of information flow control and how they work in Troupe
- implementing a property-based testing library in Troupe,
- showing how the new library is used by developing a broad suite of specifications for the core Troupe functionality and/or designing a new Troupe library, such as a dictionary or a distributed key/value store
- further extensions to the library based on the experience from the previous step.

Further reading:

- Original QuickCheck paper: https://dl.acm.org/doi/pdf/10.1145/351240.351266
- Troupe website at AU: http://troupe.cs.au.dk

Lars Birdekal (birkedal@cs.au.dk)

Polymorphic Type Inference

Type systems play a fundamental role in programming languages, both in practice and in theory.

A type system for a programming language can be understood as a syntactic discipline for maintaining levels of abstraction. For example, types can be used to enforce that clients of a module cannot get access to the data representation used in the module. A sound type system ensures that there are no runtime errors when a well-typed program is run.

For some programming languages and type systems it is possible to automatically (algorithmically) infer types, thus relieving the programmer of the burden of annotating program phrases with types. A type inference algorithm is thus an algorithm which infers types. A type inference algorithm can be thought of as a kind of static analysis: the type inference algorithm analyses the program statically, and infers which invariants (described by types) the program maintains.

There are many type systems and type inference algorithms. A very successful type system is Milner's polymorphic type system and type inference algorithm for this type system include algorithms based on unification and on constraint-solving.

The aim of this project is to learn about type systems and type inference. Specifically, the initial goal of the project is to learn about Milner's polymorphic type system and inference algorithms based on unification. The core part of the project includes implementing polymorphic type inference. Several extensions are possible, both of more theoretical nature and of more implementation-oriented nature, depending on student interest

Literature

- Tofte, M: ML Type Inference, Algorithm W. Chapter 2 of his PhD thesis titled: Operational Semantics and Polymorphic Type Inference.
- Sestoft, P: Programming Language Concepts, Ch. 6

Magnus Madsen (magnusm@cs.au.dk)

Effect-Oriented Programming

Programming with user-defined effects and handlers is a new and exciting programming technique that allows programmers to define their own control structures, e.g. async/await, exceptions, backtracking, and more. Flix has recently gained support for effect handlers. The goal of this project is to extend the Flix Standard Library with several new control-effects.

The aim of this project is to: (1) explore the design space of algebraic effects, and to (2) design and implement several algebraic effects in the Flix standard library. The work will include reading papers and implementation in a real-world programming language.

Language Server Protocol (LSP)

How does a modern compiler talk to a modern IDE? The answer is the Language Server Protocol developed by Microsoft to provide a common interface between compilers and editors. The Flix compiler implements the LSP. However, the implementation lacks several features and is not portable to other editors (e.g. NeoVim).

The aim of this project is to: (1) explore the LSP specification, and to (2) improve and extend the Flix implementation of the LSP. This includes adding new providers and improving auto-complete. It can also include bringing Flix support to other editors, e.g. NeoVim or Zed. The work will include reading a real-world specification and implementation in a real-world programming language.

Termination Analysis (co-advised with Amin Timany)

A common programming mistake is to write an infinite loop. Unfortunately, most contemporary programming languages, such as C, C++, C#, Java, Kotlin, and Scala, do not help programmers avoid such issues. Termination analysis describes a wide range of techniques that can verify that a program (or part of a program) always terminates. For example, by checking that recursive calls always operate on structurally smaller elements.

The aim of this project is to: (1) explore algorithms for automatic termination checking, and to (2) implement one of them in the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language.

Andreas Pavlogiannis

Blackbox testing of strong database consistency

(co-advised with Lasse Overgaard Møldrup)
Requires competence in: Algorithms, Concurrency, Coding.

Large-scale databases are the backbone of modern digital infrastructure. To handle the increased demand for high-throughput, low-overhead transactions, as well as resilience to network errors, they usually provide weak data consistency guarantees, formally known as isolation levels. Implementing isolation levels is tricky and error prone, and modern databases suffer from isolation errors, meaning they fail to provide the data consistency guarantees their isolation level promises.

To test whether a database suffers from isolation errors, we employ black-box testing: a number of clients connect to the database concurrently, and issue a series of transactions (reading and writing to the database). We then analyze this history to see whether the database responded correctly. *But how can we efficiently check whether a transaction history is consistent with an isolation level?*

This project aims to advance the theory and practice of testing strong database isolation levels, such as snapshot isolation, cursor isolation and serializability. This involves understanding (possibly also refining) the semantics of such isolation levels, proving the complexity of the testing problem on transaction histories (*is it NP-complete? Is it polynomial-time?*), developing new, efficient algorithms, and implementing and measuring their performance in practice. If the project delivers well, there is a possibility to publish a research paper, though this is not a requirement for a successful BSc thesis.

Related literature:

[1] https://dl.acm.org/doi/10.1145/3360591

[2] https://medium.com/nerd-for-tech/understanding-database-isolation-levels-c4ebcd55c6b9

Predicting memory bugs in concurrent programs

(co-advised with Yifan Dong)
Requires competence in: Algorithms, Concurrency, Coding.

Concurrency is hard to get right, as a programmer must have a thorough mental model of the program that accounts for all the complex ways that different threads interact. Concurrency bugs are also difficult to catch and debug for, as we typically do not have control over the scheduler. You run your program once, and it crashes – you run it a second time, and it works correctly! For this reason, concurrency bugs have been coined as “*Heisenbugs*”.

One technique for analyzing concurrent programs is called *predictive analysis*. We first run the program, and monitor its execution, thereby obtaining a concurrent trace, which is typically non-buggy. Then we analyze the trace, and reason about alternative thread interleavings that could have taken place and that would lead to a bug. This way we *predict* bugs, as opposed to waiting for them to appear by themselves.

This project aims to develop new techniques for predicting memory bugs in concurrent programs. Memory bugs arise when memory accesses are improper. For example, Use-After-Free-type of vulnerabilities occur when one thread tries to read from a file that has been closed by another thread. The task will be to extend some basic predictive techniques to this particular problem, prove them correct, implement them and test them.

Related literature:

[1] https://dl.acm.org/doi/10.1145/3180155.3180225

[2] https://doi.org/10.1145/3468264.3468572

Efficient simulation of quantum circuits

(co-advised with Jaco van de Pol and Adam Husted Kjelstrøm)
Requires competence in: Algorithms, Linear Algebra, Coding.

Quantum computers hold huge potential for efficiently solving important problems that classical computers struggle with in fields such as cryptography, drug development, and logistics. As current quantum hardware is limited in its performance, optimizing quantum software is key to pushing the boundaries for what quantum computers can achieve. Stabilizer circuits form the backbone of quantum software, and simulating Stabilizer circuits is a fundamental ingredient for both verification and optimization in the quantum setting.

This project aims to improve upon a new algorithm for simulating Stabilizer circuits. This algorithm is conceptually simpler than existing algorithms and could achieve a better run-time in big-O notation. The project aims to realize this potential by using efficient data structures where applicable, then implementing and testing the algorithm in a performant language like C++ or Rust.

Related literature:

[1] https://www.scottaaronson.com/qclec.pdf (lectures 3 and 4)

Graph reachability with counters

Requires competence in: Algorithms, Coding.

Graph reachability is a standard algorithmic problem: given a graph G=(V,E), where V is a set of nodes and E is a set of edges, determine whether there is a path from some node s to some other node t. It is well known that the problem can be solved in O(n+m) for a given node pair (s,t), while the transitive closure can be computed in O(n^{\omega}) using fast matrix multiplication (\omega is the matrix multiplication exponent).

Counter graphs (or *counter automata*) are types of graphs G=(V,E), where each edge is labeled with a weight -1,0, or 1. Now s can reach t iff there exists a path such that (i) the sum of weights along the path never goes negative, and (ii) the sum of weights is 0 at the end (when reaching t). This is a more complex model than plain graphs, and has applications to computability theory, formal languages, and program analysis.

How fast can we solve counter graph reachability? It was recently shown that the transitive closure of a counter graph can be computed in $O(n^{\omega}\log^2 n)$ time, i.e., suffering only a $\log^2 n$ factor increase compared to plain graphs.

The project will start by reading and understanding the main algorithm for counter graph reachability, and implementing it. We will then test the implementation on various input counter graphs, both random and from the literature of static analysis where they arise naturally. We will then target a parallel implementation that speeds up the computation significantly. Throughout, we will see algorithmic-engineering optimizations. The project also has an open-ended theoretical component: *Can we reduce the complexity down to $O(n^{\omega}\log n)$ or even $O(n^{\omega})$?*

Related literature:

[1] https://dl.acm.org/doi/abs/10.1145/3434315

Jean Pichon

Type-checking exception level integrity in instruction set architectures

(co-advisor Aslan Askarov)

The definition of a modern instruction set architecture (ISA) like Arm or RISCV covers thousands of instructions, and the definition of one instruction can be hundreds of lines of code in a domain-specific language like Sail. The size of these definitions means that they are likely to contain errors. One particularly important property of an ISA definition is the integrity of exception levels (aka "protection rings" or "domains"): a userland program should not be able to affect operating systems private registers, and an operating system should not be able to affect hypervisor private registers. The goal of this project is to explore how to design a type system to identify violations of exception level integrity, and to explore how to implement a typechecker so that it scales to mainstream ISAs.

https://github.com/rems-project/sail

Language-Based Information-Flow Security, Sabelfeld and Myers
https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf

Note: There is scope for other Sail-based projects.

Mechanised properties of image formats

Lossless image formats range from the very simple (for example BMP) to the sophisticated (for example PNG and lossless JPEG).
The goal of this project is to implement image formats (like BMP or PNG) in the form of their encoding and decoding functions, and of showing that these two functions are mutual inverses: decoding and re-encoding yields the same image.

Formally Verified Quite OK Image Format, Kunčar
https://ieeexplore.ieee.org/document/10026566

Mechanised definition of the Postscript language

PostScript is a page description language (like PDF, of which it is the ancestor), a language that describes the appearance of a printed page at a higher level of abstraction than a bitmap would.
Structurally, postscript is a stack language whose semantics is, unusually, an image.
The goal of this project is to mechanise the definition of (a significant subset of) the Postscript language and of its operational semantics, and (or: in the form of) an interpreter for PostScript that rasterises pages.

Postscript Blue Book: “PostScript Language Tutorial & Cookbook” https://archive.org/details/PSBlueBook
Postscript Green Book: “PostScript Language Program Design”
https://archive.org/details/PSGreenBook/page/n7/mode/2up

Safe compilation to LLVM IR and WebAssembly
(co-advisor Bas Spitters)

WebAssembly is a language designed to be the compilation target to run programs in web browsers, but it is also gaining traction for edge computing and blockchains. WebAssembly is interesting, because it is at the

same time a widely used web standard, supported by all major browser vendors, and a small and clean enough language to work with in a proof assistant.
In this area, correctness is imperative. In this project, you will write safe programs related to Wasm.
For instance:
LLVM IR is a high-level assembly language that is used as the internal representation of the optimisation phases of the LLVM suite of compiler tools. Compilers for a variety of languages, including C (clang) and Rust (rustc), are composed of a front-end compiling the language to LLVM IR, and the LLVM optimisers and backends. A large subset of LLVM IR has also been formalised in a proof assistant.
Both Wasm and LLVM can be targeted from functional languages too. Here, lambda-ANF is often used as an intermediate language. Exploring how much could be shared by two such backends would be interesting.
In short, the goal of the project is to (1) explore compilation between mainstream languages, and (2) explore safe compilation, either by using property based testing, refinements or a proof assistant.

Note: The main difference between the two languages is the type of control structure, see below; this project would assume that you are given a structured control overlay.

Two Mechanisations of WebAssembly 1.0, Watt et al.
https://hal.archives-ouvertes.fr/hal-03353748/

Modular, Compositional, and Executable Formal Semantics for LLVM IR, Zakowski et al.
https://www.seas.upenn.edu/~euisuny/paper/vir.pdf

Formal verification of a realistic compiler, Leroy
https://xavierleroy.org/publi/compcert-CACM.pdf

Relooper

WebAssembly is an unusual compilation target, because it only features structured control ("if", "while", etc.), and not unstructured control ("goto"). This means that to compile a (more typical) low-level language with unstructured control to WebAssembly, one needs to reconstruct an equivalent program with structured control. The goal of the project is to (1) explore how such a structured control reconstruction algorithm, like Relooper or Stackifier, works, and (2) explore how to use a proof assistant to prove correctness of algorithms.

https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2

Jaco van de Pol

Modeling and Solving Two-Player Games with Quantified Boolean Formulas

Traditional SAT solvers decide the satisfiability of propositional formulas. Although this is an NP-complete problem, progress in the last decade has shown that practical formulas with millions of clauses over thousands of propositional variables can often be solved. As a result, SAT solvers are a popular backend for all kinds of planning problems. We now consider QBF as an alternative to specify games. QBF are Quantified Boolean Formulas, see [1] for a quick introduction and small examples. By using quantifiers (for-all and exists), we can model the alternating moves taken in two-player games. The price is: solving QBF formulas is PSPACE-complete.

We have encoded classical planning problems (available in a PPDL, Planning Domain Definition Language [2]) into concise QBF formulas [3]. The generated formulas can be solved by any available QBF solver (like CAQE or DepQBF). More recently, we devised a domain specific language BDDL (Board-Game Domain Definition Language) to define two-player board games, like Hex, Connect-Four, Generalized Tic-Tac-Toe, Pursuer-Evader, Breakthrough, etc. We also provided an automated translator to concise QBF formulas. Our experiments show that QBF solvers can solve small and medium game instances [4].

The goal of this project is to extend the specification language BDDL to define other 2-player games. Take for instance chess: This would require to specify different pieces (BDDL currently considers only black and white stones) and it would require that pieces can move any distance over empty fields (e.g., queens, rooks, bishops). But chess is just an example, we envisage a whole library of game definitions in BDDL.

The project has a number of goals, in increasing complexity:

a) Extend the BDDL language and specify more (board) games, for instance chess
b) Develop simple tools to play, solve and visualize any game specified in (extended) BDDL
c) Extend the translation to QBF for the class of extended BDDL games (and use QBF solvers).
d) Experiment with/improve the efficiency of the solver, for instance on chess end games.

[1] https://en.wikipedia.org/wiki/True_quantified_Boolean_formula
[2] https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language
[3] I. Shaik and Jaco van de Pol, Classical Planning as QBF without Grounding. ICAPS 2022
[4] I. Shaik and Jaco van de Pol, Concise QBF Encodings for Games on a Grid. Arxiv 2023

A Protocol for Secure Key Distribution

With Jaco van de Pol (Logic and Semantics) and Diego Aranha (Security)

This problem is defined by one of our industrial partners, in a project on security of Internet-of-Things, using threat modeling and threat analysis.

Problem: key distribution and management on embedded systems is challenging. Most of the devices are resource-constrained and do not have tamper-proof hardware to generate and store keys in a secure manner.

Proposed solution: starting from traditional public-key infrastructures (PKIs), a manufacturer is assumed to hold a key pair able to authenticate long-term keys for all produced devices. These keys, together with the corresponding certificates, are then preloaded in the devices before they are deployed in the field. Using the long-term keys, the devices can run key agreement protocols to establish short-term keys for encrypted communication.

Objectives (depending on the available resources):

(a) Design the protocols and implement the software to bootstrap the proposed solution;
(b) Validate the security properties by simulating simple attack;
(c) Apply modeling and analysis techniques to investigate security (e.g., automata, attack-defense trees)

Applications: the solution could be illustrated within an end-to-end-encrypted machine-to-machine communication framework, where devices are able to authenticate public keys belonging to other nodes and establish TLS connections for secure communication.

Stretch goals: after a prototype of the basic functionality is built, extensions could include periodical key rotation or secure revocation of compromised keys. A follow-up version could also potentially implement a gossiping protocol to avoid the dependence on a centralized authority.

Roles: Industrial partner serves as technical advisor, AU serves as academic advisor.

Scope: 1-3 BSc or MSc students, with scope calibrated with the resources available.

Simplification of Quantum Circuits

With Jaco van de Pol

Quantum Computing [1] promises faster algorithms than traditional computers, but current quantum computers are quite small and noisy [2].

Quantum Algorithms are often specified by Quantum Circuits, which can be quite easily read and manipulated, using IBM's open source tool Qiskit [3].

The goal of this project is:

- To understand the basic working of quantum circuits [1]
- To simplify quantum circuits, for instance by canceling "opposite gates" systematically
- To map quantum circuits to quantum computers, by "swapping" quantum bits to physical neighbors [4]

Resources:

[1] Wikipedia on Quantum Computing, Quantum Circuits, and Quantum Gates
[2] John Preskill, Quantum Computing in the NISQ era and beyond, 2018
[3] IBM Qiskit (tools for simulation and manipulation of quantum circuits, and documentation)
[4] Our paper "Optimal Layout Synthesis for Quantum Circuits as Classical Planning", ICCAD 2023

Bas Spitters

spitters@cs.au.dk

High assurance cryptography (co-advised by Diego Aranha)

Cryptography forms the basis of modern secure communication. However, its implementation often contains bugs. That's why modern browsers and the linux kernel use high assurance cryptography:
one implements cryptography in a language with a precise semantics and
proves that the program meets its specification.

Currently, IETF standards are only human-readable. The hacspec (https://hacspec.org/) language (a safe subset of rust) makes them machine readable. In this project, you will write a reference implementation of a number of key cryptographic primitives, while at the same time specifying the implementation. You will use either semi-automatic or interactive tools to prove that the program satisfies the implementation.

The current post-quantum transition is often a twin transition to **high-assurance** PQ software. See for example the Signal messenger, or iMessage. Post-quantum implementations would be a good project topic.

There are a number of local companies interested in this technology. So, this work will be grounded in practice.

Security by Design using LLMs for code

Large Language models are used more and more for generating code. However, they are known to produce unsafe code. There are a number of suggested solutions, most of them involving formal methods/symbolic AI. E.g. involving a type checker, a testing framework, an automatic/interactive theorem prover, …
In this project, we will explore a combination of these techniques:

- Automatic synthesis of invariants, unit/property based tests, assertions, and other proof structures.
- Code repair
- Automatic translation between languages
- …

The project will focus on highly structured languages such as ocaml, rust, …
The project will consist of a combination of literature exploration, implementation and experimentation.

This is a follow-up of a project together with Alexandra Institutet, who might also be involved in this project

Amin Timany (timany@cs.au.dk)

Relational Reasoning

Relational reasoning about programs is a very powerful concept with many applications in the study of programs and programming languages, e.e., expressing compiler correctness, compiler security, etc. A particularly interesting notion of relation between two programs is contextual equivalence. Two programs are contextually equivalent if, as part of a bigger program, each could be used in place of the other without altering the behavior of the whole program. Apart from expressing properties of compilers and programming languages, one can think of contextual equivalence as the gold standard of comparing programs, e.g., when justifying program refactoring. For instance, we may, as a programmer, wish to replace one implementation of a data structure with another, more efficient implementation. In such a case, we can prove that the new implementation is contextually equivalent to the old implementation, and hence that replacing the old implementation with the new one should not change the behavior of the program. A common approach to proving contextual equivalences is to use the logical relations technique.

The aims of this project are as follows:
- Learn the formal concepts of contextual equivalence and logical relations.
- Prove a few interesting cases of program equivalence.

References

- Part 3 of the book "Advanced Topics in Types and Programming Languages" by Benjamin Pierce
- An introduction to Logical Relations by Lau Skorstengaard (https://cs.au.dk/~birke/papers/AnIntroductionToLogicalRelations.pdf)

Amin Timany (timany@cs.au.dk)

Implementation of an authoritative DNS server

Most data processed by computers are neither human-friendly nor stable, *e.g.*, IP addresses. Domain Name System (DNS) provides a way to tie data used by computers to a *domain name*, which is simple for humans to read and remember. For example, when browsing ‘au.dk’, computers connect to the server whose IP address is 185.45.20.48[1]. DNS provides a mapping between names, and the data used by computers, *e.g.*, IP addresses, what server to deliver emails to, what fallback server to use if another server is down. DNS consists of two main components: **authoritative DNS servers** and **DNS resolvers**. The former provides aforementioned mappings. The latter queries authoritative servers to obtain those mappings.

DNS is a structured distributed system. Names are a list of dot-separated strings, *e.g.*, ‘[au; dk]’, allowing partitioning the mapping in so-called zones. Zones form a tree-like structure. In our example, three zones are in use: ‘.’ (the root-zone), ‘dk.’ and ‘au.dk.’. The tree-structure can be observed, *e.g.*, at https://dnsviz.net/d/au.dk/dnssec/. This tree-structure and decentralization are reflected when making a DNS query. For example, when one wants to obtain the IP address corresponding to ‘au.dk’, they use a DNS resolver that will perform the following steps.

1. Ask where ‘au.dk.’ is located to servers managing the ‘.’ zone. The response includes a list of servers handling the ‘.dk.’ zone. This is known as delegation. That is, the ‘.’ zone delegates the query for ‘.dk.’ to these servers.
2. The resolver picks one of these servers, *e.g.*, ‘b.nic.dk’. This server returns a list of two servers managing the ‘au.dk.’ zone.
3. The resolver picks one of these servers which then provides the resolver with the IP address of ‘au.dk’.

In order to increase trust in DNS, responses to queries can be cryptographically signed by authoritative DNS servers. The protocol for signed DNS responses is called DNSSEC. In order to verify a response one gets from a server, one can check the signature of the response using the key of the zone. To ensure that a zone’s keys are legitimate, parent zones use signed/verified delegations. The blue arrows on the aforementioned Web page https://dnsviz.net/d/au.dk/dnssec/ represent signed delegations. They illustrate how one can trust the zone ‘au.dk’ (provided they trust the key of the root zone ‘.’). Hence, responses of ‘au.dk’ can be verified. However, as of this writing, answers of ‘ku.dk’ cannot be verified, as illustrated on https://dnsviz.net/d/ku.dk/dnssec/.

Finally, in order to make DNS more resilient, all zones are replicated on at least two different name servers, *e.g.*, ‘ns3.au.dk.’ and ‘ns4.au.dk.’ in the case of the ‘au.dk.’ zone. Any zone update on one server must be forwarded to other servers.

The aim of this project is for students to implement an authoritative DNS server in OCaml. The DNS server should serve one or more DNS zones and allow administrators to update zones at runtime. Moreover, the implementation should support DNSSEC and replication. Support for DNSSEC means that every zone served by the server can be signed. Moreover, should there be no data to answer to a query, the server should provide a proof of non-existence (e.g. through NSEC3 responses). Replication can take several forms. We suggest using a leader/follower model, *i.e.*, for each zone, choose one authoritative server as reference. Updates should be performed on this server, and then propagated to its replicas.

[1] Or 10.83.252.23 when connected to the university network.

References:

- Domain names concepts: https://www.rfc-editor.org/rfc/rfc1034.html
- DNSSEC specification https://www.rfc-editor.org/rfc/rfc9364
- Zone transfer-relevant RFCs:
    - https://datatracker.ietf.org/doc/html/rfc5936
    - https://datatracker.ietf.org/doc/html/rfc1995
    - https://datatracker.ietf.org/doc/html/rfc1996

  Note: Incremental zone transfer (the last two RFCs) is optional for this project.

Amin Timany (timany@cs.au.dk)

Implementation of a subset of the QUIC protocol

The Internet Protocol (IP) enables sending datagrams from one machine to a set of other machines on the same IP network. On top of this protocol run the so-called transport protocols, which enable communication between software running on different machines.

The two main transport protocols used today are TCP and UDP. TCP establishes a reliable connection between two end-points. If this connection is broken, it is lost, and needs to be re-established by applications. TCP is used by applications that need reliable communication between end-points. UDP on the other hand allows an application to send datagrams to another application on the same IP network. Unlike TCP, UDP does not provide any guarantee of delivery: data can be lost, delivered out-of-order, or duplicated. Thus, UDP is mostly used by applications that do not need reliable communication, *e.g.*, DNS servers.

QUIC is proposed as a lightweight alternative to TCP. Similarly to TCP, QUIC provides reliable communication between two applications. However, unlike TCP, QUIC is more resilient to network issues. That is, the connection can survive even if the network is temporarily unavailable. Moreover, QUIC provides more features than TCP alone. It notably encrypts data between clients and servers using TLS (Transport-Layer Security protocol), and enables multiplexing, *i.e.*, several streams of data can be sent and received through a single connection.

The core idea of QUIC is to implement reliable communication channels on top of UDP. The aim of this project is to write a QUIC implementation in OCaml. QUIC is an extensive protocol and provides numerous features. Most of them do not need to be implemented in this project. The aim is to implement a library with at least the minimal features enough to support writing simple QUIC servers and clients.

References

- QUIC-related RFCs:
  - https://datatracker.ietf.org/doc/html/rfc8999
  - https://datatracker.ietf.org/doc/html/rfc9000
  - https://datatracker.ietf.org/doc/html/rfc9001
- UDP RFC:
  - https://datatracker.ietf.org/doc/html/rfc768