Report Semesterproject
TypeDB-Client

Surname, First name:	SCHNEIDER, Fabian
Matriclenumber:	
E-Mail:	faebl.taylor@pm.me
Date:	July 9, 2021
Version:	DRAFT
Subject:	
Copyright:	Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0)

Institut für Computertechnik
Institute of Computer Technology
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Contents

1 Project Description

Aim of the project was to write an official basic working client for the graph database TypeDB [7] (version >2.0.0). This has been a personal project for quite some time but ultimately failed due to gRPC issues in Haskell (see section 4 and [2]).
The reason for there not being a Haskell client for version >2.0.0 is the refactoring of the protocol from a REST based to a gRPC based protocol. As gRPC interaction in Haskell is a pain to this day a successor client was never developed.
However, Awake Security [5] apparently uses gRPC in Haskell and provides an open sourced fix to their already on Hackage existing (yet broken) grpc-haskell package. Having learned to use nix and NixOS [3] it seemed feasible to utilize the library to implement a working client.

1.1 Project size

The project was a lot of work, yet work I do not regret at all. Reflecting this is this documentation by being quite extensive. I used the opportunity to get down the necessary background for other developers to join the project. I hope you bear with me...

2 Architecture

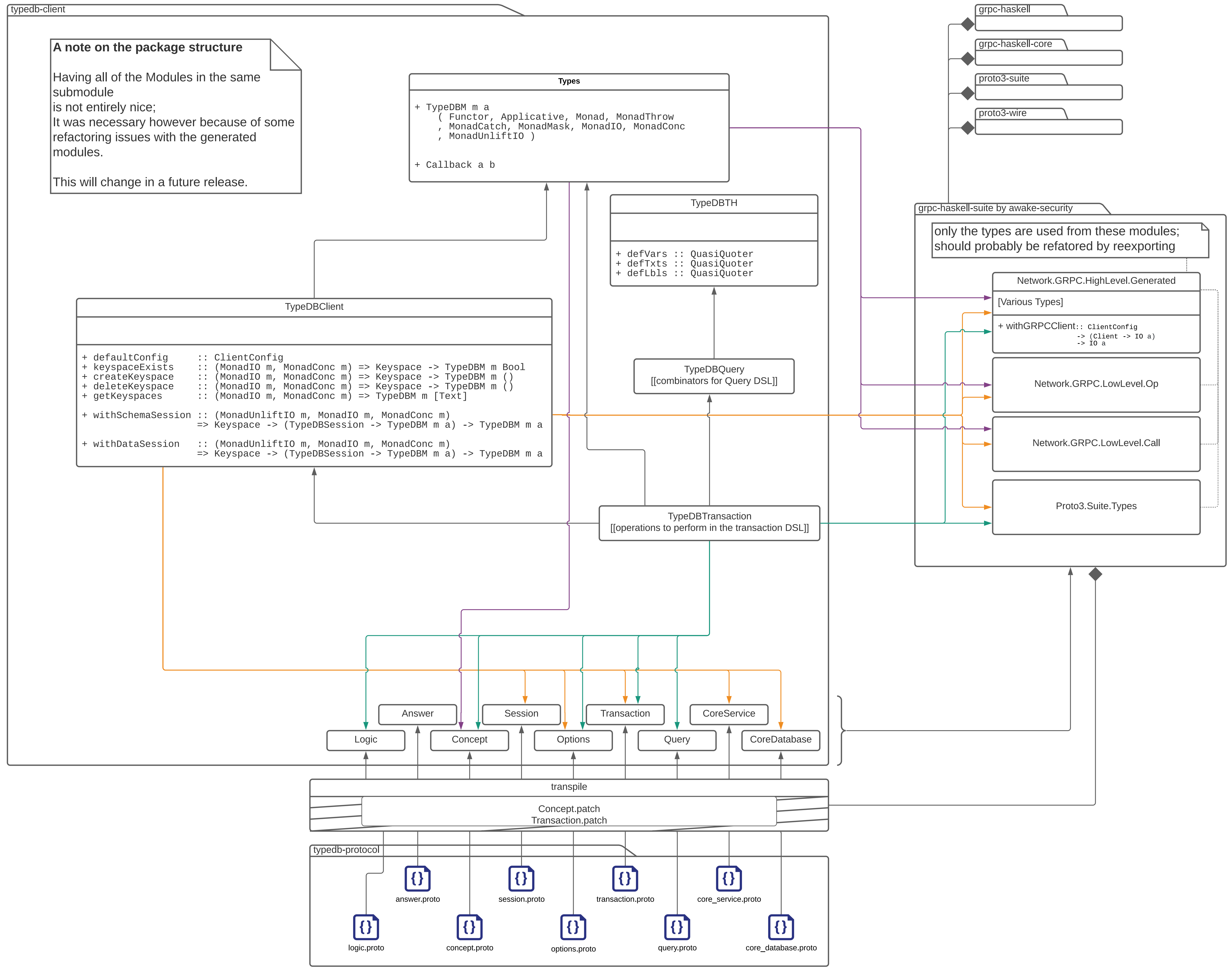
The developed database clients' software architecture is described by the diagrams in Figures 1 and 2.

A note on diagram semantics:

The diagrams are UML diagrams slightly modified for the functional paradigm and Haskell. Packages correspond to modules or indiviual libraries depending on whether they contain packages themselves. Type Signatures have been modified in a haskelly way to compensate for more complex types.
There is also an additional syntax element "}" grouping a visual cluster of modules. An import arrow from this element to another one signifies an import from all of the elements in the visual cluster to the imported element.
"[[...]]" specifies the conents of a module more closely without going into implementation details.

The components of the sequence diagram need not be execution threads. "main" and "TypeDB Server" are actual programs where the lifelines correspond to threads. All other component life threads are function calls, where the component is simply the library/module in which the function resides.

The diagrams are by no means exhaustive but show only the most relevant details to provide a complete picture. Different colors were used to clear up import paths.
The transpile process at the bottom of figure 1 symbolizes the dependency between the code and the provided protobuffer files of the database gRPC specification and the changes that had to be applied to get them working.
The functions designated in the messages of figure 2 are as close to the original names of the functions as possible. The arguments do not match exactly but set the focus on the most important parts.



A note on the package structure

Having all of the Modules in the same submodule
is not entirely nice;
It was necessary however because of some refactoring issues with the generated modules.

This will change in a future release.

```
Types

+ TypeDBM m a
    ( Functor, Applicative, Monad, MonadThrow
    , MonadCatch, MonadMask, MonadIO, MonadConc
    , MonadUnliftIO )

+ Callback a b
```

```
TypeDBTH

+ defVars :: QuasiQuoter
+ defTxts :: QuasiQuoter
+ defLbls :: QuasiQuoter
```

```
TypeDBClient

+ defaultConfig     :: ClientConfig
+ keyspaceExists    :: (MonadIO m, MonadConc m) => Keyspace -> TypeDBM m Bool
+ createKeyspace    :: (MonadIO m, MonadConc m) => Keyspace -> TypeDBM m ()
+ deleteKeyspace    :: (MonadIO m, MonadConc m) => Keyspace -> TypeDBM m ()
+ getKeyspaces      :: (MonadIO m, MonadConc m) => TypeDBM m [Text]

+ withSchemaSession :: (MonadUnliftIO m, MonadIO m, MonadConc m)
                    => Keyspace -> (TypeDBSession -> TypeDBM m a) -> TypeDBM m a

+ withDataSession   :: (MonadUnliftIO m, MonadIO m, MonadConc m)
                    => Keyspace -> (TypeDBSession -> TypeDBM m a) -> TypeDBM m a
```

only the types are used from these modules; should probably be refatored by reexporting

```
Network.GRPC.HighLevel.Generated
[Various Types]
+ withGRPCClient:: ClientConfig
                -> (Client -> IO a)
                -> IO a
```

Figure 1: Module Diagram typedb-client-haskell

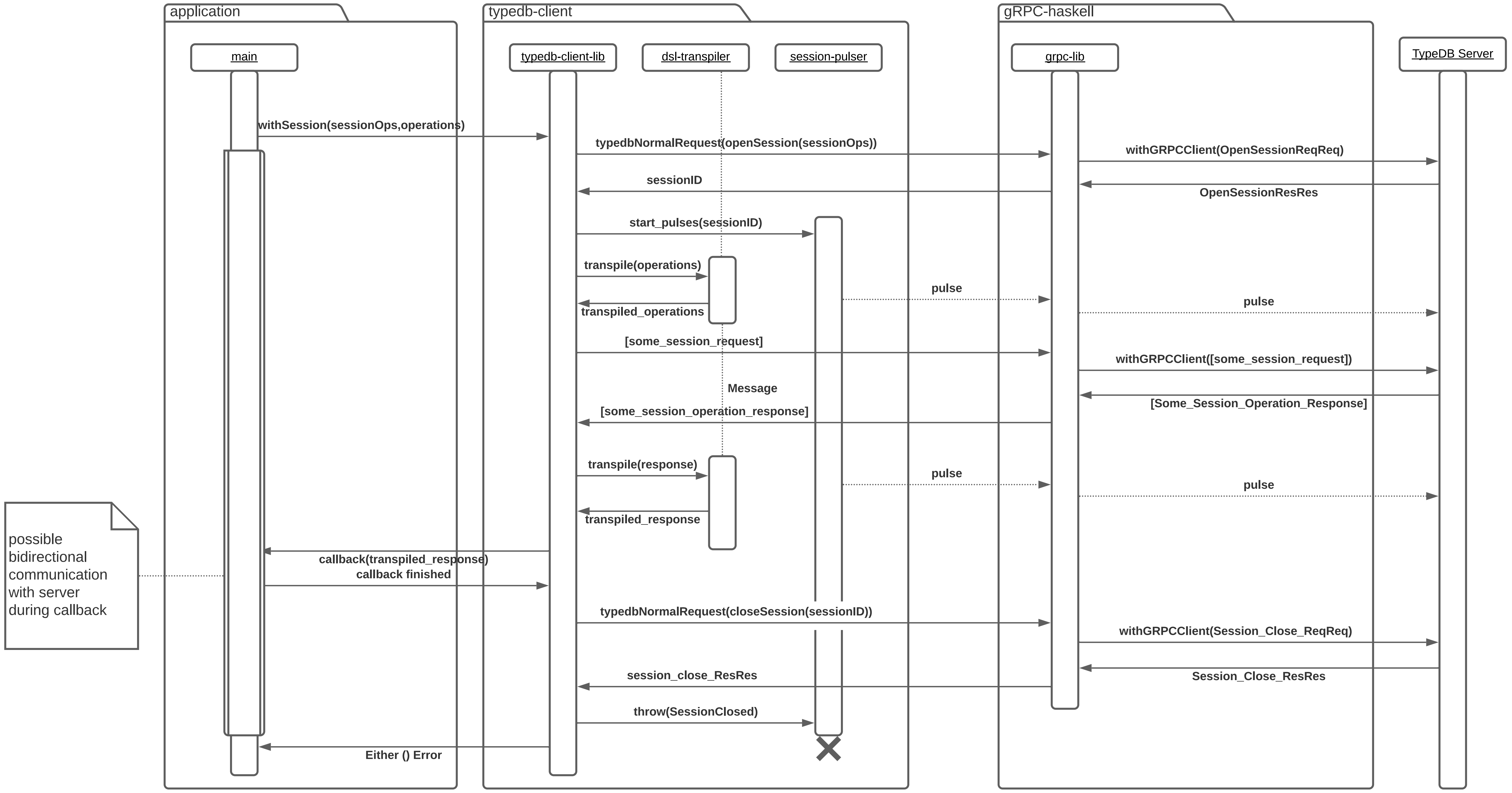

application
typedb-client
gRPC-haskell
main
typedb-client-lib
dsl-transpiler
session-pulser
grpc-lib
TypeDB Server
withSession(sessionOps,operations)
typedbNormalRequest(openSession(sessionOps))
withGRPCClient(OpenSessionReqReq)
sessionID
OpenSessionResRes
start_pulses(sessionID)
transpile(operations)
pulse
transpiled_operations
pulse
[some_session_request]
withGRPCClient([some_session_request])
Message
[Some_Session_Operation_Response]
[some_session_operation_response]
transpile(response)
pulse
pulse
transpiled_response
possible bidirectional communication with server during callback
callback(transpiled_response)
callback finished
typedbNormalRequest(closeSession(sessionID))
withGRPCClient(Session_Close_ReqReq)
session_close_ResRes
Session_Close_ResRes
throw(SessionClosed)
Either () Error

Figure 2: Sequence diagram for a standard client<->typedb interaction

As diagram 1 suggests the package structure of typedb–client is not the most efficient. More specifically the compiled proto files should be in their own submodule TypeDB.DB.Types and the rest of the modules should be in TypeDB omitting the name. Additionally, as indicated some refactoring is in order because the imports of the gGRPC types is quite inefficient and intransparent this way.

3 Build System

The build system for the library and accompanying example program is a mixture of nix and cabal. The explanation for this forced design decision is that the grpc–haskell library uses nix to build and patch the existing package from hackage (see [6]). Nix is therefore used to fetch and build the library and open up a shell with the library in scope. The rest of the build is done in the standard way using cabal.

3.1 Building

To build the library install the nix package manager [3] and then run the following command:

```
> nix-shell --command "cabal update && cabal build"
> # or to get into a dev environment and build:
> nix-shell
nix-shell> cabal update && cabal build
nix-shell> #or even
nix-shell> cabal repl
```

The method of the build process is by no means self explanatory, so here is a short descprition following loosly release.nix:

1. A specific (pinned) commit of the grpc library is fetched from github.
2. the build plan for typedb–client–haskell is computed by looking at typedb.nix. This plan requires a grpc–haskell library.
3. the normal nix package set (where all the packages (including the haskell packages) reside is "overlayed" by an overlay coming from the grpc-haskell nix build plan.
   This results in the (patched) grpc-haskell library being put into scope.
4. then the grpc–haskell library build plan is asked to build the typedb–haskell–client library with this specific overlay.
5. shell.nix specifies that we only want to get the environment where the build is actually possible.
6. using nix–shell results in us entering this environment.
   Now cabal finds all the necessary dependencies already in scope and can build the library.

There would also be a way to use nix–build to build without using cabal, however, this is not yet integrated with cabal correctly up to now and still work in progress (see section 4).

4 War Stories

Rounding off this document is ”off screen material”: Stones hit along the way and the grenades thrown towards them to wipe them off the face of the earth.

API changes
Development started for **Grakn** protocol version ~1.0.6 back when Vaticle was still GraknLabs [4]. The protocol was easily understandable and smallish. A month into development, the protocol changed to protocol 2.0.0-alpha2 which as the versioning suggests are breaking changes. The protocol got very very big and was changed during a huge rewrite of the entire database to support the typesystem of the DB better.
Solution: a complete new start to account for the architectural changes.

Using gRPC
More specifically: building and using the library. I have been using nix and NixOS since more than a year now, but the build plan of the library was extremely complex when I started. Initially the plan also didn't support building an external application.
Solution: change the build plan to compensate for this. The build plan was later changed by the implementors of the lib to allow for it which made my own build script way easier once I re-integrated it.

More using of gRPC
The library comes with an executable to transpile .proto files to Haskell modules exhibiting enough type level machinery to represent the .proto files. These files are extremely big and complex and using them with the protocol specification of course produced non-compiling modules.
Solution: Rename some of the internals of these modules to remove the name clashes and non-found definitions. This resulted in a patch script later on to do these changes automatically to update the protocol implementation easier with more and more protocol changes.

More API changes
A third through the project the company refactored the newly written DB and their company name to TypeDB and Vaticle respectively [4]. Of course it was impossible to write an official client not conforming to the naming.
Solution: refactor the complete library and build plan.

Implementing a DB-Monad
Interaction with the DB should be as easy and prototypical as possible. So a Monad was in order to conform to the sequential way of interacting with the DB and the haskell way of doing things. Using freer-simple solved the problem quite fast, however, there is **a lot of streaming and concurrency** when interacting with this specific database. To account for that I had to implement MonadConc for the database monad. However the type (while being extremely simple) did not lend itself to automatic deriving as the example of MonadConc suggested.
Solution: Spending hours to research what goes into writing/deriving MonadConc and understanding various deriving strategies later I was able to derive the instance in a slightly more annoying way.

Implementing the Query-DSL
As the Query DSL should be as closely to the actual language as possible, a lot of OverloadedStrings

and OverloadedLists were necessary. However, this still resulted in code that looked rather unnatural.
Solution: implementing helper functions to automatically generate variable names, strings and co using TH to take the burden off the developer when writing loads of queries.

Implementing the Transaction-DSL
This involved actually working with the huge generated modules to construct the correct types for the corresponding DSL commands. This was mainly jumping from definition to definition in the various files to trace down what every possible command expects value wise. Automating it would have been more effort than writing it out.
Solution: Sitting for hours on end tracing types, converting Text.Internal and en-/decoding ByteString.Internal to the corresponding normal types of the DSL because of course the library used the lowlevel interna.

Testing the Query-DSL
This one is definately on me. While implementing the DSL I started out writing example queries in the DSL to see if it actually looked how I wanted it to look. Implementing them in some module and adding a comment to their original TypeQL-Query. By the time I wanted all of it to be tested there was a huge module not suitable to be checked with a unit testing library.
Solution: Instead of changing all of it, I automated some minor changes to provide the queries as strings instead of comments and wrote my own QuickCheck.TH like module to automatically test all queries using naming conventions.

Understanding the protocol
The protocol uses many IDs for different messages. In the beginning I got the impression that the server generates SessionIDs, but TransactionIDs should be generated by the client (probably for some referencing and/or performance reason). As it turned out, there was a huge bug in the program because these IDs were actually the same. However, their naming scheme in the protocol was a bit inconsistent at times causing the client to throw exceptions in the server.
Solution: tracking down the bug involved loads of discussions with the implementers of the DB and re-tracing where I made that mistake in (at that time) around 3000 lines of code.

Getting the DB to work on my system
Using NixOS is boon and bane at the same time. The bane part being that the database was installable at the time of starting the program. Portable installation worked under certain ugly circumstances but that was inconvenient.
Solution: Writing my first NixOS package to provide a way to properly installing the DB on NixOS [1]. However, in the process I found two bugs in the implementation of the configuration handling of TypeDB and have to wait until they are resolved to commit the package to the index.

List of Figures

References

[1] Fabian Schneider et al. *Adding TypeDB #125350*. URL: `https://github.com/NixOS/nixpkgs/pull/125350`. (accessed: 30.06.2021).

[2] Fabian Schneider et al. *oneof handling #255*. URL: `https://github.com/higherkindness/mu-haskell/issues/255`. (accessed: 30.06.2021).

[3] NixOS contributors. *NixOS*. URL: `https://nixos.org/`. (accessed: 30.06.2021).

[4] Haikal Pribadi. *Introducing TypeDB, TypeQL and Vaticle*. URL: `https://forum.vaticle.com/t/introducing-typedb-typeql-and-vaticle/2418`. (accessed: 30.06.2021).

[5] Awake Security. *Awake - the NDR security division of ARISTA*. URL: `https://awakesecurity.com/`. (accessed: 30.06.2021).

[6] Awake Security. *awakesecurity/gRPC-haskell*. URL: `https://github.com/awakesecurity/gRPC-haskell`. (accessed: 30.06.2021).

[7] Vaticle. *TypeDB by Vaticle*. URL: `https://vaticle.com/`. (accessed: 30.06.2021).