Individual Report

Marius Gavrilescu

My role on the team was to mediate between the back end and the front end, to design and implement a way for these components to interact.

To this end I have designed the APIs used by the components. For the back end I provided a C header file. This does not tie the implementer to a particular programming language, as most languages have C bindings (Kaashif chose Python). For the front end I proposed a list of procedures available to the client through a `RPC` protocol (I chose `JSON-RPC 2.0` on top of `WebSocket` for its simplicity and ease of use).

After the design phase I started to implement the server-side networking code. After looking into popular `WebSocket` implementations, I settled on `libwebsockets`, a lightweight yet full-featured C library. I wrote my solution as a plugin for `libwebsockets`'s webserver (`lwsws`). This plugin took incoming messages and decoded them as `JSON-RPC 2.0`. Once decoded, the call is interpreted and validated (Does this method exist? Does it have the correct number of parameters? Do the parameters have the correct types/shapes?). Then a back end function is called, and its result is encoded into `JSON-RPC 2.0` and sent to the client. Besides error checking, this code also provides some amount of translation between the two APIs (error codes are given an error message, missing optional parameters get default values, and data formats are sometimes different and need translation).

The above code also implements `JSON-RPC 2.0` on top of the popular `Jansson` library. Another library that is used is `STB`, which provides one data structure needed in the networking code.

I have also written the front end networking code, a JavaScript module that sets up a connection to the server and contains a list of JavaScript functions that make remote procedure calls. Here I used the `JRPC` library to encode and decode messages to/from `JSON-RPC 2.0`.

This design has the advantages of loose coupling and separation of concerns. For the former, the back end and the front end never need to interact directly, so it is easy to change the API used by one of them without affecting the other. It also allows us to deal with complexity at networking level: for example we can allow multiple front end protocol versions that are all translated to calls to the same back end functions. For the latter, most of the error checking is done at this level, so the back end need not concern itself with badly formed input (only with semantically invalid input).

Other than this, I pushed for the licensing of our code as free software. I suggested we use the AGPL license, as our code (especially the back end) is accessed through a network and this license would provide the most copyleft protection. After the proposal was approved, I helped make the project comply with the AGPL.