Porting SimpleX Chat to mobile

Background and motivation

We have code that "works", the aim is to keep platform differences in the core minimal and get the apps to market faster.

SimpleX platform design

See overview for overall platform design and objectives, it is worth reading the introduction. The diagram copied from this doc:

```
User's Computer Internet Third-Party Server ------------------ |
---------------------- | ------------------------- | | SimpleX
Chat | | | | +----------------+ | | | Chat App | | | |
+----------------+ | | | SimpleX Agent | | | +----------------+
-------------- TLS ---------------- +----------------+ | SimpleX
Client | ------ SimpleX Messaging Protocol ------> | SimpleX
Server | +----------------+ ----------------------------------
+----------------+ | |
```

- SimpleX Servers only pass messages, we don't need to touch that for the app
- SimpleX clients talk to the servers, we won't use them directly
- SimpleX agent is used from chat, we won't use it directly from the app
- Chat app will expose API to the app to communicate with everything, including DB and network.

Important application modules

Modules of simplexmq package used from simplex-chat:

- a functional API in Agent.hs to send messages and commands
- TBQueue to receive messages and notifications (specifically, subQ field of AgentClient record in Agent/Client.hs)
- types from Agent/Protocol.hs).

This package has its own sqlite database file - as v1 was not backwards compatible migrations are restarted - where it stores all encryption and signing keys, shared secrets, servers and queue addresses - effectively it completely abstracts the network away from chat application, providing an API to manage logical duplex connections.

Simplex-chat library is what we will use from the app:

- command type ChatCommand in Chat.hs that UI can send to it
- UI sends these commands via TBQueue that `inputSubscriber` reads in forever loop and sends to `processChatCommand`. There is a hack that

`inputSubscriber` not only reads commands but also shows them in the view, depending on the commands.
- collection of view functions in Chat/View.hs to reflect all events in view.

This package also creates its own database file where it stores references to agent connections managed by the agent, and how they map to contacts, groups, and file transmissions.

App design options and questions

Sending chat commands from UI and receiving them in Haskell

Possible options:

- function (exported via FFI) that receives strings from UI and decodes them into ChatCommand type, then sending this command to `processChatCommand`. This option requires a single function in C header file, but also requires encoding in UI and decoding in Haskell.
- multiple functions exported via FFI each sending different command to `processChatCommand`. This option requires multiple functions in header file and multiple exports from Haskell.

Overall, the second option seems a bit simpler and cleaner, if we agree to go this route we will refactor `processChatCommand` to expose its parts that process different commands as independent functions.

On another hand, it might be easier to grow chat API if this is passed via a single function and serialized as strings (e.g. as JSON, to have it more universal) - it would also might give us an API for a possible future chat server that works with thin, UI-only clients.

In both cases, we should split `processChatCommand` (or the functions it calls) into a separate module, so it does not have code that is not used from the app.

Proposal

Use option 2 to send commands from UI to chat, encoding/decoding commands as strings with a tag in the beginning (TBC binary, text or JSON based - encoding will have to be replicated in UI land; both encoding and decoding is needed in Haskell land to refactor terminal chat to use this layer as well, so we have a standard API for all implementations).

This function would have this type:

```haskell
sendRequest :: CString -> IO CString
```

to allow instant responses.

One more idea. This function could be made to match REST semantics that would simplify making chat into a REST chat server api:

```haskell sendRequest :: CString -> CString -> CString -> CString -> IO CString sendRequest verb path qs body = pure ""```

Sending messages and notifications from Haskell to UI

Firstly, we have to refactor the existing code so that all functions in View.hs are passed to `processChatCommand` (or the functions for each command, if we go with this approach) as a single record containing all view functions.

The current code from View.hs will not be used in the mobile app, it is terminal specific; we will create a separate connector to the UI that has the same functions in a record - these functions communicate to the UI.

Again, there are two similar options how this communication can happen:

- UIs would export multiple functions however each platform allows it, as C exports, and they would be all imported in Haskell. This option feels definitely worse, as it would have to be maintained in both iOS and Android separately for exports, and in Haskell for imports, resulting in lots of boilerplate.
- UIs would export one function that receives strings (e.g. JSON encoded) with the messages and notifications, there will be one function in Haskell to send these JSON. All required view functions in Haskell land would simply send different strings into the same function.

In this case the second option seems definitely easier, as even with simple terminal UI there are more view events than chat commands (although, given different mobile UI paradigms some of these events may not be needed, but some additional events are likely to be addedd, that would be doing nothing for terminal app).

Proposal

Encode messages and notifications as JSON, but instead of exporting the function from UI (which would have to be done differently from different platforms), have Haskell export function `receiveMessage` that would be blocking until the next notification or message is available. UI would handle it in a simple loop, on a separate thread:

```haskell -- CString is serialized JSON (ToJSON serialized datatype from haskell) receiveMessage :: IO CString ()```

To convert between Haskell and C interface:

```haskell type CJSON = CString

toCJSON ToJSON a => a -> CJSON toCJSON = ...

-- Haskell interface send :: ToJSON a => String -> IO a recv :: ToJSON a => IO a

-- C interface csend :: CString -> IO CJSON crecv :: IO CJSON ```
```

Accessing chat database from the UI

Unlike terminal UI that does not provide any capabilities to access chat history, mobile UI needs to have access to it.

Two options how it can be done:

- UI accesses database directly via its own database library. The upside of this approach is that it keeps Haskel core smaller. The downside is that sqlite is relatively bad with concurrent access. In Haskell code we allowed some concurrency initially, having the pool limited to few concurrent connection, but later we removed concurrency (by limiting pool size to 1), as otherwise it required retrying to get transaction locks with difficult to set retry time limits, and leading to deadlocks in some cases. Also mobile sqlite seems to be compiled with concurrency disabled, so we would have to ship app with our own sqlite (which we might have to do anyway, for the sake of full text search support). We could use some shared semaphore in Haskell to obtain database lock, but it adds extra complexity...
- UI accesses database via Haskell functions. The upside of this is that there would be no issues with concurrency, and chat schema would be "owned" by Haskell core, but it requires either a separate serializable protocol for database access or multiple exported functions (same two options as before).

However bad the second option is, it seems slightly better as at least we would not have to duplicate sql quiries in iOS and Android. But this is the trade-off I am least certain of...

Proposal

Use the same `sendRequest` function to access database.

Additional idea: as these calls should never mutate chat database, they should only query the state, and as these functions will not be needed for terminal UI, I think we could export it as a separate function and have all necessary queries/functions in a separate module, e.g.:

```
haskell -- params and result are JSON encoded chatQuery :: CString -> IO CString chatQuery params = pure ""
```

On another hand, if we go with REST-like `sendRequest` then it definitely should be the only function to access chat and database state.

UI database

UI needs to have its own storage to store information about user settings in the app and, possibly, which chat profiles the user has (each would have its own chat/agent databases).

Chat database initialization

Currently it is done in an ad hoc way, during the application start (`getCreateActiveUser` function), we could either expose this function to accept database name or just check on the start and initialize database with the default name in case it is not present.

Multiple profiles in the app

All user profiles are stored in the same database. The current schema allows multiple profiles, but the current UI does not. We do not need to do it in the app MVP.

Notifications

We don't need it in the first version - it is out of scope of releasable MVP - but we need to think a bit ahead how it will be done so it doesn't invalidate the design we settle on.

There is no reliable background execution, so the only way to receive messages when the app is off is via notifications. We have added notification subscriptions to the low protocol layer so that Haskell core would receive function call when notification arrives to the native part and receive and process messages and communicate back to the local part that would show a local notification on the device:

```
Push notification -> Native -> Haskell ... process ... -> Native
-> Local notification
```

Notifications are the main reason why we will need to store multiple profiles in the same database file - when notification arrives we do not know which profile it is for, it only has server address and queue ID, and if different profiles were in different databases we would either had to have a single table mapping queues to profiles or lookup multiple databases - both options seem worse than a single database with multiple profiles.

For the rest we would just use the same approaches we would use for UI/ Haskell communications - probably a separate functions to receive notifications to Haskell, and the same events to be sent back.