TinySSB - Foreground-Implementation

Lars Schneider

January 31, 2025

1 Communication Model

In order to be able to split the the application into the foreground service-related topcis from the frontend, we need both of those parties to be in communication. This communication can either be happening from `WebAppInterface` to the `ForegroundService` (f.e. using `publish_public_content()`), or from frontend-requests which have a return value such as `IS_GEO_LOCATION_ENABLED` or lastly the possibility where the foreground service starts off by sending a request to the `MainActivity`, which will take place in case of `new_event()`. Note that all communication between `MainActivity` and `WebAppInterface` is happening with the help of intents. We differentiate between `ApplicationNotification` and `ForegroundNotification` depending on where the signal originated from (see yellow marked in 1). For the registration of the intents we have defined a broadcastreceiver called `mainActivityReceiver` inside the `BleForegroundService` as well as the `foregroundServiceReceiver` inside `MainActivity` whose job it is to listen for incoming intents originating from the foreground service itself (recall: `ForegroundNotificationType`). Moreover did we define there the `resultReceiver` which listens for results, which are being sent as a response from the foreground service. This intent will only occur, when the `WebAppInterface` calls `sendMessageToForegroundserviceWithOutput`. The `resultReceiver` works together with the companion object `CallbackRegistry`. Every intent, which expects a return value, will be assigned a callback identifier. This identifier is the unix-timestamp. This means, a call from the frontend expecting a return value will actually trigger two intents, which the frontend will not even witness. Moreover, pretty much every function in `WebAppInterface` is now using the keyword `suspend`

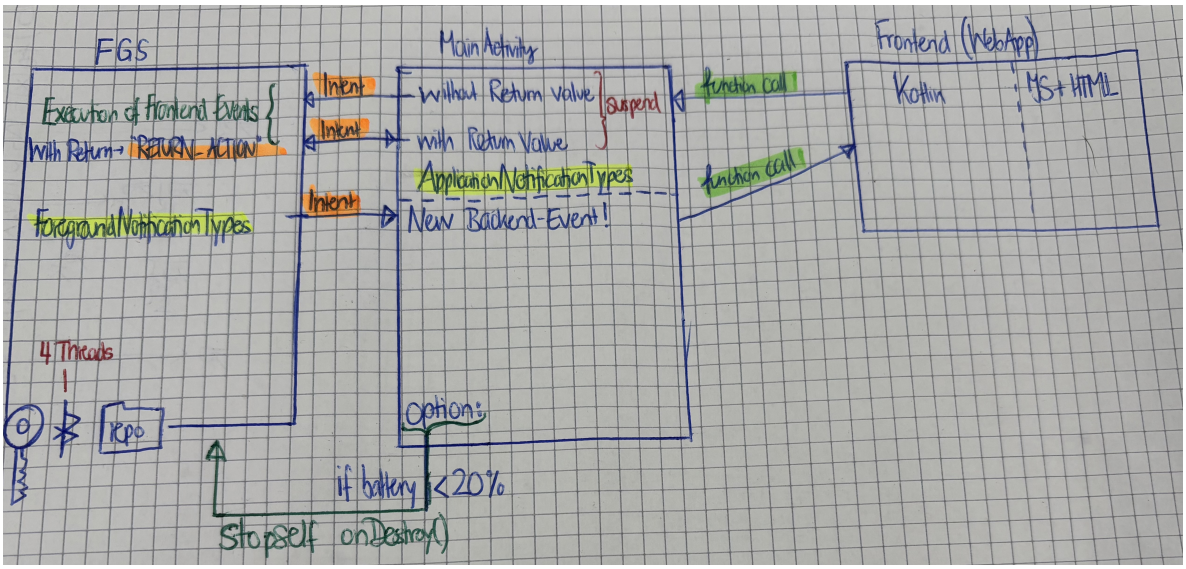

Figure 1: Communication Model composed of the foreground service, MainActivity and the frontend.

because we have calls to the foreground service, which are not immediate (i.e. similar to HTTP-Requests). There are therefore some occasions where we are required to await a response in order to continue with the execution of the function. You can only start a `suspend` function inside another `suspend` function or inside a coroutine. Because `onFrontendRequest()` cannot be turned into a `suspend` method, we use corotines if needed. Because foreground services can be very resource intensive I have added the possibility to turn it off once you exit the app. However, this would only work, if the device's battery is lower than 20%, as we do not want to limit replication.

2 Current Status

- Full-Delivery-Chain: when all components are online (Frontend, Backend, Foregroundservice, BLE, ForegroundService, Backend, Frontend)
- Remote ForegroundService correctly executes synchronization even if remote backend is not online
- Pending: Installing protocol s.t. remote backend can catch up with the remote foreground when it comes online again
- Pending: ForegroundService to provide Repo-Statistics to Backend
- WebAppInterface: Method `sendTinyEventToFrontend` uses reference to Repo Object directly (which is managed by the foregroundService) — This access should be fixed by a proper protocol design between ForegroundService and WebAppInterface
- Optimization: Currently only one timeout for sending out Beacons, regardless of communication channel (LORA to UDP). Setting these constants NODE_ROUND_LEN and GOSET_ROUND_LEN to lower values resulted in immediate better BLE synchronization experience (1000ms, 3000ms). In future these constants should be IO-channel specific.