Project Overview

The goal of Project Ellesmere is to streamline identity verification in New York airports by installing physical terminals equipped with the ability to read and verify mDL data extracted from an EWA device. The data transfer is executed via an ISO 1803-5 compliant NFC/Wifi-Aware handoff sequence. The terminals will be connected to a CAT machine via usb.

Repository Exposition

Dongle-side App (device\)

The dongle-side application (apk) is located under the master branch under the device folder. Principally, it makes use of the android.hardware.usb library provided by Google. It is triggered once the hikey is in accessory mode and enters an infinite loop in which it constantly tries to read in data from the host (using FileInputStream) and, once it succeeds in doing so, sends a USBResponse proto object (using FileOutputStream). This process is repeated for an indefinite number of iterations.

Documentation (docs\)

This file can be found at this location in addition to notes.txt, random notes I jotted down pertaining to high-level design and implementation during the early parts of this summer.

Driver Function (host\)

The function of the CAT-side usb driver that I've written is to enable effective USB half-duplex data transfer between the dongle and the CAT machine (which is the host). I implemented the Android Accessory Protocol using the low-level API of usb4java (which may have performance advantages), a java wrapped C library, for host-side operations. I was unable to verify whether or not the driver facilitated a successful data exchange between my PC and the Hikey board as my testing was cut short by a hapless corruption of crucial Hikey system files which disabled adb over usb capabilities.

I have written a few drivers that differ in style and minor implementation details.

Under the oop\ directory lie the drivers written in an object-oriented style. There are two subdirectories, async\ and sync\. The former contains a driver which transfers data in an asynchronous manner (which allows for a degree of simultaneity without multithreading) and leverages Aurelian’s protoc-generated class for data serialization. The latter holds a driver which transfers random test data (no protobuf) in a synchronous manner.

Under the procedural\ directory is a driver written in procedural style with identical functionality to its OOP counterparts.

Dependencies (libs\)

All of the project dependencies are located under the libs\ directory in .jar format.

Data Serialization (serialization\protobuf)

In terms of data serialization, I mainly used Google protocol buffers at Aurelian’s bidding. The production\ folder simply holds the UsbProtocol protoc-generated class and .proto file assembled by Aurelian. Under the testing folder lie three subdirectories: googlex, mvngooglex\autoprotoc, and objproto). The googlex\ contains tutorial material from Google’s documentation sites and is irrelevant to the project. Under mvngooglex\autoprotoc is a Maven build automating the compilation of any arbitrary amount of .proto files. Early on, I wrote a fairly naïve but functional program that transforms java objects directly into protobuf generated classes, saving the time of manually writing a .proto file. The program only works for simple cases but can handle nested objects (objects with objects (with objects… and so on) as data fields) complete with proper indentation. This program is located under the master branch at: serialization\protobuf\testing\objproto and is named ProtoConverter.java. The accompanying files are output, and templates.