KOTLIN AND BLUETOOTH

or Better Living With Coroutines

Eventually

Neal Maloney
neal.maloney@shrinkray.com.au

WHO AM I

- Mostly a Java Developer for about 20 years
- Android Developer for around 8 years (since Cupcake)
- Developing BLE code for around 4 years (since late JellyBean/ KitKat)
- Commercial projects have included:
  - AlertShirt and FanShirt football jerseys (vibrating clothing)
  - NADI X Yoga Pants early development (clothing with positional sensors)
  - SleepSense (sleep tracking and controls for bed)
  - Cochlear Nucleus Smart App

THE INTERNET OF THINGS

- Loosely defined as the world of connected computing devices embedded in every day objects.
- BLE is one of the primary ways of communicating with these objects because of it's power efficiency.
- Spans from insecure, simple devices like keychain trackers and smartphones to higher security, more complicated devices like fitness trackers and Cochlear's Nucleus 7 sound processor.

BLUETOOTH LE

- Fairly simple protocol at the application developer level.
- Supports relatively light level of encryption that is crackable with off the shelf tools.
- Robust connection and data transmission.
- Relatively simple data structure once connected. A connected client can read from, write to and receive notifications of changes to characteristics defined by the device.

ANDROID BLUETOOTH LE

- Developers need to work around and with:
  - No queue. The BluetoothGatt object will happily let you make multiple reads/writes at the same time. What happens is somewhat undefined and usually results in reads/writes getting dropped by the stack. So you need to implement some form of queuing.
  - Asynchronous callbacks to reads/writes, notifications and connection issues… but not always on defined threads. Some phones will call the ‘asynchronous’ callback from the initiating thread before returning from the function that made call responsible for it.
  - Need to not block the callback functions.
  - Bugs… e.g. Edge case allowed in the BLE specification for manufacturer data where Android was effectively dropping data. Resulted in having to parse the raw scan data ourselves.

ROBUST SOLUTIONS INVOLVE PROTOCOLS ON TOP OF BLE

- If you need a better level of security and are dealing with a complicated device, you can't just rely on the basic BLE data layout and encryption.
- The reasons for this are:
  - The encryption and negotiation at the BLE level isn't secure enough. You usually want some level of encryption on top of the transport layer.
  - Having many, many characteristics increases service discovery time and will impact connection time (and battery life for really low power solutions)
  - MTU is limited, so transferring large amounts of data (e.g. for a fitness tracker) will require splitting and re-assembly of data.

SOLUTIONS

- A rich API defined in XML and at least partially implemented with generated code on both device and in the app.
- Network service added into the mix to provide authentication for the user.
- Key exchange protocols with the device that are transparent to the app.
- A system for obtaining authentication to use the device that coordinates the authenticated user, the server, the device and some action that provides proof of possession of the device.
- A seperate BLE driver module that implements an as simple as possible interface to the device in a functional way using Rx.

BLE DRIVER

- The good parts:
  - As layered/modular as possible for easier testing.
  - Good use of generated code.
  - Use of RxJava2 throughout.
    - Good simple interface for the app to hide complexity.
- The bad parts:
  - It’s written in Java. Lots and lots of unnecessary and inelegant code.
  - Use of RxJava2 throughout. Perfectly fine in some places but a nightmare to read/maintain in others.

A TINY CHUNK OF THE CRYPTO CODE IN JAVA

```
void processVerifier(EmpheralPublicKey empheralPublicKey,
                     SecurityProcessVerifierParamVal securityProcessVerifierParamVal) {
    // First annoyance, can't overload += to perform mDisposables.add instead...
    mDisposables.add(new PerformOperation(client).execute(securityProcessVerifierParamVal)
            .delaySubscription(5000, TimeUnit.MILLISECONDS) // Device will not yet be ready.
            .subscribe(new Consumer<DeviceToServerVerifierVal>() {
                @Override
                public void accept(@NonNull DeviceToServerVerifierVal deviceToServerVerifierVal) throws Exception {
                    // Success, call "processVerifier" on the server with the details from before and the SP verifier.
                    ProcessVerifierRequest processVerifierRequest = new ProcessVerifierRequest (
                            ephemeralPublicKey,
                            deviceIdentifier,
                            toBase64(checkNotNull(deviceToServerVerifierVal.getSignature()).get())
                    );
                    mDisposables.add(mServer.getEndpoint().postDhkeProcessVerifier(processVerifierRequest)
                            .subscribe(new Consumer<ProcessVerifierResponse>() {
                                @Override
                                public void accept(@NonNull ProcessVerifierResponse ProcessVerifierResponse) throws Exception {
                                    if (processVerifierResponse.isOk()) {
                                        // Clear the device control token if we have one. It will be invalid.
                                        mServer.clearDeviceControlToken(deviceIdentifier).blockingAwait();
                                        mForceDhke = false;
                                        mServer.markSuccessfulDhke(deviceIdentifier).blockingAwait(); // Not great.
                                        checkDeviceControlToken();
                                    } else {
                                        // If the server rejects the exchange, the error is likely unrecoverable.
                                        mStatusSubject.onNext(Status.CRYPTO_ERROR);
                                    }
                                }
                            }, new Consumer<Throwable>() {
                                @Override
                                public void accept(@NonNull Throwable throwable) throws Exception {
                                    disconnectForRetry("could not send verifier to remote server", throwable);
                                }
                            }));
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    disconnectForRetry("could not send verifier to the SP", throwable);
                }
            }));
}
```

BASIC KOTLIN IMPROVEMENTS

- Lambdas instead of anonymous classes (mostly normally hidden by IDE though).
- plusAssign operator CompositeDisposable
- toBase64() extension function for ByteArray
- Type inference.

A SIMPLE CONVERSION TO KOTLIN

```
fun processVerifier(empheralPublicKey: EmpheralPublicKey,
                    securityProcessVerifierParamVal: SecurityProcessVerifierParamVal) {
    disposables += PerformOperation(client)
            .execute(securityProcessVerifierParamVal)
            .delaySubscription(5000, TimeUnit.MILLISECONDS) // Device will not yet be ready.
            .subscribe({ deviceToServerVerifierVal ->
                disposables += server.getEndpoint().postDhkeProcessVerifier(ProcessVerifierRequest(
                        ephemeralPublicKey, deviceIdentifier,
                        deviceToServerVerifierVal.getSignature().get()?.toBase64()))
                    .subscribe({ processVerifierResponse ->
                        if (processVerifierResponse.isOk()) {
                            // Clear the device control token if we have one. It will be invalid.
                            mServer.clearDeviceControlToken(deviceIdentifier).blockingAwait()
                            mForceDhke = false
                            mServer.markSuccessfulDhke(deviceIdentifier).blockingAwait() // Not great.
                            checkDeviceControlToken()
                        } else {
                            // If the server rejects the exchange, the error is likely unrecoverable.
                            mStatusSubject.onNext(Status.CRYPTO_ERROR)
                        }
                    }, { _ ->
                        disconnectForRetry("could not send verifier to remote server", throwable)
                    })
            }, { _ ->
                disconnectForRetry("could not send verifier to the SP", throwable)
            })
}
```

NOT A GREAT DEAL BETTER

- Lambdas instead of anonymous classes (mostly normally hidden by IDE though).
- plusAssign operator CompositeDisposable
- toBase64() extension function for ByteArray

WITH COROUTINES

```
suspend fun processVerifier(empheralPublicKey: EmpheralPublicKey,
                            securityProcessVerifierParamVal: SecurityProcessVerifierParamVal) {
    delay(5000)
    try {
        val deviceToServerVerifierVal = ProcessVerifierOperation(client).execute(securityProcessVerifierParamVal).await()
        try {
            val processVerifierRequest = ProcessVerifierRequest( ephemeralPublicKey, deviceIdentifier,
                                                 deviceToServerVerifierVal.getSignature().get()?.toBase64())
            if ( server.getEndpoint().postDhkeProcessVerifier(processVerifierRequest).await().isOk() ) {
                server.clearDeviceControlToken(deviceIdentifier).await()
                forceDhke = false
                server..markSuccessfulDhke(deviceIdentifier).await()
                checkDeviceControlToken()
            } else {
                statusSubject.onNext(Status.CRYPTO_ERROR)
            }
        } catch ( t: Throwable ) {
            disconnectForRetry("could not send verifier to remote server", t)
        }
    } catch ( t: Throwable ) {
        disconnectForRetry("could not send verifier to the SP", t)
    }
}
```

A DRIVER WRITTEN FROM SCRATCH IN KOTLIN

- For extending by specific device implementations.
- Built with extensibility and the core of a Kotlin only driver for the main project in mind.
- Built using Rx to control the threading.
- Still investigating channels to replace a BlockingQueue used to poll for callbacks received asynchronously from the BLE driver.
- Has a the concept of channels, for reading, writing, writing with an immediate read and notification.

CODE LINKS

https://github.com/tali3san/kotlin-bluetooth