Implementation Comparisons of a P2P Messaging Application

Bobby Dhillon

March 16, 2020

1 Introduction

The motivation for this project was to develop a peer-to-peer (P2P) messaging application on an iOS and Android device (1), with the focus of comparing the implementations of the application in different languages. The core features of the application are: finding and resolving an instance on the same wireless network, and sending and receiving messages between the two instances. The mobile application was implemented in three different languages: Kotlin (2) and Java (3) for Android, and Swift (4) for iOS. The project utilized the multi-platform Hype Mesh Networking SDK, which exposed the interface of the Hype Open Protocol (HOP) (5) life cycle in the three languages. Each mobile application followed the same design surrounding the Hype SDK interface. The languages will be introduced, and then implementations will be compared using code snippets from the project, and by lines of code (LOC) versus time to write.

2 Languages

2.1 Java

Java is a simple, object-oriented, distributed interpreted, robust, secure, architecture neutral, aportable, high-performance, multithreaded, and dynamic language (6). Java is a compiled programming language, but rather than compile straight to executable machine code, it compiles to an intermediate binary form called Java Virtual Machine (JVM) byte code (3). The byte code is then compiled and/or interpreted to run the program. Some of Java's goals are to be simple, in the sense that it would have a small learning curve and eliminate unsafe features such as direct memory allocation, and using references instead of pointers that are present in C and C++ (6). As such, the language designers of Java aimed to include only features with already known semantics, and to provide a small simple language (7).

2.2 Kotlin

Kotlin can be thought of as the successor of Java. Kotlin is an inferred, statically typed programming language that targets the JVM, Android, JavaScript and Native. The first official 1.0 release was in February 2016 (8). Kotlin has both object-oriented and functional constructs, and can be used in both object-oriented and functional programming styles, or mix of both and has first-class support for features such as higher-order functions, function types and lambdas (8). Kotlin is also very concise, with rough estimates indicating approximately a 40 percent cut in the number of lines of code as compared to Java (8). It is also more type-safe, for example it has support for non-nullable types that makes applications less prone to Null Pointer Exceptions (NPE) as compared to Java (8).

2.3 Swift

Swift is a powerful and intuitive programming language for macOS, iOS, watchOS, tvOS and beyond (4). Swift is Object-oriented, static, strongly-typed, inferred language with many modern language features such as: variables are always initialized before use, array indices are checked for out-of-bounds errors, integers are checked for overflow, optionals ensure that nil values are handled explicitly, memory is managed automatically, and error handling allows controlled recovery from unexpected failures (9). Swift can be thought of as the successor of Objective-C, and is a rather young language that was released in 2014.

```java
public class Dialog {
    private String TAG = Dialog.class.getName();
    private AlertDialog dialog;

    public void show(Context context, DialogInterface.OnClickListener listener, String title,
                     String body, boolean cancelable) {
        if (dialog != null) {
            dialog.dismiss();
        }
        Log.i(TAG, "show");
        AlertDialog.Builder alert = new AlertDialog.Builder(context)
                .setTitle(title)
                .setMessage(body)
                .setIcon(android.R.drawable.ic_dialog_alert)
                .setCancelable(cancelable);
        if (listener != null) {
            alert.setPositiveButton(android.R.string.yes, listener)
                    .setNegativeButton(android.R.string.no, null);
        }
        if (cancelable) {
            alert.setPositiveButton(android.R.string.yes, null);
        }
        dialog = alert.create();
        dialog.show();
    }
}
```

(a) Dialog.java

```kotlin
class Dialog {
    private var TAG = Dialog::class.simpleName
    private var dialog : AlertDialog? = null

    fun show(context: Context, listener: DialogInterface.OnClickListener?, title: String, body: String,
        canceleable: Boolean) {
        dialog?.let {
            dialog!!.dismiss()
        }
        Log.i(TAG, "show")
        val alert = AlertDialog.Builder(context)
                .setTitle(title)
                .setMessage(body)
                .setIcon(android.R.drawable.ic_dialog_alert)
                .setCancelable(canceleable)
        listener?.let {
            alert.setPositiveButton(android.R.string.yes, listener)
                .setNegativeButton(android.R.string.no, null)
        }
        if (canceleable) {
            alert.setPositiveButton(android.R.string.yes, null)
        }
        dialog = alert.create()
        dialog!!.show()
    }
}
```

(b) Dialog.kt

```swift
class Dialog: NSObject {
    var dialog: UIAlertController?

    func show(title: String, message: String, handler: ((UIAlertAction) -> Void)?) -> UIAlertController {
        if dialog != nil {
            dialog?.dismiss(animated: true, completion: nil)
        }
        NSLog("show")
        dialog = UIAlertController(title: title, message: message, preferredStyle: .alert)
        if handler != nil {
            dialog?.addAction(UIAlertAction(title: "OK", style: .default, handler: handler))
            dialog?.addAction(UIAlertAction(title: "CANCEL", style: .cancel, handler: nil))
        }
        return dialog!
    }
}
```

(c) Dialog.swift

Figure 1: The Dialog class

3 Implementation Comparison

The comparisons are derived from my project, as to maintain context within the P2P application. Code snippets were chosen in order to highlight some similarities and differences in syntax, semantics, and language features.

3.1 Dialog

The Dialog class is implemented as Dialog.kt, Dialog.swift, and Dialog.java, respectively. The Dialog class creates and provides access to an instance of an alert dialog, which is used to provide the user information and control over the current state of the application.

In line 3 of Figure 1(a) and Figure 1(b), we can see how Kotlin uses a nullable type for the AlertDialog instance dialog to distinguish between variables that may be null or may not be null, whereas the Java implementation does not. The nullable type is denoted by adding a ? after the data type's property (10). Java does include a class for wrapping a value as one that may or may not be null, called an optional type (11), which I chose not to implement as to avoid complexity. As such such, in line 8 of Figure 1(a) the dialog instance must be manually checked for not null, which is not required by Java. On the other hand, Kotlin allows us to do null safety checks with a safe call (?) in line 6 in Figure 1(b) (12) and enforces a dialog.show() call with the null-assertion operator (!!) in line 23 in Figure 1(b), which converts any value to a non-null type and throws an exception if the value is null (12). This allows the code to be less prone to NPEs, and also makes it easy to tell by just syntax whether a variable may be null.

In line 4 of Figure 1 (c), the handler function passed to the show method is of type $((UIAlertAction) \rightarrow Void)?$, where the ? denotes an optional value, with similar semantics to a nullable type in Kotlin, and so makes it easy to tell by just syntax whether a variable may be nil (13). We can see a similar nullable type named listener in line 5 of Figure 1(b). Both the handler and listener are now known to maybe have a value or not have a value. We can also see that in line 2 of Figure 1(c), that the dialog instance is declared as optional with ? (13), which in turn forces it to be unwrapped with ? (13) in subsequent dialog method calls, in lines 6, 11, 12 and 14 of Figure 1(c). This is the same as line 23 in Figure 1(b). The optional and nullable-type enforce null and nil checks and make it easy to discern just by syntax.

Although Swift and Java contain the optional language feature, it is easier distinguish and use optional types in Swift such as the dialog instance, as compared to Java.

3.2 ChatMessage and Message

```java
public class ChatMessage {
    private Message message;
    private MemberData memberData;
    private boolean belongsToCurrentUser;

    public ChatMessage(Message message, MemberData data, boolean belongsToCurrentUser) {
        this.message = message;
        this.memberData = data;
        this.belongsToCurrentUser = belongsToCurrentUser;
    }

    public Message getMessage() {
        return message;
    }

    public MemberData getMemberData() {
        return memberData;
    }

    public boolean isBelongsToCurrentUser() {
        return belongsToCurrentUser;
    }
}
```

(a) ChatMessage.java

```kotlin
class ChatMessage(val message: Message, val memberData: MemberData, val isBelongsToCurrentUser: Boolean)
```

(b) ChatMessage.kt

```swift
struct Message {
    let member: Member
    let text: String
    let messageId: String
}
```

(c) Message.swift

Figure 2: the ChatMessage class and Message struct

In Java and Kotlin, the ChatMessage class is a wrapper around the MemberData class and used by the MessageAdapter class to display sent and received messages in the respective view. In Swift, it is a struct named Message, since there is no naming conflict with the Message struct in the Swift Hype SDK, HYPMessage (14).

In Figure 2(a) and Figure 2(b), we can see the immediate reduction in code to provide the same functionality in Kotlin as compared to Java. Since Kotlin can auto-generate getters methods for its member variables in class definitions (15), the definition of the class is reduced to a whopping 1 line! Not only is the same meaning retained, it is more immediately understood since there is less to code to read.

When comparing Figure 2(c) to Figure 2(b), it is clear there is a feature that Swift has that Kotlin lacks: structs. Since the message construct is required to bundle three publically accessible variables, a struct is a more simple and straight forward implementation. Since Figure 2(c) is only 5 lines of code, it is still a short and easy read when compared to Figure 2(a). This comparison highlights that some abstractions may warrant the use of features that are not present in the language, in this case structs.

3.3 Resolving a Hype Instance

In Swift, Java, and Kotlin, the Hype SDK exposes methods through its *HypeNetworkObserver* interface (14) that when implemented, allows the developer to add actions for when the method is called in correspondence to the event. In the event of when a new Hype instance is resolved, *onHypeInstanceResolved* is called and will be the code snippets to be compared.

In line 1 of Figure 3(a), we see the definition of resolveHandler which utilizes currying (16), a functional programming concept in which a function is partially applied and returns another function. Currying allows a workaround to match the parameter type of handler in the dialog.show call, which is $((UIAlertAction) \rightarrow Void)?$, and also take a hype instance as an argument. On line 2 of Figure 3(b) we see the usage of higher-order functions, in this case DialogInterface.OnClickListener function type that is instantiated (17), which is the same as in line 2 of Figure 3(a) without having to explicitly pass a hype instance as an argument.

```swift
func resolveHandler(instance: HYPInstance!) -> (_ alertAction:UIAlertAction) -> () {
    return {_ in
        self.resolvedInstance = instance
        NSLog("Hype will communicate with instance %@", instance.appStringIdentifier!)
    }
}

func hypeDidResolve(_ instance: HYPInstance!) {
    NSLog("Hype resolved instance: %@", instance.stringIdentifier!)
    let message = String(format: "Instance found: %@\nDo you wish to communicate?",
        instance.stringIdentifier!)
    DispatchQueue.main.async {
        self.present(self.dialog.show(title: self.RESOLVED_INSTANCE_TITLE, message: message,
                                       handler: self.resolveHandler(instance: instance)),
                                       animated: self.ANIMATED, completion: nil)
    }
}
```

(a) Swift

```kotlin
fun resolveListener(instance: Instance): DialogInterface.OnClickListener {
    var listener : DialogInterface.OnClickListener = DialogInterface.OnClickListener {
        _, _ ->
        Log.d(TAG, "Hype will communicate with instance")
        this.resolvedInstance = instance
    }
    return listener
}

override fun onHypeInstanceResolved(instance: Instance) {
    Log.i(TAG, String.format("Hype resolved instance: %s", instance.stringIdentifier))
    runOnUiThread {
        dialog.show(this@MainActivity, resolveListener(instance), RESOLVED_INSTANCE_TITLE,
                String.format("Instance found: %s\nDo you wish to communicate?", instance.stringIdentifier,
                    false), false)
    }
}
```

(b) Kotlin

```java
private DialogInterface.OnClickListener resolveListener(final Instance instance) {
    DialogInterface.OnClickListener listener = new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialogInterface, int i) {
            resolvedInstance = instance;
        }
    };
    return listener;
}

@Override
public void onHypeInstanceResolved(final Instance instance) {
    Log.i(TAG, String.format("Hype resolved instance: %s", instance.getStringIdentifier()));
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
        dialog.show(MainActivity.this, resolveListener(instance), RESOLVED_INSTANCE_TITLE,
                String.format("Instance found: %s\nDo you wish to communicate?",
                    instance.getStringIdentifier()), false);
        }
    });
}
```

(c) Java

Figure 3: onHypeInstanceResolved in each language

In the case of Swift in Figure 3(a), currying was used since the dialog of type *UIAlertController* does not have predefined listener methods as the alert *DialogInterface* in Figure 3(b) and Figure 3(c). In Kotlin, it was more straightforward to implement with a predefined listener method and was more concise than in Java, since it omitted all the boilerplate.

3.4 Lines of Code versus Time to Write

Language	LOC	Time
Java	401	7
Kotlin	313	5
Swift	268	6.5

Table 1: LOC vs Estimated Time to Write. Time measured in hours.

After development of all the applications was completed, I ran count lines of code (cloc) (18) on each directory containing the source files. Table 1 below summarizes the LOC vs the estimated time to write in order to reach the same level of functionality in each language implementation. Table 1 does not include time to develop UI which included work with XML in Android and iOS. We can see how Java required almost 100 more LOC than Kotlin, and over 100 LOC more than Swift. The results in Table 1 follow directly from the previous comparisons, as Java was shown to be more verbose with bad null safety and so would take longer to write than Swift and Kotlin. Swift and Kotlin had similar sets of features and similarly concise syntax, which is why their respective developments and LOC were relatively low.

4 Conclusion

From the comparisons between code snippets in the P2P implementation use case, we see how structurally similar Kotlin and Swift are, while on the other hand how much more verbose Java is. This lead to a reduction in LOC and development time when implementing the application in Swift and Kotlin, since the syntax was more concise

and so easier to write and read. Although there is no perfect language for the P2P use case, this project compared similarities and differences between the different implementations and clearly favors the newer languages Swift and Kotlin.

References

[1] Al-Areeqi, Waheb Ismail, Mahamod Nordin, Rosdiadee. *Peer-to-peer communication on android-based mobile devices: Middleware and protocols. 2013 5th International Conference on Modeling, Simulation and Applied Optimization* ICMSAO 2013. 1-6. 10.1109/ICMSAO.2013.6552677.

[2] *"Kotlin Programming Language."* Kotlin, kotlinlang.org/.

[3] *"The Java® Language Specification"* Java Programming Language, https://docs.oracle.com/javase/specs/jls/se13/html/inde

[4] Apple Inc. *"Swift."* Apple Developer, developer.apple.com/swift/.

[5] *Hype Open Protocol: Decentralized Connectivity.* Hypelabs, hypelabs.io/documents/hype-open-protocol-whitepaper.pdf.

[6] Alves-Foss, Jim. *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. 1523, January 1999.

[7] Drossopoulou S., Eisenbach S. (1999) *Describing the Semantics of Java and Proving Type Soundness.* In: Alves-Foss J. (eds) Formal Syntax and Semantics of Java. Lecture Notes in Computer Science, vol 1523. Springer, Berlin, Heidelberg

[8] *"FAQ."* Kotlin, kotlinlang.org/docs/reference/faq.html.

[9] Apple Inc. *"Swift.org."* About Swift - The Swift Programming Language (Swift 5.2), docs.swift.org/swift-book/.

[10] *Safe calls(?.) vs Null checks(!!) in Kotlin* https://agrawalsuneet.github.io/blogs/safe-calls-vs-null-checks-in-kotlin/

[11] *Optional (Java Platform SE 8)*, 6 Jan. 2020, docs.oracle.com/javase/8/docs/api/java/util/Optional.html.

[12] *"Null Safety."* Kotlin, kotlinlang.org/docs/reference/null-safety.html.

[13] Abel, Agoi. *"Optionals in Swift."* Medium, Medium, 15 Jan. 2018, medium.com/@agoiabeladeyemi/optionals-in-swift-2b141f12f870.

[14] *"Hype SDK."* Hype Reference, hypelabs.io/api-reference/ios/0.8.3304/docs/index.html.

[15] Praveenruhil, et al. *"Kotlin Setters and Getters."* GeeksforGeeks, 12 June 2019, www.geeksforgeeks.org/kotlin-setters-and-getters/.

[16] jain, Aaina. *"Functional Swift: Curry Function."* Medium, Swift India, 5 July 2018, medium.com/swift-india/functional-swift-curry-function-4d26190139ed.

[17] *"Higher-Order Functions and Lambdas."* Kotlin, kotlinlang.org/docs/reference/lambdas.html.

[18] AlDanial. *"AlDanial/Cloc."* GitHub, 15 Mar. 2020, github.com/AlDanial/cloc.