Mobile Platform Architecture Review: Android, iPhone, Qt

Michael Lettner, Michael Tschernuth, and Rene Mayrhofer

Upper Austria University of Applied Sciences, Research and Development
Softwarepark 11, 4232 Hagenberg, Austria

Abstract. The recent proliferation of mobile platforms makes it difficult for developers to find the most appropriate platform for one's needs or even target multiple ones. A review of key concepts on Android, iPhone and Qt points out important commonalities and differences that will help to better understand the respective platform characteristics. As mobility is an important aspect of such phones, the ability to access mobile-specific functionality is investigated. Implications at every concept visited will then point out things to keep in mind as a developer.

1 Motivation

The last couple of years have seen smartphone sales take off. While this hype mainly started out with the introduction of the first iPhone in 2007, more mobile platforms have been introduced since, and competition is intensifying. Due to this fragmentation, targeting an application for multiple markets, devices or operators has become increasingly difficult. As cross-platform development tools are rare yet, developers have to face a decision on which platforms their native application should be supported first, or ported to later on. Market share, ease of development, popularity, an active developer community, usability and target group are all factors for consideration when favoring one platform over the other.

Developers on the other hand would prefer to implement an application once, and deploy it for many platforms with minimal effort. To enable such a scenario, programmers must know the very differences between the platforms in question. This work will try to compare popular mobile platforms in terms of capabilities and point out the most significant differences in platform concepts (which subsequently make it difficult to enable write-once, deploy-anywhere applications). Such an analysis will help the developer to understand which parts of a system may be more suitable for reuse, and which may not—the conclusion will also try to address these questions. The work will also be beneficial for somebody trying to raise an application's abstraction level to support multiple platforms—one such project, though targeted at low-cost mobile platforms, is introduced in [5].

2 Methodology

The comparison focuses on three platforms: the underlying platform of the iPhone [2], Android [1], and Qt [3,4].

The first one is the one that started the recent hype, the second one is gaining more and more momentum and becoming increasingly popular, while Qt is increasingly used for Symbian development, an operating system (OS) that has been around for a much longer time, and must be taken into account not least due to its status of the system with the most widespread use. Despite the recent announcement by Nokia[1] to use Microsoft's Windows Phone OS on Nokia's devices to create a new global mobile ecosystem, Qt will still play a key role on Nokia's expected 150 million units[2] that will ship with Symbian in the years to come, and is the development environment of choice for Intel's MeeGo platform—a platform not only targeted at mobile phones or netbooks, but intended to be used in TV sets, set-top boxes or In-Vehicle Infotainment systems.

After a basic introduction to all of the beforementioned platforms, their architecture will be analyzed, and commonalities and differences in terms of platform capabilites will be pointed out.

3 Platform Introduction

Android: is an open-source software stack created for mobile phones and other devices that includes an OS, middleware and key applications [1]. It is based on Linux kernel version 2.6 to provide core system services such as security, memory management, process management, network stack, and driver model. The Dalvik virtual machine (VM)—optimized for minimal memory footprint—executes Android applications usually written in Java that were first converted into Dalvik Executable format. From a developer's perspective, it is the Android SDK that provides the tools and APIs required for developing applications.

iPhone: iOS is the mobile OS of Apple's iPhone series. It is based on the same technologies used by Mac OS X, namely the Mach kernel and BSD interfaces, thus making it a UNIX-based system [2]. Applications are primarily developed using Objective-C—a superset of ANSI C extended with syntactical and semantic features (derived from Smalltalk) to support object-oriented programming. Cocoa Touch is the application environment to build applications for iOS and consists of a suite of object-oriented libraries, a runtime, and an integrated development environment (e.g., XCode or Interface Builder). Two core frameworks are essential to application development: the Foundation and the UIKit.

Qt: Opposed to the previously discussed platforms, Qt is not limited to one specific OS. Instead, Qt is a cross-platform application and UI framework, including desktop and mobile target platforms such as Windows, Linux, Symbian or MeeGo. Qt comes with a C++ class library and integrated development tools. Bindings for a wide range of other languages exist, e.g. Java, C# or Ruby.

[1] http://press.nokia.com/2011/02/11/nokia-and-microsoft-announce-plans-for-a-broad-strategic-partnership-to-build-a-new-global-ecosystem/

[2] http://qt.nokia.com/products/qt-for-mobile-platforms/

4 Platform Comparison

Important concepts such as memory management, user interface realization or communication, just to name a few, are provided by all platforms in one way or the other. This section will outline these basic concepts and find differences/commonalities between the platforms in question.

4.1 OS/Platform-specific

Memory Management. Proper memory management is essential to all computer systems, but due to the memory-constrained nature of smartphones it is especially important on mobile platforms. Therefore, it is indispensable to deallocate objects that are no longer needed—which could either be taken care of by the system, or be an important necessity to be carried out by the developer.

Android: Android applications rely on automatic memory management handled by Dalvik's garbage collector (GC), which, as Google states, can sometimes cause performance issues if you are not careful with memory allocations [1]. Each process uses a separate GC instance—thus, the collectors don't interfere with instances of other applications. The type employed by Dalvik is known as tracing GC and utilizes a *mark-and-sweep* approach [6]: In a first step, the collector instance keeps mark bits to indicate that a particular object is "reachable" and therefore should not be garbage collected [7]. In a second phase all objects are collected that are marked as such. The biggest advantage of the mark-and-sweep algorithm is its ability to correctly identify and collect garbage even in the presence of reference cycles [8], whereas the major disadvantage is the fact that program execution must be halted to run the algorithm.

iPhone: Objective-C supports two concepts for memory management: automatic garbage collection and reference counting [2].

- *Garbage collection:* although introduced in Objective-C 2.0, an automatic memory management mechanism is not available on iOS (i.e., it is only supported by Mac OS X version 10.5 and higher). It would allow to pass responsibility for determining the lifetime of objects to an *automatic collector*.
- *Reference counting:* denotes a technique of memory management on iOS, where an object carries a numerical value reflecting the current claims on the object. When this value reaches zero, the object is deallocated [2]. It is the developer's responsibility to determine the lifetime of objects by balancing each call that claims object ownership (object allocation/copying, or a retain message) with a call that removes that claim (i.e., release or autorelease).

Qt: Qt follows an approach that is a certain variation of object ownership [4]: As most classes are derived from QObject, instances of all such classes are placed in a hierarchy of QObject instances. Then, when a parent object is deleted, its children are deleted, too. To mimic automatic memory management, one could allocate the parent on the stack, which would automatically destruct all dynamically allocated child objects when the parent element goes out of scope.

Implications: The platforms differ in the ways they handle memory—the biggest difference being in whether a GC is provided or not. Having one, as on Android, is not an insurance against memory leaks though[3], since references can still be held that prevent the GC from collecting certain objects. If performance is a crucial criterion in an application, the overhead of a GC might adversely affect the runtime behavior. Android's mark-and-sweep algorithm required to suspend normal program execution, disallowing any runtime predictability. Applying design principles[4] is one way to workaround performance issues. In addition, there is the possibility to develop performance-critical portions in native code. On Android, starting with Native Development Kit release r5[5], entire applications (i.e., including the whole application lifecycle) can be developed this way in C++.

When a code generator is used to generate the platform code from an abstract model (as in [5]), for languages not supporting automated garbage collection memory management design patterns [9] can be incorporated into generation rules—e.g., static or pool allocation patterns might be used internally while the implementation complexity is hidden from the programmer.

Communication between Components. This section deals with ways of communication between objects or components inside applications, from a developer's perspective. It is not meant to be understood as communication in the meaning of telecommunication and/or messaging between end users.

Android: To pass information from one screen to another, Android uses *Intents*—an asynchronous way to communicate between components (in Android terms, these are Activities, Services, Broadcast receivers and Content providers). All but the last of the beforementioned components can be activated using asynchronous messages, so-called Intents, which are objects holding the content of the message.

iPhone: On iPhone, there is not a single comparable concept of how to pass information between objects. Instead, the programmer can use the built-in capabilites of Objective C, such as Delegates, Notifications or Target/Action mechanism to pass information along screens.

Qt: Qt utilizes a concept known as *Signals and Slots* to enable communication between components—another asynchronous way of message passing between objects. Qt Widgets have either predefined signals, or can be customized with own signals. On occurrence of a specified event, a certain signal is being emitted. At the receiving end, other objects can define slots (or make use of predefined ones), which are normal C++ methods that match a signal's signature. When signals and slots are connected, code defined in slots is executed on emission of events. A strength of this scheme is that signals can be connected to multiple slots, while each slot can receive signals from multiple source objects. The execution order is not defined if multiple slots are connected to the same signal.

[3] http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html
[4] http://developer.android.com/guide/practices/design/performance.html
[5] http://developer.android.com/sdk/ndk/index.html

Implications: As the mentioned communication patterns are very platform-specific, it is hard to compare them:

Commonalities:

- All three platforms have a truly asynchronous communication system in place where (multiple) events can be bound to (multiple) operations. The internal implementation is substantially different though.
- The mechanisms can be utilized to register for any notifications that report system changes to one's application (e.g., the battery level changes).
- Intents/Notifications/Signals can all be used to trigger lifecycle state changes.

Differences:

- Granularity level
  - Intents are used to pass information between Activities, Services or Broadcast receivers. They operate on a "whole screen" or service.
  - Notifications and Signals can be used to pass information between any kind of objects, be it either fine-granular to notify somebody when a single GUI element changes, or coarse-granular, e.g. on application level to react when certain system events arrive.
- Inter-process Communication (IPC)
  - Intents can be used for inter-application communication (implicit Intents, no receiving component specified, late-binding), or intra-app communication (explicit Intents, receiving component specified, early-binding).
  - Notifications are limited to intra-application communication on iOS.

Security

Android: Security mechanisms incorporated in Android span different levels [10]:

- *Operating system / VM level:* First, every application runs in its own Linux process, where each process has its own VM: Code runs isolated from all other applications' code. As each application is associated with a unique Linux user ID, permissions are set so that all application files are only visible to that user and application itself by default [1]. Together, these mechanisms create a sandbox that prevents one application from interfering with another.
- *Application level:* By explicitly declaring permissions, an application could get access to additional capabilites not provided by the basic sandbox. The user has to grant those permissions at installation time, as they have been statically declared in Android's manifest file.

iPhone: Similar to Android, iPhone applications are put in a sandbox that restricts the application to using only its own files and preferences, and limits the system resources to which the application has access. Unlike Android, there are no explicit permissions to grant on application level—all applications are equal and can use many of the provided phone capabilities (e.g., getting Internet access) without the user knowing. Due to the sandbox, they do not have direct access to the underlying communications or networking hardware though.

Qt: As Qt just comprises a cross-platform library, the security concepts inherent to the platform depend on the target OS in question. Qt itself doesn't have built-in security on a comparable level. As the focus of this review is on Qt for mobile platforms, relevant mechanisms have been discussed for Symbian [11], [12]. MeeGo will have its own security concept—still subject to definition though.

Implications

- *Open-source vs. closed-source:* Different implications of openness have been discussed. While some[6] suggest that iOS software is more secure since Android's open-source software stack can be investigated and understood by hackers, this circumstance is double-edged. It might be seen as advantage[7], since security issues can be detected and fixed faster by a bigger community. Thus, security can improve faster over time than on closed systems.
- *Approval process:* Another yet not discussed big difference on Android vs. iPhone is their strategy of application approval in their respective application stores. While Google doesn't vet applications at all, Apple is very restrictive on what is getting approved—which also adversely affects the approval time, making Android more suitable for quick prototyping, timely bugfixing or research work. Still, these measures cannot prevent somebody to publish malicious software—e.g., by activating the malicious part after a certain time period, the screening process during approval can be circumvented and tricked easily. However, what Apple could prevent is certain identity fraud—publishing an application as another author, e.g. by falsely providing the name of a big company, would most certainly be denied.
- *Sandboxing:* It should be noted that Android's and iPhone's sandbox mechanism is no means to prevent attacks, but rather a way of limiting the damage attackers could cause. Consequently, good programming practices such as carefully validating user input still apply.

4.2 Mobile-Specific

Access to Mobile-specific APIs. Functions such as telephony, texting or SIM access are at the core of each mobile phone. Their accessibility and abstraction levels are being discussed (e.g., can a call be set-up using a top-level API?).

Android: As Android was specifically designed with mobile devices in mind, the Application Framework layer in Android exports all required functionality to utilize mobile-specific services. Examples are the classes *TelephonyManager* for call- or sim-related functions (such as setting up a call, or retrieving the operator's name), the *SmsManager* (to send/receive text messages), or the *LocationManager* (e.g., to retrieve the current location).

[6] http://www.businessweek.com/news/2011-01-11/google-android-more-vulnerable-than-iphone-antivirus-maker-says.html
[7] http://tech.shantanugoel.com/2010/06/26/android-vs-iphone-security-models.html

iPhone: On iPhone, accessing core phone functionalities programmatically is more limited. For example, SMS can only be sent via the default SMS application, not from within an app (exception: iOS4 introduced some way of in-app form to prepopulate SMS form, but still, user must explicitly choose to send SMS). Another example is the limited access to network information—e.g., retrieving the signal strength is not officially supported (there are private frameworks that enable that—but it is not sure that Apple will approve apps that utilize such features). For access to other phone-relevant functionalities secondary frameworks exist, such as Core Location, Address Book or Map Kit.

Qt: Since Qt traditionally targets non-mobile operating systems such as Windows or Linux, too, it has not been designed specifically for mobile-specific use cases. To access specific features intrinsic to mobility, the Qt Mobility APIs have been introduced. These APIs grant access to the most commonly needed mobility features in a cross-platform fashion, i.e. without forcing the developer to implement platform-dependent native code like Symbian C++ [4].However, for not exposed functionality developers have to use platform-specific solutions. In Symbian, this could either be achieved by directly integrating Symbian C++ native code, or by using wrappers that expose those mobile extensions in a Qt-like API [4]—with the advantage of reducing native Symbian's steep learning curve.

Core Phone Functionality. The following section investigates how using certain phone functionality differs on the various platforms. A concrete example is used to illustrate this: placing a simple call from one's application.

Android: Uses Intent mechanism to notify other activities:

```
Intent i = new Intent(Intent.ACTION_CALL, Uri.parse("tel:+12345"));
i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
i.setClassName(mApp, PhoneApp.getCallScreenClassName());
mApp.startActivity(i);
```

Additionally, proper permissions must be granted in Manifest.xml:
`<uses-permission id="android.permission.CALL_PHONE"/>`

iPhone: Uses *Phone Link* concept (= URL scheme) to place a call (or SMS/email):

```
NSURL *phoneNumber = [[NSURL alloc] initWithString: @"tel:+12345"];
[[UIApplication sharedApplication] openURL: phoneNumber];
```

Qt: With the current version of Qt Mobility APIs 1.1, placing a telephone call is not yet supported (precisely speaking, the Telephony Events API from beta version has been removed/discontinued). Instead, one has to use platform-specific extensions. For Symbian, the easiest way is to utilize Qt Mobile Extensions:

```
XQTelephony *telephony = new XQTelephony(this);
telephony->call("+12345");
```

Implications: All platforms allow to incorporate basic phone functionality—although, despite Qt's claims to provide a cross-platform library, certain features could only be tapped into using platform-specific extensions. Another possible consequence was the possible exclusion from an application store if inofficial APIs were used for a certain functionality. It was pointed out that some features are not accessible on certain platforms at all—for example, an Emergency Service application on iPhone would not be able to send automated SMS notifications.

5 Conclusion

After a short introduction of the basic concepts on all three platforms key areas have been investigated in more detail. Due to certain limitations, it was pointed out that not every type of application is suited to be implemented on every platform. An unpredictable execution time resulting from Android's memory management concept suggested that the standard SDK might not be the library of choice for performance-critical applications such as games or applications with real-time constraints. Likewise, not obeying the application store rules of Apple, Google and the likes could lead to applications not being allowed or removed from the respective stores—a criterion that is better investigated thoroughly before chosing a platform. The APIs exposed to access mobile-specific functionality differ on the various platforms—it should be doublechecked whether the desired feature/information (e.g., reading phone signal strength) is accessible.

References

1. Android Developers Guide, `http://developer.android.com`
2. iOS Dev Center, `http://developer.apple.com/devcenter/ios`
3. Qt Developer Network, `http://developer.qt.nokia.com/`
4. Fitzek, F.H.P., Mikkonen, T., Torp, T.: Qt for Symbian. Wiley & Sons Ltd, United Kingdom (2010)
5. Lettner, M., Tschernuth, M.: Applied MDA for Embedded Devices: Software Design and Code Generation for a Low-Cost Mobile Phone. In: 34th Annual IEEE Computer Software and Applications Conference Workshops, pp. 63–68 (2010)
6. Android, D.: Source Code, `http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=vm/alloc/MarkSweep.c`
7. Ehringer, D.: The Dalvik Virtual Machine Architecture, Techn. report (March 2010)
8. Preiss, B.R.: Data Structures and Algorithms with Object-Oriented Design Patterns in Java. John Wiley & Sons Ltd, Chichester (1999)
9. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley, Reading (2002)
10. IEEE Security and Privacy 8(2), 35–44 (2010)
11. Heath, C.: Symbian OS Platform Security. John Wiley & Sons, Chichester (2006)
12. Badura, T., Becher, M.: Testing the Symbian OS Platform Security Architecture. In: Advanced Information Networking and Applications, AINA 2009, pp. 838–844 (2009)