Lessons Learned: Porting Java Applications to Android

G. Hsieh, D. Paruchuri, C. Steward, E. Nwafor and D. Gadam

Department of Computer Science, Norfolk State University, Norfolk, Virginia, USA
ghsieh@nsu.edu, [d.paruchuri, c.c.steward, e.c.nwafor, d.gadam]@spartans.nsu.edu

Abstract – *Android has become the world's most popular mobile platform. It provides a very powerful Android runtime and application framework that enable application developers to efficiently create innovative and feature-rich apps in Java. This attribute is very attractive to application developers who are familiar with Java and who may wish to port some existing Java applications to Android. However, there are significant differences between Android's Java and the Java SE environments. In addition, Android apps need to be designed and implemented with more care in order to meet the more stringent resource and performance constraints for mobile devices than those assumed for the Java SE environment. As a result, porting non-trivial Java applications from the SE to Android environments may not be as easy and straightforward as one may assume. In this paper, we discuss our experiences and lessons learned in our efforts to port two Java-based applications/systems - each utilizing an extensive set of open-source Java libraries - to Android from the SE environment.*

Keywords: *Android, Java, application porting.*

1 Introduction

Android has become the world's most popular mobile platform [1]. It has gained widespread acceptance since the announcement of the Open Handset Alliance in late 2007 [2]. Seeing a tremendous growth of internet usage and search in mobile devices, Google acquired Android, Inc. in 2005.

Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base and fastest growing of any mobile platform [3]. There are more than one hundred different makes of Android devices on the market currently, including smartphones and tablets, from more than fifteen manufacturers worldwide [1].

Android is also an open-source platform optimized for mobile devices. It is made available through the Android Open Source Project (AOSP) [4] which is led by Google, Inc. Android builds on the open-source Linux kernel, and its openness has made it very attractive for consumers and developers alike.

In addition, Android truly is a complete stack, from boot loader, device drivers, and libraries, to software APIs, included applications, and SDK [5]. Figure 1 shows the system architecture for the Android platform [6].

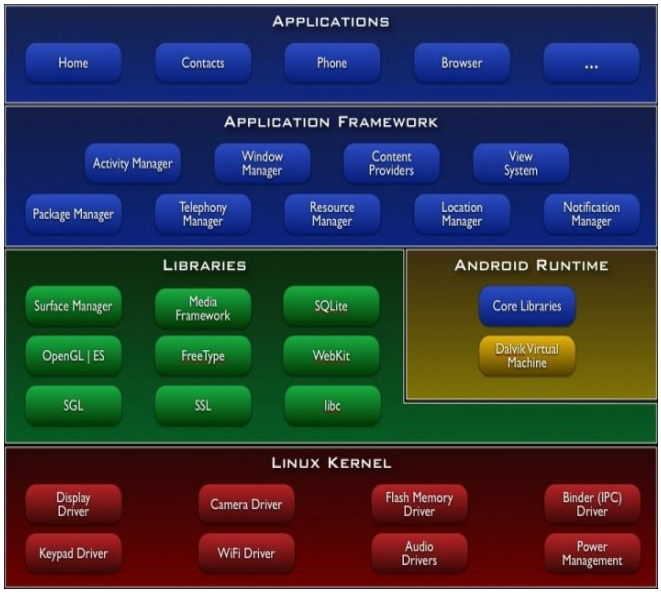

Figure 1. Android platform architecture

Android applications are typically written in Java. The application framework provides a tightly integrated part of the platform SDK and APIs that allow for high-level interaction with the system from within applications (e.g., accessing network data) [5]. Beneath the application framework is the middleware layer which contains the Android runtime and system libraries.

Android's runtime environment is similar to the Java runtime environment (JRE) provided by Sun/Oracle. First, it provides a core library which bundles all classes that are part of the specific Java platform, including language utilities, networking, concurrency, etc. Second, the runtime environment provides a Java virtual machine (JVM), called *Dalvik*, for running Java applications [5].

Android's integrated support for Java application development and deployment makes it very attractive to application developers, especially those who are familiar with Java and who may wish to port some existing Java applications from the SE to Android environments.

However, it is important to note that Android's Java is not equal to Sun/Oracle's Java SE. First, Android's core libraries do not bundle the same packages as in Java SE. Second, Dalvik is a JVM optimized for mobile platforms which accepts a different bytecode called Dalvik executable (*Dex*). This requires that the regular Java bytecode produced by a standard Java compiler needs to be translated into Dex

code in advance such that the latter can be executed by Dalvik VM on Android.

These two major differences can have varying degrees of impact when attempting to port existing Java applications from Java SE to Android environments. Some may be able to reuse many existing Java libraries with Android applications while the bytecode translation is merely a procedural issue that is automatically taken care of by Android SDK.

On the other hand, porting of more complex and larger scale applications may not be as easy and straightforward due to these two major differences in the Java platforms. There are also additional Java language/API-level differences which require the modification of Java application code. For example, the entry point to a Java program on Java SE is its *main*() method, while an Android app is not allowed to have a main() method. Another example is that Android does not support the AWT or Swing widget toolkits that are standard in Java SE for developing graphical user interfaces in Java.

Furthermore, Android apps need to be designed and implemented with more care in order to meet the more stringent resource and performance constraints for mobile devices than those assumed for the Java SE environment. For example, the application may need to be restructured or optimized in order to reduce the memory and storage requirements, or to improve the response time by performing tasks asynchronously.

Hence, porting of complex and larger scale Java applications/libraries from Java SE to Android environments can be very challenging. In some situations, it can be too difficult or impossible without major redevelopment, and thus it no longer qualifies as a "porting" effort.

In this paper, we present our experiences and lessons learned in our efforts to port two non-trivial Java applications (libraries) from Java SE to Android environments, hoping to invite more systematic and comprehensive discussion and information sharing among software engineering professionals on this interesting topic of "to port, or not to port".

Both of our efforts are related to the self-protecting security framework research program which began in 2005 at Norfolk State University [7] [8] [9] [10] [11] [12]. The fundamental concept underlying this framework approach is the use of a variety of XML-based open standards that are commonly used for web services security [13], including eXensible Access Control Markup Language (XACML) [14] for expressing access control policies.

This self-protecting security framework approach can be applied in a general-purpose fashion by using XACML as the container for all related information. Or it can be applied in a domain-specific fashion to use an open XML-based standard, such as Clinical Document Architecture (CDA) [15] for electronic health/healthcare information, as the container for all related information.

For experimentation and demonstration purposes, we have continued to develop prototype software for the self-protecting security frameworks [7] [9] [16] [17]. Our prototype software is written primarily in Java and it involves extensive processing of XML documents.

One of our objectives is to provide similar self-protecting security for apps and documents on Android. Our first effort was centered on the open-source XACML Java libraries, both Version 1.2 and 2.0, implemented by Sun/Oracle [18]. We successfully ported, after some difficulty, the Version 1.2 of Sun's XACML Java library which has been used for our prototype software on Java SE. However, we abandoned porting the Version 2.0 after running into so many problems.

Our second effort began with the open-source Model-Driven Health Tools (MDHT) Runtime Jars for Java (Release 1.0) [19], which has been used for developing our initial prototype Personal Health Record (PHR) application for the Java SE environment [16]. We successfully ported, after a period of trial and error, a subset of the Jars to meet the needs for our PHR application. Equipped with the ported MDHT Runtime Jars, we next attempted to port our PHR Java application code to Android. Due to the reasons mentioned above, we ended up practically redeveloping the PHR application as a native-architecture Android app throughout [17].

The remainder of the paper is organized as follows. In Section 2 we provide an overview of Android's Java application architecture and runtime environment, focusing on the implications for porting Java apps. In Section 3 we discuss the activities, results, and experiences in our case studies of porting efforts. In section 4 we conclude the paper with a summary.

2 Android Java

In this section, we discuss some of the most common and important factors affecting the degree of reuse of existing Java libraries or applications for Android. These factors include the Android core libraries and Dalvik VM which combine to form the Android Runtime, and the structure and performance considerations for Android apps which affect the scope of restructuring of Java application code.

2.1 Android Core Libraries

Android's Java core library implementation is based on Apache Harmony [20] which is an open source Java SE implementation by the Apache Software Foundation. Although Harmony is the basis for Android's core Java library, they are not exactly the same.

The Android core library implementation includes only those Harmony packages that are useful for Android mobile devices. It also includes Android-specific implementation of Java SE, replacing comparable packages in Harmony.

As a result, not all of Java SE runtime library is implemented in Android. The degree of potential reuse of existing Java apps or libraries is significantly determined by what is supported, partially supported, or not supported at all by Android's core Java library.

One obvious example is the user interface toolkits. Android provides its own user interface components that are optimized for mobile devices, and does not support AWT or Swing which are considered the standard user interface components for desktops. Thus, an existing Java SE app which uses AWT or Swing will need to have its user interface re-developed to replace AWT or Swing with Android's own user interface components.

As mentioned earlier, XML processing is fundamental to our self-protecting security framework approach and prototype implementation. Again, Android provides most, but not all, of the many XML support classes in Java SE.

Android supports both Document Object Model (DOM) and Simple API for XML (SAX) parsing of XML documents, and includes all core Java classes that those parsers require. On the other hand, the Java API for XML Binding (JAXB) is missing from Android completely [5].

2.2 Dalvik VM

Dalvik VM [21] is in charge of executing Java applications running on Android. It is developed through an open-source project with support from Google. Dalvik is optimized for mobile devices which have limited resources and power comparing with the desktop environment.

For efficiency considerations, Dalvik does not interpret Java bytecode directly. Instead, it uses the custom Dex bytecode. The *.class* files produced by a Java compiler needs to be converted to this Dex format. This conversion can be easily done by the Android SDK took, *dx*. So it is not necessary to have the source code for a Java library in order to use it in an Android application.

The main difference between the Dalvik and Oracle/Sun Java bytecodes is in the packing of code [22]. With Dex, all the classes of the application are packed into a single Dex file, as shown in Figure 2 [21]. In addition, all the classes in the same Dex file share the same constant pools for strings, fields, methods, etc.

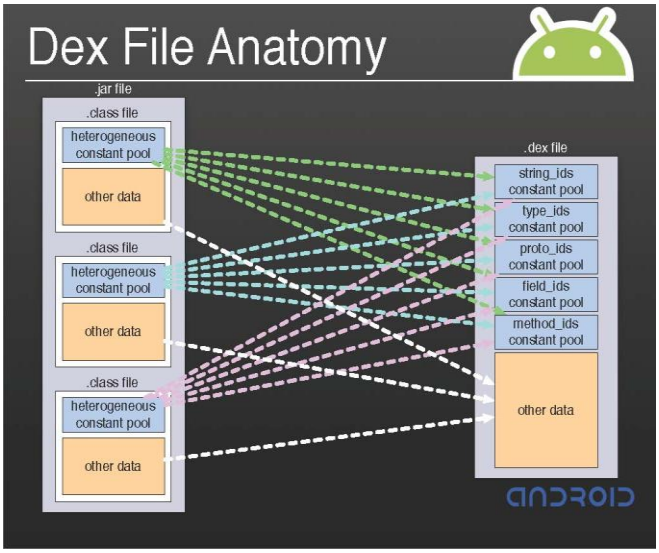

Figure 2. Dex file anatomy

The Dex approach helps reduce duplication of internal data structures and cuts down on the file size. On the other hand, classes from the same Dex file are loaded by the same class loader instance. In other words, these classes cannot be loaded using different class loader instances [22] as what can be done with Java SE. This restriction can pose a problem for porting those Java applications or libraries which require the manipulation of multiple classloaders.

This also means that all the classes in the same Dex file belong to the same namespace, and thus duplicated names across multiple Java classes can be a problem for Dex even when they are fine for the Java SE environment.

2.3 Android Applications

As mentioned earlier, Android applications are typically written in the Java programming language. Unlike applications on most other systems, Android applications don't have a single entry point (there's no *main*() function, for example) [23].

Android applications are composed of one or more application components. There are four types of application components: activities, services, content providers, and broadcast receivers. An *activity* is an application component that provides a screen with which users can interact in order to do something, such as dial the phone or view a map. Thus, it is commonly used by Android apps which provide user interfaces.

An activity is created as a subclass of the public class *android.app.Activity* (or an existing subclass of it). The lifecycle of an activity is managed by implementing callback methods that the system calls when the activity transitions between various states, such as when it is being created, stopped, resumed, or destroyed [23].

In summary, the structure of Android Java applications is quite different from that of Java SE. Thus, some restructuring of the application code is required when porting existing Java SE apps to Android.

Furthermore, Android apps need to be designed and implemented with more attention towards performance than typical Java SE applications, in order to meet the more stringent resource and energy requirements for mobile devices.

In Android, the system guards against applications that are insufficiently responsive for a period of time by displaying an "Application Not Responding (ANR)" alert and may even force the non-responding application to close [24]. It is critical to design responsiveness into the application so the system never displays an ANR alert to the user.

Android applications normally run entirely on a single thread (by default the "UI thread"). This means anything the app is doing in the UI thread that takes a long time to complete can trigger the ANR alert because the app is not giving itself a chance to handle the input event or intent broadcasts. Therefore, any method that runs in the UI thread should do as little work as possible on that thread. Potentially long running operations should be done in a

worker thread, which can be most effectively created with the *AsyncTask* class [24].

Again, an existing Java app code may need to be restructured for performance consideration, as we have done for the Android version of our PHR prototype application.

2.4 Android SDK

The Android SDK [25] provides the API libraries and developer tools necessary to build, test, and debug apps for Android. The recommended IDE is Eclipse with the ADT (Android Developer Tools) plugin.

Since we have been using Eclipse IDE for Java Developers for our prototype software development, Android's Eclipse+ADT IDE is very convenient for us. To test our Android apps and libraries, we used a variety of Android device emulators, smartphones, and tablets.

3 Case Studies

In this section, we discuss our efforts, results, and experiences in two cases: 1) porting Sun's XACML Java libraries and a sample application; and 2) porting MDHT Runtime Jars for Java and our prototype Personal Health Record application.

3.1 Sun XACML Jars and Sample App

Oracle/Sun Lab released its Version 1.2 of XACML Java Library in July 2004 and Version 2.0 in July 2010 [18]. We have been using the Version 1.2 of the *sunxacml* library for our prototype software. As a matter of fact, we have extended the library to add new features and conventions for our self-protecting security framework approach.

As we were already planning to upgrade our prototype software to leverage the Version 2.0 of sunxacml, we first attempted to port this version to Android in late 2011. After running into so many problems with this version, we went back to the Version 1.2 of sunxacml with which we had more knowledge and experiences.

In the end, we managed to port the Version 1.2 of sunxacml library to Android. However, the process was not easy, nor straightforward. The main challenges were due to the fact that the sunxacml library requires a set of Java core (java.* or javax.*) classes that were not supported by Android runtime.

The sunxacml Version 1.2 release contains the source, data files, documentation, and the produced libraries. The main library, *sunxacml.jar*, for producing and reading XACML documents is 191 KB in size. The source needed to build sunxacml.jar is contained in a */src/sunxacml* folder which contains 243 files in 23 subfolders taking up a total space of 1.14 MB. The distribution also contains a *samples.jar* (7 KB in size) which includes a sample program called *simplePDP* that can be run to demonstrate XACML applications while using sunxacml.jar. The source and XML data files needed to build samples.jar and run simplePDP are contained in a */sample* folder which contains 22 files in 4 folders taking up a total space of 104 KB.

To port Sun's Version 1.2 XACML Java API library (sunxacml.jar) and its sample application (simplePDP) to Android, we undertook the following major activities:

(1) Set up a new Android application project also called *simplePDP*, using the Eclipse-integrated Android SDK (r6 or newer). The project target was set for Android API Level 6 (Android 2.0.1 Release 1) which was released in December 2009 and represented the Android platform that was broadly supported by Android devices in 2010-2011 timeframe.

(2) Set up the source for the simplePDP application project. This step was quite straightforward as sunxacml already used the same Apache Ant build tool and a very similar project structure as required by the Eclipse-integrated Android SDK.

(3) Restructure the code for the simplePDP class. The original class for Java SE contains a *main*() method which is not allowed for an Android application. Thus, we created a new *simplePDPActivity* class, which extends the Android Activity class, to serve as the entry point and to provide a user interface for the Android simplePDP app.

The relevant initialization code contained in the main() method was implemented inside the *onCreate*() method for simplePDPActivity, such that the necessary and equivalent initialization functions can be performed when the activity is created after the app is launched by the user. Also contained in the onCreate() method is the code to start an instance of the modified simplePDP class which no longer contains a main() method. Note that the sample program contains six other helper classes for the simplePDP class. Those classes did not require any code modification for Android.

(4) Restructure the file I/O. The sample application for sunxacml takes two XML files as input to produce another XML file as output. On Java SE, the input files are stored under the */sample/policy* and */sample/request* folders, respectively, within the project's file structure, and they can be easily accessed by using *java.io* APIs on the same Java SE host.

For Android, we prefer to have these input files distributed with the app such that no separate file transfer or configuration actions are required. To accomplish this goal, we have not found a working solution other than including these files as resources or assets for the app such that they can be packaged and installed as part of the app.

Unfortunately, this solution requires a different set of APIs, namely *Resources* or *AssetManager* classes, instead of the *java.io.File* class, to access the contents. This posed a problem for the sample application as it relies on File operations extensively. Instead of modifying the app code to use AssetManager operations everywhere and thus causing more widespread changes, we chose to isolate the changes within the onCreate() method by adding the code to read the contents through the AssetManager and then store them into files on internal storage. The references to the internal files (e.g., fully-qualified file names) are then used in the rest of the application in the same way as before.

(5) Bundle the missing core Java libraries. With all the preparations done, we proceeded to build and run the app using the Eclipse-integrated Android SDK. After fixing application-level errors, a compilation or execution could still fail due to "unresolved symbol" compilation errors or "NoClassDefFoundError" runtime errors, both indicating that some core Java classes were needed but missing from the Android runtime.

To resolve these types of errors, we chose to bundle the missing core Java classes with the app itself, instead of extending the core runtime library for Android platform, to facilitate our porting and experimentation efforts without modifying Android platform releases. We also used an iterative process to find appropriate solutions if possible. For each missing core Java class or package, we first used online resources, such as findJAR.com [26], to find available Jar(s) that contain the missing element. After further investigation, we next added such a Jar to the list of external libraries used to build the simplePDP app. Then we proceeded to build and run the app with the added external Jar which in turn might need additional Jars that were missing from Android runtime. This process was repeated until there was no core Java class that was apparently missing. After a working set was assembled, we next worked to reduce the memory and storage requirements for the app by eliminating redundant or extraneous classes from the working set.

For the sunxacml 1.2 Java API library and sample app, we added three additional Jars: xml-apis.jar, jndi.jar, and jndi-properties.jar, which combine to take 290 KB in size.

(6) Work around the "Conversion to Dalvik format failed with error 1" problem. According to the error message, this error indicates an "ill-advised or mistaken usage of a core class (java.* or javax.*) when not building a core library. This is often due to inadvertently including a core library file in your application's project, when using an IDE (such as Eclipse)." On the other hand, the Android app building tool does provide a --core-library option which can be set to suppress this error message and allow the build to proceed even when core classes are present in the application project. However, the ADT plugin for Eclipse does not allow this option to be set through Eclipse. It is interesting to note that the Android Maven Plugin does allow this option to be set through Maven.

For our porting effort, we needed to include these missing core library files (e.g., xml-apis.jar) in our application's project. However, we did not want to change our build tool from Ant to Maven. Therefore, we modified the default *build.xml* file to set this --core-library option through a custom shell script that we developed. Using this approach, we managed to work around the problem with a relatively simple custom solution. However, it was not ideal as it required modifying the default build file, and running the final application packaging tool through the command line interface outside of Eclipse.

In summary, we managed to port Sun's XACML v1.2 Java Library and sample application to Android. The size of the Android application package (.apk) file is about 187 KB.

Our efforts to port the Version 2.0 of Sun's XACML library and sample program did not succeed. One major reason for our difficulties was due to the fact that the Version 2.0 library was re-implemented using the JAXB technology.

JAXB is very powerful as it provides a fast and convenient way (using automation tools) to bind XML schemas and Java representations, making it easy for Java developers to map Java classes to XML representations [27]. On the other hand, it also adds a great deal of complexity to the runtime environment. As an indication, the size of the source-only release of Version 2.0 Sun XACML library is already approximately 570 KB in size.

Since JAXB was not supported by Android's runtime core library, it was very challenging and time-consuming trying to bundle all missing core classes (e.g., *java.xml.bind*) and their dependencies in the application's project. As a result, we abandoned this approach after putting in a good amount of effort without ever gaining enough confidence that this approach could work from both functional and performance viewpoints. For example, the size of the none-functional .apk file had already reached a size of approximately 500 KB for the same "application".

3.2 MDHT Runtime Jars and PHR App

Our interests in MDHT runtime Jars and personal health record applications centered on our efforts in developing the self-protecting security framework and associated prototype software for securing electronic medical records [8] [28] [16] [17].

As mentioned earlier, our approach leverages the CDA which is an XML-based document markup standard that specifies the structure and semantics of a clinical document. For our prototyping effort, we chose to leverage the runtime Jars provided by the open-source MDHT project which was initiated, by the Veterans Health Administration in April 2008 in collaboration with IBM as the co-lead of the project, to promote interoperability in healthcare infrastructure.

The MDHT runtime distribution contains JAR files with generated Java code from template models, plus all necessary dependencies for Eclipse-based modeling framework and code generation facility. It is intended for application developers who are using MDHT Java libraries created from models (e.g., CDA), not for creating or editing model specifications.

We first implemented a prototype PHR application for the Java SE environment [16]. This application used the MDHT runtime distribution Release 1.0, which became available in September 2011 timeframe, for processing CDA documents. Although the MDHT runtime release contained 24 Jar files with a total size of 9.68 MB, it was not an issue for the prototype PHR application running on Java SE.

With our interest in providing self-protecting security capability for Android, we undertook an effort to port the MDHT runtime Jars to Android. The first major roadblock we encountered was due to the duplicated file names. Each of the Jar files contained a text file named *plugin.xml* and/or another text file named *plugin.properties*. Given the

duplicated file names, the Eclipse-integrated Android SDK would fail to build an Android application with these Jars in the application's project.

To work around this problem, we chose to delete these files from all of MDHT runtime Jars, as they were descriptor files used for describing how the plugin (Jar) extends the Eclipse platform, etc. [29]. After the files with duplicated names were removed from the Jars, the Android application could be built successfully. Note that the file removal could be easily accomplished by using the Java *jar* command without modifying or recompiling any source files.

Since our focus was on using the MDHT Java libraries that were already created from models (and not on creating or editing the models themselves), we believed that the impact of removing these types of descriptor files would not be significant for our purposes. Our experiences in running the Android application with the modified Jars seemed to confirm this assumption, as we have not observed any side effect due to the removal of these descriptor files.

After resolving the major roadblock caused by duplicated file names, we undertook optimization effort to reduce the number and total size of the Jars required for our application which did not need all the capabilities provided by all the Jars collectively. We used an iterative and (more or less) a trial-and-error approach to select the minimal subset of Jars that we needed for our Android application. It turned out that the final subset contained 11 (vs. 24 originally) Jars with a combined size of 4.35 MB (vs. 9.68 MB originally). This optimization effort and results were very beneficial to our prototyping program as they helped to reduce the application's runtime memory and persistent storage consumption on Android devices.

Our Java SE personal health record prototype application had GUI-based user interfaces that allow users to enter, view, modify, encrypt, and digitally sign their records maintained in CDA documents. These user interfaces were implemented using Swing.

To develop a similar PHR application for Android [17], we used the final subset of modified MDHT Jars to provide the same CDA processing functions. However, we did major restructuring of our application level code for both functional and performance considerations.

First, we restructured the code based on Android's application architecture. The Android PHR application now consisted of five Android activities plus additional helper classes. These activities allowed us to organize the code in a very modular fashion, and they provided the main and submenu user interfaces for starting the app, entering data, viewing data, editing data, and emailing data, respectively.

Second, we developed the user interfaces for our Android PHR application using the View-based components for Android.

Third, we implemented the Android PHR application with multi-threading capabilities in order to improve user responsiveness and avoid the much dreaded ANR problem. We used the AsyncTask construct to execute potentially long-running operations (e.g., encrypting or saving a CDA document which could be large) in separate threads away from the UI threads.

Fourth, we used the SAX-based XML parser for the Android PHR application, in contrast with our using the DOM-based XML parser for the Java SE based PHR application. This approach allowed us to conserve memory usage when parsing large CDA documents. However, it did add a great deal more complexity in our application code in order to handle the SAX events asynchronously. In addition, the CDA structure is very flexible and hence complex, and it is difficult to use SAX-based parser to extract information from CDA documents [30].

The MDHT runtime Jars were implemented using the in-memory, DOM-based programming model, and it did a very good job of hiding the low-level details and complexity from the application developers. Using the SAX-based parser, the application developers had to handle the low-level details themselves and this increased the application programming complexity significantly. To save time, we implemented only a subset of the data fields for the Android based PHR application.

In summary, we managed to migrate our PHR app from Java SE to Android. For performance consideration, we used multi-threading and memory-efficient parsing of XML documents. In the end, we practically redeveloped it as a native-architecture Android app which bears little resemblance with the Java SE based PHR app, while reusing the MDHT runtime Jars.

4 Summary

In this paper, we presented our efforts, results, and experiences in porting two Java applications/libraries from Java SE to Android environments. These software packages involved open-source Java libraries for processing XML-based documents.

Overall, we found these experiences very educational, as we encountered numerous problems along the way, including those caused by the differences in the core Java runtime library and virtual machine, IDE restrictions, etc. We were able to overcome those problems in all cases except the one involving the sunxacml v2.0 library. In addition, we learned important lessons in dealing with the more stringent resource and performance constraints for mobile devices which are not the same as desktops. Using techniques such as multi-threading and event-driven XML parsing helped to improve the resource and performance aspects; on the other hand, they added more complexity and required more effort in developing the applications.

We like to close the paper with the following observations:

(1) Migrating Java applications from Java SE to Android is more complicated than what might be assumed, except for small and trivial programs perhaps. Minimally, the app needs to be restructured to conform to Android's application model (e.g., activity versus main() method).

(2) The complexity increases if the application has extensive user interfaces implemented with AWT or Swing.

These user interfaces need to be practically rewritten using Android's View components.
(3) Third-party Java libraries could be a problem, especially if they use many of the core Java libraries (java.* or javax.*) that are not supported by Android.
(4) Files for initialization, configuration, or information could present a problem. Android has different classes and APIs to handle "resource" type of content which are treated differently from "files". Duplicated file names could cause additional problems.
(5) For performance and resource usage considerations, Android implementations may require more efficient or user-responsive techniques such as multi-threading and asynchronous/event-driven processing.
(6) Do not ignore the fact that Android-powered mobile devices are not the same as desktops, let alone servers. Be careful not to overload Android devices with apps requiring heavy-weight processing or storage.

5 Acknowledgement

This research was supported in part by U.S. Army Research Office, under contract no. W911NF-12-1-0081, and U.S. Department of Energy, under grant no. DE-FG52-09NA29516/A000.

6 References

[1] "Android," android.com, [Online]. Available: http://www.android.com/. [Accessed 27 May 2013].

[2] "Open Handset Alliance," [Online]. Available: http://www.openhandsetalliance.com/. [Accessed 27 May 2013].

[3] "Android, the world's most popular mobile platform," [Online]. Available: http://developer.android.com/about/index.html. [Accessed 27 May 2013].

[4] "Android Open Source Project," [Online]. Available: http://source.android.com/. [Accessed 27 May 2013].

[5] C. Collins, M. G. Galpin and M. Kaeppler, Android in Practice, Manning Publications Co., 2011.

[6] "Android Architectural Diagram," [Online]. Available: http://developer.android.com/images/system-architecture.jpg. [Accessed 27 May 2013].

[7] G. Hsieh and E. Nwafor, "A Self-Protecting Security Framework for CDA Documents," in *Intl' Conf. on Security and Management (SAM'13)*, Las Vegas, NV, 2013.

[8] G. Hsieh, "Towards Self-Protecting Security for e-Health CDA Documents," in *Proc. Int'l Conf. on Security and Management 2011 (SAM'11)*, Las Vegas, NV, 2011.

[9] G. Hsieh and M. Masiane, "Towards an Integrated Embedded Fine-Grained Information Protection Framework," in *Proc. 2011 Int'l Conf. on Information Science and Applications (ICISA'11)*, Jeju Island, Korea, 2011.

[10] G. Hsieh, R. Meeks and L. Marvel, "Supporting Secure Embedded Access Control Policy with XACML+XML Security," in *Proc. 5th int'l Conf. on Future Information Technology (FutureTech'10)*, Busan, Korea, 2010.

[11] G. Hsieh, K. Foster, G. Emamali, G. Patrick and L. Marvel, "Using XACML for Embedded and Fine-Grained Access Control Policy," in *Proc. 4th Int'l Conf. on Availability, Reliability and Security (ARES'09)*, 2009.

[12] G. Hsieh, G. Patrick, K. Foster, G. Emamali and L. Marvel, "Integrated mandatory access control for digital data," in *Proc. SPIE 2008 Defense + Security Conf.*, Orlando, FL, 2008.

[13] E. Bertino, I. D. Martino, F. Paci and A. C. Squicciarini, Security for Web Services and Service-Oriented Architectures, Springer-Verlag, 2010.

[14] "eXtensible Access Control Markup Language (XACML) Version 2.0," OASIS, 2005. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml. [Accessed 31 May 2013].

[15] R. H. Dolin, L. Alschuler, C. Beebe, P. V. Boyer, D. Essin and E. Kimber, "The HL7 Clinical Document Architecture, Release 2," *J. Am Med Inform Assoc.*, vol. 13, no. 1, pp. 30-39, Jan-Feb 2006.

[16] D. Gadam, "Generating CDA Documents and Embedding XML Security," M.S. Thesis, Department of Computer Science, Norfolk State University, Norfolk, VA, March 2012.

[17] D. Paruchuri, "Developing a Personal Health Record Application for Android Platform," M.S. Thesis, Department of Computer Science, Norfolk State University, Norfolk, VA, April 2013.

[18] "Sun's XACML Implementation," [Online]. Available: http://sourceforge.net/projects/sunxacml/. [Accessed 27 May 2013].

[19] "Model-Driven Health Tools (MDHT)," [Online]. Available: https://www.projects.openhealthtools.org/sf/projects/mdht/. [Accessed 27 May 2013].

[20] "Apache Harmony," [Online]. Available: http://harmony.apache.org/. [Accessed 29 May 2013].

[21] D. Bornstein, "Dalvik VM Internals," Google, 29 May 2008. [Online]. Available: https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf. [Accessed 29 May 2013].

[22] G. Paller, "Understanding the Dalvik bytecode with the Dedexer tool," 2 Dec 2009. [Online]. Available: http://www.slideshare.net/paller/understanding-the-dalvik-bytecode-with-the-dedexer-tool. [Accessed 29 May 2013].

[23] "Application Fundamentals," [Online]. Available: http://developer.android.com/guide/components/fundamentals.html. [Accessed 30 May 2013].

[24] "Keeping Your App Responsive," [Online]. Available: http://developer.android.com/training/articles/perf-anr.html. [Accessed 30 May 2013].

[25] "Get the Android SDK," Android Developers, [Online]. Available: http://developer.android.com/sdk/index.html. [Accessed 31 May 2013].

[26] "findJAR.com," [Online]. Available: http://www.findjar.com/index.x. [Accessed 31 May 2013].

[27] "Lesson: Introduction to JAXB," [Online]. Available: http://docs.oracle.com/javase/tutorial/jaxb/intro/. [Accessed 31 May 2013].

[28] G. Hsieh and R.-J. Chen, "Design for a secure interoperable cloud-based Personal Health Record service," in *IEEE 4th Int'l Conf. on Cloud Computing Technology and Science (CloudCom'12)* , Taipei, Taiwan, 2012.

[29] "FAQ What is the plug-in manifest file (plugin.xml)?," eclipse.org, [Online]. Available: http://wiki.eclipse.org/FAQ_What_is_the_plug-in_manifest_file_(plugin.xml)%3F. [Accessed 31 May 2013].

[30] Keith Boone, The CDA Book, Springer, 2011.