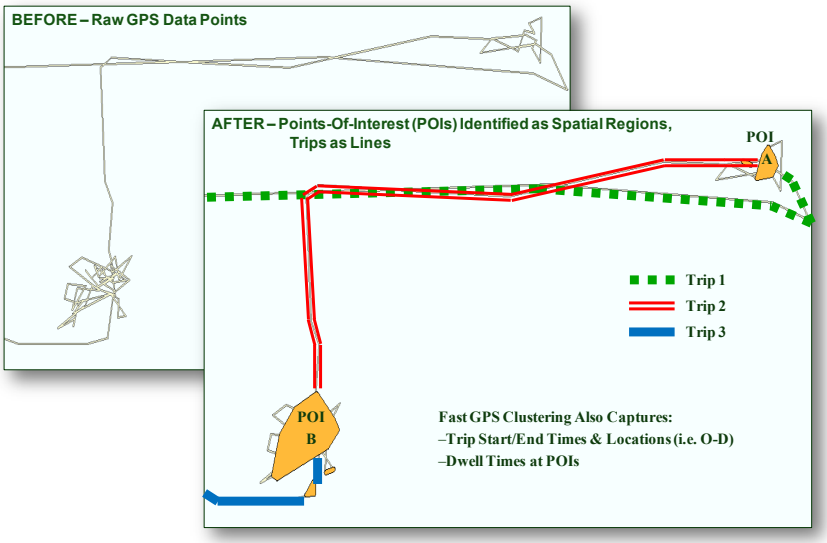
Prepared by

National Center for Transit Research

USF UNIVERSITY OF SOUTH FLORIDA

Dynamic Travel Information Personalized and Delivered to Your Cell Phone

Addendum



Funded by

Florida Department of Transportation

FDOT BDK85 TWO 977-14

March 2011

DYNAMIC TRAVEL INFORMATION PERSONALIZED AND DELIVERED TO YOUR CELL PHONE

ADDENDUM

FDOT BDK85 Task Work Order #977-14

Prepared for:

Florida Department of Transportation
Amy Datz, Project Manager

Prepared by:

Sean Barbeau, Senior Research Associate
Nevine Labib Georggi, Senior Research Associate
Philip Winters, Transportation Demand Management Program Director
USF Center for Urban Transportation Research

FINAL Addendum

March 2011

CONTENTS OF ADDENDUM

LIST OF FIGURES

LIST OF TABLES

REACHING THE LARGEST NUMBER OF POTENTIAL MOBILE PHONE USERS

Real-time travel information must reach a significant amount of travelers to create a large amount of travel behavior change. For this project, since the TRAC-IT mobile phone application is used to monitor user context in terms of location, the mobile phone application should be targeted to span across many different cellular carriers and various mobile device manufacturers. The Java programming language is the most platform-independent programming language for mobile devices that reaches the most phones from different manufacturers and on different cellular networks. Java Micro Edition (Java ME) has been the Java-programming platform of choice from early 2001 through 2009 and was used to implement the initial mobile phone version of TRAC-IT.

In late 2008, the Android platform created by Google and its partners in the Open Handset Alliance emerged in the mobile industry. Since 2008, use of Google Android has exploded, appearing on many different smartphone devices from a variety of manufacturers, including Motorola, HTC, Samsung, Kyocera, Sanyo, and LG, and is available on every major U.S. cell network, including Verizon Wireless, AT&T, Sprint, Nextel, and T-Mobile. In the second quarter of 2010, Android took the lead as the top operating system for smartphones sold in the U.S., with 33 percent of the share of phones purchased in the seocnd quarter of 2010, while RIM Blackberry came in second at 28 percent and Apple's iOS for iPhone held on at 22 percent [1]. In September 2010, Gartner declared that Android will capture the number two worldwide operating systems title in 2010, ahead of RIM's Blackberry and Apple's iPhone, and challenge Nokia's Symbian operating system for the No. 1 position by 2014 [2]. Android as a mobile device programming platform is also beginning to push into the tablet PC market as well as the "feature phone" or the less capable and cheaper "flip phones."

Android will be a crucial platform for the deployment of mobile applications that provide travel information to users, as Android reaches many users on all major U.S. carriers and on devices that tend to be more capable devices than those that have Java ME as the programming platform. Therefore, an early version of TRAC-IT for Android was created as part of this project.

Basic capabilities in Android devices generally include:

- Higher speed processor
- Better user interface
    - Larger screens
    - Touch screens
    - Accelerometers
    - Haptic feedback
- More capabilities built into platform by default:
    - E.g., encryption and decryption algorithm suites for symmetric and asymmetric algorithms RSA, AES, DES, SHA1, MD4, MD5, DSA
- Extensible applications and modules (e.g., third party libraries)
    - Intents, Services, ContentProviders
- Networking

    - Built in support for IPv6
    - Wi-Fi is standard on touch-screen devices (but may not appear on lower-end phones)
    - Includes Multicast and Broadcast support
- Location (i.e., Positioning Technologies):
    - Implementation of assisted GPS is changing to gpsOneXTRA to reduce impact on carrier network resources. Uses predictive ephemeris information for GPS instead of real-time, which reduces required communication with network to once every seven days instead of once every two hours in traditional assisted GPS. Reduced network overhead is important to carriers since they pay a per-transaction fee to positioning technology implementer's assistance servers, and Android is an open platform which allows unlimited location queries. This differs from traditional assisted GPS, to which access was restricted by carriers to only important vendors.
    - Some hybrid networking positioning technologies are emerging, such as Skyhook

This section presents a brief history of Java ME and Android, including the motivating factors behind Android's design and its relationship to Java ME. The major differences between the two platforms are discussed, as well as some basic information needed to develop applications on Android. Finally, features of the Location platform on Java ME and Android are compared and contrasted.

These differences are important for mobile software developers to understand as they transition applications, such as TRAC-IT, from feature phones (i.e., "flip phones") to smartphones. This background research for Android also formed the basis of the design of the prototype Traffic Text-To-Speech (TTS) mobile application, a prototype version of TRAC-IT for Android which is discussed in another chapter that delivers traffic information to users while trying to minimize distraction to the user while they are traveling. No matter how relevant a piece of information is to a traveler, the information should be provided to the user in the least distracting method possible. Traffic TTS is one method of information delivery via a mobile phone that attempts to minize user distraction.

A BRIEF HISTORY OF JAVA ME

Java ME is the most widespread mobile device programming platform since around 2001, with over 2.6 billion Java ME enabled mobile phones worldwide [3]. Java was initially created by Sun Microsystems in order to have a programming language that could be compiled across platforms such as Windows and Linux. Java ME, the mobile edition of the Java programming language and platform, has been used by most major cellular carriers and device manufacturers. Java ME allows developers to use their existing Java programming skills for the desktop and server environment in order to quickly learn how to develop applications for mobile phones. The Java community takes advantage of expert groups from the industry to expand various Java platform Application Programming Interfaces (API), such as a Location API that allows programmers to access GPS positions, via Java Specification Requests (JSRs).

THE ANATOMY OF JAVA ME

Java ME is divided into two major categories of devices: Connected Limited Device Configuration (CLDC) for less capable "feature phones" and Connected Device Configuration (CDC) for higher-end smartphones. CLDC is usually combined with the Mobile Information Device Profile (MIDP) 2 for mobile phones. MIDP 3 is the next major step in evolution for feature phones, which was finalized in December 2009, but is not yet available. CDC was planned to be combined with "Foundation" Profile and Open Services Gateway initiative

(OSGi) (JSR 232) for smartphones. CDC with OSGi and Foundation Profile is backwards-compatible for apps on MIDP 2.0 and CLDC 1.1. CDC has not been widely implemented yet by mobile device manufacturers, although a Sprint "Titan" proof-of-concept with IBM and Sun emerged in 2008 to 2009.

DRAWBACKS OF JAVA ME CLDC

While Java ME was the best platform for the largest number of devices from different manufacturers and carriers at the time, it had several drawbacks. First, Java ME remained tightly controlled by carriers in the United States. This meant that the carrier was the entity that held the final control over what applications would be allowed to run on consumer handsets. Carriers had to digitally sign applications before they could be installed on mobile phones. For certain sensitive APIs, such as the Location API which could affect the carriers' assisted GPS servers, cellular carriers even restricted access to developing applications based on these APIs to a few select trusted partners.

Another limitation of Java ME was its phone-centric design. Java ME was designed to run on devices that were mobile phones first, and computing devices second. This led to some early design considerations for application lifecycles that left ambiguities in how various applications would react in a multitasking environment when multiple applications were running in the background. When Java was first developed, phones were fairly primitive computing devices that weren't even capable of floating-point calculations. As cell phones became more powerful, they began to have behaviors that fell outside of the initial design considerations for Java ME.

Another limitation to the Java ME environment is that applications were sandboxed, limited, and difficult to troubleshoot. This meant that up until CLDC 1.1/MIDP 3 (which still has not been commercially deployed at the time of this report) there was no interaction allowed between mobile applications running on the same device. This means that no third party APIs or libraries could be installed on a device for other applications to take advantage of, which restricts applications to the functionality that comes integrated in the mobile device at the time of manufacturing and the functionality of the source code bundled inside of the application. Debugging applications on Java ME is also a painful process. While various manufacturers have created emulators that simulate a device in a software application running on the developer's computer, these emulators often have different behavior than the actual device. Additionally, no general standard debugging tools exist for Java ME, and the only tools that do exist are customized applications created by device manufacturers that aren't always designed for use by third party application developers.

Since Java ME was created when mobile devices were fairly primitive, it also has a limited scope for certain functionality on-board devices. For example, Java ME exposes a general Canvas object for developers to draw images on. It also has some basic textboxes, list elements, and other basic user interface elements. However, it does not expose any other types of interactions with the user that many mobile users are now accustomed to, including touch-screen interfaces, animated interfaces, and haptics (i.e., vibration) feedback. Java ME also offers little in the way of persistent storage on the device, with only a byte record store exposed for applications to store data that is not lost after an application exits.

While many of these limitations of Java ME could have been changed, another major limitation of the platform essentially prevented this from happening in a timely way. As Java uses a Java Specification Request (JSR) process with an expert group appointed by industry for new APIs and modifications to old APIs, this process can take years to complete by the time a general consensus is reached. With many mobile phone lifecycles lasting less than a year, Java ME was not able to evolve fast enough to keep up with demand for new handsets by customers and new features by device manufacturers and mobile application developers.

The JSR process also brought about limitations to the platform in the way of fragmentation. Since separate JSR expert groups existed for each type of API (e.g., Location API, Messaging API, Mobile Media API, etc.), a silo design of Java ME emerged that did not always include full integration and intuitive behavior across APIs. Additionally, since an application would have to depend on multiple JSRs being supported on a device to support various components of the app (e.g., GPS access, SMS messaging access, camera access), determining which devices might be compatible with an application was not always a straightforward process and forced customers to go through a large check-list of requirements to determine compatibility. Additionally, since on the mobile platform the CLDC/MIDP virtual machine and JSR APIs may be implemented by different manufacturers (unlike desktop and server versions of Java), different implementations of the same API (e.g., Location API) may react differently on different devices. The Mobile Services Architecture (MSA) 1 spec (JSR248) was created to help unify all the different individual JSR APIs and define expected application and device behavior in areas that other JSRs vaguely defined. While MSA 1 did help customers identify compatibility of devices with applications (e.g., a device would be either MSA1 "Full" or "Subset" compliant), it was not enough so resolve all of the fragmentation problems. Mobile Service Architecture (MSA) specification 2 (JSR249) was then created in order to try to limit fragmentation again and unify the diverging Java ME platform again. MSA 2 is now under development for new JSRs\ and at the time of this report is currently in Public Review stage. The large amount of various APIs that are fragmented across Java ME devices which MSA 2 attempts to consolidate is shown in Figure 1.

Additional considerations also made Java ME problematic for device manufacturers and customers. Since each JSR requires licensing from Sun Microsystems, and usually the lead company on each API spec for TCK (validation software) and/or RI (actual JSR implementation), there are high licensing costs for device manufacturers to make "Java ME-compliant" devices. These costs must then be passed down to the consumer. Another limitation which makes mobile application distribution from developers to consumer difficult is that there has not been a widely accepted "market place" to download applications from. GetJAR and several other marketplaces have recently become more popular places for Java ME application distribution, which helps alleviate distribution problems.

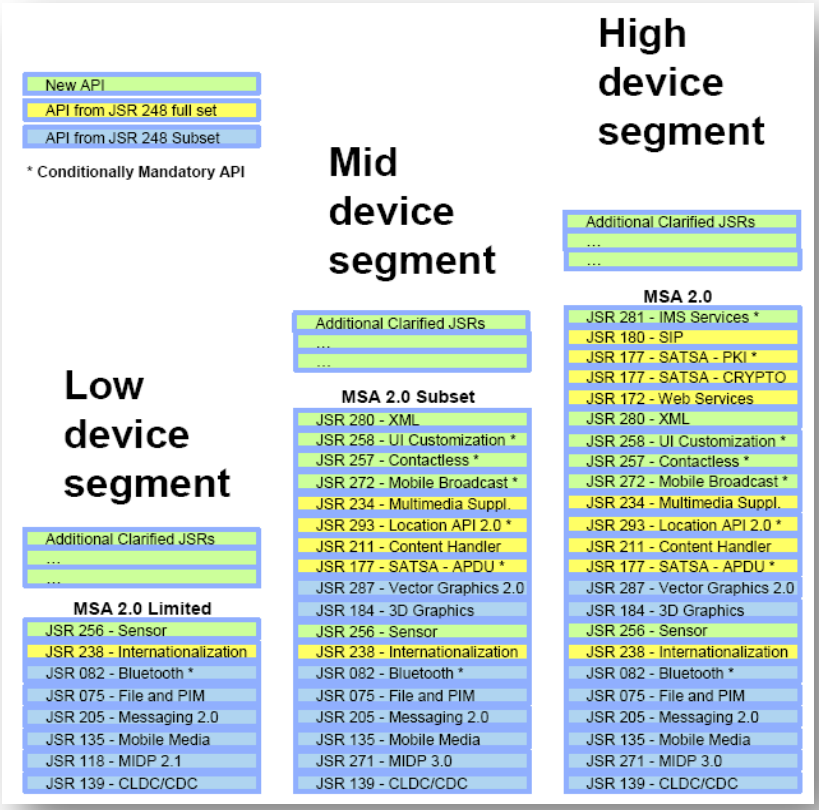


Figure 1 - Java ME's Mobile Service Architecture (MSA) Specification (JSR249)

THE PLANNED EVOLUTION OF JAVA ME

Java ME was originally planned to evolve from more limited "feature phones," or "flip phones," to more capable smartphones with touch screens and faster processors. The Connected Device Configuration (CDC) and Open Services Gateway Initiative (OSGi) were planned to be the base Java programming platform for Java ME on smartphones. Open Services Gateway initiative (OSGi) allows a Service-Oriented Architecture which should allow open services on devices, including 3rd party libraries/APIs. However, it was not clear that CDC would resolve some of the other limitations and challenges from CLDC, including:

1) The slow evolution process of JSRs
2) Fragmentation issues
3) Licensing issues
4) Carrier control of Java ME platform in the United States

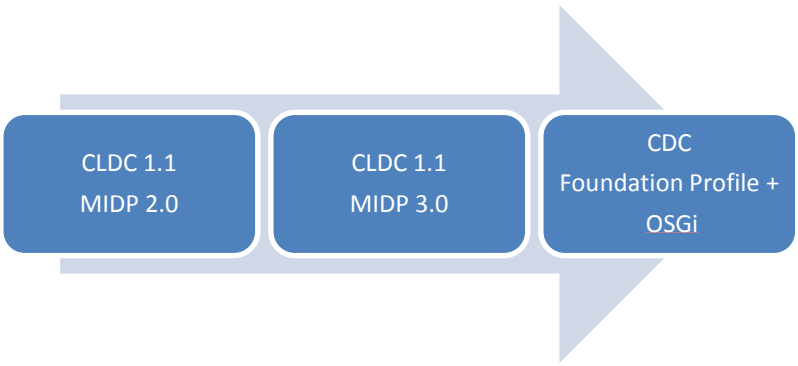


Figure 2 - Planned Evolution of Java ME

MOTIVATION BEHIND ANDROID

Google wanted a powerful, unified, open platform so Google and developers could provide services to users without control of a middleman (i.e., cellular carrier) for all interactions. The company believed that a platform should be designed from the ground-up for current mobile devices with a less phone-centric, and more computer-centric, approach. CLDC, and even CDC, are being adapted from early thoughts of what mobile devices would be like circa 2001. Google also wanted a platform that could evolve quickly to keep pace with the market. For example, MIDP 2 was released in June 2006, MIDP 3 was finalized in December 2009 (almost three years, and MIDP 3 is still not to market). In contrast, Android evolved from v1.1 in February 2009 to v2.1 in January 2010 (less than a year). Google also wanted a marketplace that could provide easy access for users to find and download apps (e.g., iPhone AppStore). But, unlike the iPhone AppStore, no one company should control what is available in a store.

To achieve this vision in 2005, Google bought a company named "Android." By 2008-2009, Android had:

1) A track record of success from late 2008 (T-Mobile G1)
2) Open Handset Alliance (OHA) – 65 hardware/software/telecom companies along with Google that backed Android
3) A track record of fast innovation for platform versions
4) An Android Market filled with apps

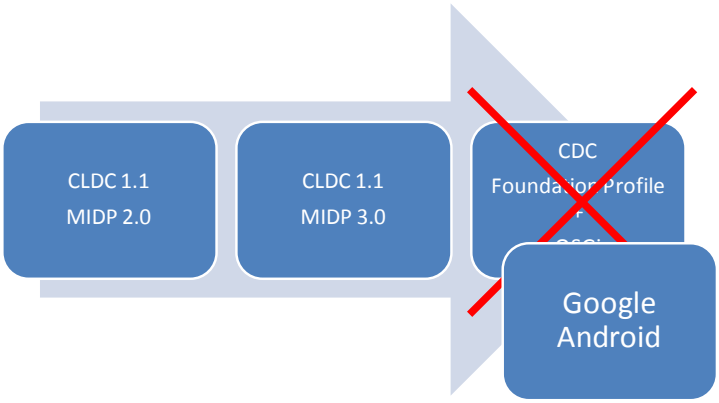


Figure 3 - Apparent Evolution of Mobile Java

Given the virtual overnight success of Android, the relevance of the emerging CDC + Foundation + OSGi was questionable. Android had solved most problems which CDC + OSGi was designed to address, plus added a slew of new functionality that was natively supported on the platform instead of bundled together from a wide variety of JSRs. Android was much faster to market and had buy-in from most major U.S. cell carriers within the first two years of its public existence. As a result, it appears that Android has seized hold of the market that CDC + Foundation + OSGi was designed to address before this Java ME platform was ever fully launched.

Java ME CLDC and MIDP will likely still remain relevant for low-end feature phones (e.g., flip phones), but a reduced feature-set of Android may begin to creep into this market as well. Only time will tell if Android can emulate its success in the smartphone market on lower-end feature phones.

ANDROID OVERVIEW

In Android, Java is the primary programming language. Some native coding to Linux is possible through the Java Native Interface (JNI), but JNI is not encouraged for general Android application development. It is important to note that while Java is the primary Android programming interface, a Java ME application will not run on Android without modification. This is because although Android has several APIs (e.g., Location API) that are similar in design to Java ME, Android does not conform to JSR API specifications. However, given that the same Java programming language is used on both platforms, it is possible to port a Java ME application to Android and keep some of the same source code and application design.

Android Java is much richer than Java ME CLDC and includes support for many data structures (e.g. linked lists, priority queues, etc.), rich Graphical User Interface (GUI) interaction with animations and 2D and 3D graphics, dynamic linking, and SQLite local database storage. The Dalvik Virtual Machine was built for Android and is a common VM implementation across most Android devices, which reduces fragmentation in the general behavior of the platform. Table 1 provides a list of the differences between the Java ME – CLDC and Android platforms.

Eclipse (currently the Galileo version v3.5.2, http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/galileosr2), is primary development environment for Android applications, along

with the Android SDK (http://developer.android.com/sdk/index.html) and the Android SDK plug-in for Eclipse (http://developer.android.com/guide/developing/projects/projects-eclipse.html).

Table 1 - Comparison of Java CLDC vs. Android Features

Feature	Java ME - CLDC	Android
Java language	Java Micro Edition (subset of Java v1.3)	Most of Java Standard Edition v1.5
Platform	Various JSRs define APIs, many restricted by carrier	Full, rich platform and open APIs defines by Google (with OHA)
Multitasking Virtual Machine (MVM)	Only certain phones	All Android devices
Data storage	MIDP RecordStore – byte storage	SQLite database – SQL support
Browser integration	None	Webkit support
3rd party APIs/libraries	MIDP 2.0 – none MIDP 3.0 – Added on via "Liblets, Inter-Process Comm"	Intents, services allow dynamic linking with complete platform support
Signing	Requires carrier approval, carrier-controlled certificate	Must be signed by **user-generated** certificate
Licensing by device manufacturer	Costly, various venders, Sun	Free – Apache, open-source on Linux kernel v2.6

A significant advantage of mobile application development using Android and Eclipse is the presence of standard on-device debugging via a USB cable. Java ME provided only manufacturer-specific debugging capabilities, typically only printing out simple *System.out.println()* statements which programmers can insert into the code. Android includes a much richer set of development and debugging tools that are standard on all Android devices. Over-the-Air (OTA) deployment is also possible by downloading a compiled application in the form of an APK file from a normal web server. An exception to this capability are current AT&T Android devices, as at the time of writing this report Android application can only be deployed to AT&T Android devices via USB debugging cable and tethering to a computer, or via the Android Market. The Android Market is the primary distribution channel for most Android applications, as it provides an environment where users can visit, browse, and purchase applications that have been posted by Android application developers.

KEY ANDROID OBJECT COMPONENTS

- Activity – every User Interface (UI) screen is its own Activity with its own lifecycle (see Figure 4)
- Task – consists of one or more Activities. Tasks may include Activities from outside applications (e.g., 3rd party APIs, libraries, plug-ins) via Intents
- ContentProviders – Can serve data to all applications (e.g., phone book)
- Services – Background activities with no GUI
- Intents – used for dynamic linking to include other Activities and Services in current app

APPLICATION LIFECYCLES

Since Java ME is a "phone-centric" programming model, the lifecycle reflects a very simple design where an application can be "paused" or "started" based on the occurrence of phone calls (Figure 4). As phones became more mature and began running multiple applications in the background, it became apparent that a more complex lifecycle is required that is more "software" centric. Android reflects this more complex

lifecycle design in order to address the different states an application can have when running on a mobile phone.

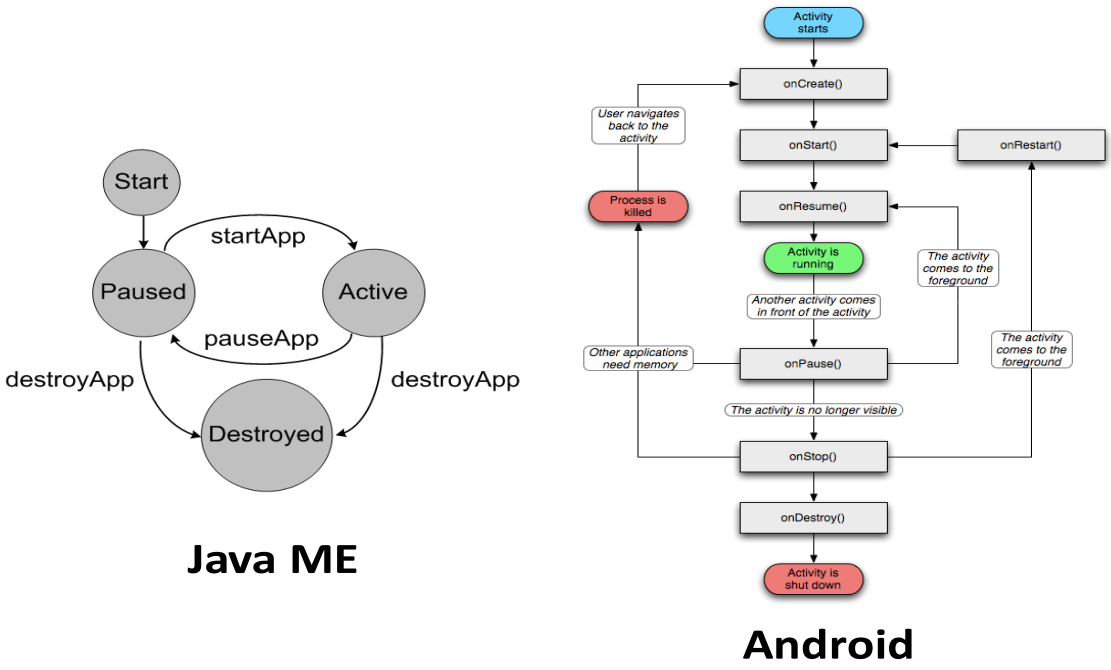


Figure 4 - Android and Java ME Application Lifecycles

DEPLOYMENT DETAILS

Java ME produces two files after a mobile application has been compiled: a Java Application Descriptor (JAD) file containing a text description of various attributes of the application (including permissions), and a Java Archive (JAR) file, which contains the compiled byte code. The main application class which acts as an entry point for execution on Java ME is the MIDlet. Table 2 outlines some of these details.

Table 2 - Deployment Details of Java ME Compared to Android

Feature	Java ME - CLDC	Android
Application descriptions file (for permissions, etc.)	Java application descriptor (JAD) file	AndroidManifest.xml file
Packaged installable files	JAD + JAR	APK file (includes Android Manifest.xml within APK)
Main application class	MIDlet	Activity and tasks (task is defined by one or more activities)

Android produces a single Android Packet (APK) file, which contains an AndroidManifest.xml file holding XML-based descriptions of various attributes of the application (including permissions). The AndroidManifest.xml file can be viewed as an equivalent to the Java ME JAD file. For testing application on device, a developer needs to enable device settings to allow installation of "non-Market" apps, and allow features related to development (e.g., "Settings->Applications->Unknown sources" should be checked, "Settings->Applications->Development" options are (typically) all checked).

One item in Android that is different from Java ME, and perhaps a step backwards, is the Android security model as it is exposed to general users of applications. Android has a user-permission security model based on install-time permissions. In other words, when an Android application is installed, it asks the user if it is okay that the application accesses a variety of information. For example, if the application uses GPS data, the user is asked if they want to allow the application to access the user's "fine (GPS) location" (i.e., fine-grained positioning technology with high precision). If the user replies yes, the application is granted unlimited access to the GPS location of the phone after the application is installed. In other words, the Android platform will never again ask the user whether or not to allow monitoring of the GPS location. While the Android platform does show a satellite icon on the task bar if the GPS position of the phone is being calculated, there is no direct indication what application might be performing these actions in the background. Java ME, on the other hand, provides install-time as well as run-time user permissions. So, even if the user installs a location-based app, the first time the application runs the user will be prompted with a notice that the application is requesting the GPS location of the user, and the user is asked if they want to "Allow Always" (i.e., always allow GPS and do not ask the user again), "Allow this Session" (i.e., allow for the remaining time that this application is running, but then ask again the next time the application is started), "Allow Once" (i.e., only allow the application to request one position from GPS, and then prompt the user again the next time the application wants to request a GPS position, and "Never Allow" (i.e., prevent the application from retrieving the GPS position of the user). These run-time user permissions allow a user to test out an application and decide, based on the functionality of the application, whether or not the user wants to give this application access to sensitive information in future use. As of Android 2.2, the Android user permission model does not allow the user to "try out" an application before deciding whether or not it wants to allow that application permission to access sensitive information. The Java ME runtime user permissions provide an extra buffer of permissions that give the user context within the application of when the application is performing an action related to a sensitive API.

For example, consider an application in which a user can post what they are doing in real-time on the web to be shared with other users. In this application, the user can input a written description of what he is doing (e.g., "At the store"). Immediately after the user inputs this data, the Java ME platform would ask him, "This application wants to retrieve your GPS location. Do you want to allow this?" This context allows the user to infer that the application is going to tie the GPS location to the post that they just entered into the application. If they are comfortable with this, the user can allow the action. In Android, the user is not given any such context for when the application is using sensitive information about the user. In the application example above, as long as the user granted the Android application tracking permissions when they installed the application, the application could turn on GPS as soon as the user launches the application and be constantly tracking them even when they are not posting updates to the website. The user would have no direct notification that this behavior is happening.

It should be noted that Android and Java ME both enforce a wide-variety of security constraints on mobile applications, as do other platforms such as the iPhone, that prevent mobile applications from accessing sensitive information WITHOUT the user's permission. In other words, the only method for an application to gain access to sensitive information on a device on Java ME and Android is for the user to directly confirm that they allow this action. Runtime user security permissions, which Android does not provide, serve as an additional buffer between the user and the application which can help users understand when and how their information is being accessed. In other words, on Android, install-time user permissions are the first defense against unwanted application behavior. Run-time user permissions, which Android does not provide, would serve as a secondary defense to provide the user more context of the use of their permissions.

CONTINUING CHALLENGES IN MOBILE DEVELOPMENT

Even though Android mobile devices are more capable than previous Java ME devices, there are a number of continuing challenges which are common across almost all mobile software development platforms. The following list outlines some of these existing challenges.

- Energy-constraints
- Processors – Even as chipsets become better at managing energy, faster processors require more energy. This can lead to shorter battery life, especially as users keep the phone active for longer periods as they run different applications for different purposes.
- Wireless – using wireless communications (e.g., cellular, Wi-Fi) to send data and voice still consumes battery energy. Therefore, the use of wireless communications should be minimized when possible.
- Location – Currently, the primary two categories of general positioning technologies remain the same: GPS and cell tower/sector location. However, some hybrid networking positioning technologies are emerging, such as Skyhook. Implementation of assisted GPS is changing to gpsOneXTRA to reduce impact on carrier network resources. Other GPS hybrid satellite solutions based on GLONASS, the Russian satellite positioning system, and Galileo, the European Union satellite positioning system, are emerging, and new embedded sensors may provide better support for "dead-reckoning" when satellite positioning fails. However, just like in less capable devices, all of these location technologies use a significant amount of battery energy, and must be managed carefully.
- Simultaneous applications – smartphones now have many applications running in the background to manage email, text, location, messaging, etc. These apps all consume energy as they wake up the device and force the CPU, and possibly other hardware, to power up. While this problem existed in a limited fashion on Java ME, it is very much amplified on Android due to the larger number of average applications a typical user executes.
- Networking support
- HTTP(S), TCP, SSL, and UDP remain the basic supported networking protocols for mobile devices.

THE ANDROID LOCATION APPLICATION PROGRAMMING INTERFACE

The Android Location API is designed to loosely mimic Java ME JSR 179 Location API, although the format of the two APIs is not identical. Some of the main attributes of the Android Location API follow:

- Adds LocationManager class, instead of handling location management within LocationProvider class
- Two mandated LocationProviders for phones:
  - GPS
  - NETWORK
- Android 2.2 adds a "PASSIVE" location provider. This location provider only returns location data to an app if another app directly invokes a location provider, such as GPS or Network. In other words, the passive LocationProvider scavenges location data off of other apps requests, and does not directly impact battery life since it does not trigger location updates.
- Can use Criteria to get "best" LocationProvider, or can ask for specific LocationProvider

- Same LocationListener concept as JSR179
- onLocationChanged(Location location) provides location updates to application
- Similar to JSR 179: locationUpdate(LocationProvider lp, Location location)
- ProximityAlert detects entry and exit from defined circle
- JSR 179 ProximityListener only detects entry into circle
- Rough accuracy of location in meters given using Location.getAccuracy()

The Android platform also supports Android Maps, which allows an application to easily display a map view to a user:

- Supports MapView via "Maps External Library"
- Requires registration for Google Maps API key for each digital key you use to sign application
- One Maps API key for debugging on emulator/device
- One Maps API key for device deployment
- Overlays (points are easily defined, lines and polygons are defined manually)
- Transformation between pixels and geographic projection for detecting interaction with map
- MyLocationOverlay to simply show real-time location

The Android Location API has several advantages over the Java ME JSR179 Location API):

- Open API that anyone can use
- Supports standard & reverse geocoding
- Address->Lat/long, Lat/long->Address
- Better standard support for getting GPS satellite info:
- GPSStatus Listener
- GPSStatus NMEA Listener
- Supports user-created mock location TestProviders
- Adds LocationProvider.sendExtraCommand()
- Allows application to clear or refresh assistance data
- Allows extension of Location API by device manufacturers
- LocationListener can listen for distance changes, in addition to time interval location updates

However, the Android Location API also has some drawbacks when considering the Java ME Location API:

- No distinction between Wi-Fi and cell tower locations in NETWORK LocationProvider
- Location.getProvider() always returns "NETWORK"
- NETWORK LocationProvider calculations are a black box
- No distinction between assisted/unassisted GPS in GPS LocationProvider
- Location.getProvider() always returns "GPS"
- Exact behavior, reliability isn't defined for distance-based LocationListener updates
- No fallback from one LocationProvider to another within same LocationListener
- JSR179 can provide network info when GPS is not available
- No bound on accuracy info for implementations

Since Android Location API is open for use by all application developers, carriers needed to reduce the impact of assisted GPS on network assistance servers. For example, the Java ME JSR179 refreshes assistance data for GPS at least every two hours. Carriers have limited access to that technology as they incur costs for every

refresh of assistance data. Android uses a new technology "gpsOneXTRA," which refreshes assistance data every seven days [4]. It uses predictive ephemeris information from assistance servers which actually predicts satellites positions several days in advance. In theory, accuracy decays as assistance data ages and the potential difference between predicted and actual satellite locations increases. Therefore, different levels of GPS accuracy may be observed in similar devices which use the different types of assisted GPS. Figure 5 shows a Sanyo Pro 200 and an HTC Hero with Google Android 1.5 as they were carried along the same pedestrian path near a building. The Sanyo Pro 200 had a much more accurate representation of the true path.

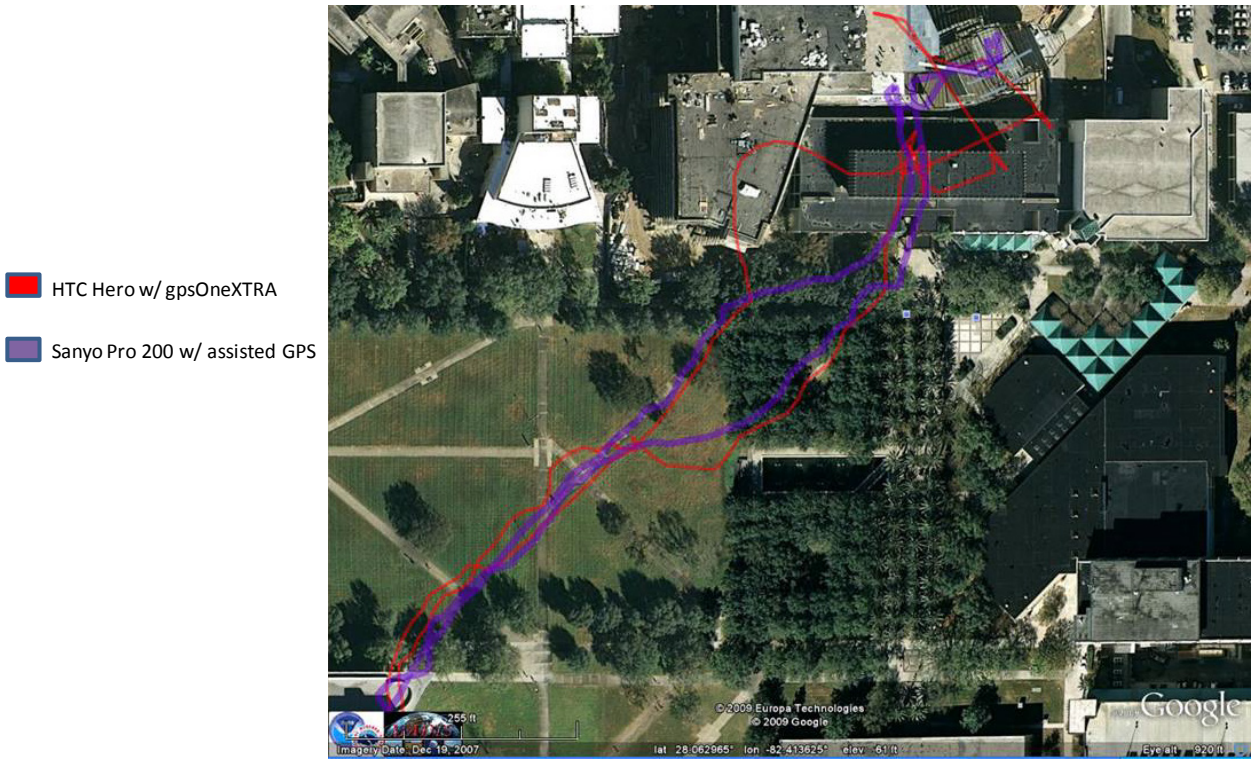


Figure 5 – Sanyo Pro 200 and HTC Hero with Google Android 1.5 Paths

LOCATION-AWARE INFORMATION SYSTEMS CLIENT (LAISYC) FRAMEWORK FOR ANDROID

The TRAC-IT mobile application utilizes the LAISYC framework, which supports intelligent real-time applications for GPS-enabled mobile phones [5]. As cellular data networks achieve broadband speeds, constrained device resources (e.g. battery life) replace bandwidth as a primary limiting factor for mobile applications. IP-based networking protocols, now widely available in Java Micro Edition and Android devices, must be carefully integrated into existing architectures to meet application needs and maximize system efficiency. Evolving location-aware architectures require intelligent clients for low-latency real-time applications and efficient management of device resources, as well as server-side intelligence to analyze past user behavior.

LAISYC is a general framework that can support a variety of location-aware applications with many different requirements. These requirements can differ from time-sensitive second-by-second real-time tracking to more delay-tolerant applications focused on recording the accurate representation of travel paths. Hybrid

applications that support both real-time and delay-tolerant features are also possible, since module settings can be manipulated dynamically to adjust the framework according to real-time application needs.

LAISYC was originally created and implemented for the Java Micro Edition platform, but its design remains very relevant for the Android platform as well. While the server-side portions of LAISYC remain unchanged for any location-aware client, due to the enhanced capabilities of the Android platform and some specification differences from Java ME implementation, details of some LAISYC components on Android devices differs from that on Java ME devices. The following section addresses each of the device-side components of LAISYC and states any implementation differences for Android devices.

POSITIONING SYSTEMS MANAGEMENT COMPONENTS

POSITION RECALCULATION MANAGEMENT

There are no conceptual changes to the framework from Java ME. Since the concept of the LocationListener exists in Android, much of this module to dynamically vary the interval between position requests to the platform remains the same. However, after testing with several Android devices it is apparent that certain devices ignore the "minTime" parameter for the LocationListener, which is supposed to communicate the frequency of position recalculation desired by the application to the platform (Table 3). For example, the Samsung Epic, Samsung Moment, HTC Hero, and HTC EVO update every second for ~9-11 GPS fixes, then sleep for "minTime". Therefore, application developers will have to implement their own timing system for these devices via threads to activate and deactivate the positioning technology (e.g., GPS) every X seconds, or take into account that their application may receive multiple location fixes within the requested interval period. However, other devices, such as the Motorola i1, refresh GPS exactly at the required interval.

Setting "minTime", "minDistance" = 0 in a LocationListener always provides updates as frequently as possible.

On some Android devices, extensive background tracking may require registering/canceling LocationListener, or filtering data from API, to achieve desired behavior of refreshing GPS data as a specified interval.

Table 3 –Android Phone Behaviors for GPS Update Rates

Device	Obeys "minTime"?	Requires extra code
Motorola i1	Yes	No
HTC Hero	Yes*	Optional*
HTC EVO 4G	Yes*	Optional*
Samsung Moment	Yes*	Optional*
Samsung Epic 4G	Yes*	Optional*

* Actual interval between updates may vary significantly,
which needs to be handled properly by application software

POSITION ESTIMATION

No conceptual changes to framework from Java ME. Android supports more location providers than Java ME due to the presence of wi-fi, which provides more data to the Position Estimation module for more complex but potentially more accurate position estimates.

PRIVACY FILTER

No changes to framework from Java ME.

LOCATION DATA SIGNING

No conceptual changes to framework from Java ME. While Java ME developers would have had to use external cryptographic libraries such as Bouncy Castle to implement public and private key cryptography on Java ME devices, Android supports many cryptographic algorithms in the java.crypto package [6]. Therefore, since these algorithms can be optimized for the underlying hardware, it is likely that encryption and decryption will be more efficient on Android devices than on Java ME devices. RSA, AES, DES, SHA1, MD4, MD5, DSA are supported by the Android platform on the HTC Hero and Motorola i1.

COMMUNICATIONS MANAGEMENT COMPONENTS

CRITICAL POINT ALGORITHM

No changes to framework from Java ME.

ADAPTIVE LOCATION BUFFERING

No conceptual changes to framework from Java ME. While the only persistent storage on a Java ME device was the RecordStore, using byte storage, Android supports a more robust SQLite relational database. Android devices also tend to have more on-board memory, therefore expanding the capabilities of the device to temporarily store location data if this module detects a situation where location data communication with the server is likely to fail given the current conditions.

LOCATION DATA ENCRYPTION

No conceptual changes to framework from Java ME. While Java ME developers would have had to use external cryptographic libraries such as Bouncy Castle to implement symmetric cryptography on Java ME devices, Android supports many cryptographic algorithms in the java.crypto package. Therefore, since these algorithms can be optimized for the underlying hardware, it is likely that encryption and decryption will be more efficient on Android devices than on Java ME devices. RSA, AES, DES, SHA1, MD4, MD5, and DSA cryptographic algorithms are supported by the Android platform on the HTC Hero and Motorola i1.

ASSESSMENT OF LOCATION ACCURACY OF ANDROID DEVICES

Even though the Android platform is much less fragemented in terms of varying application behavior on different Android devices, fragmentation in certain areas, such as positioning technologies, does still occur. GPS characteristics can be very different in different devices due to a variety of influences, including:

1. Mobile Device Hardware & Software
    - > GPS hardware sensitivity
    - > Antenna quality and device integration
    - > Assisted vs. Unassisted GPS
        - MS-based vs. gpsOneXTRA
    - > Firmware/software filters
2. Environment
    - > Indoor / Outdoor
    - > "Urban canyons" – areas surrounded by tall buildings
    - > Building materials
    - > Shielding by enclosure (e.g., purse, car)

The best method of determining how a location-based mobile application is going to perform on a mobile phone is to benchmark the accuracy of location (e.g., GPS) by gathering and analyzing GPS data produced by the device.

Figure 6 illustrates possible differences in GPS accuracy, showing GPS data captured from the Samsung Moment and the HTC Evo 4G. The Samsung Moment and the HTC Evo 4G have drastically different accuracy levels, and therefore applications developers should not expect location-based applications to perform similarly on these two devices. However, the HTC Hero, compared ot the HTC EVO 4G in Figure 7 is much closer in GPS accuracy to the HTC Evo 4G. By grouping mobile devices that have similar GPS accuracy levels application developers can expect similar application behavior on those devices.

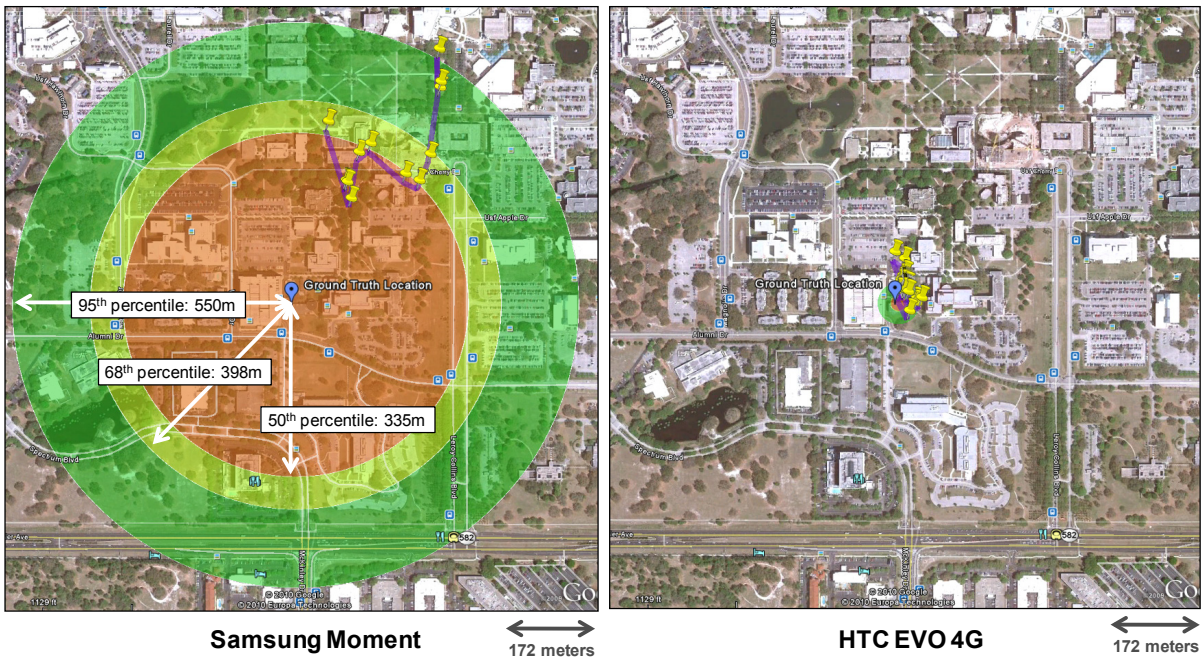


Figure 6 - GPS Benchmarking shows that the Samsung Moment is significantly less accurate than the HTC EVO 4G

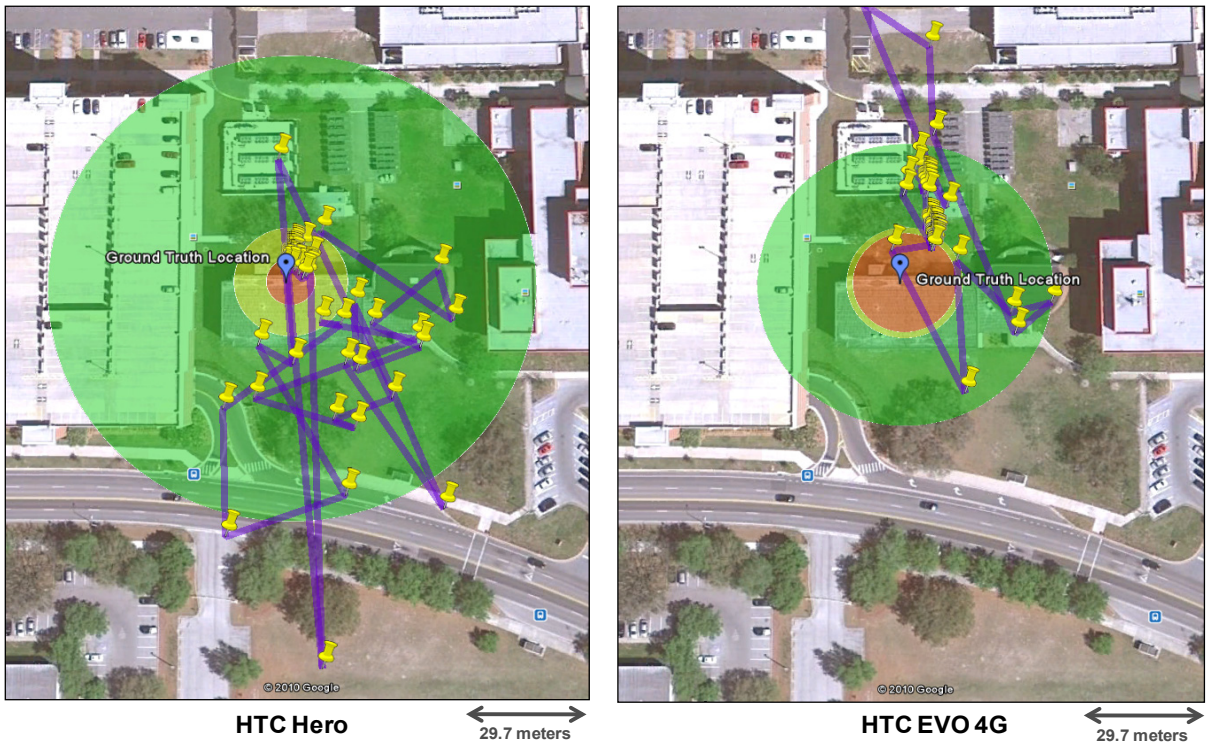


Figure 7 – GPS Benchmarking shows that the HTC Hero and HTC EVO 4G have similar accuracy

REFERENCES

[1] NPD Group. 2010. "Motorola, HTC drive Android to Smartphone OS lead in the U.S.," accessed August 4, http://www.npd.com/press/releases/press_100804.html.

[2] Gartner, Inc., website, http://www.gartner.com/it/page.jsp?id=1434613.

[3] Sun Microsystems. 2009. "Sun Opens 2009 JavaOne Conference with Debut of Java Store, a Global Marketplace for Cutting-edge Java Applications." Accessed July 13, 2009, http://uk.sun.com/sunnews/press/2009/2009-06-02c.jsp.

[4] gpsOneXTRA positioning technology, http://www.qualcomm.com.au/news/releases/2007/070212_gpsonextra_assistance_expand.html

[5] Sean J. Barbeau, Rafael A. Perez, Miguel A. Labrador, Alfredo Perez, Philip L. Winters, and Nevine L. Georggi. In process. "LAISYC – A Location-Aware Framework to Support Intelligent Real-time Applications for GPS-enabled Mobile Phones." IEEE Pervasive Computing.

[6] Legion of the Bouncy Castle website, http://www.bouncycastle.org/.