Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 31 (2008) 1091–1103
computer communications
www.elsevier.com/locate/comcom

Location API 2.0 for J2ME – A new standard in location for Java-enabled mobile phones

Sean J. Barbeau [a,*], Miguel A. Labrador [b], Philip L. Winters [a], Rafael Pérez [b], Nevine Labib Georggi [a]

[a] Center for Urban Transportation Research, University of South Florida 4202 E. Fowler Avenue CUT100 Tampa, FL 33620, USA
[b] Department of Computer Science and Engineering, University of South Florida 4202 E. Fowler Ave. ENB108 Tampa, FL 33620, USA

Available online 31 January 2008

Abstract

Key aspects in realizing the maximum potential of advanced Location-Based Services (LBS) are the standardization and cross-platform availability of an Application Programming Interface (API) for mobile phones that allows access to real-time location information. To shorten the development time of advanced LBS, such an API should also provide essential features such as map user interfaces, geocoding, and navigation to be used as building blocks in the context of larger mobile applications. Using these available services, application developers can focus on building innovative location-aware applications rather than re-creating existing services. This article's main goals are to emphasize the importance of such an API and to describe the Location API for Java 2 Micro Edition (J2ME). This description includes the main features of the current "JSR179-Location API v1.0" as well as the significant enhancements and new services included in the development of "JSR293-Location API v2.0." These new features, illustrated using coding examples, will help software developers create next-generation location-aware J2ME applications.
Published by Elsevier B.V.

Keywords: Location based services; Mobile phone; Standard; Application programming interface; Java; J2ME; JSR179; JSR293

1. Introduction

The availability and pervasiveness of powerful mobile phones along with advances in software development platforms and communication networks promise to change the way we receive and process information. Once large, awkward devices owned only by the wealthy, mobile phones are now becoming accessible to the majority of the world. In 2007, global mobile phone use will hit a record 3.25 billion users, which is over half the world's population [1]. Continued escalation is being driven by growth in emerging markets such as China, India, and Africa. In established markets such as the United States, Europe, and Japan, innovative mobile phone applications that promise to revolutionize the future seem to be emerging daily. However, ubiquitous access to real-time data threatens to put mobile users into information overload. Subscription-based services push out messages that may be insignificant in the context of the user's current position, such as traffic alerts about a local highway when the user is out of town. Location-Based Services (LBS) will likely play an important role in filtering down the deluge of information to relevant items based on real-time or historical location data as well as the relationship to the position of other users or places connected to the Internet. Additionally, LBS are making new applications possible, such as location-tagged photos, location-based social networking, and personalized navigation for transit users or pedestrians.

Due to the important role LBS will serve in future software systems, global commercial demand of such services continues to skyrocket. Market research confirms that the world population of global positioning system (GPS)-enabled location-aware services subscribers will grow from

* Corresponding author. Tel.: +1 813 974 7208; fax: +1 813 974 5168.
E-mail addresses: barbeau@cutr.usf.edu (S.J. Barbeau), labrador@cse.usf.edu (M.A. Labrador), winters@cutr.usf.edu (P.L. Winters), perez@cse.usf.edu (R. Pérez), georggi@cutr.usf.edu (N.L. Georggi).

0140-3664/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.comcom.2008.01.045

12 million in 2006 to a projected 315 million in 2011 with North American growth reaching 20 million users up from 500,000 users in 2006 [2,3]. The software engineering industry must be properly prepared to develop effective mobile location-aware applications to serve this growing market.

The many different device manufacturers and wireless carriers involved in the mobile software development process can create a significant barrier to entry for software engineers new to mobile development. The application developer has the option to utilize multiple programming languages, many of which contain features restricted to certain operating systems. Therefore, an application may work well on one mobile phone, but must be re-created if it is to run on other unsupported models.

A cross-platform language that has emerged on different handsets from many manufacturers and wireless carriers is Java 2 Micro Edition (J2ME) [4]. J2ME emulates the cross-platform nature of its desktop (Java 2 Standard Edition) and server (Java 2 Enterprise Edition) counterparts through the implementation of a subset of the Java programming language and libraries on mobile devices. While not a perfect "write-once-run-anywhere" solution, J2ME is not restricted to a single chipset or operating system. J2ME applications can run on any "Java-enabled" mobile device without substantial changes to the code or structure of the application. Additionally, since many traditional application programmers are familiar with the Java language, the barrier of learning a new language specific to mobile development is removed. These features have increased the popularity of J2ME and have pushed it to the forefront of mobile application development.

An important task for software engineers new to location-aware application development is to understand how to create software that can execute on a mobile phone and access real-time device-location information. Since there are currently over 1 billion Java-enabled mobile phones in the marketplace, understanding how to access location data through J2ME applications is critical to the future of LBS [5]. This article discusses the features of a new J2ME software standard, "JSR293-Location API 2.0" that will help LBS application developers create next-generation location-aware J2ME applications.

2. The J2ME-Location API – How it differs from other "Location" standards

The sheer number of location-related standards can be overwhelming to software engineers. Several popular standards fall under the OpenGIS Specifications [6] as defined by the Open Geospatial Consortium, Inc. (OGC). Geography Markup Language (GML) [7] is an XML-based markup language for geographic data such as latitude, longitude, and altitude. Another standard defined under OpenGIS is Location Service (OpenLS) [8], which is a protocol that defines how location information may be requested and returned from a location data repository such as a database. These standards may be useful for the developer when persisting or transferring location data, but are secondary in importance for developers interested in obtaining real-time location data from a mobile phone. Other standards, such as the European Telecommunications Standards Institute (ESTI)'s "GSM 03.71: Location Services (LCS)" [9] and the 3rd Generation Partnership Project (3GPP)'s "TS 22.071: Location Services (LCS)" and "TS 23.171: Functional stage 2 description of location services in UMTS" describe how location is calculated for handsets and handled internally for a wireless carrier's network and concern mainly telecommunications engineers, not the common software developer. Additional standards exist for "network-initiated" location requests, or a method of using web applications to query a wireless carrier's location server to retrieve a mobile device's position. These protocols include the Open Mobile Alliance (OMA) Location Working Group's Mobile Location Protocol (MLP) [10] as well as a set of telecommunication web services known as Parlay X [11]. These "network-initiated" requests do not require any software to be installed on the device, but also have many limitations including a restriction on the refresh rate of new location data per device, the number of phones that can be queried simultaneously, the characteristics of location-related data (e.g. velocity, heading, etc.), and strict access control by the wireless carrier. Therefore, while useful for some applications, network-initiated location requests are subject to many limitations and are unusable for advanced location-aware applications such as real-time navigation.

The above standards should not be confused with "handset-initiated" location requests, or methods that request real-time location data inside software running on a mobile device. The majority of advanced mobile location-aware applications are driven by the ability to programmatically retrieve real-time location data from the local device. Immediate access to location information allows the handset software to work autonomously from any application server with an extremely low latency between the position calculation and action taken by the software on the device. Handset-initiated APIs also allow the software developer greater control over properties such as the frequency of position re-calculations or characteristics of the data returned (e.g. velocity, heading, etc.) The standardization of this type of on-device API is the focus of this paper.

Accessing location information from software running on a mobile device has changed significantly over the last decade. In the past, location-aware applications were created by tethering a stand-alone Global Positioning System (GPS) device to a laptop or Personal Digital Assistant (PDA). The GPS device would then stream strings formatted according to a standard, such as National Marine Electronics Association (NMEA) 0183 [12], over a serial connection, which could be implemented over a cable or wirelessly via Bluetooth™. The location-aware software application was responsible for listening for these

"sentences" and parsing the data to yield information such as the current latitude, longitude, and altitude.

New mobile phones with embedded GPS chips and carrier-based localization systems have changed the way location information is obtained and increased the complexity of the procedure. For example, Assisted GPS (A-GPS) retrieves data wirelessly from a server that is used to lessen the time required to calculate a position. Alternate positioning mechanisms such as trilateration based on cellular signals can also be used to calculate the position of the device when GPS is not available. In addition, final position calculation can now take place on the device or on a server in the network. To utilize these advanced positioning technologies, software applications must provide certain information to the mobile device such as the requested frequency of updates, requested type of positioning technology, and any other parameters required to provision these technologies. Therefore, modern LBS applications require more interaction with the device than their traditional autonomous GPS-driven counterparts. Since this interaction must be accessible from software running on the mobile phone, this functionality is exposed through an API to make the underlying complexity transparent to the 3rd party software developer.

As J2ME location-enabled mobile phones began to evolve, it soon became evident that simple, interactive, standardized programmatic access to real-time location information on the device would be necessary. Fortunately, J2ME provides a method for continuous improvement of the J2ME platform through the Java Community Process (JCP). Any JCP member, including device manufacturers and wireless carriers, can submit a Java Specification Request (JSR) which outlines the need for an additional optional feature, or API, in the J2ME environment. If other JCP members agree that this JSR is needed, an expert group is then formed to create a specification. As long as a mobile software developer uses J2ME APIs defined by JSRs, the application should run seamlessly across any J2ME platform that fully supports that particular JSR. Following this process, in 2002 Nokia proposed the addition of "JSR179: Location API for J2ME" [13] that would allow J2ME applications to utilize a standardized interface to access location data from the device. An expert group including IBM, Nokia, Symbian, Intel, Motorola, Sony-Ericsson, ESRI, Sun Microsystems, Inc. and others deliberated on the standard and released the final version in September 2003.

3. JSR179 – Location API v1.0

JSR179 allows standardized access for J2ME applications to location data that represents the real-time position of the mobile phone. JSR179 can be compared against similar J2ME APIs including Motorola's OEM Position API [14] for integrated Digital Enhanced Network (iDEN) devices as well as Qualcomm's Java Application Extensions (QJAE) API [15] for their Code Division Multiple Access (CDMA) devices. However, JSR179 is the only standardized J2ME "location-related" API that works across different devices, chipsets, and wireless carriers. Both Motorola's Position API and QJAE are proprietary in format and will only work on Motorola iDEN devices and Qualcomm CDMA devices, respectively, and therefore fall into the "Proprietary APIs" section of the J2ME architecture shown in Fig. 1. A J2ME application utilizing JSR179 to access location information should require few changes to execute across different manufacturers' devices or wireless carriers' networks. As shown in Fig. 1, JSR179 is an optional API in the J2ME architecture and will only be supported on devices with access to positioning technologies such as GPS.

JSR179 can also be compared to the proprietary location API available on Google's new Android platform [16]. Android, an open-source software stack based on Linux, is expected to span several mobile device and chipset manufacturers, and is therefore similar in concept to J2ME. A main strength of Android is its support for modular service-oriented applications and inter-application communication (capabilities also planned for the J2ME platform in MIDP 3 [17] and OSGi [18]). However, since Android is new to the mobile industry and does not yet exist on any commercially available mobile phones, it is not nearly as widespread or as generally accepted as J2ME. Android applications are written using Java, although Android APIs are proprietary in nature and do not conform to J2ME standards. For example, Android's proprietary location API is very similar in functionality to JSR179 but does not conform to the JSR179 specification. Technical and political issues will dictate how widespread and accepted Android becomes and therefore more time is needed to see if it will be a worthwhile platform for mobile application development. Currently J2ME remains the accepted standardized platform for Java development on mobile devices and JSR179 remains the only standardized location API on the J2ME platform.

3.1. Criteria, LocationProvider, and Location

JSR179 is driven by a software object referred to as a *LocationProvider*, which is the source of location data for the application. Additionally, all interactions with the underlying technology that provides location data are handled through this object. Since a mobile phone may support multiple types of positioning technology (e.g. A-GPS, cellular signal trilateration, or cellular base station ID), multiple *LocationProviders* may exist. *Criteria* specified by the application indicate what type of *LocationProvider* is required to meet the application's needs. For example, a navigation application may require highly precise positioning data, it may require speed and course information, and it may allow some kind of cost to be involved (e.g. cost to access the network for A-GPS assist data). A *LocationProvider* with this type of description would be obtained through the following code:

```
Criteria criteria = new Criteria();
criteria.setHorizontalAccuracy(30);//Indicate that required estimated accuracy is 30 meters
criteria.setSpeedAndCourseRequired(true);//Indicate that speed and course are required
criteria.setCostAllowed(true);//Indicate that cost is allowed
LocationProvider lp = LocationProvider.getInstance(criteria);//Request a LocationProvider that meets these Criteria
```

The mobile phone will return a *LocationProvider* that typically is able to meet these criteria, or *null* if a *LocationProvider* meeting these criteria does not exist. This process allows the mobile phone to return the "best" positioning technology based on the application's needs. Therefore, if a weather information application has a horizontal accuracy requirement of 500 m, very coarse location information may be sufficient. In this case, a mobile phone could quickly return the location of the cellular base station's coverage area, or Cell ID, with very little latency and a minimum impact on the phone's battery, since the precision of GPS was not required.

Once a *LocationProvider* has been obtained, the application can get information about the real-time location of the device through the following code:

```
Location location;
location = lp.getLocation(20); //Request a new location object with a timeout of 20 seconds
```

The *Location* object holds important information about the current location, including an encapsulated *QualifiedCoordinates* class that contains information about the estimated latitude, longitude, altitude of the current position as well as estimated horizontal and vertical accuracies associated with the calculated position. This accuracy information is important, since no positioning technology can give results with 100% accuracy. Therefore, the estimated horizontal and vertical accuracy uncertainty values provide a means to estimate how far the calculated position is (in meters) from device's true geographic location. Other important information available through the *Location* object include:

```
public class LocListener implements LocationListener {
...
public void locationUpdated(LocationProvider provider, Location location) {
// This code will be triggered with updated location data at the defined interval
}
...
}
```

- the timestamp when the location was calculated,
- the current speed and course of the device,
- whether or not the position data is valid (i.e. whether or not a latitude and longitude could be determined),
- the location method that was used to calculate the position.

The location method is defined by several different constants including *MTE_SATELLITE* (i.e. GPS), *MTE_TIMEDIFFERENCE* (i.e. cellular-signal based positioning), and *MTE_CELLID* (i.e. Cell ID) that represent particular positioning technologies. Also specified is whether the technology was assisted or unassisted (i.e. *MTA* constants), and whether the technology is network or terminal (i.e. device)-based (i.e. *MTY* constants). Using all of this location information, an application can evaluate its current state and decide if action needs to be taken.

3.2. LocationListener

The *getLocation()* function works well for applications that require location information once, such as software that geo-tags an image or text message. However, many location-aware applications require a constant knowledge of their current location to trigger certain actions. For these types of applications, JSR179 defines a *LocationListener* that allows an application to be updated at a defined interval with new location data. The LocationListener is a Java interface and therefore must be implemented by a class that defines several methods, including the most important method which receives the updated location information from the device:

The *LocationListener* is then set through the following lines of code:

```
LocListener locListener = new LocListener( );
int interval = 4;// Interval between location updates is 4 s
int timeout = 2;// Timeout after location request is 2 s
int maxAge = 2;// Maximum age allowed for a duplicate location value to be returned is 2 s
lp.setLocationListener(locListener, interval, timeout, maxAge);
```

As a result of these instructions, every 4 s the *locationUpdated( )* method of *LocListener* will be fired by the J2ME platform, which will pass in both the *LocationProvider* that calculated the new location as well as the new *Location* which was calculated. Every time that the *LocationListener* fires, the application can take action using that new *Location* information, such as sending the new location data to a server to create a real-time tracking application. The *LocationListener* provides a very simple mechanism for the 3rd party developer to monitor device location and therefore removes the need to implement complex threads and timers.

3.3. ProximityListener

Another common feature of LBS applications is to trigger an event when the device nears a particular location. JSR179 provides a *ProximityListener*, also defined via an interface, which allows the application to register a specific set of *Coordinates* (i.e. latitude and longitude) as well as a proximity radius via the *LocationProvider* class. When it is determined that the device has entered the area defined by the coordinates and radius, the *proximityEvent( )* method of the class that has implemented *ProximityListener* is fired by the J2ME platform. Any code that the application developer has inserted into this method will be called upon this event:

```
public class ProxListener implements ProximityListener{
...
public void proximityEvent(Coordinates coordinates, Location location){
// This code will be triggered when proximity to given coordinates is detected
}
...
}
```

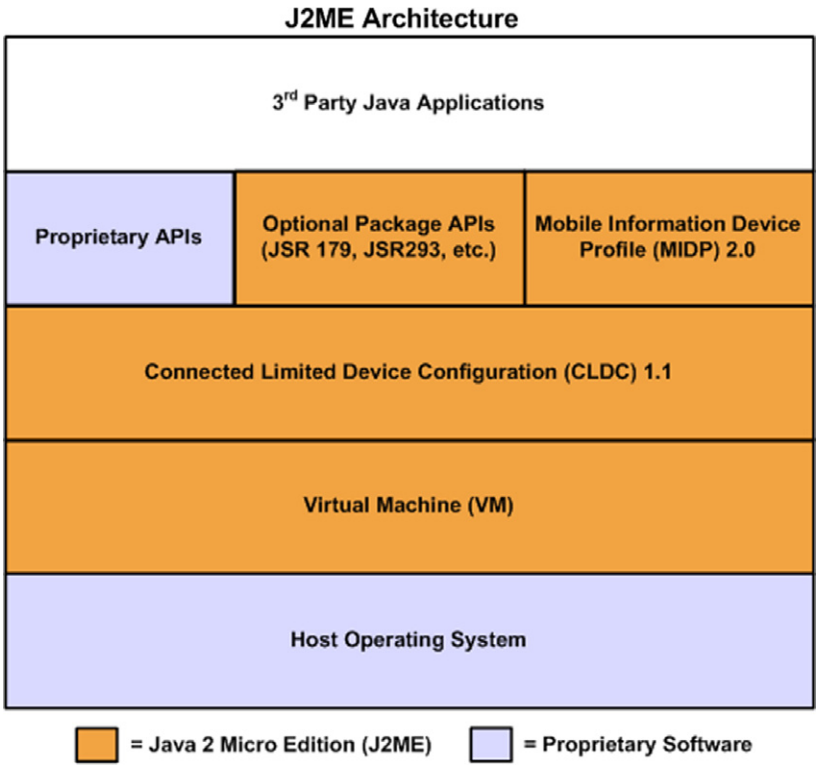

Fig. 1. J2ME Architecture.

3.4. Landmark and LandmarkStore

JSR179 also contains an on-device database known as the *LandmarkStore*, which is a persistent record of *Landmarks*. *Landmarks* are defined by a latitude and longitude along with other human-readable properties such as name, address information, and description. The *LandmarkStore* provides basic functionality of storing and retrieving *Landmarks*, as well as creating categories to which *Landmarks* can be assigned.

4. JSR293-Location API v2.0

While JSR179 is a significant contribution to J2ME as a standardized API to access device location, it does have several limitations. First, the use of JSR179 in real-world scenarios has exposed areas in the API where improvements are needed. Since location-aware applications operate in dynamic environments and with various positioning technologies, it is very difficult to replicate all potential scenarios and use cases on emulators. Secondly, JSR179 was designed as a compact API for early J2ME mobile phones that were severely resource-constrained. With the advancement of mobile phone technology, there are new and more advanced use cases such as real-time navigation, geo-coding, map user interfaces, and the exchange of landmark information between devices. In JSR179, developers would have to code all of these elements from scratch. A new API should provide these elements as building blocks to software developers so they can spend more time developing innovative applications and less time constructing the basic pieces of location-aware software. Both of these issues are beyond the scope of JSR179, and therefore a new JSR is required. Nokia proposed "JSR293: Location API 2.0" [19] in early 2006 to address the needs of advanced location-aware applications. An expert group including Nokia, Motorola, Sprint-Nextel, the University of South Florida, GPS hardware manufacturer SiRF, Samsung, Cingular, Sun Microsystems, Inc., Sony Ericsson, European wireless carriers Orange and Telecom Italia, and China Mobile Communications Co. Ltd. has been deliberating on this standard, with JSR293 currently in the public review stage. JSR293 is significantly more complex than JSR179 and a thorough discussion is beyond the scope of this article. Therefore, the following sections focus on two major areas: critical improvements of elements from JSR179 Location API v1.0 and significant new features in JSR293 Location API v2.0.

4.1. Improvement of features from v1.0

One of the challenges encountered when improving existing features of JSR179 through JSR293 is that JSR293 must be fully backwards-compatible with JSR179. In other words, an application developed to run on a handset with JSR179 must exhibit identical behavior when running on a handset with JSR293. JSR179 is well-designed in general, so this issue arises only in a few cases, primarily the concept of *Criteria* in *LocationProvider* selection, the functionality of the *ProximityListener*, and the abilities of the *LandmarkStore*. Solutions implemented in these areas were carefully designed to allow backwards compatibility with legacy JSR179 applications while providing much needed improvements for future applications.

4.1.1. Criteria and LocationProvider

Criteria were used in JSR179 to hide the complexity of different location technologies from the 3rd party software developer and to prevent an application from binding its functionality to a specific technology. Ideally, developers do not have to know anything about the underlying positioning technologies and simply specify the needs of their application through the *Criteria* object to the *LocationProvider*, which then returns the "best" technology for this circumstance. However, the ambiguity of this design leaves an opening for vastly different behaviors on different JSR179 devices. For example, if conflicting values are set in the *Criteria* object such as a $PREFERRED\_POWER\_CONSUMPTION = LOW$ and $HORIZONTAL\_ACCURACY = 5$ meters, the *LocationProvider* of one implementation may choose A-GPS (thereby giving preference to the desired accuracy) while another implementation of JSR179 may choose Cell-ID (thereby giving preference to power consumption). This varying behavior forces the application developer to construct software that handles a variety of cases on different JSR179 devices based on the technology type returned by that platform. Given the sheer number of permutations of criterion settings compounded by new permutations for different platforms and the increasing number of available positioning technologies on mobile phones, this issue defeats the cross-platform concept of J2ME as well as the desired simplicity of hiding implementation details from the developer.

Two solutions to this problem have been constructed in JSR293. First, the *Criteria* object now supports setting a priority from 1 to N for each criterion, with the lowest numbered criterion (1) having the highest priority. This reduces ambiguity by allowing the application developer to clearly communicate to the device what criterion is the most important for his or her application. Secondly, a new method of retrieving a *LocationProvider* is supported via a new *getInstance()* function that does not rely on *Criteria*. Instead, an array of prioritized location method constants, or technology types defined in the *Location* object, can be input to the API to specify the desired fallback order of positioning technologies to be used by the *LocationProvider*. For example, a tracking application may wish to use GPS, and if GPS is not available use cell signal-based positioning, and if cell signal-based positioning is not available use Cell ID:

```
int[] preferredLocationMethods = new int[3];
preferredLocationMethods[0] = MTE_SATELLITE;//First preference of positioning technology
preferredLocationMethods[1] = MTE_TIMEDIFFERENCE;//Second preference of positioning technology
preferredLocationMethods[2] = MTE_CELLID;//Third preference of positioning technology
LocationProvider lp = LocationProvider.getInstance(preferredLocationMethods, parameters); //Get the LocationProvider
for given preferred location technologies
```

This method will be a welcome addition for developers who understand the pros and cons of different positioning technologies so they can clearly communicate their application needs to the device while still maintaining platform independence through the use of *MTE*, *MTA*, and *MTY* constants already defined in JSR179. By allowing applications to explicitly request a certain type of positioning technology, application developers can easily create their own optimized application-specific positioning request sequences by simply requesting a new *LocationProvider* during application execution. For example, if an application knows that the device is likely to be indoors where GPS will fail at certain times of the day, it can directly request a network-based cell signal trilateration calculation instead of requesting a GPS fix and waiting for it to failover to a different type of positioning technology. While this was technically possible in JSR179, it involved keeping track of complex proprietary *Criteria* mappings to technology types. Allowing smart application-level switching of positioning technologies, both network and terminal-based, will likely be important for future mobile devices that will have an increased number of different wireless capabilities, including cellular, Wi-Fi, Bluetooth, ZigBee, and Ultra-Wideband, as well as various positioning options available within a single wireless technology. The *LocationProvider* is able to hide network communication used to implement assisted, network-based, or hybrid positioning technologies from the application and allow the developer to concentrate on simply requesting the right technology at the right time. In a related improvement to *LocationProviders*, a new location method technology constant *MTE_INERTIALNAVIGATION* is defined in JSR293 to support positioning via "dead-reckoning" when the primary positioning technology (e.g. GPS) is not available. This feature will allow mobile devices with embedded accelerometers or other technology to provide estimated positioning updates to the application in harsh wireless environments (e.g. indoors or underground) where the primary positioning technology has lost the ability to provide accurate location data.

This new method of requesting *LocationProviders* in JSR293 also allows the input of a "parameters" string as a secondary argument that can define values necessary for a specific platform to function, such as a location server IP address and port number ("*SERVER_IP = XXX.XXX.X.XX; SERVER_PORT = Y*") for network-assisted or network-based positioning methods. JSR179 does not support any parameter strings and therefore this functionality had to be accomplished through proprietary extensions to JSR179. The parameters string also allows devices and platforms to create settings and fallback patterns optimized for specific types of applications on their platform. For example, a tracking application could simply specify "*APPLICATION_TYPE = TRACKING*" in the parameters string and the device would return a *LocationProvider* optimized for tracking on its platform.

4.1.2. ProximityListener

The JSR179 *ProximityListener* also needed improvement in several areas. First, when the JSR179 *ProximityListener* was registered for a particular circular area defined by a point and a radius, the application had to wait until the implementation fired the *proximityEvent()* method before taking action. There was no method to set or discover the periodic refresh rate used to check the real-time position against the registered position or any other kind of communication from the implementation if the proximity was not detected. Therefore, if the method did not fire when expected while testing on a real device, debugging the application was extremely difficult. Causes for failure could vary from a momentarily dropped GPS signal, to a temporary loss in GPS accuracy, to a low refresh rate that could not capture the device's entrance into the circular area prior to its exit. Secondly, the *ProximityListener* lacked the ability to detect an exit from a particular area, as it only allows detection of an entrance into the area. Lastly, the only geographic shape supported by the ProximityListener was a circle, represented by a point and a radius. There are many use cases where the detection of a long rectangle area or an irregular polygon defined by multiple points is desired.

Proximity detection has been greatly enhanced in JSR293. For example, an interval and timeout value can be defined by the application when the *ProximityEnterAndExitListener*, which has replaced the JSR179 *ProximityListener*, is registered. At this interval, a new *locationUpdated()* method is called so that the application can tell how frequently the device is checking proximity to the registered location. Information about the positioning technology being used by the implementation

for proximity detection is also provided so application developers can better troubleshoot detection failures. Additionally, detection of departure from a specific area is now supported through the *ProximityEnterAndExitListener*. Finally, JSR293 allows the registration of different types of geographic areas, including *CircularGeographicAreas*, *RectangleGeographicAreas*, and *PolygonGeographicAreas* for all JSR293 features that handle geographic areas, including the *ProximityEnterAndExitListener*. For example, any polygon that is represented in a server-side GIS database can now be directly transferred to a mobile phone, constructed into a *PolygonGeographicArea*, and registered with the *ProximityEnterAndExitListener*. This feature alone is a huge leap ahead for precise LBS using areas that could not be easily represented with a center point and radius.

4.1.3. Landmark and LandmarkStore

The *Landmark* and *LandmarkStore* also received an overhaul in JSR293 to lend better support for future LBS applications. The *Landmark* now features new fields such as author, identifier, and timestamp and the *LandmarkStore* has methods to search the store based on these properties. Wildcard searches are also allowed. This expanded search ability should make it much easier for applications to synchronize on-device landmarks with a database server and keep track of record updates. Additionally, the Landmark now has a geographic area in addition to a simple latitude and longitude to better differentiate the coverage of landmarks, such as the concept of the "University of South Florida" as a landmark versus "Bus Stop 27". An "ExtraInfo" field has also been added as a dedicated field for the storage of application-specific data related to landmarks, which can be used for landmark properties that do not directly map to existing JSR293 *Landmark* fields. Generic global landmark categories are now defined and localized for a better end user experience and consistency among LBS applications. To better support a large on-device database that may be accessed by multiple J2ME applications simultaneously, JSR293 also adds a *LandmarkStoreListener* that can inform the application when the contents of the *LandmarkStore* are modified by another thread or application. In an effort to improve the privacy and security of location-aware applications, *LandmarkStore*s can now be declared as "private." Private *LandmarkStores* are only accessible by the application that created the *LandmarkStore*, in contrast to JSR179 *LandmarkStores* which were always shared among all J2ME applications on the device. Private *LandmarkStores* will allow secure LBS applications to easily store sensitive location-related data (e.g. user-generated locations, business customer locations, etc.) on the device without worry that the information could be easily accessed by other applications. Public *LandmarkStores* are still allowed and encouraged in JSR293 for publically available landmark datasets that can be shared among applications, such as a points-of-interest database.

4.2. Features new to v2.0

JSR293 also includes many completely new features that will significantly accelerate LBS software development for handsets as well as improve interoperability between mobile devices and server-side systems. These features provide basic location-aware services to 3rd party J2ME applications that developers would otherwise have to manually code, thus drastically shortening the development time for advanced location-aware applications. The following features are presented in the context of a "Tourist Guide" mobile application designed to download tourist attractions for a city, show these landmarks to the user on a map, and then guide the user on a tour of the city using real-time navigation.

4.2.1. Landmark exchange formats

An important new feature in the area of interoperability is "Landmark Exchange Formats," implemented in an *ExchangeFormatFactory*. In JSR179,*Landmarks* existed primarily inside a single device and were created and accessed by the same single application. This created a "walled garden" of landmarks that could not easily be shared among other mobile phones or desktop applications, or downloaded from websites. In JSR293, the *ExchangeFormatFactory* can support multiple exchange formats that promote interaction between mobile phones and with desktop or server applications. This ability significantly increases the usefulness of LBS applications that handle landmarks, allowing users to share favorite places via email or text-message and allowing applications to share landmarks via communication on specific ports. Additionally, this method allows exchanges with non-Java devices as well and promotes general interoperability between location-aware systems. Two exchange formats are included in the JSR293 specification as examples of different formats that serve various application needs: vCard (a light-weight format derived from the popular vCard standard) and LMX (an XML-based format). These formats will likely be implemented by most device manufacturers and will be available for 3rd party developers to utilize on most JSR293-compliant devices. Other formats, such as KML or GML, may also be included by device manufactures. Additionally, the inclusion of an *ExchangeFormatHandler* interface allows 3rd party developers to create their own formats in addition to the formats that device manufacturers decide to implement. These capabilities make landmark exchanges virtually limitless in scope and interoperability. The following example from the "Tourist Guide" application shows the simplicity of downloading popular tourist attractions (i.e. landmarks formatted in the vCard format) from a website and importing them into a *LandmarkStore*:

```
// Get format handler for vCard format
ExchangeFormatHandler handler = ExchangeFormatFactory.getExchangeFormatHandler("text/x- vcard");
// Get reference to landmarkstore used to store landmark data for this application
LandmarkStore store = LandmarkStore.getInstance
("MyFavoriteLandmarks");
// Open stream connection to website to download landmarks
StreamConnection conn = (StreamConnection) Connector.open("http://www.tampabaytourism.com/tourist_
attractions.vcf");
InputStream is = conn.openInputStream();
// Import landmarks into a specific category in the landmark store
handler.importLandmarks(is, store, "Tampa Tourist Attractions", true);
```

4.2.2. Geocoding

In JSR293, three categories of services are derived from new *ServiceProvider* and *ServiceListener* superinterfaces: Geocoding, Maps, and Navigation. The use of interfaces allows device manufacturers and software developers the opportunity to explore many different types of *ServiceProviders* in addition to the main three groups listed above, and therefore adds significant expansion opportunities to JSR293.

Geocoding, or the translation of address information into latitude and longitude, as well as reverse geocoding, or the translation of latitude and longitude information into an address, are both supported through a *GeocodingServiceProvider*. This feature is very important for interactions with the user, since positioning technologies provide coordinate information (i.e. "Latitude = 28.058425, Longitude = −82.416170") which is not meaningful to the end user, and users provide location information in the form of an address (i.e. "4202 E. Fowler Ave. Tampa, FL 33620") which is not useful to software and positioning technologies. Geocoding and reverse geocoding bridge the gap between the end user and the positioning technology and enable fluid user interaction with applications, as well as enable the other types of services by supplying location information to software in a usable format. In the "Tourist Guide" example, the user could type in an address of a tourist attraction recommended by a friend and the application could use the *GeocodingServiceProvider* to generate a latitude and longitude for that address. This data could then be used to created a new *Landmark* that would then be added to the *LandmarkStore* of existing "Tampa Tourist Attractions."

4.2.3. Map user interfaces

Another important feature of many location-aware applications is the ability to display location information in the form of a map. JSR293 features a *MapServiceProvider* that will allow 3rd party software developers to rapidly build solutions that include rendering map information, including landmarks and routes, to the mobile phone screen. This ability will have various levels of control. So, if the application simply wants to show something on a map to the user, a single function call will hand over control to the *MapServiceProvider*, which will then show the map to the user in its default format. In the following example, the "Tourist Guide" application shows the user all "Tampa Tourist Attractions" that were imported into the *LandmarkStore*, including the attraction recommended by a friend:

```
// Get list of landmarks from store
Enumeration enum = store.getLandmarks("Tampa Tourist Attractions", null);
// Create an array of landmarks to pass into map method
Landmark[] landmarks = new Landmark[10];
while (enum.hasMoreElements()) {
   landmarks[i] = (Landmark)enum.nextElement();
   i++;}

// Find map service providers
ProviderCapabilities[] providers = ProviderManager.findServiceProviders(ProviderManager.MAP, null);
// Connect to MapServiceProvider
MapServiceProvider mapProvider = (MapServiceProvider)ProviderManager. connectToServiceProvider(providers[0].
getName(), ProviderManager.MAP, null);
// Request MapServiceProvider to display landmarks on the map
mapProvider.displayMap(null, landmarks, null, null,
MapServiceProvider.MAP_TYPE_REGULAR, 0, false, false, this);
```

If the application wants more control over what is rendered to the screen, there is also an option to retrieve a *BaseMap* from the *MapServiceProvider* and then render this information, along with various *MapOverlays*, to a graphics object defined by the application. Through the manipulation of *MapOverlays* the application could add or remove certain related features on a map, such as showing only restaurant or museum attractions to the user. A *MapServiceListener* is also defined to allow applications to react to changes on a map such as the user selecting certain landmarks or features. Users will also be able to select different map views such as regular, satellite, and hybrid that are commonly found on desktop map interfaces. Having a standardized map view will provide a consistent feel to JSR293-compliant applications and will encourage map providers to create new and advanced features that build on the basics outlined in JSR293. This feature vastly simplifies the work of the 3rd party software developer as a map can be rendered in just a few lines of code.

4.2.4. Navigation

Navigation is perhaps one of the most significant new additions to JSR293. The ability to easily add real-time guidance and directions to any mobile applications will push the LBS industry forward and spur new types of location-aware applications. The *NavigationServiceProvider* supports two primary modes of operation in JSR293. If applications simply want to use a turn-key navigation solution, it can make a simple call and allow the service provider to take control of the user interface as well as application flow to navigate to a particular location. For applications that want to handle the navigation logic and take more control over the navigation process, the *NavigationServiceProvider* can act as a route planner that returns a new *Route* object that contains all the information an application needs to navigate. Both methods allow the use of *NavigationServicePreferences*, which can specify anything from a mode of transportation preference (including walking and public transit), a desire to obtain the route with the least traffic, certain geographic areas to avoid, or voice or text directions. The *Route* object, which consists of *RouteSegments*, allows more queries so that an application can determine whether it is suitable for its use, including whether costs such as toll roads are involved, total travel time, and total distance. If the application chooses to control the actual navigation process, the *RouteSegments* contain the instructions as well as the locations where the instructions should be given. These features allows the 3rd party application developer to focus on creating a better navigation application, or integrating navigation into software in new ways, instead of worrying about the logistics of planning a route and getting the geographic data to the cell phone. The Route object can also be created by the 3rd party application, which allows 3rd party developers to combine Routes with the landmark exchange format to allow the exchange of favorite hiking trails, bike paths, bus routes, or scenic tours. In the following example, the "Tourist Guide" application uses the "Tampa Tourist Attraction" landmarks to generate a NavigationServiceProvider that will take the user on a guided tour of all the landmarks using real-time turn-by-turn directions.

```
// Get list of landmarks from store
Enumeration enum = store.getLandmarks("Tampa Tourist Attractions", null);
// Create an array of coordinates to pass to NavigationServiceProvider as waypoints
Coordinates[] coords = new Coordinates[10];
Landmark lm;
while (enum.hasMoreElements()) {
   lm = (Landmark)enum.nextElement();
   coords[i] = lm.getQualifiedCoordinates();
   i++;}

// Find navigation service providers
ProviderCapabilities[] providers = ProviderManager.findServiceProviders(ProviderManager. NAVIGATION, null);
// Connect to NavigationServiceProvider
NavigationServiceProvider navProvider = (NavigationServiceProvider)ProviderManager. connectToServiceProvider(providers[0].getName(), ProviderManager.NAVIGATION, null);
// Set Navigation Service Preferences
NavigationServicePreferences prefs = new NavigationServicePreferences();
prefs.setRouteType(ROUTE_SCENIC);//Set preferred type of route as the one that is the most scenic
prefs.setTransportMode(TRANSPORT_CAR);//Set preferred transportation mode as car
prefs.setNavigationType(NAVIGATION_TYPE_REAL_TIME); //Directions to destination will be given in real-time
prefs.setInstructionType(INSTRUCTION_VOICE);//Set instructions to be given to the user via voice announcements
prefs.setMapShown(true);//Requests that a map is shown to the user while traveling
prefs.setLocationShown(true);//Requests that the user's real-time location is shown on the map while traveling
```

```
navProvider.setServicePreferences(prefs);//Set preferences in NavigationServiceProvider
// Create new object that "listens" for updates from the NavigationServiceProvider
NavListener listener = new NavListener()
// Request NavigationServiceProvider to navigate through these waypoints – Nav. Provider will take over screen here
navProvider.navigate(coords, listener);
```

Another class *NavListener* must be defined by the application that implements the *NavigationServiceListener* interface. The *NavigationServiceProvider* will call methods in the *NavigationServiceListener*, and thereby trigger actions in the application, when certain events occur. These events include encountering a waypoint or reaching a final destination. Below is a simplified version of a *NavListener* class:

```
public class NavListener implements NavigationServiceListener{
...
public void waypointReached(Coordinates coordinates){
// This code will be triggered when the waypoint defined by these Coordinates is reached
}
public void destinationReached(){
   // This code will be triggered when the final destination is reached
   // Control of screen would be handed back to application after this method exits
}
...
}
```

In the above scenario the user would be shown a navigation-mode screen on their mobile phone with a map showing the user's current location as well as the tourist attractions, referred to as waypoints. The JSR293 platform would then deliver audible instructions (e.g. "Turn right in 150 feet") to the user to guide them through to waypoints. Each time the user reached a waypoint, the JSR293 platform would call the NavListener.waypointReached() method and pass in the coordinates of the detected waypoint. The application could then compare this location against the list of tourist attractions to determine which attraction the user was near, and perform an action such as showing a picture of the attraction or announcing its name. Navigation would continue until the last attraction was reached, at which would the NavListener.destinationReached() method would be called by the JSR293 platform. The "Tourist Guide" application could then take action to wrap up the tour experience.

For applications that wish to take more control over the actual navigation process, below is the code that will request *Route* information from the *NavigationServiceProvider*:

```
Route route = navProvider.getRoute(coords);
```

Once the application has the *Route*, it can access the *RouteSegments* that compose the *Route* and deliver notifications to the user according to the supplied information. 3rd party applications can then utilize route information and integrate it into the application in many different ways.

As mentioned earlier, the *MapServiceProvider* and *NavigationServiceProvider*, along with all other objects in JSR293, allow the use of more complex geographic objects beyond simple circles, including *RectangleGeographicAreas* (i.e. rectangles) and *PolygonGeographicAreas* (i.e. irregularly-shaped polygons). These geographic objects can be used for many advanced features in JSR293, including requesting an area to be displayed on a map or defining geographic areas to be avoided when requesting a *Route* from a *NavigationServiceProvider*. Since few geographic features in the real-world can be accurately represented using only circles, this seemingly simple characteristic in JSR293 should make the API more adaptable to real-world problems.

5. Implications for stakeholders

JSR293 has many implications for various stakeholders, including device manufacturers, wireless carriers, geographic data providers, and the 3rd party application developers.

Device manufacturers are primarily responsible for implementing the hardware and software capabilities of the handsets, as well as coordinating with wireless carriers when assisted and network-based positioning technologies are supported. JSR293 will create some work for device manufacturers that are responsible for implementing JSR293 on the device, since the implementation of new capabilities exposed in v2.0 are not trivial. JSR293 should provide enough guidance to device manufacturers to ensure consistency between different JSR293 platforms, while still allowing platforms to implement various features that improve performance or add value on their particular

platform. Since the basic *ServiceProvider* and *ServiceListener* are defined as interfaces in JSR293, there is a potential for device manufacturers and/or wireless carriers to create new location-aware services based on these objects. Even though these services would be proprietary extensions to JSR293 (i.e. not available on all JSR293 implementations), there may be some providers that would take the initiative to create and promote their own types of services for their specific platforms as a competitive advantage. Since the basic Java *ServiceProvider* interface already exist in JSR293, this would allow a common base with other JSR293 service providers and familiarity to JSR293 programmers, thus promoting rapid adoption of extensions to JSR293.

Coordination with geographic data providers must also take place in order to make certain JSR293 services (e.g. *GeocodingServiceProvider*, *MapServiceProvider*, and *NavigationServiceProvider*) available to the developer. Geographic services are of little value without up-to-date geographic information defining road networks and points-of-interest, so GIS databases must be maintained and provisioned for access from mobile devices through JSR293 service providers.

Third party application developers are responsible for creating unique applications that will make efficient use of device resources, the cellular network, and geographic services exposed in JSR293 in order to create advanced location-aware applications that will generate revenue for all parties involved, either directly or indirectly through increased use of the mobile device and cellular network. JSR293 should facilitate the integration of data from multiple sources in order to add value to applications. For example, an application could download a list of landmarks that represent real-time areas of dense traffic and feed these areas into a *NavigationServiceProvider* as places the application would like to avoid when generating a route. Multiple sources of information and services allow programmers to create "mash-ups" for location-aware applications that combine data from providers operated by multiple organizations. Each of these providers may have a cost, or they may be free. JSR293 allows a mechanism for the 3rd party application developer to query service providers and determine the impact a provider might have on the end user (i.e. will it generate network traffic billable to the user, is there fee for the service, etc.).

The availability of different types of JSR293 *ServiceProvider* objects directly accessible to application developers on specific J2ME implementations will likely be decided by device manufacturers and wireless carriers in coordination with commercial partners. It is hoped that all parties involved in realizing JSR293 will support an open system that will also allow 3rd party software developers to create "middleware" *ServiceProviders* that could be registered and accessed through mobile applications utilizing JSR293. An open environment would likely allow free versions of a provider with only basic functionality as well as a more advanced provider with advanced services at a subscription or per-use fee, thereby stimulating competition and ensuring that costly service providers add an appropriate value for the end-user of the application. Closed systems accessible only to commercial partners may stifle innovation and slow down the adoption of such services.

It is also hoped that device manufacturers and wireless carriers will provide an open-access model that would allow any properly registered 3rd party application developer to create and test applications that utilize JSR293 on real mobile phones. Development of location-aware applications using emulators is limiting at best, since application performance and behavior can only be properly evaluated using the actual hardware device. Such open access would fuel the rapid development of many different types of location-aware mobile applications, thus accelerating the adoption of such software by mobile phone users. Google's Android platform is based on these open-access concepts, and it is hoped that parties involved in the implementation and distribution of JSR293 devices follow suit.

6. Conclusions

"JSR293: Location API 2.0" is a new standard for location-aware J2ME mobile phones that improves v1.0 features ("JSR179: Location API") and adds many new advanced location-aware capabilities. At a basic level, JSR293 allows Java applications running on board a mobile device to access real-time location information from a positioning technology such as A-GPS. JSR293 also promotes the rapid development of advanced location-aware applications by exposing powerful functionality to 3rd party software developers. These capabilities include sharing information about favorite locations or routes through new Landmark Exchange Formats, viewing the location of nearby friends using on-screen maps in a new *MapServiceProvider*, and providing users with customized navigation services through a new *NavigationServiceProvider*. To support these services a new*GeocodingServiceProvider* for geocoding and reverse geocoding is supported, as well as support for advanced geographic areas such as rectangles and polygons in addition to simple circles. While mobile phones are the primary target of JSR293, any mobile device that implements J2ME's Connected Limited Device Configuration (CLDC) 1.1 or Connected Device Configuration (CDC) can utilize this API. This covers a large range of mobile devices, including Personal Digital Assistants (PDAs), sensors in wireless sensor networks, and embedded systems in vehicles.

JSR293 is currently in the public review stage with the final release expected Q2 2008. While the core functionality of this API is expected to remain the same, there is still a possibility for changes in details until the final release date. As a result, it should be noted that this article is illustrative in purpose, and JSR293 is subject to change. To view the specification as it currently exists, or to submit comments to be considered by the expert group, please visit the JCP

webpage for "JSR293: Location API 2.0" at http://jcp.org/en/jsr/detail?id=293. Comments and suggestions are strongly encouraged, as the API can be strengthened by input from anyone working in the area of location-aware information systems.

The next generation of mobile applications will likely all be "location-aware" in some manner to protect mobile device users from information overload. Mobile application developers are beginning to understand that location in itself is not a killer application, but a killer attribute. Location, when used appropriately, can be a significant indicator as to what information is currently relevant to the user. As a result, complex LBS systems and applications are beginning to emerge that provide traditional services such as multimedia messaging with the added attribute of location. JSR293 will help push this trend forward and allow mobile application developers to quickly and easily integrate advanced location-aware features into future applications. The end products will be location-aware applications that deliver information to you based on who you are, what you want, when you want it and, finally and perhaps most importantly for mobile applications, *where* you are.

References

[1] K. Ridley, Global Mobile Phone Use To Hit Record 3.25 Billion, Reuters, June 27, 2007. © Reuters 2007. Available online at http://www.reuters.com/article/email/idUSL2712199720070627.

[2] ABI Research, GPS-enabled Location-Based Services (LBS) Subscribers Will Total 315 Million in Five Years, New York, September 27, 2006. © 2007 ABI Research. http://www.abiresearch.com/abiprdisplay.jsp?pressid=731.

[3] ABI Research, Personal Locator Services to Reach More than 20 Million North American Consumers by 2011, New York, November 28th, 2006. © 2007 ABI Research. http://www.abiresearch.com/abiprdisplay.jsp?pressid=766.

[4] Sun Microsystems, Inc., The Java ME Platform – the Most Ubiquitous Application Platform for Mobile Devices, http://java.sun.com/javame/, © 1994-2007 Sun Microsystems, Inc.

[5] Sun Microsystems, Inc., Java Technology: The Power of the Java Brand, http://java.com/brand. © 2007 Sun Microsystems, Inc.

[6] Open Geospatial Consortium, Inc., OpenGIS® Specifications (Standards), http://www.opengeospatial.org/standards. © 1994-2007 Open GeoSpatial Consortium, Inc.

[7] Open Geospatial Consortium, Inc., Geography Markup Language, http://www.opengeospatial.org/standards/gml. © 1994-2007 Open GeoSpatial Consortium, Inc.

[8] Open Geospatial Consortium, Inc., Location Service (OpenLS): Core Services, http://www.opengeospatial.org/standards/olscore © 1994-2007 Open GeoSpatial Consortium, Inc.

[9] European Telecommunications Standards Institute, http://www.etsi.org/. © ETSI 2004.

[10] Open Mobile Alliance (OMA) Location Working Group, TS 101 – Mobile Location Protocol Specification – Version 3.0.0, http://www.openmobilealliance.org/tech/affiliates/lif/lifindex.html © 2007 Open Mobile Alliance Ltd.

[11] The Parlay Group, Parlay/OSA Specifications, http://www.parlay.org/en/specifications/. © Parlay 2007.

[12] National Marine Electronics Association, NMEA 0183 Interface Standard, http://www.nmea.org/pub/0183/ © NMEA 2003.

[13] Sun Microsystems, Inc., Java Specification Request (JSR) 179: Location API for J2ME™, http://jcp.org/en/jsr/detail?id = 179. © Sun Microsystems, Inc. 2007.

[14] Motorola, iDEN J2METM Developer's Guide 2005, Version 1.98, © Motorola, Inc. 2005, pp. 487–498.

[15] Qualcomm, Qualcomm Java Application Extensions, http://www.cdmatech.com/products/qvm_qjae.jsp. © Qualcomm 2007.

[16] Google, Android – An Open Handset Alliance Project, http://code.google.com/android/documentation.html. © Google 2007.

[17] Sun Microsystems, Inc., Java Specification Request (JSR) 271: Mobile Information Device Profile 3, http://jcp.org/en/jsr/detail?id = 271. © Sun Microsystems, Inc. 2007.

[18] Sun Microsystems, Inc., Java Specification Request (JSR) 232: Mobile Operational Management, http://jcp.org/en/jsr/detail?id = 232. © Sun Microsystems, Inc. 2007.

[19] Sun Microsystems, Inc., Java Specification Request (JSR) 293: Location API 2.0, http://jcp.org/en/jsr/detail?id = 293. © Sun Microsystems, Inc 2007.