Enhancing JSR-179 for Positioning System Integration and Management

Paolo Bellavista, Antonio Corradi, Carlo Giannelli
Dip. Elettronica, Informatica e Sistemistica - Università di Bologna
Viale Risorgimento, 2 - 40136 Bologna - ITALY
Phone: +39-051-2093001; Fax: +39-051-2093073
{pbellavista, acorradi, cgiannelli}@deis.unibo.it

Abstract

Several heterogeneous positioning systems are more and more widespread among client wireless terminals, thus leveraging the market relevance of Location Based Services (LBSs). Positioning techniques are very differentiated, e.g., in terms of precision, accuracy, and battery/bandwidth consumption, and several of them are simultaneously available at clients. That motivates novel middleware solutions capable of integrating the dynamically accessible positioning techniques, of controlling them in a synergic way, and of switching from a positioning system to another at service provisioning time by choosing the most suitable solution depending on application-level LBS context. In this perspective, the paper proposes the original PoSIM solution, which significantly extends the emerging JSR-179 standard specification to allow differentiated forms of visibility/control of low-level positioning characteristics, greater flexibility in location change-driven event triggering, and the simultaneous management of multiple and dynamically introduced location techniques.

1 Introduction

The growing availability of powerful mobile devices with relatively high wireless bandwidth, e.g., via UMTS, IEEE 802.11, and Bluetooth 2.0 connectivity, is going to leverage the widespread diffusion of Location Based Services (LBSs). LBSs can provide service contents depending on current user positions, on the mutual location of clients and accessed server resources, and on the mutual position of users in a group [1]. To enable LBSs, positioning techniques are crucial. Several research activities have deeply worked on evaluating mechanisms and technologies for positioning: some solutions have been specifically designed for determining location, e.g., the well known Global Positioning System (GPS); other proposals try to estimate positioning information by monitoring characteristics of general-purpose communication channels, such as the IEEE 802.11-based Ekahau [2]. For a more exhaustive positioning system survey please refer to [3, 4].

Currently available positioning solutions greatly differ on capabilities and provided facilities. For instance, they diverge in:

- the representation model of the provided location information. That model could be either physical (longitude, latitude, and altitude triple), or symbolic (e.g., room X in building Y), or both;
- the applicable deployment environment. For instance, GPS can work only outdoor, Ekahau primarily indoor;
- accuracy and precision of the positioning information. Accuracy is defined as the location data error range (10 meters for GPS), while precision is the error range confidence (95% for GPS);
- power consumption, which usually depends on location update frequency;
- user privacy. For instance, client nodes that exploit IEEE 802.11-based positioning have to disclose their location, to some extent and at a certain granularity, to be capable of communications (they must associate to an AP for communication purposes);
- and additional supported features, which can be peculiar of specific positioning systems. For instance, some positioning solutions can provide location data as a probability distribution function.

That heterogeneity among the available positioning systems, together with the fact that current wireless clients tend to simultaneously host several wireless technologies useful for positioning (e.g., terminals with Wi-Fi and/or Bluetooth connectivity and/or equipped with GPS), motivate the need for novel middleware solutions capable of integrating the available position-

ing techniques, of controlling them in a synergic way, and of dynamically selecting the most suitable solution depending on context. First of all, that middleware should allow to seamlessly switch from a positioning system to another based on availability, e.g., GPS outdoor and Ekahau indoor. Then, it should suggest exploiting, at any time, the positioning technique which best fits user preferences, application requirements, and device resource constraints: for instance, the positioning system with lower power consumption in the case of priority given to battery preservation, or the one with greater accuracy and precision, or the one with most frequent updates, or the one providing physical/symbolic location information. Moreover, when several positioning systems can concurrently work, the middleware could perform fusion operations on location data, e.g., to increase accuracy and/or confidence.

For all these purposes, there is also the need to make low-level characteristics of positioning systems easily accessible to the upper layers (middleware and/or application levels), thus enabling application-specific control of positioning techniques, possibly by avoiding to complicate LBS development and deployment.

The paper extensively discusses the integration and management of heterogeneous positioning systems. Section 2 describes related work about middleware solutions for positioning integration, while Section 3 focuses on JSR-179, an emerging standard API for positioning. Section 4 rapidly sketches our original PoSIM middleware and its main components, while Section 5 compares the proposed PoSIM API with the JSR-179 one. Conclusions and on going work end the paper.

2 Related Work

Several academic research activities have recently addressed the area of dynamically fusing positioning information from different sources. Here, we only present a few of them to point out the primary solutions related to context/location information and positioning system integration.

Some positioning middleware proposals have the main goal to support the easy development and deployment of LBSs. The main idea is to hide the complexity due to the adoption of several, heterogeneous positioning systems, by providing integrated positioning services in a transparent manner. Every low level information and detail is hidden and there is no possibility to control positioning system behavior. Most of them make positioning system integration completely transparent from the LBS point of view. [5] focuses on the integration of several positioning systems through appropriate wrappers exploited to provide a uniform API to heterogeneous positioning systems. The goal is to exploit the positioning system which is currently available or which best fits accuracy requirements, eventually performing location data fusion. Moreover, [5] provides user-controlled privacy actively, by requesting explicit user permission before disclosing location information. [6] has the primary goal of seamless navigation, i.e., to provide location information regardless actually exploited positioning systems and maps. Its main solution guideline is to exploit middleware components called mediators-wrappers, to abstract from each exploited positioning system peculiarities and map implementations. In addition, it permits to dynamically change the exploited positioning system and map, in a transparent way from the user/application point of view. [7] supplies a specific interface to develop new LBSs. Moreover, it supports the introduction of new positioning systems through a plug-in architecture; the middleware kernel interacts with positioning systems in a standardized manner, via OSA/Parlay. Similarly, [8] tries to abstract from the adoption of several positioning system: it performs information abstraction through a multi-step architecture for location data fusion, generation of geometric relationships, and event-based information disclosure. [9] goes further by proposing different abstracting steps to provide high-level location data: positioning, modeling, fusion, query tracking, and intelligent notification. Moreover, it ensures privacy and security management, by controlling information disclosure, similarly to [5]; the positioning system integration is achieved by the Common Adapter Framework that provides standard APIs to fetch the location information of the mobile devices.

The previously described middlewares integrate several positioning systems with the goal to facilitate LBS development. They tend to propose transparent approaches that hide applications from positioning complexity, but do not support any application-specific form of control of currently available positioning techniques. A few proposals start to delineate cross-layer supports that provide visibility of low-level details and control features at the application level. In the following part of the related, we will focus on middleware solutions that, to some extent, propagate the visibility of low level details at the application level.

MiddleWhere [10] offers some abstracting facilities, like previously described solutions, but also provides applications with low-level details. In particular, it can provide requesting clients with additional data about location resolution, confidence, and freshness. An adapter component acts as a device driver allowing MiddleWhere to communicate with positioning system implementations: the adapter makes location description uniform by hiding any positioning implementation

peculiarity. [11] supports the integration and control of several positioning systems providing low-level details at the application layer. However, it performs integration and control in a hard-coded and not flexible manner. In addition, the visibility of data/features peculiar to a specific positioning system requires its full static knowledge, thus significantly increasing the LBS development complexity. Location Stack [12] represents a state-of-the-art model of solution for location/context fusion: it identifies several sequential components, deployed in a layered manner, which provide increasing abstraction: Sensors, Measurements, Fusion, Arrangements, Contextual Fusion, Activities, Intentions. However, the Unified Location Framework [13], a possible implementation of Location Stack, demonstrates that such a layered system does not easily allow to propagate the visibility of useful low-level data such as accuracy and precision. In fact, [13] points out that cross-layering is required both to supply low-level details to LBSs and to control positioning systems from the application level.

In conclusion, most proposed middlewares mainly address the location fusion issue and tend to hide any low-level detail depending on positioning technique and implementation. [10], [11] and [13] offer some low-level details, but they have to be statically predetermined. To the best of our knowledge, no middleware solution in the literature addresses the challenge of dynamic and integrated control of available positioning systems by considering application-level requirements in a flexible and extensible way.

3 The JSR-179 Location API

In the last years, the industrial research activity has primarily focused on the development of standards to address the wide heterogeneity of available positioning systems. The JSR-179 API [14], also known as Location API for J2ME, represents the most notable result of that standardization effort for Java-based LBSs on mobile phones. JSR-179, inspired by the usual and widespread interface of the GPS solution, provides a standardized API to perform coarse-grained integration and control of positioning systems (*location providers* according to the JSR-179 terminology). To better understand how JSR-179 provides location information, here we rapidly report its main characteristics and offered functions.

The `LocationProvider` class is the JSR-179 API entry point. Applications invoke the `getInstance()` method of `LocationProvider` to retrieve an actual location provider implementation among the currently available ones. The actual location provider is the selected positioning system that returns location information to applications.

When invoking the `getInstance()` method, an application optionally specifies particular criteria (`Criteria` class) that the actual location provider must satisfy. If several actual location providers are compatible with the passed criteria, `LocationProvider` selects the one which best fits the requirements according to a predetermined strategy. `Criteria` can specify that the actual location provider must supply speed and altitude, and/or that the provided horizontal/vertical coordinates have to respect a minimum accuracy level. Moreover, it is possible to specify the desired power consumption (low, medium, or high). Let us notice that the passed criteria are exploited only at the moment of the selection of the actual location provider; they are completed neglected at provisioning time.

Figure 1 depicts an example of application that requests an actual location provider implementation, by specifying the desired selection criteria. The result is the activation of the positioning system best fitting the criteria among the currently available ones (Location Provider 2 in the figure). Location Provider 2 is associated with the application until a new explicit request of location provider selection to the JSR-179 API.

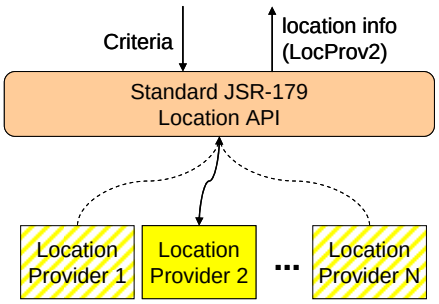

Figure 1. The JSR-179 API for criteria-based selection of an actual `LocationProvider` implementation.

Location providers return location data in three different ways:

- *on demand*, via the `getLastKnownLocation()` and `getLocation(timeout)` methods, which respectively provide cached and just updated location information, the latter actively requesting for new data to the underlying positioning system;
- *periodically* at fixed time intervals, via the method `setLocationListener(listener, interval, timeout, maxAge)`. Only one periodical listener at a time can be registered with each location provider instance;
- *in an event-driven fashion* via the `addProximityListener(listener, coordinates, proximityRadius)` method. The only triggering event that can be exploited in JSR-179 is the proximity of the located client to specified co-

ordinates. Several proximity listeners may contemporarily indicate multiple coordinates close to which a location provider triggers the events.

The provided location information specifies qualified coordinates (physical location), address info (symbolic location), or both. Moreover, it may include additional data such as speed, timestamp, and the technology of the actual location provider.

JSR-179 is a good example of standardization effort in the industrial research area to leverage the adoption of positioning systems and LBSs. Its architecture and API have the goal of representing a standardized model for every developer willing to provide new positioning systems or LBSs. However, we claim that JSR-179 does not provide a sufficiently expressive API to perform efficient integration and control of positioning systems. In particular, it supports neither the dynamic management of multiple location providers nor the provisioning of low-level system-specific details to the application level as required by many LBSs.

First of all, it does not support the dynamic and flexible management of dynamically retrieved location provider implementations. On the one hand, JSR-179 only permits to exploit one location provider at a time among the ones currently available at a client, even if several of them satisfy the specified criteria. On the other hand, according to the JSR-179 specification, LBSs have the full duty of monitoring the performance of the selected location provider and of taking suitable management operations consequently, e.g., requesting for a new location provider selection in response to accuracy degradation. In other words, once JSR-179 has selected a location provider, the specified criteria are no more considered even if the capabilities of the actual location provider do not satisfy the LBS requirements any more or if a new more suitable location provider becomes available at the client.

In addition, the JSR-179 API assumes that the characteristics of location providers are statically identified and do not considerably change over time: that is partially true for static features, e.g., ability to provide speed/altitude or not, but not applicable to dynamic characteristics such as horizontal/vertical accuracy. For example, GPS accuracy may abruptly decrease when the user moves from an outdoor to an indoor environment. Moreover, JSR-179 has dynamicity and flexibility limitations also due to its impossibility to accommodate new positioning systems newly introduced at service provisioning time. The actual location provider implementation is determined only once at the moment of location provider instantiation; JSR-179 does not consider any context change after that instantiation, until a new LBS request for actual location provider determination. Another limitation of JSR-179 is that selection criteria are limited to few and statically predetermined elements. It is possible to specify as requirements only the features defined in the `criteria` class before service provisioning. Moreover, also the event handling functions of JSR-179 exhibit non-negligible limitations, as already pointed out: only one type of triggering event is supported, the one related to proximity to a fixed location.

But, according to our opinion, corroborated by our experience in developing and prototyping LBSs, JSR-179 exhibits the most relevant lack in its limited capabilities to propagate the visibility of low-level details of underlying location providers when needed. In fact, the only state information available about location providers is their availability status (available, temporarily unavailable, or out of order). This full and uniform transparency of low-level positioning system features does not always fit the requirements of application-level visibility typical of LBSs. For example, a LBS would get and control peculiar positioning system functions, such as to get and possibly change the location provider privacy level.

The academic research on the extension of JSR-179 capabilities to achieve greater flexibility and dynamicity is still at its very beginning, also due to the novelty of the standardization effort. [15] proposes the integration and management of multiple positioning systems via a JSR-179 fully compliant API. It tries to increase dynamicity by transparently switching among available positioning systems: in particular, it alternatively exploits either GPS/Bluetooth-based positioning dependently on client outdoor/indoor location. However, the proposal does support neither the dynamic change of positioning selection criteria (only system availability), nor the integration with new positioning systems at provisioning time. Moreover, it does not provide any function at all to control integrated positioning systems from the application layer.

4 The PoSIM Middleware

Our goal is to go further than just hiding positioning systems integration behind the JSR-179 API. The objective is to provide a middleware solution that significantly extends JSR-179 with new and more powerful features to support positioning system integration and management in a flexible, dynamic, and extensible way. At the same time, our middleware proposal should adopt an API similar to the JSR-179 one, at least when possible, to facilitate its adoption by developers of both positioning systems and LBSs.

This section describes our Positioning System Integration and Management (PoSIM) middleware for the efficient and flexible integrated management of different positioning systems. In particular, PoSIM focuses

on three aspects. First of all, it is capable of integrating positioning systems at service provisioning time in a plug-in fashion, by exploiting their possibly synergic capabilities and by actively controlling their features. Secondly, PoSIM allows positioning systems to flexibly expose their capabilities and location data at runtime and without requiring any static knowledge of positioning-specific data/functions. Third, it can perform location data fusion depending on applicable context, e.g., application-specific requirements about accuracy or client requirements about device battery consumption.

Furthermore, PoSIM enables differentiated visibility levels to flexibly answer all possible application requirements stemming from different LBS deployment scenarios and application domains. On the one hand, PoSIM enables LBSs to access and control the whole set of available location providers in a transparent way at a high level of abstraction: LBSs can simply specify the behavior positioning systems must comply with via declarative policies; PoSIM is in charge of actually and transparently enforcing the selected policies. On the other hand, PoSIM allows LBSs to have full visibility of the characteristics of the underlying positioning systems via a PoSIM-mediated simplified access to them. In this case, PoSIM provides LBSs with a uniformed API, independently of the specific positioning solution, that permits to access/configure heterogeneous location providers homogeneously and aggregately. We call *translucent* the original PoSIM approach that supports LBSs with both transparent and visible integrated access to available positioning solutions.

Thanks to the translucent approach, two different classes of PoSIM-based LBSs can properly manage heterogeneous positioning systems: simple LBSs and smart ones. Via PoSIM, simple LBSs can interact transparently with location providers perceived as a single service exposing a JSR-179-like API. They can control positioning systems easily, just specifying the required behaviors via declarative policies or simply selecting the policies to enforce among pre-defined ones, e.g., by privileging low energy consumption or high location accuracy. Instead, smart LBSs, i.e., applications willing to have direct visibility and manage location information or peculiar capabilities of positioning systems, interact in a middleware-mediated aware fashion: they can have a PoSIM-based uniform access to all functions of underlying positioning solutions, even the system-specific ones, e.g., the possibility to limit Ekahau accuracy to reduce network overhead.

Let us stress that we distinguish between positioning *features* and *infos*. Features describe positioning system characteristics and capabilities, possibly with settable values and useful for positioning system control, e.g., power consumption or ensured privacy level. Infos are location-related information, e.g., actual positioning data and their accuracy, not modifiable from outside the positioning systems. Infos are the only data provided to simple LBS.

In the following, the section briefly presents all the PoSIM components, in order to point out how the translucent approach is implemented to provide a more dynamic, extensible and powerful version of the JSR-179 API. For further implementation details about PoSIM components, please refer to `http://lia.deis.unibo.it/Research/PoSIM`.

To interact with positioning systems in a transparent manner, simple LBSs can exploit the Policy Manager (PM) and Data Manager (DM) components depicted in Figure 2. Via those APIs, simple LBSs can ask for pre-defined behaviors implemented as declarative policies, without any knowledge of how actually the integrated positioning systems are exploited. For example, the `POWER_USAGE_LOW` policy turns off all the positioning systems with high energy consumption by preserving application-specific requirements about precision and accuracy. PM is in charge of maintaining pre-defined policies and enforcing active ones; it is implemented on top of the Java-based rule engine Jess [16].

Via the DM component, PoSIM provides integrated positioning system info in an aggregated way as a single XML document, where tags are exploited to specify the content semantics, thus permitting a significantly higher level of dynamicity.

In addition, PoSIM can offer location data for any integrated and currently active positioning system. Location data access retrieval is possible either on request, or specifying a time period, or via event notification. LBSs can easily specify the conditions to trigger XML document delivery. For instance, the pre-defined `atLocation` condition triggers location data notification only when the current physical location of the user is close to a known location, similarly to the only possibility available in JSR-179 via the proximity listener. In addition, LBSs may request DM to work as a filter, e.g., the pre-defined `highAccuracy` data filter discards location information with accuracy below a given threshold. Note that the proper exploitation of filtering rules permits to reduce the network overhead due to non-relevant changes of location data. PoSIM implements triggering events and filtering rules as Java classes, which can be easily sub-classed to specify specialized triggers and filters.

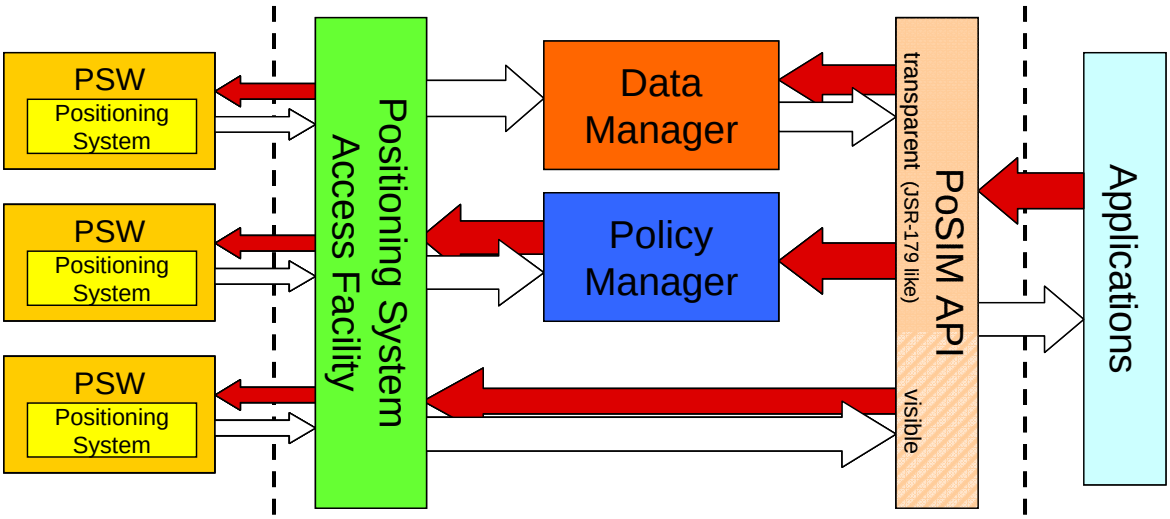

Figure 2 The PoSIM architecture (white arrows represent data flows, grey arrows control flows).

Let us stress that expert users, such as PoSIM administrators, can develop and deploy new policies, i.e., selection/fusion criteria, triggering events, and filtering rules. The PoSIM behavior can thus be specialized and extended with impact on neither its implementing code nor the application logic code. That permits to easily extend and personalize the PoSIM middleware. For instance, it is possible to dynamically extend PoSIM capabilities by introducing the `atChanges` condition that triggers location notification only when current and previous physical location differ more than a specified distance. Anyway, simple LBSs and novel developers can also work, simply and rapidly, by selecting among the existing set of most common policies, events, and filters.

Smart LBSs and PM/DM can directly control positioning systems by exploiting the lower level API of the Positioning System Access Facility (PSAF). PSAF supports API to dynamically handle the insertion/removal of positioning systems and to retrieve/control data/features of all the currently available positioning systems. The only requirement is that positioning systems provide their data/features via a specified interface; that interface is the result of the wrapping of another PoSIM middleware component, i.e., the Positioning System Wrapper (PSW). PSAF exploits Java introspection to dynamically determine and access the set of data/features actually implemented by the underlying positioning solutions currently available in its deployment environment.

5 Comparing PoSIM and JSR-179 API

This section aims at pointing out and discussing the main differences between the PoSIM API and the JSR-179 one. As depicted in Figure 2, PoSIM offers two levels of visibility to LBS developers: a transparent API, which is similar to the JSR-179 one, and a visible API, which extends traditional JSR-179 by providing the capability to directly interact with integrated positioning systems.

The transparent part of the PoSIM API, provided by PM and DM, is similar to the JSR-179 one. However, since PoSIM offers extended and richer integration functions, there are necessarily a few API differences also in the transparent part.

Delving into finer details, PM supplies many capabilities that JSR-179 does not provide. As already depicted, the JSR-179 API only exploits the location provider which best fits the criteria specified once at request time. On the contrary, PoSIM permits to specify and modify criteria at service provisioning time. In fact, PM accepts declarative criteria similarly to JSR-179, but it actively and dynamically controls positioning system behaviors instead of simply selecting the one which best fulfills the specified requirements. Furthermore, since PoSIM criteria are implemented as Jess rules, it is possible to create new criteria and provide them at runtime, without either recompiling or restarting the system.

Also DM exposes many capabilities that the standard JSR-179 API cannot provide. First of all, since PoSIM can exploit several positioning systems at a time, it can also perform location fusion, for instance to possibly increase location information accuracy. Then, LBSs may take advantage of every available positioning system suitable to their requirements. For this reason, PoSIM provides location information as an XML document, not as a single `Location` class like JSR-179 does.

Both JSR-179 and PoSIM may perform data deliv-

ery in a event-driven fashion. However, while the JSR-179 API only supports statically determined triggering events, i.e., proximity-based event notification, PoSIM also provides the capability to exploit new events, specified and deployed at service provisioning time, thus relevantly increasing system flexibility and extensibility. For instance, it is possible to specify the aforementioned `atLocation` and `atChanges` triggering events, the former similar to the only supported JSR-179 proximity event, the latter not available through the standard JSR-179 API.

Capabilities provided by PSAF are completely absent from JSR-179. First of all, PSAF offers the possibility to integrate new positioning systems at service provisioning time. Newly integrated positioning systems can be immediately exploited by PoSIM. Active criteria will be dynamically applied to new positioning systems and their location information automatically inserted in the provided XML document. On the contrary, to exploit a new positioning system through JSR-179, LBSs must explicitly request for another location provider instance, by actively providing their specific selection criteria again.

Finally, the JSR-179 API tends to hide application developers from low-level positioning system details. On the contrary, if necessary, the PoSIM API provides full visibility of and fine-grained control over the integrated positioning systems, by permitting the access to both standard and system-specific features/info.

Table 1 Relationships between the primary functions of JSR-179 and PoSIM API.

API category		JSR-179 API	PoSIM API	PoSIM Component	Comparison
n.a.		getInstance(criteria)	getInstance()	PoSIM API	extended
transparent	info delivery	getLastKnownLocation()	onDemand(listener)	DM	equivalent
		addProximityListener(...)	addEvent(event,listener)	DM	extended
		setLocationListener(...)	periodical(interval,listener)	DM	equivalent
		n.a.	addFilter(filter,listener)	DM	additional
	control	getInstance(criteria)	activateCriteria(criteria)	PM	extended
visible	control	n.a.	insertPosSys(newPosSys)	PSAF	additional
		getState()	getFeatures(posSys) getFeature(posSys, aFeature) setFeature(posSys, aFeature)	PSAF	extended

Table 1 reports and compares the main functions for info delivery or positioning system control available in the JSR-179 and PoSIM API, categorized as either transparent or visible. For each JSR-179 API method, the table reports the corresponding PoSIM one, by underlining which PoSIM component provides it and by pointing out possible differences between JSR-179 and PoSIM implementation. In particular, a PoSIM method is classified as i) equivalent to the correspondent JSR-179 one if and only if they offer exactly the same capability, ii) extended if providing more expressive and powerful features, and iii) additional if introducing completely new behaviors not available in JSR-179.

Most PoSIM methods offer the capability to control and interact with integrated positioning systems in a transparent manner. By considering these first transparent functions and going into finer details:

- both PoSIM and JSR-179 offer a `getInstance()` method, but with significantly different expressiveness. While JSR-179 selects only one location provider among the currently available ones dependently on given criteria, PoSIM just returns a middleware-mediated interface instance. Let us stress that the PoSIM `getInstance()` method provides LBS developers with the capability to get multiple simultaneous location data from any integrated positioning system, while JSR-179 allows the access to only the actual location provider.
- PoSIM `onDemand(...)`and JSR-179 `getLastLocation()` methods behave similarly. Both immediately provide the last known location information, even if PoSIM returns the data obtained by possibly fusing information from every integrated positioning system, while JSR-179 the data from the previously selected actual location provider.
- `addEvent(...)` relevantly extends the expressive power of the correspondent `addProximityListener(...)`. In fact, the former provides the capability to specify which kinds of event trigger location information delivery, while the latter can exploit only proximity events.
- `setLocationListener(...)` and `periodical(...)` are almost equivalent. Both periodically provide location information at a given time interval. However, while the JSR-179 API specifies that only one location listener can be registered at a time, `periodical(...)` permits to register several listeners, also by possibly serving multiple ap-

plications with the same location data at a time.

- The PoSIM `addFilter(...)` method permits to define new filters for location information (possibly after fusion). That capability is not supported at all in JSR-179 API.
- The PoSIM `activateCriteria(...)` could seem similar to JSR-179 `getInstance(...)` since both permit to specify selection criteria. However, they are greatly different since the former activates a management policy exploited to control integrated positioning systems at service provisioning time, while the latter simply selects the actual location provider at invocation time.

In addition to the above transparent functions, the following methods provide LBS developers with full but middleware-mediated visibility of the integrated positioning systems.

- `insertPosSys(...)` is available only in PoSIM. JSR-179 does not provide any method to add new positioning systems at service provisioning time.
- The JSR-179 `getState()` simply provides coarse-grained information about the availability of the actual location provider. PoSIM extends this function by providing a method to get all the available features of a given positioning system, `getFeature(...)`, and a method to configure their values, `setFeature(...)`, if allowed by the underlying positioning systems. Features are described in a portable and interoperable way according to the representation described at the PoSIM Web site `lia.deis.unibo.it/Research/PoSIM`.

Let us rapidly observe that the PSAF `getInfo(...)` method is equivalent to the JSR-179 `getLocation()` one to get just updated info. The only difference is that the former provides information in a visible manner, the latter transparently.

6 Conclusions

The widespread diffusion of several and heterogeneous positioning systems pushes towards the adoption of widely accepted standard to provide location information. The already proposed JSR-179 tries to addresses issues raised from positioning system heterogeneity, by hiding positioning systems behind a well standardized API. However, it does not address the crucial issue of dynamically and flexibly integrating, with full access to fine-grained control features, several positioning systems at a time. The paper proposes the original translucent PoSIM approach: our middleware permits to control integrated positioning systems both in transparent and non-transparent way, respectively fitting simple and smart LBS requirements. In particular, the paper focuses on similarities and differences between the PoSIM API and the standard JSR-179 one, by pointing out how PoSIM relevantly extends JSR-179 capabilities, while mimicking its API to facilitate and accelerate adoption.

The encouraging results already obtained in the PoSIM project are stimulating further related research activities. We are extending the middleware openness by including an additional wrapper for our original Bluetooth-based positioning system (at the moment the PoSIM prototype includes wrappers for GPS and Ekahau). Moreover, we are extending the set of pre-defined set of criteria, filter rules, and triggering events, to fit all the personalization requirements of most common LBSs by simply requesting developers to select the integration/control strategies to apply.

Acknowledgements

Work supported by the MIUR FIRB WEB-MINDS and the CNR Strategic IS-MANET Projects.

References

[1] G. Chen, D. Kotz, "A Survey of Context-Aware Mobile Computing Research", Dartmouth College Technical Report TR2000-381, http://www.cs.dartmouth.edu/reports/, 2000.

[2] Ekahau, http://www.ekahau.com

[3] J. Hightower, G. Borriello, “Location systems for ubiquitous computing”, Computer, Vol. 34, No. 8, Aug. 2001, pp. 57-66.

[4] J. Hightower, G. Borriello, "Location Sensing Techniques", UW CSE 01-07-01, University of Washington, Department of Computer Science and Engineering, Seattle, WA, July 2001.

[5] J. Nord, K. Synnes, P. Parnes, “An Architecture for Location Aware Applications“, 35th Hawaii Int. Conf. on System Sciences, Hawaii, USA, Jan. 2002.

[6] Y. Hosokawa, N. Takahashi, H. Taga, “A System Architecture for Seamless Navigation”, Int. Conf. on Distributed Computing Systems Workshops (MDC), Tokyo, Japan, Mar. 2004.

[7] M. Spanoudakis, A. Batistakis, I. Priggouris, A. Ioannidis, S. Hadjiefthymiades, L. Merakos, “Extensible Platform for Location Based Services Provisioning”, Int. Conf. Web Information Systems Engineering Workshops, Rome, Italy, Dec. 2003.

[8] G. Coulouris, H. Naguib, K. Samugalingam, “FLAME: An Open Framework for Location-Aware Systems”, Int. Conf. on Ubiquitous Computing, Goteborg, Sweden, Sept. Oct. 2002.

[9] Y. Chen, X.Y. Chen, F.Y. Rao, X.L. Yu, Y. Li, D. Liu, “LORE: An infrastructure to support location-aware services”, IBM Journal of Research & Development, Vol. 48, No 5/6, Sept./Nov. 2004.

[10] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R. Campbell, M. D. Mickunas, “MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Ap-

plications", ACM/IFIP/USENIX Int. Conf. on Middleware, Oct. 2004, Toronto, Ontario, Canada.

[11] J. Agre, D. Akenyemi, L. Ji, R. Masuoka, P. Thakkar, "A Layered Architecture for Location-based Services in Wireless Ad Hoc Networks", IEEE Aerospace Conf., Big Sky, Montana, USA, Mar. 2002.

[12] J. Hightower, B. Brumitt, G. Borriello, "The Location Stack: A Layered Model for Location in Ubiquitous Computing", IEEE Work. on Mobile Computing Systems and Applications, Callicoon, NY, USA, Jun. 2002.

[13] D. Graumann, W. Lara, J. Hightower, G. Borriello, "Real-world Implementation of the Location Stack: The Universal Location Framework", IEEE Work. on Mobile Computing Systems and Applications, Monterey, CA, USA, Oct. 2003.

[14] JSR-179, http://www.jcp.org/aboutJava/communityprocess/final/jsr179/index.html

[15] C. di Flora, M. Ficco, S. Russo, V. Vecchio, "Indoor and outdoor location based services for portable wireless devices", Int. Conf. on Distributed Computing Systems Workshops (SIUMI), Columbus, Ohio, USA, Jun. 2005.

[16] Jess, http://herzberg.ca.sandia.gov/jess/