Place Lab — An Open Architecture for Location-Based Computing

Timothy Sohn[†] William G. Griswold[†] James Scott[‡] Anthony LaMarca[*]
Yatin Chawathe[*] Ian Smith[*]

[†]Computer Science and Engineering
University of California, San Diego
{tsohn,wgg}@cs.ucsd.edu

[‡]Intel Research Cambridge
james.w.scott@intel.com

[*]Intel Research Seattle
{anthony.lamarca, yatin.chawathe,
ian.e.smith}@intel.com

ABSTRACT

Location-based computing (LBC) is an emerging hot topic in both industry and academia. A key challenge is the pervasive deployment of LBC technologies; to be effective they must run on a wide variety of client platforms, including laptops, PDAs, cell phones, and even embedded devices, so that location data can be acquired anywhere and accessed by any application. Moreover, LBC as a nascent research area is experiencing rapid innovation in sensing technologies, the positioning algorithms themselves, and the applications they support. Lastly, as a newcomer, LBC must integrate with existing communications and application technologies, including web browsers and location data interchange standards.

This paper describes the Place Lab architecture, a first-generation open platform for client-side location sensing. Using a layered, pattern-based architecture, it supports modular innovation in any dimension of LBC, enabling the field to move forward more rapidly as these innovations are shared with the community as plug-in components.

Several uses of Place Lab are described to demonstrate the architecture's effectiveness and limitations. These experiences are instructive for future developers of mobile context-aware systems.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures – Domain-specific architectures; D.2.13 [Software Engineering]: Reusable Software – Domain Engineering;

General Terms
Algorithms, Design

Keywords
Location-based computing, pervasive computing, ubiquitous computing, software architecture

---

This research supported in part by a gift from Intel Research.

Submitted To FSE 2005

1. INTRODUCTION

Location-based computing (LBC) is now possible on a variety of platforms for use in developing and deploying rich context-aware applications. However, location-based computing depends heavily on the technologies on which it is deployed and how it is applied. In order to achieve effective, pervasive deployment of location technologies, the supporting software must run on a variety of platforms including laptops, PDAs, and mobile phones. These devices vary widely in their computing power, operating system environment, sensing technologies, and in the types of application deployed on them. Developing a portable location-based computing software architecture to support these platform demands is challenging. Moreover, LBC is a nascent research area, and the positioning algorithms are still in a period of rapid innovation. Examples of recent positioning algorithms involve using particle filters [9] or fingerprinting techniques [4]. A driver for innovation in positioning algorithms is the emergence of new sensing technologies. Radio beacon technologies such as 802.11, Bluetooth, GSM, and infrared are all being used for positioning. The positioning capabilities of other technologies are also being actively explored, including new technologies such as ultra wide-band, and with novel uses of existing technologies such as sound hardware [15]. Lastly, location-based computing must integrate with existing communications and application technologies in order to prove useful. The ability to incorporate location into an application without significant effort is useful in promoting the greater aspect of context-awareness to affect application behavior.

In this paper we describe the software architecture of Place Lab, a widely used first-generation open source toolkit for client-side location-based computing. Place Lab supports multiple platforms and innovation in three different dimensions: applications, positioning algorithms, and sensing technologies. We first detail the requirements for pervasive client-side deployment of a location-based computing architecture, and then describe Place Lab and several case studies that demonstrate the architecture's effectiveness as well as its limitations. These experiences are instructive for future developers of mobile context-aware systems.

2. RELATED WORK

Place Lab falls into the general category of *fusion architectures*. Conceptually, a fusion architecture refines raw streams of data from possibly many sources into a sequence of high-level inferences. Fusion architectures have a place in wide-scale defense systems, context-aware computing, and sensor networks, to name a few examples. The purpose of such an architecture is to separate the different aspects of the data processing into logical algorithmic components that can be independently improved, replaced, or composed. A dominant theme in fusion architectures is the pipelining, stacking, or layering of the components into a sequence of processing stages that successively refine a data stream into inferences.

An example fusion architecture for defense systems is the U.S. Department of Defense's JDL[1] data fusion conceptual architecture, which contains five levels (phases) of situation modeling, proceeding from top to bottom [18]:

0. *Sub-Object Data Association and Estimation*: signal-level data association and characterization.
1. *Object Refinement*: combines data from multiple sensors and other sources to determine position, kinematics, and other attributes.
2. *Situation Refinement*: develops interpretation of the relationships among the objects and events in the context of the operational environment.
3. *Significance Estimation*: intent prediction and consequence prediction.
4. *Process Refinement*: monitors the fusion process to refine the process itself and guide acquisition of additional data.

The seven-layer Location Stack architecture focuses on the inference of location-related information [10], and provides an infrastructure for location-sensing based on Bayesian inference [9]. The six software layers in the stack are sensor measurements, fusion, arrangements, contextual fusion, activities, and intents. The difference in this architecture from the JDL is that the fusion of location measurements ("fusion") is distinguished from fusion across context categories ("contextual fusion"). In short, the composition is understood to be more like a tree than a pipeline.

ActiveCampus is a server-centric database-oriented fusion architecture for extensible, integrated application design [8]. It employs a multi-stage mediator-observer design pattern [19] to create the stages of processing. The event-driven database model provides for decoupling of components yet tight integration: the storing of a lower-level data element into the database triggers an event that causes the next stage of processing to begin; the storing of that stage's results triggers another event that starts the next stage of processing. New inference components can be added by registering for the appropriate events. Normalization of the database tables supports incremental extension of the objects being modeled and the components that process those objects.

The Context Toolkit is a small set of generic base classes from which a programmer can derive specific subclasses for the development of a streaming peer-to-peer networked context-aware application [6]. The primary classes are a *Context Widget*, which abstracts away a sensor as a data stream, a *Context Interpreter*, which provides a mapping of one type of context element to another, and a *Context Aggregator*, a context widget that fuses data streams from multiple widgets. The data element streamed between widgets is an aggregation of generic key-value pairs. The Context Toolkit's primary value lies in the generic services of storing and forwarding data between peers, as well as the flexible interoperability of the classes that are developed by the programmer.

Place Lab follows the general lines of a layered event-streaming fusion architecture. Like ActiveCampus, it makes heavy use of the hybrid-mediator design pattern. Place Lab's components map on to those of the Context Toolkit. What distinguishes it from these systems is its focus on location sensing, client-side inference, and the expected presence of the application itself on the client. These unique characteristics yield a distinct set of requirements, and enable the deployment of not only a toolkit, but also an infrastructure that can infer location on a wide variety of today's client computing platforms.

3. LOCATION-BASED COMPUTING REQUIREMENTS

The research community is particularly active in three aspects of location-based computing: sensing, sensor fusion in positioning algorithms, and applications. There is also substantial innovation in the personal computing devices that might deploy location-based applications. Our motivation is to provide a toolkit to serve as a "playground" for researchers in each area, minimizing unnecessary overhead in exploring their aspect of location-based computing. At the same time, we want to provide modularity for software components in each area, facilitating interoperability. Ideally, a new sensor fusion algorithm and a new sensor type could be developed independently, but would be able to operate together without any modification to either.

While modularity is a goal, providing this through pure abstraction is not useful to the research community. For example, the 802.11 and GSM radio technologies have very different characteristics. If data from these two sources is abstracted so that they are indistinguishable, this hinders the development of algorithms that handle those sources differently to achieve better accuracy. We therefore desire "lossless" abstractions between modules, in which useful abstractions can be made, but essential details remain accessible.

Our final priority is supporting a wide range of platforms, so that an application relying on many different form factors are all supported. A cross-platform toolkit has the additional advantage of providing platform independence for the code developed, thus allowing code to be more easily reused between applications that are otherwise very different.

3.1 Sensing

Sensing involves observations about the environment, such as nearby radio access points. There are many different ways of conducting sensor measurements. Some of these result in direct positioning information, such as the Global Positioning System (GPS). Others provide data that can indirectly indicate location, such as observing an 802.11 access point that is known to be mounted on a particular building. Even more indirect sources of location data might include using a microphone to monitor ambient noise levels to provide an estimation of whether a user is outdoors or indoors.

We wish to provide an API that allows different sensor types to be easily integrated into the architecture. Beyond the type of data that sensors produce, we identify at least three distinct ways in which sensor types differ and a LBC computing architecture must flexibly support. First, some sensors are implemented such that their natural interface is synchronous, polling the environment in some way, while others are asynchronous, generating events in reaction to the environment. Second, some sensors may generate groups of simultaneous data that are connected by belonging to the same "scan" in a given timeframe, while other sensors may generate

---

[1] JDL stands for "Joint Directors of Laboratories".

individual readings that are conceptually independent of one another. (Note that, according to our principle of not hiding potentially important distinctions in the data, we cannot simply present a group of readings as multiple individual readings). Third, sensors may be local (i.e., running on the same device as the user's application) or remote (i.e., running on another device that the user is carrying).

3.2 Fusion

The fusion stage includes any sort of transformation from raw sensor measurements to information such as a coordinate position or a place name. The architecture should allow for flexible fusion of the sensed data. In addition to the sensor data, it might need to make use of persistent information about the environment. For example, observing several nearby access points and relating them with persistent information about where they are located can help determine one's current position.

As with the sensor stage, fusion algorithms may be developed that naturally operate synchronously or asynchronously. Some may support sensor data from only a certain sensor type, others from broad classes of sensor types. Also, the fusion might be carried out on the same device as the application, or on a remote device to the application (e.g. on a computing server, if the application device is underpowered). Similarly, any persistent storage that is necessary for fusion might be available either locally or remotely (making use of network connectivity).

3.3 Applications

The range of potential location-aware applications is quite broad [16], [17], and it is unrealistic to expect that a single toolkit could provide seamless support for every unanticipated need. Nonetheless, in support of innovation, a LBC architecture must aim to make it easy to prototype or "upgrade" a wide range of applications.

One important way in which applications differ is subset of platforms on which they run. In addition to PCs and PDAs, location-aware applications can run on "smart phones", or on custom platforms such as embedded sensors. The latter would be very difficult to support, since it is impossible to predict the properties of a future custom platform. It would also be very difficult to design a single architecture that is interoperable across many custom platforms. We therefore chose to initially focus on the commodity platforms of PCs, PDAs, and smart phones.

Two dimensions of existing work that we wish to support in the space of location-based computing are existing applications and existing location data standards that applications use. A significant number of location-aware applications have been developed alongside a particular brand of location sensor. For example, location-enhanced map applications typically use GPS for positioning, but GPS is limited to outdoor environments. A powerful ability would be to simply plug in an indoor positioning technology without any software changes. In the latter category, we find location standards such as NMEA [1] and JSR-179 [3], which numerous applications are built on (e.g., many applications relying on GPS information understand NMEA).

A third dimension, application support, is in how location information is presented to applications. Although some applications might understand global latitude/longitude coordinates, others might expect locations relative to some base point (e.g., the corner of a building).

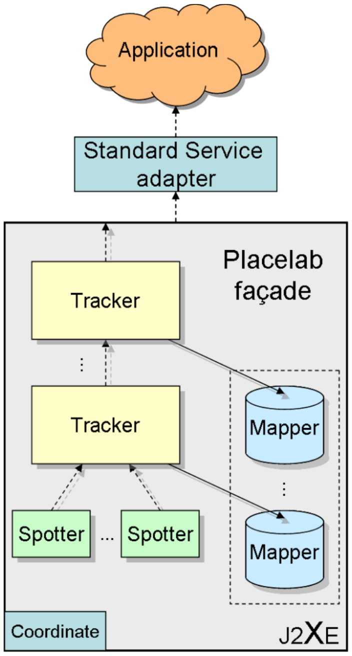

Figure 1. The Place Lab Architecture. Boxes are major components. Solid arrows are calls, dashed arrows are events. Coordinate is effectively a library extension of the Java environment. The events are all of subtype Measurement, permitting flexible composition of Spotters and Trackers. Each Tracker effectively has its own Mapper, but they may be combined for ease of implementation. Nominally run in a J2ME environment, the Coordinate abstraction hides the possible absence of floating point number support. The Placelab object hides the separate components, and a separate adapter can provide a standard location-reporting interface (e.g., GPS serial port emulation) to the application.

4. THE PLACE LAB ARCHITECTURE

Place Lab is a client-side location-inferencing architecture that was designed with respect to the requirements outlined. In this section we begin with an architectural overview, then describe how the platform is abstracted away, and finally discuss the components of the architecture in detail.

4.1 Overview

Place Lab is a fusion architecture based on a layered mediator-observer hybrid design pattern [19] (Figure 1). Conceptually, in each layer of the architecture a location *Tracker* receives locative *Measurement* objects from the layer below (e.g., {timestamp, remote beacon ID, signal strength}), correlates it to persistent location meta data from a read-only repository called a *Mapper* (e.g., {beacon ID, {latitude, longitude}}), infers a location, and then publishes a location inference event as a higher-level Measurement, known as an *Estimate* when an actual location is included (e.g., {timestamp, latitude, longitude, error}). Feeding the Trackers at the bottom of the layered architecture are one or more *Spotters* that gather raw sensor outputs and abstract them as initial Measurement events. The Placelab façade object groups and hides the above components. Optionally, a separate adapter can provide a standard location-reporting interface to the application (e.g., GPS serial port emulation). At the top of the architecture, loca-

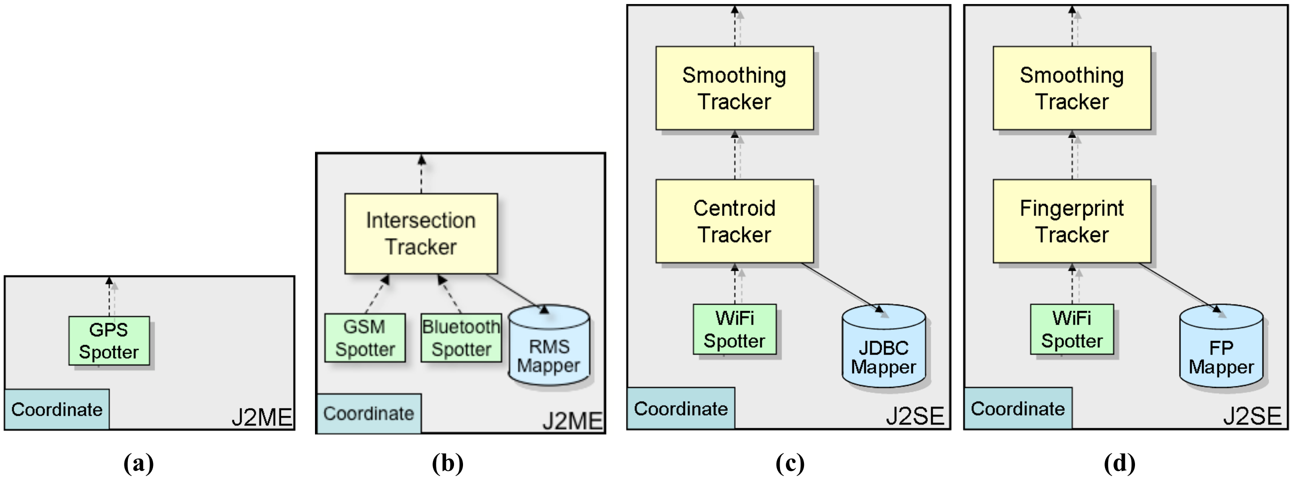

Figure 2. Four Actual Place Lab Instantiations. **(a)** Using only a GPS Spotter **(b)** Running on a phone platform using a GSM Spotter and a Bluetooth Spotter with an Intersection Tracker and a Record Management System (RMS) Mapper **(c)** 802.11 Spotter with a Centroid Tracker and a Smoothing Tracker stacked on top using a Java DataBase Connectivity (JDBC) Mapper **(d)** 802.11 Fingerprint Tracker and a Smoothing Tracker stacked on top.

tion-based applications process a stream of location events from the service or directly from the Placelab object.

The rules governing the use of architecture make it uniquely flexible in its ability to be extended or adapted. For one, the distinction of a read-only Mapper from a dynamic Tracker separates data-oriented and algorithm-oriented innovation in location tracking. This permits greater mixing and matching of innovations, and also isolates platform-independent tracking algorithms from store-dependent mapping services. Two, the ability to stack Trackers on top of Spotters permits independent innovations in different aspects of tracking to be composed. At its simplest, Place Lab could be instantiated with a GPS Spotter and no Trackers (Figure 2a). Using one Tracker, Place Lab could be instantiated with a GSM Spotter and a Bluetooth Spotter feeding an IntersectionTracker that performs fusion of these measurements (Figure 2b). On a PC, it could be instantiated with an 802.11 (WiFi) Spotter, a CentroidTracker, and a SmoothingTracker above that smoothes the incoming Estimates into a more probable path (Figure 2c). Such a configuration could be painlessly upgraded by replacing the CentroidTracker with a newly developed FingerprintTracker, with no change required to the Spotter or SmoothingTracker (Figure 2d).

This conceptual view of the architecture only tells part of the story. Abstracting away the vagaries of the underlying platform is addressed in the next subsection. Three other overarching architectural issues are briefly discussed here, and then details on the components of the architecture are provided in the following subsections.

One, an asynchronous event-driven model is not appropriate to all applications. For example, some applications update their location information only on request from the user. Others are implemented sequentially and use polling to acquire updates. Consequently, all Spotters and Trackers provide an alternative synchronous method-call interface. Generally, superclasses implement the emulation of one in terms of the other, so that subclasses are not burdened with satisfying these error-prone details.

Two, a consumer of Estimate events may need the source data from which they were computed, especially in a research environment. Therefore, when a Tracker creates a new Estimate, it provides a link back to the Measurements or Estimates that contributed to it. Consequently, each Estimate inexpensively references its provenance, making it available to subsequent trackers.

Three, for performance reasons, the Mappers in a particular instantiation of the architecture might be fused, perhaps as one big hash table, a database with multiple tables, or a sequential tuple store. These implementation details are of course abstracted away from the Trackers, each of which views the Mapper as its own. This abstraction of independence is assisted by the fact that the Mappers are effectively read-only.

4.2 Platform Abstraction

We decided to implement Place Lab on Java 2 Micro Edition (J2ME). This is a subset of the Java 2 Standard Edition (J2SE) framework, which only uses Java 1.1 facilities. J2ME was chosen because many mobile phones support it, using the Mobile Information Device Profile (MIDP) and Connected Limited Device Configuration (CLDC) libraries. Since Java virtual machines are available for PC and PDA platforms, this decision allowed much of Place Lab's core code to be directly reusable across these three platforms. The upwards compatibility of J2ME with J2SE also permits PC-specific components to take advantage of the full J2SE facilities without loss of flexibility in the overall architecture.

4.2.1 Real Number Support

There are a number of differences between Java implementations on the PC/PDA and phone platforms that required special attention. The most notable of these is that floating point arithmetic is not available on many smart phone models, but location coordinates, notably latitude/longitude, are normally represented as real number quantities. Five digits of decimal precision are required to achieve one-meter location precision with decimal latitude/longitude measurements.

Many of the solutions considered were determined to be untenable. Using integer representations of coordinates throughout Place Lab was rejected since programmers would not be able to use the coordinate systems that were familiar to them. Using an abstracted representation for a number, instantiated as a fixed-point or floating-point number depending on the platform, was rejected since Java does not allow the basic arithmetic operators like + and * to be defined for new types. All arithmetic operations would have to be coded using long-hand method calls (i.e., x.add(y).times(z)), which was deemed to be too inconvenient.

The chosen solution was based on the observation that most manipulations of coordinates do not need to access the numerical values of the coordinates themselves. A Coordinate abstract data type class, with suitable method definitions, can hide the fixed/floating distinction from much of the code. For example, application code that needs to compute the distance between two coordinates A and B can invoke A.distanceFrom(B) to obtain an integer value in meters. Programmers whose needs are not supported by existing methods have a choice between writing new methods (allowing their code to operate seamlessly across fixed and floating platforms) or casting the Coordinate to the true fixed or floating subtype, and sacrificing portability for simplicity of development. We incorporated a factory class called `Types` that detects the availability of floating point (using Java's `System.getProperty` method) and manufactures the appropriate Coordinates for the platform, thus abstracting away this particular platform difference from the programmer.

4.2.2 *Cross-Platform Libraries*

Another difference between PC/PDA and smart phone platforms is in the libraries available. In particular, persistent storage access and user interfaces are both provided by different libraries on the two types of platforms.

Persistent storage is treated similarly to real numbers in that the supported storage abstractions are one level up from the typical primitive abstractions (e.g., open, read, write, seek, close), which would not perform well on many platforms. However, there isn't one appropriate high-level abstraction with two obvious implementation alternatives; the anticipated usage patterns over the persistent store affect which storage structure would be most efficient. Consequently, storage-centric Place Lab services are declared as Java interfaces (e.g., User Preferences and Mapper (4.3.2)) and a few obvious class implementations are provided.

User interface abstraction is more difficult to achieve, given the richness of function available (and expected) today. Since it is the applications and not Place Lab itself that interacts with users, the problem of cross-platform user interfaces was not addressed in this framework.

4.2.3 *Native Interfaces*

The final issue with using the Java platform is that many location sensor types are not intrinsically supported; hence Java classes cannot directly access these sensors. The Java Native Interface (JNI) system is useful here, allowing platform-specific sensor "drivers" to be written in another language and accessed by Java. For current smart phones, no JNI support is available; instead, a "loopback networking" paradigm is used to virtualize a sensor as a generic operating system service that Java can access, such as a network stream. More details on implementing various sensors are found in the next subsection.

4.3 Architecture Components

We now describe the main architectural components of Place Lab, namely Spotters, Mappers, Trackers, and the interfaces provided for applications.

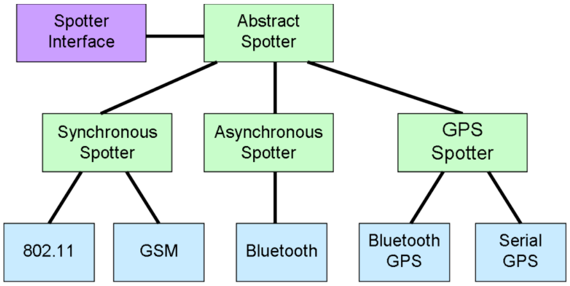

Figure 3. Spotter Hierarchy Diagram. Beacon technology spotters extend the Synchronous or Asynchronous spotter depending on the interface. GPS devices are treated as serial ports that produce NMEA data. The GPS Spotter class handles NMEA parsing and allows for synchronous or asynchronous access.

4.3.1 *Spotters*

Spotters are the components that abstract away the hardware that senses the environment. In the cases where native code is required to interface with the hardware, we have implemented the smallest feasible native part, and performed as much logic as possible in the Java component. This facilitates code reuse; for example, our 802.11 spotter uses a different native part on the Windows Mobile (PDA), Mac OS X (PC), Windows XP (PC) and Linux (PC) platforms, but share the same Java part. Maximizing the reuse opportunities required careful design. The four standard spotters implemented in Place Lab are 802.11, GSM, Bluetooth, and GPS. These technologies are varied in how they obtain data from their data source. The 802.11 and GSM spotters require a native code module that is accessed synchronously; however for Bluetooth, a Java API standard (JSR-82 [2]) is available that returns measurements asynchronously. Supporting these different data access methods, as well as exposing a flexible synchronous or asynchronous interface to outside components led us to the spotter class hierarchy shown in Figure 3.

At the top level, the `Spotter` interface exposes synchronous and asynchronous modes of interaction for outside components to use. The interface also defines the generic methods to support these operations. The `AbstractSpotter` class implements the `Spotter` interface, establishing a framework for the emulation of synchronous calls with asynchronous events, and vice versa. The `AbstractSpotter` is extended by the `SynchronousSpotter` and the `AsynchronousSpotter` classes. The `SynchronousSpotter` provides facilities for emulating the asynchronous interface with synchronous hardware. The `AsynchronousSpotter` provides the converse emulation. The result of this hierarchy is that spotter implementations can subclass either the synchronous or asynchronous spotter class, whichever is more natural for the spotter, and the other interface is automatically emulated.

The `GPSSpotter` superclass handles both the serial port streaming and NMEA data formats provided by GPS devices. Properly speaking, the `GPSSpotter` hierarchy should appear under `AsynchronousSpotter`, as it would eliminate a largely redundant implementation of the synchronous interface.

Spotters communicate with other components using `Measurement` objects. A `Measurement` captures a spotter's observed readings and a timestamp of when the readings occurred. Beacon-based spotters (e.g., 802.11, GSM, Bluetooth) construct `Bea-`

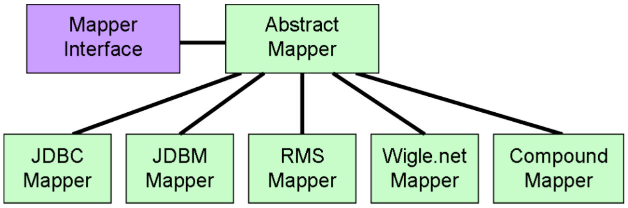

Figure 4. The Mapper Hierarchy. Each class that extends AbstractMapper is able to hold any Beacon type. JDBC and JDBM run on PCs, RMS runs on phones. The Wigle.net mapper uses 802.11 data from the Wigle website. A CompoundMapper can combine any of these other Mappers.

`BeaconMeasurement` objects that are made up of one more `BeaconReading` objects, while the GPS spotter streams `PositionMeasurement` objects that contain `Coordinate` objects.

4.3.2 Mappers

Mappers are static databases of information that are used by trackers to retrieve location information for spotter measurements. The data stored in a mapper always includes a location coordinate, but may include other useful information such as coverage radius. The data to populate a mapper can come from a mapping database, or user-defined files containing known beacon locations. Mappers can also be populated by war-driving data.[2] Constructing the dataset for a mapper can be non-trivial [13]. The cache of data stored in a mapper can be for any size area scale ranging from single cities to the entire world.

Mappers that reside on different systems will require a different method of persistent storage. For example, a mapper using the Java DataBase Connectivity (JDBC) or Java DataBase Manager (JDBM) libraries would work well on a PC, but would not function on a mobile phone. The `Mapper` interface defines the methods a class must implement to insert, query, and retrieve data from the persistent store (Figure 4). The `AbstractMapper` class implements the `Mapper` interface to provide a superclass for all Mapper classes to extend. The superclass also implements caching of data for quick accesses. To date we have implemented several mappers for the PC using JDBC and JDBM, a mapper for the mobile phone that uses MIDP's Record Management System (RMS) interface, and a mapper that draws data from Wigle.net, a world wide 802.11 beacon database. Mapper objects can be composed through a `CompoundMapper` to search through multiple sources of data.

Mappers are generic with respect to the data they store. To achieve this, each entry in the database is represented as a serialized object that includes the name of the class–a subclass of `Beacon`–that represents it. The `Beacon` abstract class is a factory that uses reflection to construct and initialize the appropriate Beacon subclass object for the mapper.

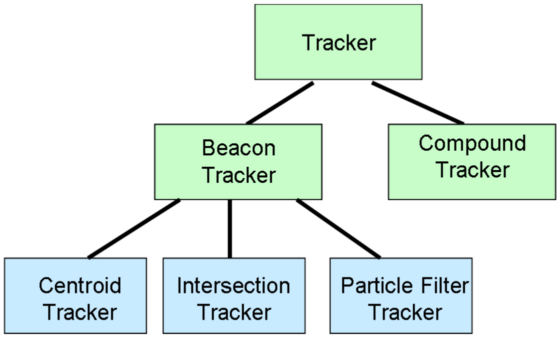

Figure 5. An Excerpt of the Tracker Hierarchy. All trackers extend the Tracker class. Most trackers are single beacon-based and extend the Beacon Tracker class. A Compound Tracker can combine several Trackers together.

4.3.3 Trackers

Trackers are the system components that produce position estimates. The tracker utilizes the stream of spotter observations as `Measurement` objects, together with persistent data from Mappers, to calculate a single position `Estimate`. In doing so, Trackers may perform sensor fusion by combining data from multiple types of sensors with different characteristics. `Estimate` objects are a subclass of `Measurement` allowing the estimates of one tracker to be used as input to another tracker (Figure 2c). The complexity of trackers varies enormously, from simply finding the centroid of recently seen beacons' positions to trackers that take into account signal strength, propagation models, environment information, and physical world models.

The `Tracker` class defines the methods that all trackers must implement (Figure 5). Each tracker must implement a method to update its position estimate when receiving a new spotter measurement, filtering out any unwanted measurements that may be provided. For example, some trackers may not be able to understand GPS Measurements. If an application or another tracker is registered with the tracker, the update of a tracker's estimate will result in estimate event being announced. Regardless, the updated estimate is available through a procedural interface as well. Multiple trackers can be composed using a `CompoundTracker`. The `CompoundTracker` updates each individual tracker separately and returns a compound estimate that contains the estimates from each tracker. Numerous trackers have been implemented in Place Lab [5].

4.3.4 Platform/Application Adapter—Façade

When Place Lab is instantiated, it must be adapted to the platform, available sensors, and the application. In a few cases runtime checks are used to detect the available sensors, but generally the configuration is determined by how the `Placelab` adapter object is subclassed and instantiated. The `Placelab` constructor accepts a tracker, mapper, and list of spotters, and plumbs them into the specified configuration. An application then obtains location information by communicating with the `Placelab` object by one of several means, described below.

Place Lab currently runs in many different platform configurations, as shown in Figure 6. Several `Placelab` objects and subclasses exist to provide convenient preconfigured combinations for several platforms. For example, because of platform limitations and available spotter technologies, the `PlacelabPC` object for the PC platform instantiates a different set of spotters than the `PlacelabPhone` object for the phone platform.

---

[2] War-driving is the act of driving around with a mobile device equipped with a GPS device and a radio (typically an 802.11 card but sometimes a GSM phone or Bluetooth device) in order to collect a trace of network availability.

Operating Systems	Architectures	802.11	GSM	Bluetooth
Windows XP	x86	●	●*	●
Linux	x86, ARM, XScale	●		
Mac OS X	Power PC	●		
Windows Mobile	ARM, XScale	●	●*	●
Symbian	Series 60 phones		●	●

Figure 6. Platform configurations that Place Lab currently runs on. All platforms also can access GPS devices for location. Place Lab is able to use GSM on the Windows XP and Windows Mobile platforms because of a remote GSM spotter over Bluetooth, discussed in Section 5.2.

Place Lab provides five interfaces for communicating location information to applications; one directly connects to the Placelab object, and the others provide the Placelab data as an existing standard service. The availability of these services means that an application that already uses location via an existing standard may require no modification to use Place Lab.

1. **Direct Linking.** Applications may communicate with the `Placelab` object directly. For applications that use a preconfigured Place Lab object, they can invoke a single method to start the location tracking service. The application can use either an asynchronous or synchronous interface to obtain position estimates from Place Lab.
2. **Daemon.** For some applications, it may be desirable or necessary to not link them directly to Place Lab. To support such applications, Place Lab can be run as a daemon and be queried via a simple HTTP interface. This interface allows programs written in a wide range of languages and styles to use Place Lab.
3. **Web Proxy.** A web proxy interface uses Place Lab functionality to support location-enhanced web services by augmenting outgoing HTTP requests with extension headers that denote the user's location. By configuring web browsers to use this proxy (in the same way one uses a corporate firewall's proxy), web services that understand the extension headers can provide location-based service to the user.
4. **JSR 179.** To support existing Java location-based applications, Place Lab can provide location through the JSR-179 Java location API [3].
5. **NMEA 0183.** Place Lab provides a virtual serial-port interface that mimics an external GPS unit by emitting NMEA 0183 navigation sentences in the same format generated by GPS hardware. Since many applications (e.g., Microsoft MapPoint) already understand NMEA, they can seamlessly take advantage of location functionality developed using Place Lab (which might operate indoors, unlike GPS).

5. DISCUSSION

To review, the Place Lab architecture and infrastructure was designed to serve as a playground for researchers to explore the domain of location-based computing. First, we wanted to enable application designers to focus on the details of their applications rather than the hardware, data management, and algorithms of location sensing. Second, for those performing research in location sensing, we wanted to support modular innovation, enabling them to freely exchange and compose their components with others' into the configurations best suited for their work.

In this section we evaluate the architecture in achieving these goals. First, we provide some data to shed light on the level and kinds of use Place Lab is seeing in the research community. Second, we discuss three informal case studies on three unanticipated extensions of PlaceLab.

5.1 Place Lab in the Wild

The Place Lab toolkit, available through SourceForge.net and placelab.org, has been downloaded more than 4585 times in the year since its initial release in April 2004. The download activity reflects the high interest by the research community to explore location-based computing.

The effectiveness of Place Lab is shown by the activity among researchers who are using it to innovate in positioning algorithms and prototype location-aware applications. Place Lab is enabling researchers to quickly experiment with their algorithms, and take many ubiquitous computing applications beyond the confines of the controlled research setting and into the wild. At the University of Washington and Dartmouth, Place Lab has been used as a part of several class projects in location-aware computing. Researchers are currently using Place Lab to conduct experiments with graph-based tracking algorithms, multi-floor location estimation and GSM fingerprinting. In addition, several location-aware applications using Place Lab have been developed by us and the user community:

- **Topiary.** Topiary is a rapid prototyping tool developed at UC Berkeley for designing location-enhanced applications [14]. A Topiary prototype can be run on one mobile device while the designer monitors the user's interactions from a second mobile device. In this mode, the user's location is determined in a Wizard-of-Oz-style by the designer who can change the user's current location by clicking on a map. Topiary has been extended to allow the use of live location estimates from Place Lab running on the user's device. Place Lab has proven especially useful because it can operate indoors and, permitting Topiary to be used in a wide variety of settings.
- **ActiveCampus.** The ActiveCampus project is one of the more widely used 802.11-based location-enhanced systems [8]. ActiveCampus offers a suite of community-oriented applications to students on the UC San Diego campus. The Active Campus project is currently using the Place Lab architecture for their location technology. Place Lab is enabling ActiveCampus to run on a wider array of devices than was possible with their earlier in-house spotter technologies.
- **Place Bar.** PlaceBar is a demonstration application developed at Intel Research Seattle that uses a browser toolbar to manage a user's interactions with Google's location-based search engine, http://local.google.com/. In addition to the query terms, Google Local accepts an address or latitude/longitude, and the

results are filtered to return pages relevant to nearby places. (Google estimates a pages location by extracting information like addresses and phone numbers from the page content.) When a query is performed in the PlaceBar, the user's location is obtained from Place Lab via the web proxy adapter, and it is automatically used as the location for the query.

- **A2B.** A2B is an online catalog of web pages that allows users to add new geocoded pages (pages tagged with location metadata) or query for nearby relevant pages (http://a2b.cc/). A2B can be queried either by manually entering a location or with a custom client that talks to a GPS unit. A2B extended their interface to support HTTP requests from clients running the Place Lab web proxy. Devices running the Place Lab proxy can now talk directly to A2B in any web browser and automatically use their location-based lookup service.

5.2 Case Studies

5.2.1 Motorola V300

The Motorola V300 is a popular phone supporting Java J2ME, with several hardware and software differences from the Symbian Series 60 phones already supported. We now discuss the relevant differences and their implications for the Place Lab toolkit.

The V300 does not provide native programmability like the Symbians, and instead provides for directly accessing GSM beacon information within Java. However, this method only provides access to the Cell ID variable, as opposed to the cell ID, area ID, network code, and country code variables available on the Series 60 phones. Without these three other pieces of information, it is impossible to form a unique key to look up a beacon's location in the Mapper. This is because cell IDs may be reused across different areas, telephony providers, or countries.

We first dealt with the different means of access, using a runtime-detection approach in `GSMSpotter` (Figure 3), which expects to get the location via a native component accessed through a loopback. The code was extended to initially call `System.getProperty("Cell ID")` to see if it returned a valid (e.g, non-null) cell ID. If so, this means the software is running on a device that does not need a native component. Otherwise the spotter will attempt to use the native component to obtain GSM information. For this change, one method was modified in `GSMSpotter` and another added, for a total change of 11 Non-Comment Source Statements (NCSS).

Second, we modified the `RMSMapper` component (Figure 4) to handle non-unique keys. Since the V300 only provides one part of a four part unique key (cell ID:area ID:MCC:MNC) the `RMSMapper` cannot do a direct lookup to find matching beacons. Consequently, the `RMSMapper` was modified to find the relevant beacons with only a matching cell ID. If more than one beacon matches the cell ID, all of the matching beacons are returned. Receiving a list of matching beacons is already expected by the trackers, so no modification to a tracker is necessary unless the tracker algorithm specifically depends upon uniqueness.[3] One method was modified and another method was added, for a total change of 39 NCSS.

With these small and local modifications the Place Lab software was successfully ported to the V300 device. No modifications were needed for the Tracker or existing applications.

5.2.2 Remote GSM Spotter

Providing a local interface to an existing remote spotter displays a unique dimension of flexibility. A remote spotter provides the ability to combine the strengths of two platforms to achieve a superior result. In this case, we demonstrate making GSM measurements available on a laptop, thus achieving virtually ubiquitous location sensing of the mobile phone platform [13] on a device with considerable computational power and GUI capabilities.

In particular, we extended Place Lab to provide a GSM-over-Bluetooth spotter. The remote spotter requires a new class that runs on the master device and an application on the phone to obtain the needed GSM measurements.

The first change was to develop a J2ME MIDlet for the phone that advertises itself as a remote GSM spotter over the Bluetooth interface. The `GSMBTMidlet` application uses `GSMSpotter` without modification to obtain the cell measurements, and stores them in a buffer. The application required 210 NCSS. With the `GSMSpotter` extension discussed in the previous section, the remote spotter runs on both the Symbian Series 60 and Motorola V300 phones.

The second modification was to add a `RemoteGSMSpotter` class that discovers the remote GSM spotter service and periodically polls the phone via Bluetooth to read the buffer of cell readings. The `RemoteGSMSpotter` extends the `SynchronousSpotter` (Figure 3), fitting easily into the Spotter abstraction. Since much spotter functionality is abstracted away in `SynchronousSpotter`, the `RemoteGSMSpotter` required only 108 NCSS. It can be instantiated on any device that is equipped with a Bluetooth radio. It is currently in use on the Windows XP and Windows Mobile platforms (Figure 6).

5.2.3 Fingerprint Tracker

The location-aware computing literature is full of location estimation algorithms. Not all algorithms fit the typical Place Lab model of estimating a device's position from the positions of well-known beacons. For example, RADAR uses a technique known as *fingerprinting*: it relies on the fact that at a given position, a user may hear different beacons with certain signal strengths; this set of beacons and their associated signal strengths represent a fingerprint that is unique to that position [4]. RADAR compares the readings generated by the spotter to a database of pre-collected fingerprints from previous war drives, and places the user at (or near) the fingerprint(s) that most closely match the readings obtained from the spotter. RADAR uses Euclidean distance in signal space as its comparison function. A related algorithm, RightSpot uses relative rank ordering based on signal strength as its comparison function [12]. Thus, adding a fingerprinting tracker to Place Lab is a good test of its adaptability.

The fingerprint tracker depends on a different kind of mapper that, instead of aggregating information for each beacon into a single location estimate, keeps track of all the raw fingerprints gathered during previous mapping war drives. Each fingerprint is composed of a set of { beacon-id, signal-strength } tuples obtained in a scan and the location where the scan was taken. The mapper is queried with a measurement to find all fingerprints that share beacons with the supplied measurement. By not requiring a strict fingerprint match, the algorithm is tolerant to missing or newly

---

[3] Trackers are generally written in a defensive manner, since inconsistencies abound, such as access points being moved or reporting non-conformant ID's.

deployed beacons. To support efficient retrieval of this kind from the large fingerprint corpus, a modular hashing method using MySQL's bitwise comparisons was formulated. As a consequence, a special fingerprint mapper was implemented, rather than using the existing JDBC mapper or JDBM mapper.

The `FingerprintTracker` receives a set of readings from a spotter, queries the `FingerprintMapper` for all matching fingerprints, and estimates the position of the user based on either the RADAR or the RightSpot algorithm. Details of these algorithms and their use in Place Lab are available [5].

The `FingerprintTracker` is 106 NCSS, and the `FingerprintMapper` is 315 NCSS. The resulting tracker is an interoperable component of the Place Lab infrastructure, usable on any PC/PDA platform that can provide 802.11 measurements. However, the novel performance and functional requirements for the mapper entailed implementing a new one from scratch, making this case study a limited success. Another iteration on this project could result in the mapper being subclassed from one of the existing mappers, or perhaps generalizing the fingerprint mapper to be independent of the fingerprint data representation, admitting wider reuse.

5.3 Future Work

Place Lab is a first-generation architecture for LBC. Experience has been a good teacher, but not all its lessons have been incorporated into Place Lab, and some lessons have yet to be taught. Minor issues include the proper integration of GPS spotters into the framework, as discussed in Section 4.3.1.

As future work, we might consider the implementation of more sophisticated factories for helping users build instantiations of Place Lab, perhaps using Open Implementation [11]. However, platforms like phones dictate that much of the configuration be determined at compile time, both to exclude the burden of storing unneeded (and non-J2ME) code, but also for performance reasons.

A larger issue is the integration of *place* into Place Lab. By place, we mean personal or conceptual places like "home" or "can buy stamps here". There may be no coordinate system per se for place (and no apparent distance between two places), but coordinates pervade Place Lab. The place-based applications that have used Place Lab to date have simply appropriated its spotters. Thus Place Lab has been enabling to those projects, but their results could not be distributed as interchangeable Place Lab components. Place is certainly permitted in the architecture, but it is unclear what impacts place might have on the design of Place Lab's core interfaces if it were to become a first class citizen. Mappers return "beacons" and trackers emit coordinate-based estimates. The orthogonality of location and place, naïvely treated, could result in considerable dynamic type checking (i.e., Java downcasts), among other issues. Java 5's generics provide an opportunity, but our use of J2ME to support lightweight clients currently precludes its use.

6. CONCLUSION

Location-based computing is a nascent, active research area comprising several major research topics, in particular sensing, inferencing, and applications. The Place Lab client-side architecture for LBC was designed to support portable modular innovation in each of these topics. Location is only one type of context to appear on personal devices, and our experiences provide an informal roadmap for future developers of context-aware systems.

The principled use of design patterns, notably the type-compatible stacking of sensors, trackers, and mappers into a multi-level mediator-observer design pattern, provides exceptional independence, interoperability, and composability. The Placelab façade hides these details from applications, and a separate toolbox of proxies to standards-based services enables location-based applications to adopt Place Lab with no change. The pervasive availability of asynchronous event interfaces and synchronous procedural interfaces–provided at no development cost to those who extend Place Lab–further minimizes the constraints placed on developers.

Place Lab's early successes are encouraging, although challenges remain. The incorporation of conceptual places is a major consideration. Also, the mobile phone market is unruly, and may provide unexpected challenges in future ports.

7. ACKNOWLEDGMENTS

The authors thank the many contributors and users of Place Lab for their support.

8. REFERENCES

[1] NMEA 0183. http://www.nmea.org/pub/0183/

[2] Java Bluetooth API (JSR-82). http://www.jcp.org/en/jsr/detail?id=82

[3] Java Location API (JSR-179). http://www.jcp.org/en/jsr/detail?id=179

[4] Bahl, P. and Padmanabhan, V. RADAR: An In-Building RF-based User Location and Tracking System. In *Proceedings of IEEE Infocomm* 2000, pp. 775-784.

[5] Cheng, Y., Chawathe, Y., LaMarca, A., Krumm, J. Accuracy Characterization for Metropolitan-scale Wi-Fi Localization. In *Proceedings of Mobisys 2005.*

[6] Dey, A.K., Salber, D., Abowd, G.D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *HCI Journal* 16(2-4) 2001, 97-166.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software* Reading, MA, Addison-Wesley, 1995.

[8] Griswold, W.G., Shanahan, P., Brown, S.W., Boyer, R., Ratto, M., Shapiro, R.B., Truong, T.M. ActiveCampus – Experiments in Community-Oriented Ubiquitous Computing. *IEEE Computer*, Vol. 37, No. 10., pp. 73-81, October 2004.

[9] Hightower, J., Borriello, G. Particle Filters for Location Estimation in Ubiquitous Computing: A Case Study. In *Proceedings of Ubicomp 2004*, pp. 88-106.

[10] Hightower, J., Brumitt, B., Borriello, G. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *Proceedings of WMCSA 2002.*

[11] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G., Open Implementation Design Guidelines, *1997 International Conference on Software Engineering (ICSE)*, May 1997.

[12] Krumm, J., Cermak, G., Horvitz, E. RightSPOT: A Novel Sense of Location for a Smart Person Object. In *Proceedings of Ubicomp 2003*, pp. 36-43.

[13] LaMarca, A., Chawathe, Y., Consolvo, S., Hightower, J., Smith, I., Scott, J., Sohn, T., Howard, J., Hughes, J., Potter, F., Tabert, J., Powledge, P., Borriello, G., Schilit, B. Place Lab: Device Positioning Using Radio Beacons in the Wild. In *Proceedings of Pervasive 2005*.

[14] Li, Y., Hong, J.I., Landay, J.A. Topiary: A Tool for Prototyping Location-Enhanced Applications. In *Proceedings of User Interface Software and Technology 2004*.

[15] Madhavapeddy, A., Scott, D., Sharp, R. Context-Aware Computing with Sound. Proc. of Ubicomp 2003, pp. 315-332.

[16] Marmasse, N., Schmandt, C.Location-aware information delivery with *comMotion*. In *Proceedings of Handheld and Ubiquitous Computing (HUC) 2000*, Bristol, England, pp. 157-171.

[17] Smith, I., Consolvo, S., LaMarca, A., Hightower, J., Scott, J., Sohn, T., Hughes, J., Iachello, G., Abowd, G. Social Disclosure of Place: From Location Technology to Communication Practice. In *Proceedings of Pervasive 2005*.

[18] Steinberg, A.N., Bowman, C.L., and White, F.E. Revision to the JDL data fusion model. *In Proceedings of SPIE Aero-Sense (Sensor Fusion: Architectures, Algorithms, and Applications III)*, pp. 430-441, Orlando, Florida, 1999.

[19] Sullivan, K.J. and Notkin, D. Reconciling environment integration and component independence. *In Proceedings of the SIGSOFT '90 Fourth Symposium on Software Development Environments*. pp. 22-33.