Data Store Issues for Location-Based Services

John Krumm   Steve Shafer
Microsoft Research
Redmond, WA 98052
{jckrumm, stevensh}@microsoft.com

Abstract

Location-based services (LBS) provide resources or information based on the user's own location or an indicated location of interest. This paper introduces a number of data store and access requirements for LBS based on examples from our own work and that of others in the field. We consider specifically the areas of location representation, gathering location data, assets, location-based queries, and location system architecture, all of which are important to location-based systems and are extensions and specializations of current database issues.

1 Introduction

Location-based services (LBS) provide resources or information based on the user's own location or an indicated location of interest. For example, an automobile's navigation system may display the locations of nearby gas stations when the car's tank is getting low, a cell phone may switch to vibrate when a person enters a theater, or a document may be directed to print near the location of a person's next meeting. A data store of some sort is needed to represent the locations in the world, as well as their attributes and relationships, and the resources available. This data store is used for interpreting sensor readings, performing spatial queries and inferences, and triggering actions. In geographic information systems (GIS), the data store is usually a geospatial database; in many indoor ubiquitous computing systems, the data store may be a simple as a drawing file. In any case, location-based services have requirements that challenge traditional data representation systems. This paper introduces a number of these requirements based on examples from our own work and that of others in the field. We consider specifically the areas of location representation, gathering location data, assets, location-based queries, and location system architecture, all of which are important to location-based systems and are extensions and specializations of current database issues.

2 Location Representation

Geometry and Objects: All location-based systems need to represent locations. The representation can be as simple as a few items in a list or as complex as a full geographic information system. Two of the more natural representations of location are metric and object-oriented. A metric representation consists of numerical coordinates (e.g. (x,y,z) and (latitude, longitude)) to specify points and shapes. In our EasyLiving project [BMK+00, BS01],

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

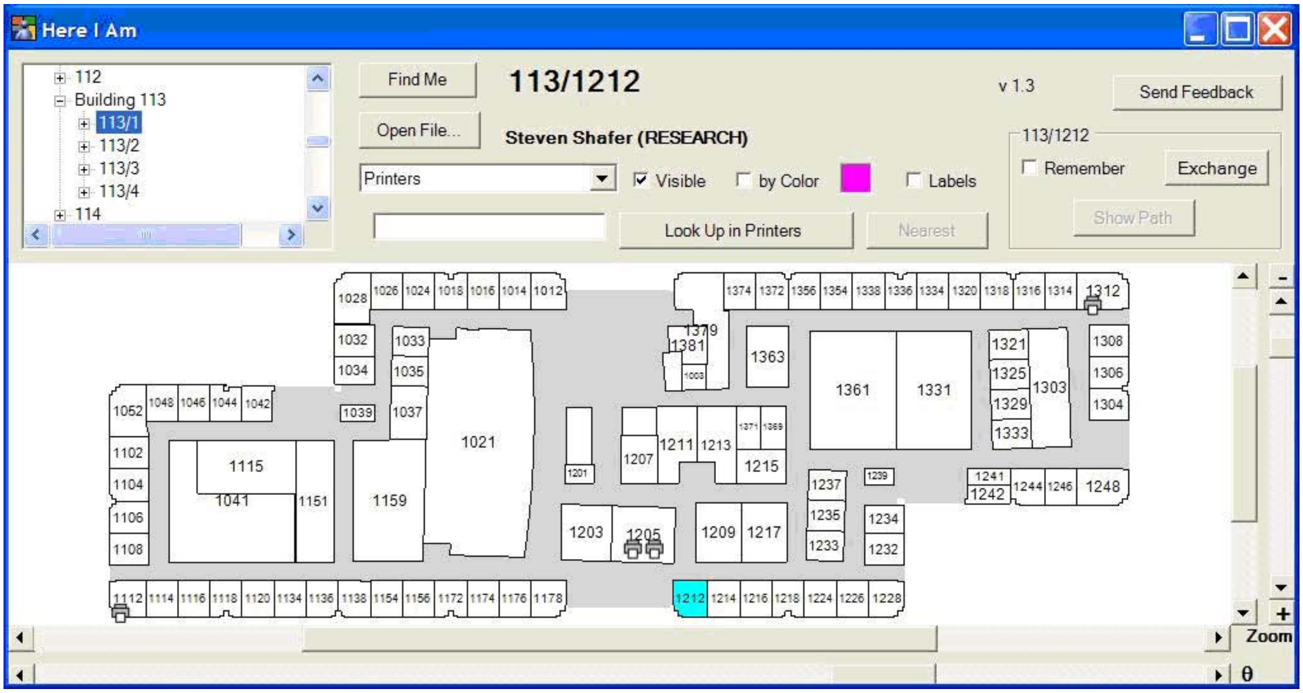

Figure 1: Our “HereIAm” program displays floor plans and contents of rooms. It represents spaces with coordinates and containment relationships.

we used a metric representation to give the locations of devices, furniture, and people on a 2D grid on the floor. This made it easy to compute when, for instance, a person passed within a certain distance of a computer screen. A metric representation may have to support multiple coordinate systems and transformation between them.

A common alternative to a metric representation is an object-oriented representation which, at its simplest, is just a list of locations, such as “kitchen”, “living room”, “bedroom”. An object-oriented representation makes it easy to specify containment relationships in a hierarchy, such as the fact that a number of different offices are in a given wing of a building, which is on a given floor, which is in a given building. Representing spaces as objects, however, leaves out critical data for computing metric relationships like distance. This can be fixed with a hybrid approach that represents spaces with both coordinates and objects. As an example, we built the “HereIAm” program, shown in Figure 1, to represent building floor plans. HereIAm uses a hierarchical representation to relate the spaces on a given floor, the floors of a given building, the buildings on a given campus, etc. It uses a metric representation to draw all the spaces to scale. Some systems that use objects have a type definition for various types of location objects (building, floor, room, conference room, etc.) - in HereIAm, the appearance attributes of a room were given in such a type definition, but these attributes could be overridden for individual objects, for example to highlight the currently selected room in blue in Figure 1.

The Third Dimension: Another key issue is the representation of the third dimension. In many systems, the third dimension is completely absent, for example a mapping program that treats the Earth’s surface as a spheroid. When the third dimension is introduced outdoors, complexities arise from the need for GPS (or other location sensor) to have more points of reference in order to determine the third dimension; the lack of standardization of the unit of height in standard outdoor coordinate system definitions; and of course the storage and rendering complexity of 3D polyhedra v. 2D polygons. Many systems do record and/or display true 3D, for example game worlds and topographic maps. Even modern aerial photograph systems are beginning to map the images onto a 3D world map with elevation, such as Google Earth (`http://earth.google.com`). Indoors, a single floorplan may be naturally represented as a 2D drawing, and this is satisfactory for many purposes. 3D indoor models are much more complex, typically used for the construction industry but not so much for consumer applications. The complexity of 3D is frequently not warranted for a given application. However, there are other choices that capture some of the elevation information such as “2-1/2 D” geometry. The definition of this term is idiosyncratic, but an example would be to store space definitions as 2D polygons in XY, with the Z coordinate a simple ordinal quantity (1 = first floor, 2 = second floor, etc.) Our EasyLiving system worked entirely in 2D, though HereIAm had a simple version of 2-1/2 D as presented here.

Associated Data: Frequently, a location data store records more than just the location descriptions, it records additional data associated with the locations. One kind of associated data is properties or attributes of each location, such as whether it is a space or a route segment; or whether it is displayed by outline, displayed by coloring its extent, or not displayed at all. Attributes can also describe how a space is used, how it is classified for tax or rental purposes, who is responsible for administering it, what its name is, and so on. Space attributes can be grouped, and sometimes this grouping forms a “type” system with spaces of each type sharing certain attributes. The type system can even allow subtyping with inheritance of attributes. Another kind of associated data found in location systems is relationships between locations, such as containment or adjacency. Storing such relationships makes it more efficient to search through the location data store, but the relationships must be computed and stored. One of the beauties of geometric coordinates is that a great many relationships - including containment and adjacency - can be computed dynamically, without the need to pre-compute and store them in the data store explicitly. However, performing such computations at run-time can be expensive - potentially extremely expensive. So, a design engineering process is needed to decide what relations to record explicitly in the data store, and what relations can be calculated dynamically.

Uncertainty: Many times location data comes from sources that are not completely reliable, such as the video cameras we used in EasyLiving [KHM$^+$00] or the Wi-Fi access points we used in LOCADIO [KH04b] to track the locations of people and devices in our space. In such cases, it is important to explicitly represent the probabilistic nature of the measured locations to avoid hiding the inherent uncertainty from down stream processes. Uncertainties can be represented as parameterized distributions like Gaussians, which are easy and compact to represent. Or they can be represented nonparametrically as, say, a cloud of particles [DdFG01], which is less efficient but more expressive.

Federation: No one database will contain all location data for all locations. Different databases will cover different geography as well as different objects. For instance, an enterprise’s buildings and campuses may be represented in different files as we did in HereIAm. Different authorities may control data for different types of assets: the IT department may maintain the locations of printers and wireless access points, while the facilities department may maintain floor plans. In these cases it is important that the representation allow for the federation of different databases, perhaps by supporting pointers to data maintained by other authorities. When federation occurs across different representation systems, some interoperability standard is needed to allow locations in one system to point to locations in another. The goal of federation is to implement functional computations smoothly, as if disparate location data stores were parts of a single “virtual” unified data store.

3 Generating Location Data

Authoring and Editing: Location data can come from many sources and go to many destinations. For inputting primarily static data, such as floor plans and geographic entities, a one-time conversion into the chosen representation is sufficient, if sometimes tedious. Once in the database, location data sometimes needs manual editing to stay up-to-date. Instead of making the human editor work with traditional database update statements, it is much more practical to use a friendly, graphical front end to the database so editing becomes more like running a drawing program. When editing an existing location data store, a common approach is to update the source data (e.g. architectural drawing), and rerun the analysis program that creates the location data store representation. However, this can lead to changing the internal ID numbers associated with locations, which may be used by applications. Maintaining location ID across edits is a key problem. Similarly, it may be desirable to maintain a history of ID and other changes as an attribute of locations (current or past), such as when a wall is added or removed from a structure and the room numbers are reassigned accordingly. HereIAm uses drawings as source data; the room numbers are assigned by hand in the source drawing, and the internal IDs are derived from building names and room numbers for continuity across floorplan modifications.

Self Maintenance: Sometimes location data can be generated from the database itself. Our NearMe system [KH04a] finds nearby people and resources based on Wi-Fi access points. As people submit sets of detected access points, the NearMe server recomputes every hour the topology of overlapping access points based on which ones have been detected together. The new Location Finder in Microsoft's Virtual Earth (`http://virtualearth.msn.com`) anonymously logs access point detections to try to infer the locations of access points it has not yet seen. In these self-maintenance schemes, it is important to guard against mistaken data, either as a result of inaccurate sensors or malicious data submissions.

Use-Based Location Definition: Self Maintenance can be extended to actually identify locations of interest based on observations of "breadcrumb" data (the history of people's movements). Whenever a region has a pattern of occupancy or movement very different from its surroundings, it may be useful to define a boundary and declare it to be a place of interest. An object can be created, and it can be entered into the location representation system. For example, the Lachesis project [HT04] automatically finds a user's frequent destinations which an experimenter can manually name, while Opporunity Knocks [PLG$^+$04] asks the user to take a picture of any place it recognizes as a frequent destination. In this way, from breadcrumb data, a network of interesting places and paths can be defined based on the actual usage of spaces - transit areas, meeting areas, resting areas, etc. Three tasks that are particularly challenging are defining the boundaries of the new space, interpreting what it is used for so its attribute values can be assigned, and giving it a name that can be advertised and used in external references.

4 Locations and Assets

Assets: One key function of location-based systems is to provide access to assets of various kinds based on location. Assets can include physical things like printers or goods in a warehouse; information such as a building directory or facts about a specific painting on the wall; services such as requesting maintenance or obtaining navigation directions; or even people whose locations may be determined by sensors. Generically, we can think of an "asset roster" as a list of assets and for each, its location or service region. However, in practice, there are many ways to represent an asset roster. In one extreme, only locations are represented, and each asset is recorded as a location just like any other. For example, a printer in a room is represented as a new location object at the appropriate place and with the appropriate relations with other locations. The attributes of this location contain the detail about the fact that it's a printer. This makes queries easy, but shoehorning all interesting facts about the world into a location data store is awkward. The other extreme is to say that there is only a "data store of things", and all locations are represented as "things", with some relations between them. This also simplifies some aspects of design; but it is awkward, for example, to perform geometric computations or route selection if your only data store is generic for all kinds of things. These extremes can be characterized as "all things are locations" or "all locations are things". A more sophisticated design distinguishes between locations in the location data store and asset rosters which provide a list of assets and their corresponding locations. HereIAm distinguishes locations from assets, and allows "click-through" on the icon of any displayed asset to view or edit its properties. However, HereIAm uses a declarative representation of asset rosters, which provides no means to propagate changes back to the source data store that generated the asset roster.

Asset Roster Representation: Asset rosters can be represented in many very different ways. The simplest presentation might be a file or database table with one row for each asset, and columns for the asset ID; for other asset information such as a name, type, and pointer to more information; and for the location of the asset. An asset roster can be a file, a table in a database, or a composite of numerous tables or files. An asset roster can be a drawing file with things drawn onto it as icons or as geometric figures. In some systems, each asset is a device that maintains its own location property, which can be queried, so the "asset roster" is implicitly the distributed set of location properties maintained by all devices. There may be many asset rosters for a given location domain, and they may be administered by different authorities - for example, the Real Estate staff of an enterprise may keep the floorplan drawing database, whereas the IT staff maintains a roster or printer locations within the buildings. Issues of Federation (see above) are particularly important for asset rosters. In addition, the jurisdictional scope of an asset roster may not match that of the location store, for example the Real Estate staff at each site may maintain the building floorplans for that site, but the printer database may span several sites.

Mobile Assets: Some assets move in the world, so their location must be queried dynamically, and may be remembered in a historical record. Location updates can come from live sensors, such as the people- and device-trackers we have developed. In these cases, the database must support programmatic, real time updates. For objects that move, it is important to represent dynamics such as velocity to answer queries related to speed and to make predictions about where an object will be. In LOCADIO we used the variance of Wi-Fi signal strengths to infer probabilistically whether or not a client was moving. Representing dynamics for location-based services starts down a slippery slope of full context representation, such as a user's current activity or a conference room's availability. Asset rosters are frequently queried by copying all or part of the roster into the client application's memory space, but for dynamic assets, this may be prohibited, requiring that locations be queried every time they are to be used by the application, or according to some schedule or plan.

Own Location: Perhaps the most challenging kind of mobile asset is people, whose locations may be determined by external sensors, by a sensor they carry, or through a computing device whose location is measurable. People move around, sometimes in relatively unpredictable ways, sometimes using spaces in ways the designers did not intend, with many variations in the significance of their location. A person's location can be viewed from the standpoint of the person or the environment. From the person's standpoint, "own location" means where the person is located right now. For example, a building directory application might use the own location to select which floor's floorplan to display (this is done in HereIAm). From the environment's standpoint, all people's locations can be collectively considered to form a single asset roster of mobile assets, i.e. people. There are deep privacy issues around the disclosure of own location to others, including the centralization of own locations into a single server or data store. In addition, some location sensing systems determine own location on each mobile device (e.g. using Wi-Fi signal strengths in LOCADIO), while others are central systems that determine the locations of all mobile devices through facilities in the infrastructure of the environment (e.g. WhereNet Corporation, `http://www.wherenet.com/pdfs/VSS.9.9.03.pdf`). In the latter case, a person or device that desires to know its own location must obtain it by querying an asset roster in the environment.

5 Location-Based Queries

Geometric Queries: Location-based services are challenging in terms of database querying because of the specialized geometric nature of the queries and because of the multitude types of triggerable events. Clearly geometry is important for spatial queries. In our EasyLiving project, one of our most frequent queries concerned whether or not a person, respresented as an (x,y), had entered a particular space, such as the area around a computer monitor. Our XRay local search program [HKH05] makes queries about points of interest inside a cone-shaped region emanating from a user's current (latitude, longitude) pointing toward wherever the user is pointing his mobile device. Queries that span coordinate systems have to take into account coordinate transformations, including possibly discovering and computing long chains of transformations to connect the relevant frames of reference. Queries involving multiple shapes are also important. As an example, the region from which an audio speaker can be heard is the intersection of its audible region in free space and the walls of the room around it. Incorporating dynamics introduces the element of time, such as a query asking when a certain velocity vector will intersect a certain region. Probabilistic representations need probabilistic queries, such as, "Show me the minimum area over which the probability of a given person's location integrates to 0.99".

Precomputation: The answers to common queries can be saved for efficiency. For instance, a common query on floor plans is to list all the spaces on a floor of a given building. Similarly for routing, commonly requested routes can be stored to avoid recomputing them. One recent example of the power of precomputation for location-based services is the new Google Maps (`http://maps.google.com/`) and Microsoft's Virtual Earth (`http://virtualearth.msn.com`). Previously, Web-based maps were generated on demand, so panning even a few pixels required a complete rerendering. With precomputed tiles, new Web map cites deliver fast, interactive panning. However, prerendering may require that the zoom levels be coarsely quantized, and map labels may often be obscured by superimposed annotations.

Privacy: With location-based services receiving more attention, location privacy is the subject of increasing concern and research. Databases can help by enabling location queries that return locations of specified resolutions. This would allow, for instance, a user to specify that he is willing to share his location down the city level, but not at any higher resolution. Likewise, it can be valuable to have a database deliver outright lies about a user's location in order to reduce the confidence of anyone spying. These techniques are sometimes called "location dithering", and representative work appears in [DK05]. Privacy can also include hiding information within the location data store, for example certain buildings in a corporation may contain sensitive resources and their floorplans might not be available for viewing by all employees.

Triggers: Some queries run in the background in order to trigger location-related events. In the EasyLiving project, our database triggered events based on the room's behavior rules, such as automatically adding a user's music play list to the room's list of playable songs whenever the user entered the room. Sometimes events trigger actions outside the database, such as redrawing a space when something or someone moves. While it is usually sufficient to recheck trigger conditions whenever the database is updated, triggers can also be conditioned on inaction. For instance, in elder care, it is important to be notified if a subject has not left his or her residence for a long period of time. Just as with editing a spatial database, authoring and editing location-based triggers would be easier with a graphical interface that helps a programmer define the spatial preconditions for an event to fire.

6 Location Computing Architecture

Location Engine Structure: A location-based service needs a body of code that understands the representation and can evaluate queries such as "find nearest thing" and "find route". This body of code can be called a "location engine". One of the key design issues in location-based services is where the location engine resides - in the client (application), or in the location data store (server), or split across the two. When the data store is a true database system, or presents a server interface, the location engine can be inside the data store. A client need not understand the location representation, it can simply send a query to the data store, whose location engine will compute the desired result. This centralized approach involves small but relatively frequent network traffic, and allows very thin client applications. However, it also requires that all location calculations for all clients share the same server to do the heavy work, and it requires connectivity to the network in order for the application to execute (unless the client has precomputed and stored the results of certain queries before becoming disconnected). Such a design also gives the client a very limited language in which to express its queries, i.e. the network interface presented by the data store. An example of this is the MapPoint Web Service (`http://www.microsoft.com/mappoint/products/webservice/default.mspx`), an on-line server (XML Web Service) providing client applications with maps and routing information about outdoor locations. On the other hand, if the location engine is inside the client, then the data store is simply a repository for blocks of location data that can be uploaded by the client at will. This allows the client to pose essentially arbitrary "queries", since the programming language serves as the means of access to the cached location data. Network traffic will be "bursty", but clients are expected to cache whole chunks of the location model, thus eliminating the need to go back to the data store for every query. Predictive prefetching can be used to cache more, reducing latency in answering likely future queries. But, any degree of caching introduces the problem of maintaining consistency when the location data store changes. A client-based location engine can be thought of as a single distributed location engine, which is part resident on every client. HereIAm uses a file repository as the data store, with the location engine compiled into client applications. Alternatively, a system could have a hybrid design, with the location engine split into parts that run on the client system, and other parts that run on the data store server. In a hybrid system, the division of labor between the server and client sides can be viewed as a continuum, and might be tuned during the design process for the data store, or tuned dynamically based on system performance, e.g. delivering larger data chunks when server and network are busier.

Location-Independent Applications: It would be desirable to have a location-based service application that can run against many different location data stores, for example an application to display a building directory on your cell phone or laptop, which will work in any building. Each building is its own administrative jurisdiction, and different buildings may use different systems to represent their location data stores. An interoperability interface to the location data store would be needed to achieve location-independence in the application. If the location engine is in the data store, then this interoperability interface would present generic location queries such as "find nearest", and all the work would have to be done by the server. On the other hand, if the location engine is in the client, then the interoperability interface governs the fetching of chunks of location data, which must all be decodable by the same location engine in the client. A hybrid system would be more challenging to design in this context, with the location engine split into client and server parts, because the interoperability interface would need to dictate the interaction between the parts.

Application-Independent Location Data: Today, most location-based systems are really designed to support a single application. Within a large enterprise, for example, there may be many copies of the building floorplans and campus maps maintained by different branches of the organization for different purposes. For location-based services to become economically viable, it would be desirable to have a single location data store that can serve many applications. This is relatively easy to achieve within an enterprise, in which all data accessors could be expected to use the same client interface software. The structure of the location engine is irrelevant, since all applications use the same software components and can therefore interoperate freely. But for public venues, such as a shopping mall presenting a mall directory to the general public, the expectation must be that different clients may be based on different software systems. This is one reason that publicly available location-based services today generally place the location engine entirely in the server, with a very simple and universal query interface such as a web browser displaying a picture or linking to a specific URL to provide information for the specified location, e.g. the Web-based maps at `http://virtualearth.msn.com`. Unfortunately, the application is therefore designed into the location data store itself, so there is in effect only a single application supported by the data store - while client-system-independence is achieved, application-independence is sacrificed. True application-independence requires a careful design to expose a rich, flexible view of the location data, while not imposing an undue burden of computational intensity on the client.

7 Conclusions

Location-based services are critically dependent on data stores. Today's LBSs are primarily deep but narrow or broad but shallow. As these services grow in breadth and depth, the complex issues of storing and accessing a sometimes messy landscape of location data will present database developers and researchers with interesting challenges.

References

[BMK+00] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. Easyliving: Technologies for intelligent environments. In *Second International Symposium on Handheld and Ubiquitous Computing (HUC)*, pages 12–29, 2000.

[BS01] Barry Brumitt and Steven A. Shafer. Better living through geometry. *Personal and Ubiquitous Computing*, 5(1):42–45, 2001.

[DdFG01] A. Doucet, N. de Freitas, and N. Gordon. An introduction to sequential monte carlo methods. *Sequential Monte Carlo Methods in Practice*, pages 4–11, 2001.

[DK05] Matt Duckham and Lars Kulik. A formal model of obfuscation and negotiation for location privacy. In *Pervasive*, pages 152–170, 2005.

[HKH05] Ramaswamy Hariharan, John Krumm, and Eric Horvitz. Web-enhanced gps. In *LoCA*, pages 95–104, 2005.

[HT04] Ramaswamy Hariharan and Kentaro Toyama. Project lachesis: Parsing and modeling location histories. In *GIScience*, pages 106–124, 2004.

[KH04a] John Krumm and Ken Hinckley. The nearme wireless proximity server. In *Ubicomp*, pages 283–300, 2004.

[KH04b] John Krumm and Eric Horvitz. LOCADIO: Inferring motion and location from wi-fi signal strengths. In *MobiQuitous*, pages 4–13, 2004.

[KHM+00] John Krumm, Steve Harris, Brian Meyers, Barry Brumitt, Michael Hale, and Steve Shafer. Multi-camera multi-person tracking for easyliving. In *Proceedings of the Third IEEE International Workshop on Visual Surveillance (VS'2000)*, 2000.

[PLG+04] Donald J. Patterson, Lin Liao, Krzysztof Gajos, Michael Collier, Nik Livic, Katherine Olson, Shiaokai Wang, Dieter Fox, and Henry A. Kautz. Opportunity knocks: A system to provide cognitive assistance with transportation services. In *Ubicomp*, pages 433–450, 2004.