# Aalborg University

**Department of Computer Science**


**Database and Programming Technologies**

**Title:**
Online Aalborg Guide - Development of a Location-Based Service

**Project period:**
1. February 2003 - 7. July 2003

**Project group:**
DAT6 - E3-215A


**Group members:**
Kristian V. B. Andersen
Michael Cheng
Rasmus Klitgaard-Nielsen

**Supervisor:**
Chistian S. Jensen


**Total number of pages:** 102

**Number of reports printed:** 7


**Abstract:**

This report documents the development of a push-based LBS framework. In a prior semester a LBS framework that supports pull-based LBSs was developed. This framework has been extended to support the development of push-based LBSs.
Using the framework a prototype LBS, Online Aalborg Guide, is developed. In order to access the service an application, GPSOne, has been developed for Nokia 7650.
A design criterion is to minimize the client's number of position updates to the server. This is achieved by letting the client download data based on the location and preferences of the user. This way the client can operate on data relevant to the user.
Another design issue that needs to be handled is the limited CPU power and storage capacity on the client. Since only a part of the data is stored at the client, this frees space, which can be used for other purposes, such as storing map images. The client application caches map images, so maps does not have to be fetched from the Internet every time. This reduces the network usage and increases the performance of the client application.
A model for storing time-related information is developed and used to deliver advertisements based on time and location.


Kristian V. B. Andersen

Michael Cheng

Rasmus Klitgaard-Nielsen

# Preface

This report documents group d601a's work on the Dat6 semester in the period from the 1st of February 2003 to the 7th of July 2003 at the Unit for Database and Programming Technologies at the Department of Computer Science at Aalborg University.

Knowledge of databases and Object Oriented Analysis and Design are recommended for understanding the report.

Bibliographic references are displayed in brackets, e.g [Por03]. These are referred to in the bibliography at the end of the report.

We would like to thank the following participants for contributing to the project:

Nykredit and Aalborg Tourist Bureau for providing data for the project. The assistant programmer Jevgenij Gagach for answering questions and help. Our supervisor Christian S. Jensen for guidance.

# Contents

# Chapter 1

# Introduction

Pervasive computing is expected to be the next generation computing environment where information and communication technology is present everywhere at all times. Information and communication technology will be integrated into every day life and everyday products such as dishwashers, cars, electric circuits of houses, wrist watches etc. Pervasive computing is expected to be the third IT wave, the first wave being the computer (mainframes and PC's) and the second wave being the Internet, mobility and wireless communication. Pervasive computing is expected a development time frame of 5-10 years before affecting peoples lives [Min03].

One type of application within pervasive computing is Location-Based Services (LBS). LBSs are a powerful way to deliver highly personalized services. LBSs are mobile Internet services that utilize Location-Determining Technologies (LDTs) such as GPS technology to obtain the user's position. The position of the user is fundamental and crucial for delivering these highly personalized services. The user's position is obtained from a mobile device with a built in LDT that the user carries around. The position is transferred wirelessly to a service provider and the user receives location-based information on the mobile device. A mobile device could be a Personal Digital Assistant (PDA), a laptop or a mobile phone.

## 1.1 LBS Technology

Advancements in wireless communication, mobile devices and LDTs in recent years enable the opportunity to develop LBSs. Next, the core technologies available today for developing LBSs, are presented.

### 1.1.1 Location-Determining Technologies

LDTs are essential for enabling LBSs for users. One type of LDT is cell based positioning. Cell based positioning is a technology used in GSM networks, where the

position of a mobile phone is found using the known location of the base station that the phone is connected to. This technology determines the location of a person within several hundred meters. A more precise LDT is the Global Positioning System (GPS), which uses a collection of 26 satellites to pinpoint a position. GPS provides an accuracy of 5-10 meters [GPS03]. These are the most common LDTs used to develop LBSs. More sophisticated LDTs, such as Server Assisted GPS, provides better accuracy than GPS technology [DR02].

### 1.1.2 Communication Technology

LBSs require communication technologies to handle the communication between the user's mobile device and the service provider. Today the most common used communication standard for mobile devices is GSM (Global System for Mobile communication). GSM is a Second Generation (2G) mobile telecommunication technology and provides data transmission rates of 9.6 Kbps. This transmission rate is adequate for transferring text such as SMS (Short Message Service) or small images.

However, almost all new mobile devices support GPRS (General Packet Radio Service). GPRS has a theoretical transmission rate up to 171 Kbps, while practical transmission rates lie between 40-50 Kbps [Wik03a]. Unlike GSM, GPRS users can stay online permanently, since fees are taxed based on the amount of data sent or received and not by transmission time. GPRS is often referred to as 2.5G mobile telecommunication. GPRS enables the possibility of incorporating images, animations and small video clips into mobile services.

The Third Generation (3G) mobile network is already being deployed. 3G is equivalent with UMTS (Universal Mobile Telecommunications System). UMTS promises theoretically transmission rates of up to 2 Mbit/s [Wik03b]. Actual rates are estimated to be lower. Services such as multimedia presentations, video clips and video conferences become possible in 3G mobile networks.

### 1.1.3 Mobile Devices

The mobile devices must support mobile communication technologies and LDTs in order to take advantage of LBSs. A large variety of such mobile devices are available on the market. The devices can be grouped into the following three categories: *mobile phones, PDAs* and *smartphones*. These devices support mobile communication technology and LDT either embedded or via expansion modules.

- ***Mobile Phones*** Many new mobile phones on the market support GPRS and Bluetooth. With Bluetooth it is possible to connect wirelessly to a Bluetooth

GPS receiver. However, mobile phones have limited computing power, and the multimedia facilities are often limited to displaying low-resolution images.

- ***PDAs*** The PDA devices have more computing power than mobile phones. This enriches the devices with multimedia possibilities such as digital imaging, animations and video clips. However, most PDAs do not have mobile communication technologies embedded. Extension cards usually provide the support for these technologies. PDAs often have larger displays than mobile phones, which make them suitable for multimedia purposes. However, PDAs are not practical for use as mobile phones, since the size of the display results in relatively large devices compared to mobile phones.

- ***Smartphones*** A smartphone is a fusion of a mobile phone and a PDA. A smartphone has many of the multimedia facilities of the PDA, and at the same time it has embedded mobile communication technologies such as GPRS and wireless technologies such as Bluetooth. A smartphone has more computing power than a mobile phone and similar to that of a PDA. The size varies from pocket size to PDA size.

## 1.2 LBS Categorization

Depending on what kind of service a company wishes to provide, certain accuracy requirements need to be met. The Location Interoperability Forum (LIF) has categorized LBSs into 3 categories: Basic Service level, Enhanced Service level and Extended Service level [Nok03c]. Table 1.1 shows the different service levels and their associated positioning technologies. For instance, LBSs at the Basic Services level imply the us-

| Categories | Terminal Support | Network Support |
|---|---|---|
| **Basic Service level** | Legacy Terminals | Based on cell or improved accuracy (accuracy kilometers) |
| **Enhanced Service level** | Location of all new terminals | Improved accuracy (accuracy tens of meters) |
| **Extended Service level** | Location of new terminals | High Accuracy (accuracy meters) |

Table 1.1: The categorization of LBSs defined by LIF

age of cell based technologies. The accuracy is fairly poor (kilometers of accuracy) but already established terminals are available to support this level of services. Services of the Enhanced Service level are services that require higher accuracy (tens of

meters accuracy). However, in order to provide such accuracy, more advanced positioning technologies are required and new terminals must be developed to support such services. Figure 1.1 is an illustration of the evolution of LBS by LIF's categorization. This evolution scheme is introduced by Nokia. As shown in the figure pull-based services are categorized at the Basic level while push-based services and monitoring services are at the Extended level. According to Nokia's estimation the market maturity for LBSs at the Extended level will be reached from 2003.

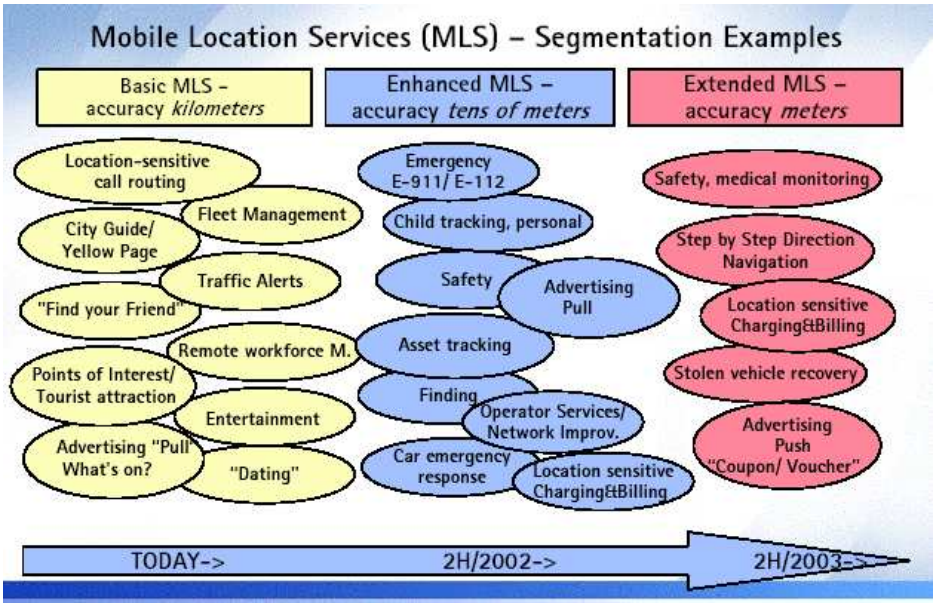


Figure 1.1: Illustration of Nokia's estimation of the development of LBSs [Nok03c]

Devices with high accuracy LDT have not yet reached maturity in the market. In order to develop LBSs at the Extended level various mobile devices have to be combined with LDT technologies such as GPS modules. At the moment many handhelds and mobile devices have interfaces to extend the device with e.g. GPS receivers.

## 1.3 Push and Pull Services

The *push* and *pull* concepts are used and defined in various disciplines. The concepts are especially used in the area of economics, production and marketing. In general push and pull are often used to interpret the relationship between the consumer, the actor that consumes or demands the information, and the provider, the actor that provides the information. A definition of pull is: *Users pull information to them wherever and whenever needed* [Jan01]. A definition of push is *Location service utilizes users*

*location to make user a customer or service recipient* [Jan01].

In this project the push and pull concepts are interpreted differently from the prior definition. The variation lies in the definition of push. Push is defined as *Information delivered to the consumer where the consumer has no control of when information is delivered.* In push-based LBSs the user can subscribe to a certain service. The service could be information delivery to the user that happens if certain criteria are satisfied. When the criteria are satisfied, the information is sent to the user at a time not controlled by the user. This kind of service involves pull elements (the user subscribes to the service) and push elements (the information is delivered to the user). However, according to the definition of push in the project, this service is denoted as a push-based service, since the user has no control of when the criteria are satisfied, and when the information is delivered. The division of LBS into pull and push services reduces the complexity of understanding the interaction and nature of LBSs.

## 1.4 Push-Based Service Development

In [ACKN03] a framework for LBSs is developed. The LBS framework, illustrated in Figure 1.2, is based on a component-oriented structure that ensures modularity and the possibility of modular expansion. Additionally, the components in the structure are ordered in layers. These layers ensure transparency in the system, which means, that a developer does not need to know all the logics and processes in the entire framework. The developer only needs to know the interfaces of a particular layer. For instance a server developer does not need to know about GPS technology at the client side, but only the interfaces in the client layer. The LBS framework is an approach to developing a framework that is capable of creating and adapting various LBSs within one framework using a layered and component oriented structure.

The LBS framework has been implemented and utilized to develop pull-based LBSs. The features of the pull-based LBSs consist of retrieving step-by-step directions, with or without a map, to a certain Point of Interest (PoI) and retrieving the nearest PoIs to the user. However, no LBS components in the LBS framework support push-based services. From this point of view, the main purpose of the project is to continue the work on the LBS framework by extending the LBS framework to support push-based services. Due to the modularity of the LBS framework no major modifications are required to implement push-based features. A set of push components needs to be developed. The push-based components that are developed in this project support the prototypic example of a location-based Online Aalborg Guide. The Online Aalborg Guide is an online city guide where users can get information about restaurants, events and popular sites in Aalborg. The Online Aalborg Guide is the entire system, consisting of a server and a client application, GPSOne. The client application runs on the client, which is a mobile device with an Internet connection. The information in the

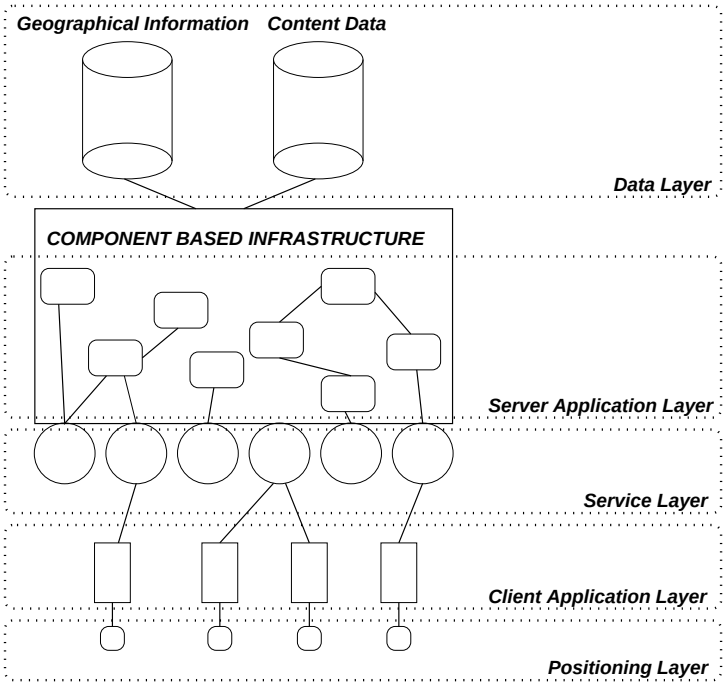


Figure 1.2: Illustration of the developed LBS framework of [ACKN03].

system is based on the users' location and preferences. The user is "pushed" information and advertisements about the nearest PoI, which matches the users preferences. In addition the user is able to "pull" and browse through further information about PoIs. Furthermore, a map is continuously displayed, indicating where the user and the PoI is located. The content and specification and design issues of the Online Aalborg Guide is described in Chapter 2. Next, related work to the push-based LBS framework and the Online Aalborg Guide is described.

## 1.5 Related Work

Some companies have developed push-based services for mobile use, but the number of push-based services is not as large as the number of pull-based services.

A push-based LBS developed by Met Office [Met03] is a weather forecast service. The user enters a city name or zip code, a time and a date, and sends the information using a SMS or a WAP page. A weather forecast for the region is then delivered at the time specified by the user.

A range of products with pull-based LBSs are on the market. Some of the most popular products are digital city guides for handhelds. The digital city guides *Vindigo*[Vin02]

and *Citysync*[Lon03] provide updated reviews, PoIs, step by step directions and simple color maps to PalmOS and PocketPC based handheld devices. The user needs to input the current location in order to use the LBS. If a GPS device is attached to the handheld device the location is received from the GPS. The reviews and PoIs in the applications can be updated via an Internet connection.

A more advanced digital city guide is *Trekker*[Vis03]. *Trekker* is an application for PocketPC devices, adapted to blind people. It offers features that enable blind persons to determine their position, create routes and receive information on navigating to a destination. Besides that, blind people are able to find PoIs. Compared to *Vindigo* and *Citysync*, *Trekker* has an advanced speech/voice interface for enabling blind people to use the application. *Vindigo*, *Citysync* and *Trekker* are handheld city guides that provide location-based features to consumers. However, all these systems are limited to pull-based services and due to the limited storage capacity, the digital city guides only cover a limited number of cities. In addition, an update of the system when entering a new city requires large amounts of data to be transferred, since all information is stored on the handheld device.

Contrary to pull-based LBSs, the number of push-based LBSs is limited in the market. The push-based LBSs that exist in the market today are fleet management and warning services. The company *IntelliWhere* offers a range of fleet management solutions for the business-to-business industry. One of the their solutions is *IntelliWhere TrackForce* [Int03]. With this solution fleet oriented companies are able to track and communicate with their units in the field. The TrackForce system allows the users to have alerts that are triggered by certain conditions, i.e. traffic jams. The alerts can be given to the people working in the field, thus enabling them to avoid delays.

The mobile phone manufacturer Nokia has recently introduced a comprehensive middleware framework *mposition* for developing LBSs. The middle-ware framework provides comprehensive APIs for managing information in the cellular network and gateway layer. Figure 1.1 illustrates *mposition*'s End-to-End architecture. The framework is very similar to the framework proposed in [ACKN03]. The main difference between the two frameworks is the variety of LDTs. While the LBS framework only supports GPS technology, *mposition* supports four different cell based LDTs and A-GPS technology.

The rest of the report is organized as follows: Chapter 2 describes the features of the Online Aalborg Guide and related design issues. The chosen architecture and design approaches are described in Chapter 3. The chapters after that describe the layers of the developed push-based LBS framework. Chapter 4 describes the Data Layer and contains a description of the data model. The Server Application Layer containing specification of components and services at the server, is described in Chapter 5. The interface between the Service Layer and the Client Layer is described in Chapter 6.

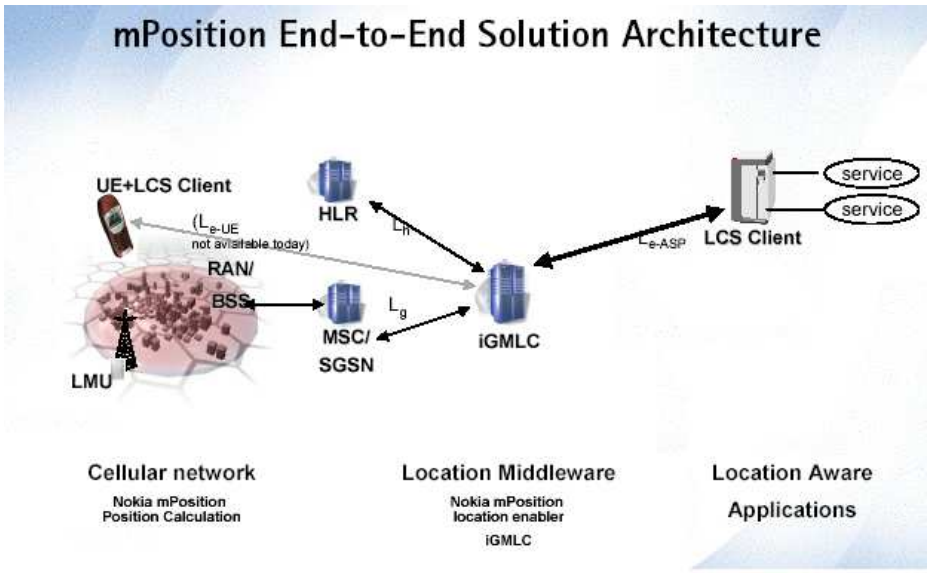


Figure 1.3: Illustration of Nokia's mposition architecture

Chapter 7 describes how a client application is implemented at the Client Application Layer. A use-case scenario for the GPSOne application is introduced in Chapter 8. An evaluation of the push-based LBS framework and GPSOne application is presented is Chapter 9. Conclusion of the report is given in Chapter 10. Chapter 11 discusses future work for the system.

# Chapter 2

# Online Aalborg Guide

As mentioned in the introduction the push-based LBS framework is used to implement a Online Aalborg Guide LBS. Section 2.1, describes design issues and principles for developing location-based city guides. Section 2.2 describes how a complete and comprehensive LBS tourist guide is imagined. A specification of features and sub services of the prototype "Online Aalborg Guide" is described in Section 2.3

## 2.1 Design Issues

The Online Aalborg Guide concept proposed is based on [Mar03]. Creating applications for mobile devices imply different design principles than traditional applications for stationary desktop computers. A LBS is inherently intended to be used in mobile situations and must take advantage of this fact, that the application for the mobile device must be able to adapt to different and dynamic contexts. In the following the results from [Mar03] are compared to the Online Aalborg Guide features.

The main target group for the Online Aalborg Guide is tourists visiting Aalborg. A way to develop an application targeting this group is to perform studies of such groups in order to understand their needs and requirements for such a system. Research from the GUIDE project [Mar03] is based on semi-structured interviews with members of the Tourist Information Center of Lancaster and observations of tourist visiting the tourist office over a period of several days. This research shows that tourists were very interested in having access to cultural, historical and architectural information when visiting Lancaster. Online Aalborg Guide users can retrieve information about all PoIs and events stored in a database maintained by the Danish Tourist Board.

Furthermore, the tourists expressed interest for dynamic and flexible information and support for interactive services. The tourists were, for example, pleased with the possibility of receiving dynamic information, such as the "specials" of a cafe. The Online Aalborg Guide handles dynamic time-related location-based information in form of

advertisements, which can change over time, from nearby PoIs.

Also visitors to Lancaster were more inclined to trust a system provided by a reliable source such as the local tourist information board. The data used for the Online Aalborg Guide is obtained from the Danish Tourist Board.

The Lancaster visitors would also like the possibility of booking services through their guide. However, some visitors wanted to speak to someone to be sure that reservations had been confirmed. Such services would require integration of booking systems, payment and security systems. Such services are not included in the Online Aalborg Guide. Users of Online Aalborg Guide can request address information and booking phone number and use this information to contact the PoI for reservations.

According to [Mar03], mobile users do not want to waste time interacting with mobile devices, however they do not mind navigating through options that go deeper into areas that they want to know more about. Other studies show that the most suitable systems for mobile users incorporate user interfaces, that require minimal attention. This allows for the user to perform activities in mobile situations without much attention on the device interaction.

The Online Aalborg Guide uses a mix of push and pull technologies in order to comply with this behavior. The nearest PoI is continuously updated and displayed on the screen. This way the user does not have to interact with the device in order to see which PoI is nearest. However, if the user wants further information about the PoI, the user must make a request and the information is displayed in a window where the user can browse through it.

The report [Mar03] emphasizes the importance of understanding how users behave differently in different situations and under different circumstances. For this purpose the report proposes a user profile. In order to use the Online Aalborg Guide each user must create a profile in which the user has the option of adding and removing categories of PoIs. The Online Aalborg Guide then utilizes this profile information and delivers PoI-information within the specified categories.

Another wish that many travelers expressed according to [Mar03] was the possibility to plan a trip before the trip. This could be in the form of deciding what to see, getting information and locations of PoIs, find out how to get to the PoIs, book services and plan a tour. Other issues not directly expressed in studies of tourists' behavior are tasks, which could be performed after a trip. A feature to support registration of travel memories is proposed in [Mar03]. Furthermore, features for sending electronic postcards and the ability to rate and comment PoIs are proposed. Online Aalborg Guide users have the option of planning ahead to the extent that it is possible to access a Web interface and edit the users profile. The user has the possibility to add the categories

of PoI that the user is interested in, before a trip.

Based on the mentioned design principles and issues the next section describes the features of the Online Aalborg Guide.

## 2.2 Features

These are the features that form the basis of the Online Aalborg Guide:

- Nearest PoI Information: Is the main feature of the Online Aalborg Guide, where information is displayed to the user, according to the user's position and time.
- Further Information: Users can retrieve additional information about the nearest PoI such as address, phone number and description.
- Favorites: Allows the user to save PoIs for later use.
- User PoIs: Gives the user an opportunity to add user-generated content to the service, in the form of new PoIs.
- Reviews: Allows the user to add content to already existing PoIs, by writing reviews of individual PoIs.
- Pictures: Is an extension to Reviews and User PoIs, where the user can submit photos of PoIs.
- Push Advertisement: Allows data providers to send out advertisements based on the users' interests and positions.
- Route planner: Allows the user to find the shortest route to a certain PoI.
- Buddy Finder: Extends the push service to retrieve information about other users.
- Profile Handling: Allows the user to edit profiles before and during a trip.
- Record Note of PoI: Allows the user to record a travel note of a PoI visited.
- Map Service: A map of the surroundings, is at all times displayed for the user, with an indication of the user's position and nearest PoI position.

In the following the features are described in more detail.

## Nearest PoI Information

If a PoI matches the criteria specified in the profile, the nearest PoI is displayed for the user. The system allows for the user to set the distance within which PoIs should be located, in order to be displayed. The information is continuously updated (pushed) so that the user doesn't have to interact with the device in order to receive the nearest PoI. The information displayed is the name of the PoI and the distance from the user to the PoI.

## Further Information

The user can retrieve further information about the nearest PoI. The message retrieved contains information about the name of the PoI, address, phone number and a description of the PoI. The user can navigate through the information and find the needed information.

## Favorites

When the user gets information pushed during the day, the number of nearby PoIs may be large on some occasions, making it impossible for a user to deal with all of them. If the service has an option to save PoIs for later use, the user can use the Favorites feature in the same way as a bookmark function in a Web browser. The user can save interesting PoIs for later use, if the user is busy with other tasks or just doesn't need the information provided at the present time.

## User PoIs

This feature allows a user to submit new PoIs to the service. The user needs to enter the name and a description of the PoI, and is then able to submit the data to the service. This makes the content of the system more flexible and meaningful for the user, since the user partakes in adding content to the service. Some samples of user content could be:

**Sønder Tranders Church** This is a nice little church, which is located on a hill.

**My Spot** Nice open spot with beautiful oak trees.

The PoIs that a user enters will not necessarily be meant for the public. Some PoIs are only important to the user, for instance the user's home or the place where the user works.

## Reviews and Pictures

Another type of user content are reviews. The reviews, which are submitted by users, give other users a better way of telling if a PoI is to their liking. The chances are that

if other users for instance find that a hotel is sub standard, then the hotel is in fact of a low standard, despite what the tourist brochure shows. Letting users submit photos of PoIs, so that in addition to reviews, other users can actually see the PoI before going there, may enhance the reviews.

### Push Advertisement

Some PoIs could have special offers that the data provider wants to advertise. Especially commercial PoIs, such as shops or restaurants, will find a certain value in paying the service provider to allow them to advertise through the service. This push service allows the advertisements to be location and time based, so that offers is made available only at certain time intervals, and at certain places.

### Route Planner

This feature provides the user the ability to get route guiding to a certain PoI. The route guiding could be materialized in textual form or in pictures. For instance, a map that shows how the user reaches a destination. This feature is pull-based since it is the user who requests the information.

### Buddy Finder

The Buddy Finder feature allows users to find other users using the service. When a user is within a specific range of another user, the user is notified. As a part of the Online Aalborg Guide, the Buddy Finder allows traveling companions to find each other in foreign places. For instance, the Buddy Finder allows parents to let their children wander around more freely, as the service allows both to find each other.

The Buddy Finder service can be expanded to cover many different applications, where people need to find other people.

### Profile Handling

The profile handler is a key feature. It gives the user the possibly to personalize the application to the user's wishes.

The profile is intended to consist of the following options and features:

- Create new user.
- Create and delete profiles. Each user can have several profiles, for instance a work profile, shopping profile or a sightseeing profile.

- Set the maximum number of advertisement that the user wishes to be pushed at a time, so that the user is not overburdened with advertisements.
- The user is able to add and remove categories of PoIs from a profile. For instance, this way a user can personalize the application to only show information about beaches and restaurants.
- The user can, for each category of PoI, specify the range within a PoI of this category should be located within, in order for advertisements to be pushed. For instance, this allows users to specify that advertisements from restaurants are allowed within 500 m while advertisements from amusement park are allowed within 10 km.

The profile handling is a solution, which allows the user to edit profiles before a trip and also during a trip.

### Record Note of PoI

Many tourists would like to write notes of places they have visited. With this feature a user can create a note, which is attached to a certain PoI and write a description of this place. Pictures taken near the PoI with a digital camera, possibly built in, can also be attached. After the trip is completed the tourist can extract these notes and pictures from the device and create a collection of travel memories. The user has the option of sending the travel notes to friends and family.

### Map Service

This feature provides a map of the surrounding streets and gives the user assistance in finding a way to a PoI. The location of the user is indicated in the center of the map and the map scrolls as the user moves. The location of the nearest PoI, if one is present, is also indicated on the map, along with an arrow indicating in which direction the PoI is in relation to the user. This is particularly useful if the PoI is located outside of the visible map. This way the user can follow the direction of the arrow and know that the PoI is getting closer.

## 2.3 The Prototype

To implement all the mentioned features is beyond the scope of this project and hence a prototype Online Aalborg Guide system and a prototype application “GPSOne” for Symbian OS operated phones has been developed. The prototype system includes the following features:

- Nearest PoI Information

- Further Information
- Push Advertisement
- Web Based Profile Handling
- Map Service

Some of the introduced features are very similar to features provided by traditional digital city guides. What makes the Online Aalborg Guide different in relation to other city guides is the push-based features that are especially emphasized in the **Nearest PoI Information**, **Buddy Finder** and **Push Advertisement** features. Hence in order to utilize the push-based LBS framework, the prototype "GPSOne" application is able to display nearest PoI based on a user profile and receive time-related location-based advertisements from PoIs.

The application also provides users with a **Further Information** feature. The feature retrieves further information about the nearest PoI. This is an important feature for the users, since this information has great value for tourists and other users. For instance, this information could be phone number booking and the address of the PoI. The profile concept, **Profile Handling**, has also been developed since this is a key concept in order to provide a highly personalized service. The profile handling is a Web-based solution which enables the user to edit profiles before a trip and at the same time edit profiles on the mobile device since most new mobile devices has a built in Web browser. The **Map Service** has been implemented to provide users with assistance in finding a way to a PoI and in order to provide users awareness of their surroundings.

## 2.4 LBS Issues

In order to develop the Online Aalborg Guide some technical and practical issues need to be addressed. Some issues are fundamental for developing push-based services and some are specific for the Online Online Aalborg Guide. In the following sections, some of the issues and side effects of developing push-based services and the Online Aalborg Guide are introduced. Other issues may be relevant for the project. However, the issues described here are the main focus issues in this project.

### 2.4.1 Continuous Update

As mentioned earlier, it is fundamental to know the user's position in order to deliver LBSs to the user. By knowing the user's location it is possible to find the nearest PoIs within a predefined range. This is achieved by comparing the user's position with the position of the PoIs in the database.

Depending on the size of the database and the design approach, the database is placed either at a central server or at the client terminal. Assume that the database resides at a central server. This means the central server is responsible for performing the logic of the LBSs. Hence, the central server has to be informed about the user's location continuously in order to provide accurate information to the user. To inform the server, the client has to perform continuous updates to a central server. This task can cause performance decrease for the whole system if the number of users is significant. This can also result in an overload of the transmission bandwidth.

### 2.4.2 Storage Issue

Before a practical LBS implementation is possible, some technical requirements have to be met. Some of the crucial requirements concern processing capabilities and storage capacity of the mobile devices. This rises some issues. For instance, how is it possible to optimize the usage of the limited storage capacity and what data should be stored?

LBSs need geo-referenced data such as PoIs and graphical maps. The PoI data may be the most dynamic content since information such as address, phone number and other PoI information changes relatively often. Maps, on the hand, change very seldom. This means that PoI data stored on the client needs to be validated more often than map data. In general graphical map data takes up more space than PoI data. Hence, there is a limit to the number of maps that can be stored on the mobile device. A solution to handle the storage issue is needed.

### 2.4.3 Temporal Information Delivery

When delivering location-based information to the user in the Aalborg Guide, it is preferable to provide the user with the opportunity to receive information according to time and location. For instance, if the user's nearest PoI is a cafe that is closed at the user's time of arrival, the user may prefer to get information about the next nearest PoI instead. This type of service is denoted as *time-related location-based service* in this report. From the information provider's point of view it is a preferable option to be able to send information at different times. For instance, a cafe may have different offers during a day. From 11 AM to 3 PM the cafe may have a lunch offer, and from 6 PM to 10 PM the cafe may have a musical event. The feature of sending different information at different times can be used in the information provider's marketing strategy. In order to provide such a feature in the Online Aalborg Guide a time scheduling system needs to be developed for the purpose.

In order to extend the LBS framework with push features to support the Online Aalborg Guide, the mentioned issues have to be managed. In the next section the design approaches and solutions to develop the Online Aalborg Guide is introduced.

# Chapter 3

# Architecture and Solutions

This chapter describes the design approaches and the design solutions to achieve the goal of developing push-based LBSs. In the following, various design approaches are introduced.

## 3.1 Design Choices

Devices with high accuracy LDTs are crucial for developing push-based LBSs. However, technical constraints such as CPU power and storage capacity at the client terminals have great impact in the consideration of architecture design. In the following, design approaches to developing push-based LBSs are described. Figure 3.1 illustrates the various approaches.

Using the *Client Approach* the whole solution resides at the client platform. LBSs often contain geo-referenced data and a set of function, which operates on the data. This design approach demands large storage capacity and powerful CPUs at the client terminal. At the moment, the only candidate client terminals that fulfill the requirements are laptop PCs and embedded car navigation systems. If the solution has to be implemented in handheld devices and smartphones, only a limited set of data can be stored due to the limited storage capacity in such devices. This rises several problems in relation to update policies and data validity. Another crucial disadvantage in the *Client approach* is the lack of fleet management capabilities, where mobile units are administered from a central place. Since each client has all the required data and functions installed, there is no need for a central unit to administer the clients, and hence larger distributed monitoring and warning applications are not possible.

In contrast to the *Client Approach*, the *Server Approach* separates the database and the LBS functions from the client platform. The database and the LBS functions reside at a central server. This forms a client/server architecture where the task of the client device is to request the needed service and data from the server and display it to the

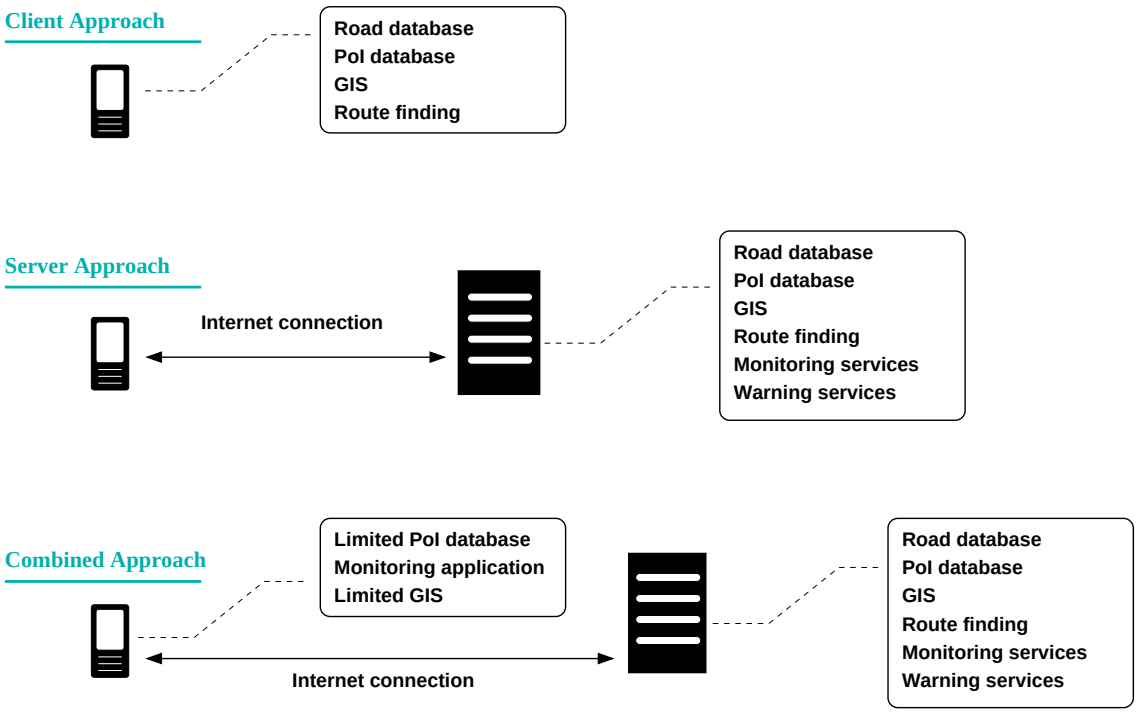


Figure 3.1: Illustration of the three design approaches. The mobile phone at the left side represents the client and the server is represented at by the larger black box.

user. The LBS framework mentioned in Section 1.4 is an implementation of this design approach.

The approach of storing the database in a central server avoids the data validity problem, since updating the data from a central server, it is assured that the clients will always obtain the newest data available. This ensures data validity and data consistency. Comparing the two approaches, the *Server Approach* can support more features than the *Client Approach*.

One of the advantages of the *Server Approach* is the possibility to provide monitoring and warning services. When the client uses the LBS, the client position is reported to the central server. This enables the possibility of developing more advanced monitoring and prediction services such as traffic warnings, location-sensitive advertisement and fleet management. In addition, it is possible to create client independent services, which means that different client terminals can access the same service. However, in order to provide these services, the server must know the location of the client. This requires the client to continuously update the server with its position. This can result in an overload of the server if a significant amount of users are logged onto the system. Continuous updates from the clients may also result in congestion of network traffic

to the server, hence delaying the overall traffic to the server for the user. In addition, the users have to pay for the network traffic, making the LBS less interesting for the user. An alternative solution to solve the bandwidth and server overload problems is to provide more memory and CPU power to the server and improve the existing bandwidth. However, the performance will still be worse than in a *Client Approach* due to the network transmission delay.

As seen in Table 3.1 the *Client Approach* lacks the capability to handle Client Independency and Data Validity. The *Server Approach* covers these features though the solution suffers from large numbers of updates and bandwidth overload, for large numbers of users of the system. The two concepts have different developing perspectives; centralized vs. decentralized design.

| | **Client** | **Server** | **Combination** |
|---|---|---|---|
| Client Independence | | X | |
| Data Validity | | X | (X) |
| Update minimization | X | | (X) |
| Monitoring options | | X | X |

Table 3.1: Comparison of the advantages of the three design solutions. An X indicates an advantage, a (X) indicates a partial advantage.

To claim the benefits from the two different perspectives, the *Combined Approach* is introduced. The architectural structure of this approach is based on the client/server design from the *Server Approach*. In order to minimize the continuous updates, a small subset of the database is replicated at the client side. This approach reduces the CPU power usage at the server side, since parts of the computations are carried out on the client device. As long as the data at the client device is not invalid, the client does not need to issue an update. This reduces the bandwidth usage. When less updates are performed, the validity of data suffers, since some data will become out of date. The *Combined Approach* has all the features of the former approaches. However, a tradeoff has to be taken into consideration since less updates imply that the server positions the client less accurate, which in turn means less precision in services. In this project, the *Combined Approach* has been chosen as the design principle for the Online Aalborg Guide. The next section describes the architectural design of the *Combined Approach*.

## 3.2 Architecture Design

Figure 3.2 illustrates the overall design of the *Combined Approach*. The LBS Application Servers use the LBS framework architecture. This provides the LBS Application Server with pull-based service features such as shortest route calculations and step by step direction descriptions. The LBS Application Server has a geo-referenced PoI

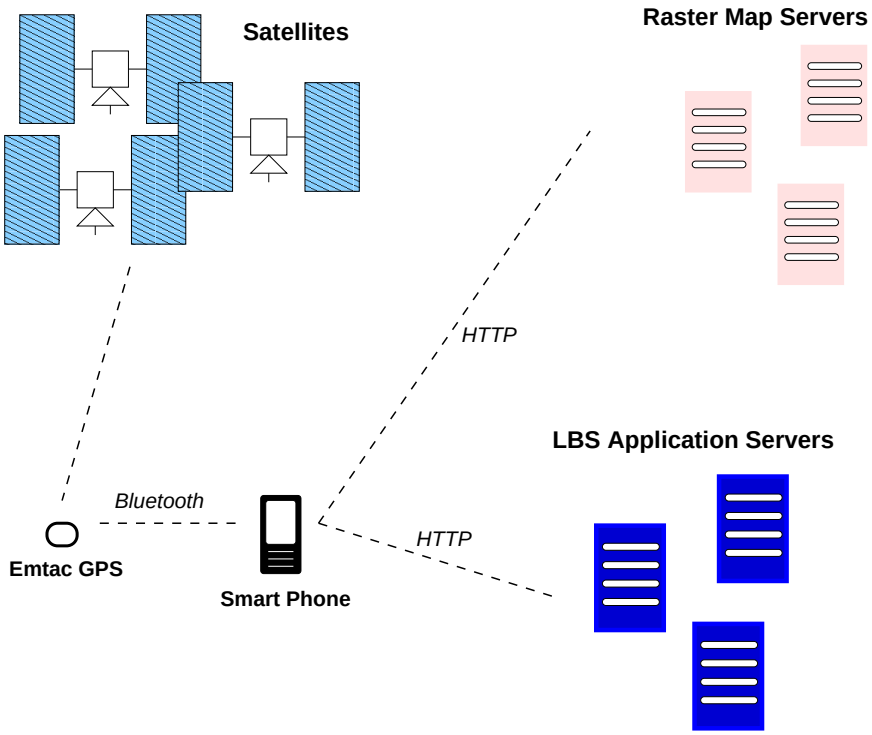


Figure 3.2: Overview of the design architecture

database and a road network database that supports the main roads in Denmark. The LBS Application Server does not support push-based LBSs. Hence additional components for enabling push-based features are developed. The raster maps to display the roads are supported by using an external raster map servers. In this case, the Kort & Matrikelstyrelsen (KMS) raster map server is used for the purpose.

As the client part in the architecture, a Symbian smartphone is selected as the target device for developing GPSOne. One of the reasons for selecting the Symbian smartphone instead of other available devices is the stability and the comprehensive communication support of the Symbian OS. Furthermore, the Symbian OS provides a wide range of multimedia and multitasking options. Among Symbian smartphones, the Nokia 7650 is selected. The Nokia 7650 uses the Symbian OS and is built upon Nokia's Series60 platform [Nok03b]. This platform has been licensed to various vendors whose combined market share is more than 55 percent [Por03]. This means that all Series60 phones can use the GPSOne application.

The Nokia 7650 does not support GPS technology. GPS support is crucial for developing push-based LBSs. In order to provide GPS support for the Nokia 7650, an Emtac Bluetooth GPS receiver is used. Since the Nokia 7650 and any Symbian smartphones

with Symbian OS version 6.1 has Bluetooth support, the phone can connect to the GPS receiver and extract positions from the Emtac GPS receiver.

GPSOne is developed upon the *Symbian LBS framework* [Gag03]. The *Symbian LBS framework* is a research project developed at Aalborg University and provides fundamental communication APIs for developing LBSs for Symbian smartphones. The *Symbian LBS framework* has been extended with a *LBS Engine* to create a client framework for developing the Online Aalborg Guide.

As seen in Figure 3.2, the Emtac GPS receiver provides coordinates to the Nokia 7650 continuously via a Bluetooth connection between the two devices. When the Nokia phone receives the coordinates, GPSOne starts providing the LBSs to the user. During execution of GPSOne, a service request is needed to the *LBS Application Server*. The Nokia phone connects to the server via the HTTP protocol. The *LBS Application Server* processes the client's request. When the server has accomplished the task for the client, the result is send XML format or a customized text format. If GPSOne needs to download raster maps, a connection to a raster map server is established and the map is retrieved in JPEG format.

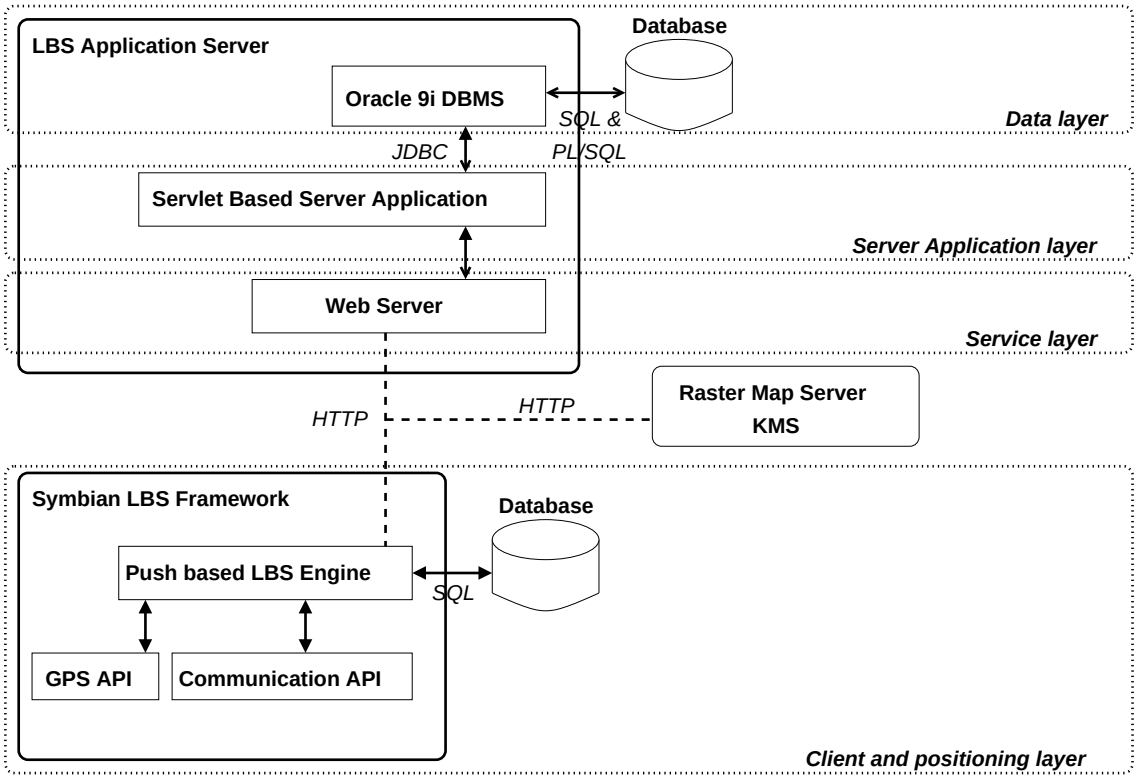


Figure 3.3: A layered illustration of the design

While Figure 3.2 provides an overall design of the *Combined Approach*, Figure 3.3 illustrates the technical components of the LBS Application Server and the Symbian LBS Framework. Furthermore, the figure illustrates the abstract layers of the components. The LBS Application Server consists of a Web server, a set of server applications, and a DBMS. The Web Server is in charge of handling requests to Java Servlets from client applications. The Web Server is categorized in the Service layer. The server applications, categorized in the Server Application layer, are the set of LBS functions that execute the user's request. The LBS functions are implemented in Java 2 Enterprise Edition. This enables the usage of Java Servlets to support LBSs. At the Data layer, Oracle9i DBMS is used to store the data. The features that Oracle9i provides in comparison to other available DBMSs are the spatial indexes and spatial functions. These features are fundamental for the LBS Application Server and for deploying LBSs. As mentioned, geographical raster maps are downloaded from external servers. Depending of which kinds of services the LBS Application Server provides, the raster maps are either directly downloaded to the client or processed by the LBS Application Server, before send to the client.

At the Client and Positioning layer the *Symbian LBS framework* supports fundamental communication. This involves a TCP/IP connection to the LBS Application Server and Bluetooth communication for GPS coordinates extraction. The communication support API is divided into a communication API and a GPS API, which the push-based LBS Engine can access. The push-based LBS Engine contains the essential functions to support the GPSOne application. This involves update routines, database management, image management, range and PoI monitoring. The Symbian OS provides a lightweight database that supports basic SQL. This is used for storing and managing the PoIs received from the server. The LBS Engine is the extension developed for the *Symbian LBS framework*.

In order to develop GPSOne some issues need to be addressed. The issues are *Continuous Update* and *Storage*. The design approaches that deal with these issues, are described in the next sections. The design approaches for handling the *Temporal information delivery* is described in Section 4.2.

## 3.3 Solutions to Update and Storage reduction

The *Combined Approach* stores a subset of PoIs from the LBS Application Server database in the client's database. This minimizes the continuous updates to the server from the client, since the client can use the local PoI subset to find the nearest PoI. The subset of PoIs is generated based on the user's profile, which has been defined during LBS subscription. The profile contains information about the user's desired PoIs, configuration and service settings. The subset of PoIs stored in a PoI index and sent to the client. In the following the approaches for generating the subset of data are

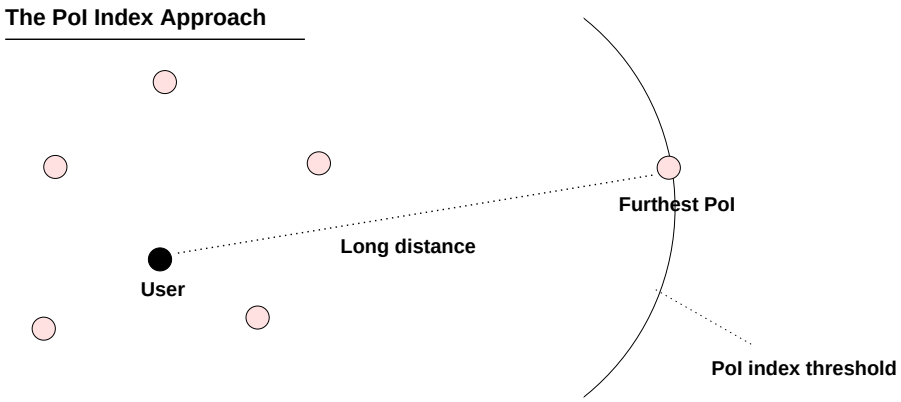


Figure 3.4: The figure illustrates the PoIs included in the PoI index. The problem arises when a PoI in the index is too far away from the user. This PoI may not be relevant in relation to the user's location.

presented.

### 3.3.1 Index generation

Storing a subset of PoIs at the client terminal reduces the number of updates to the server. However, if the user defines all the categories in the database or a very large category of PoIs, the size of the PoI index may exceed the client terminal's storage capacity. To prevent this case, a limitation of the PoI index must be set. The limitation setting depends on the client's storage capacity and the CPU power at the client terminal.

Assume that $N$ is defined as the maximum storage PoI capacity in the index. When the client requests a PoI index, the location of the user and the user name is sent to the LBS Application Server. With these information it is possible for the LBS Application Server to generate an index with $N$ nearest PoIs according to the user's position and profile. When the user receives the PoI index, an update algorithm at the client terminal monitors the user's location in relation to the PoIs in the PoI index. The update algorithm continuously finds the nearest PoI until a certain threshold has been exceeded. This threshold denoted as **PoI index threshold**, has been defined as the distance between the user's location and the location of the furthest PoI in the PoI index. When the PoI index threshold has been violated the present PoI index is no longer valid in relation to the user's current location. Hence, a request for a new PoI index is issued. This approach is denoted as the *PoI index approach* and is illustrated in Figure 3.4.

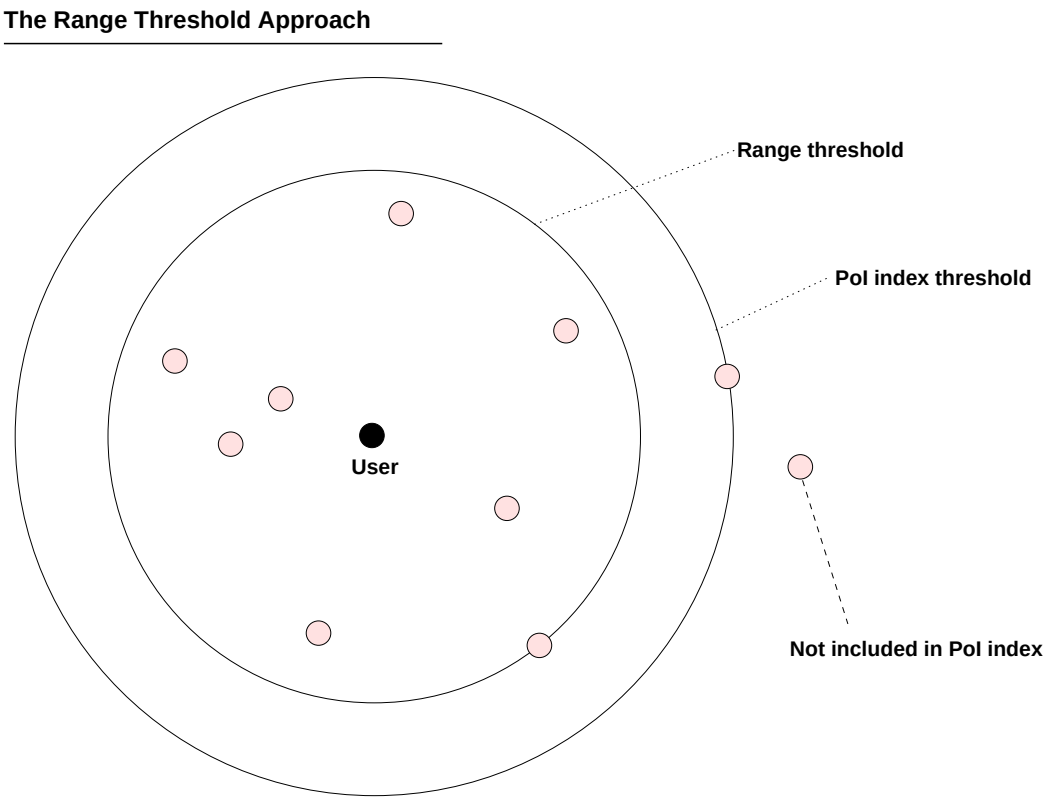


Figure 3.5: Illustration of the range threshold approach. The PoI outside the range threshold is not included in the PoI index.

The *PoI index approach* is suitable in very PoI concentrated areas but insufficient in sparse areas. The reason is the PoI index threshold definition. The PoI index threshold in *PoI index approach* depends on the furthest PoI in the PoI index which means that the PoI index threshold varies from time to time. The problem in sparse areas arises when the furthest PoI in the PoI index is too far away from the user's location. The result will be a very long PoI index threshold and less updates to the server, which means less accurate PoI index. The following scenario clarifies the sparse area problem: A user in London would like to find churches in the city. $N$ is defined to be 100 in the LBS Application Server. However, the are only 50 churches in London so additional churches in England have to be included in order to fill the PoI index. The result is the 100 nearest churches in England, and the PoI index threshold distance is larger than 500 miles since the furthest church in the PoI index lies in Manchester. The church in Manchester may not be in the user's interest due to the long distance. Another problem with the *PoI index approach* is, that it is not guaranteed that the nearest PoI in the PoI index is the actual nearest PoI in relation to the user's location. The nearest PoI in the PoI index could be many miles away, even though the actual nearest PoI is close by. This is illustrated in Figure 3.6.

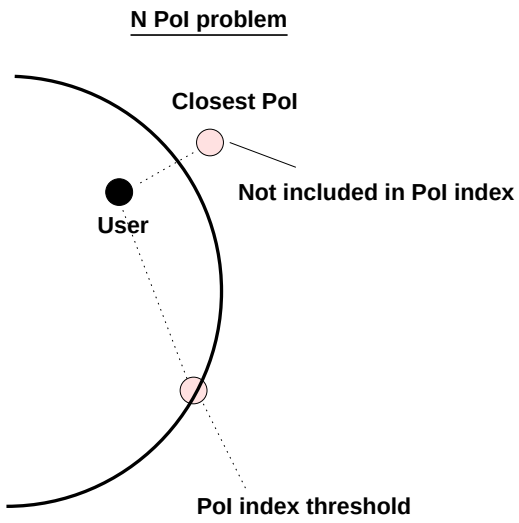


Figure 3.6: Illustration of the PoI index threshold approach. The nearest PoI is not included in the PoI index.

The *Range threshold approach* avoids generating PoI index with PoIs too far way, by defining a fixed threshold denoted as the **range threshold**. The range threshold assures that only the PoIs within in the range threshold are added to the PoI index. If there are more than *N* nearest PoIs within the range threshold, only *N* PoIs are added. If there is less than *N* PoIs within the range threshold the available PoIs will be added. This range threshold prevents the problem from the prior approach illustrated in Figure 3.5. However, the *Range threshold approach* used to generate the PoI index is inaccurate in finding the nearest PoI, if the user approaches the range threshold. The problem is illustrated in Figure 3.7. As seen in the figure the PoIs outside the range threshold are not included in the PoI index. If the user approaches the range threshold, the nearest PoI located nearby is not necessarily the nearest PoI. There is the possibility of a PoI being nearer than the selected one from the PoI index. The problem with the candidate nearest PoI is that the candidate nearest PoI may has not been included in the PoI index due to the range threshold.

To prevent this situation, an **update threshold** has been defined in the *Update threshold approach*, illustrated in Figure 3.8. In comparison with the earlier approaches where the PoI index/range threshold determines when a server request is issued, the update threshold is now in charge of the task. When the user passes the update threshold, an update is issued to the server. Since the PoIs on both side of the update threshold are known it is always possible to find the nearest PoI. By setting the update threshold to be the half of the range threshold it is guaranteed that it is possible to find the nearest PoI. A larger update threshold distance implies a greater risk of not finding the nearest

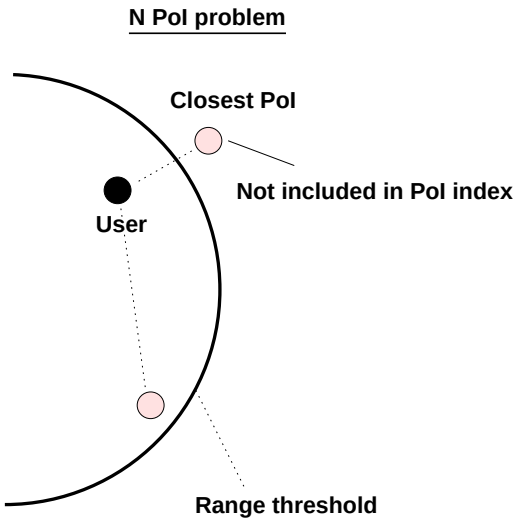


Figure 3.7: This illustrates the problem of finding the nearest PoI. Another PoI candidate may be nearer but not in the client's database.

PoI, and a smaller update threshold reduces the risk but implies more frequent server updates.

The PoI index generation procedure in the project utilizes the *Update threshold approach*. The range threshold limits the distance of the PoIs and the update threshold prevents the nearest PoI inaccuracy problem. The update threshold is defined to be one third of the range threshold or PoI index threshold, depending on which threshold that sets the upper boundary of the PoI index range.

### 3.3.2 PoI Index Update Algorithm

When the client receives the PoI index from the server, the PoIs in the index are stored in the client's database. The client is then ready to start the monitoring process. The monitoring process finds the nearest PoI based on the user's current location, check whether the nearest PoI is within the user's defined threshold. The user can define a threshold, which is a range that determines within which range the user wishes to receive information about PoIs. For instance if the user would only like to receive information of a nearby PoI if the PoI is within 500 m.

If the monitoring process is handled by the LBS Application Server, finding the nearest PoI can be solved by arranging the geo-referenced data in spatial indexes. Arranging geo-referenced data in spatial indexes provides fast access to the data. However, since Symbian OS only provides a thin DBMS, advanced features such as spatial indexes

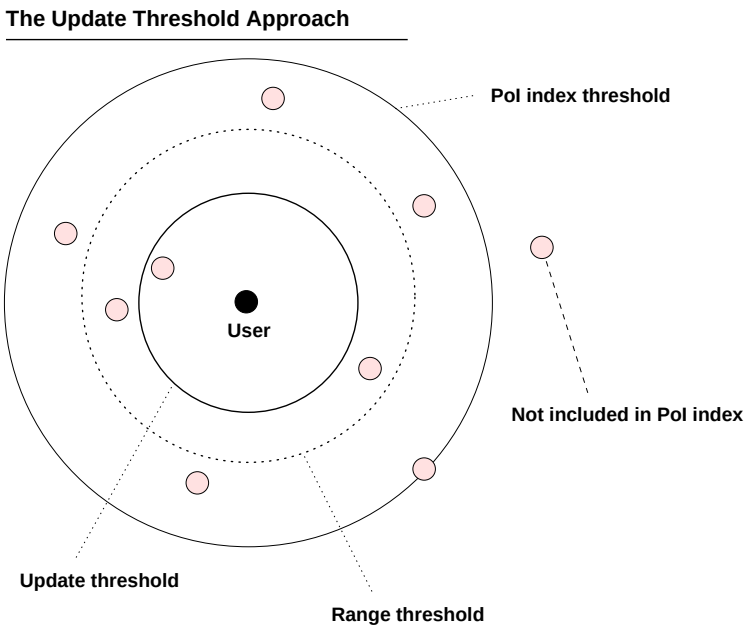


Figure 3.8: The update threshold approach has three thresholds. The PoI index, range and update threshold. The update threshold decides whether the client should issue an update. The Update threshold range is included in the PoI index.

and finding the nearest PoI functions are not available. In order to find the nearest PoI an Update Algorithm has been developed. The task of the algorithm is to find the nearest PoI and monitor whether the PoI is within the user's defined threshold. In the following the algorithm is described.

### The Update Algorithm

In order to find the nearest PoI the algorithm has to traverse the PoI index and calculate the distance to each of the PoIs. In order to traverse the PoIs from the PoI index, the PoIs have to be loaded into an array in the main memory. The PoIs are initialized as objects in the array. The algorithm is decomposed into the following task:

- Calculate the distance to each PoI in the array.
- Compare the calculated distance with the shortest distance so far.
- Store the shortest distance.
- When the end of the array is reached the object with the shortest distance is found.

- Check whether the object is within the user's defined threshold.
- If the object is within the user's defined threshold report it to the user.

The algorithm works as follows: For each PoI, object the algorithm traverses the PoI array, the Euclidean distance between the user's current location and the PoI object is calculated. If the distance is the shortest distance so far, the object's index and distance are stored. This is repeated until the algorithm reaches the end of the array. After the algorithm has traversed the PoI array, the shortest distance and the index to the corresponding object is found. The shortest distance is then compared with the user's defined threshold. If the shortest distance is within the user's defined threshold, the object with the shortest distance is retrieved.

The algorithm is a linear time algorithm, which means that the CPU time consumption of the algorithm is proportional with the length of the PoI array. Comparing the CPU power and the storage capacity of present smartphones, this performance is adequate for the monitoring process. The implementation of this algorithm is an essential component of the push based LBS Engine. The implementation of the algorithm is described in section 7.3.

### 3.3.3 Raster Map Caching

Raster map data changes very seldom, which means that the raster the map data is the most static data in the LBS application. However, the raster map data demands the most storage capacity, due to the size of the raster maps. Storing all the raster maps at the client terminal would not be possible, due to the limited storage capacity at the client. Storing a subset of raster map covering a whole city or region would limit the LBS application. Hence, a method that is able to store all the raster maps, while minimizing the storage capacity, is needed.

In the project, a *Raster Map Caching Approach* has been implemented. Instead of storing all the raster maps at the client, only the needed raster maps are stored. This approach minimizes the storage usage and does not limit the LBS application to be used in certain areas. In order to store the needed raster maps, the maps are ordered in an uniform grid, as illustrated in Figure 3.9.
Depending the user's location, only the raster maps around the user are downloaded from the raster map servers. However, this would preferably require a high speed Internet access from the client. At the moment, GPRS connection is available for mobile phones and is adequate for downloading raster maps. After a period of time, the LBS application will download a large number of raster maps and the memory will fill up. In order to avoid this situation, the storage is purged for unused raster maps. The common caching strategy is to purge the *Least Recently Used* raster maps [Sta97]. This can be achieved by setting a counter on each raster map downloaded. Each time a raster

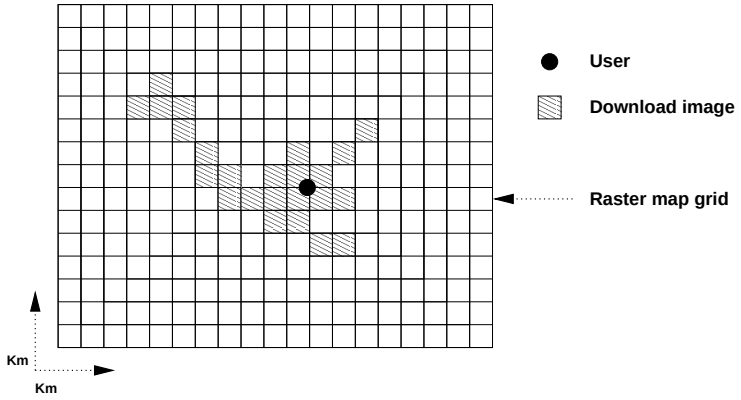


Figure 3.9: Illustration of the grid division. Only the raster maps nearby the user id downloaded to the client.

map is loaded into the memory the counter will be incremented. When the purge cycles begin, the raster map with the smallest counter will be deleted from the storage until a certain storage criterion is met. The implementation of the Raster Map Caching Approach is described in Section 7.6. After proposing the various solutions for implementing the *Combined Concept* and for developing a push based LBS the interaction between the client and the server will be presented.

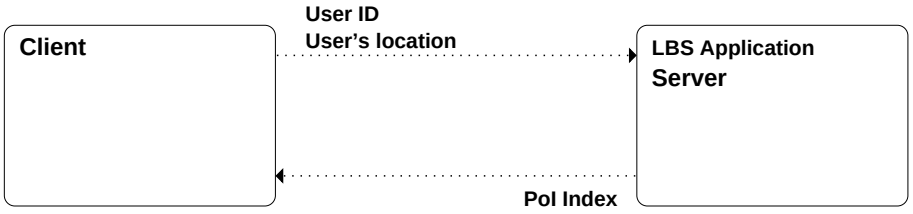


Figure 3.10: When the user starts the client LBS application, the *User ID* and the user's current location is sent to the server. The server generates and provides a PoI index for the user.

## 3.4 Overview

Figure 3.10 illustrates the interaction when the user start the LBS application. In order to use the LBS, the user needs to subscribe to the LBS. This is done using a Web page supported by the LBS Application Server. The subscribing process involves configuring the LBS and setting up a profile. The profile settings concern information about the user's desired PoIs, number of advertisements to receive, etc. After the subscription process the user is provided with a *Username* that is essential for accessing the

LBS. A user can define several profile settings depending on the user's usage of the LBS. When a user starts the client LBS application, the user has to enter the provided *Username* to access the LBS. When the user has entered the *Username*, the client LBS application requests a PoI index from the LBS Application server containing the user's desired PoIs. In order to generate a PoI index, the LBS Application server needs to receive information about the user's current location and *Username*, which is sent by the client LBS application. With this information and a *Profile ID*, which the user has defined as the current profile on the Web page. The PoI index is generated and sent to the client LBS application. The client LBS application starts the LBS.

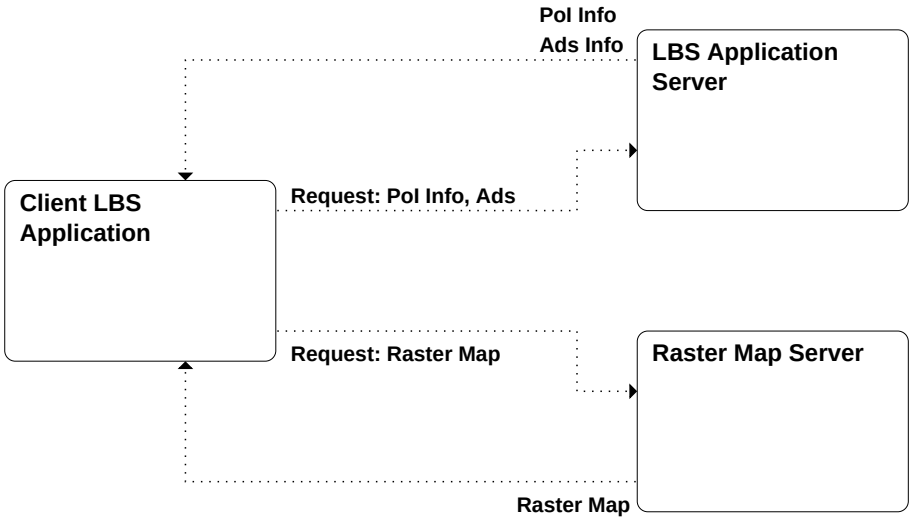


Figure 3.11: An illustration of the interaction between the Raster Map Server and the LBS Application Server.

The PoI index Update Algorithm, mentioned in Section 3.3.2, informs the user about nearby PoIs within a certain threshold defined by the user. If the user needs to acquire more information about the Nearest PoI such as address and description details, the client LBS application can send a *PoI Info* request to the server. The server retrieves the required information and if the user is nearby PoIs within a defined *advertisement distance* a location-based advertisement is pushed to the user. The *advertisement distance* defines the range the user would like to receive advertisements within. The advertisement mechanism is performed by the Client LBS application that request advertisements from the LBS Application server in certain intervals. The LBS Application server will retrieve the location-based advertisements if there are any available from the PoIs within the advertisement distance. The geographical raster maps are downloaded to the Client LBS Application by sending a request to the Raster Map Server. The Raster Map downloading policy is handled using the Raster Map Caching approach mentioned in Section 3.3.3. The interaction concerning PoI info, advertise-

ment info and raster map requests is illustrated in Figure 3.11.

In the next chapters, the technical details of design proposed are presented. Figure 1.2 in Section 1.4 illustrates the LBS Application Framework. The LBS Application Framework has been extended with additional components in order to enable push based LBSs. In the following Chapters the respective layers of the extended LBS Framework are described. Next the Data layer is described.

# Chapter 4

# Data Layer

This chapter describes the Data Layer of the LBS framework, which consists of content PoI data obtained from Nykredit and Aalborg Tourist Bureau. The PoIs from Nykredit includes PoIs such as supermarkets, kinder gardens and sport centers. The data from Aalborg Tourist Bureau are tourist PoIs such as hotels, monuments, museums, amusement parks, etc. Section 4.1 describes how the complete data model is designed and Section 4.2 goes into further detail with the temporal event handling used for PoI advertising.

## 4.1 Data Model

The model must contain the PoI data, which can originate from different data sources including the users. One approach is to create one PoI table, which holds all the attributes needed to describe the different kind of PoIs. This way the queries become very simple, since all PoI attributes are in the same table. For instance when a user asks for further information about a particular PoI it is not necessary to check which kind of PoI it is since they can all be extracted from the same table. However, for many different types of PoIs with different attributes, this would mean that the table would hold many attributes, which are relevant only for a small subset of PoIs. A more flexible approach is one where it is possible to relate tables of PoIs, which can originate from different data sources, to a main table containing only the attributes which are common for all PoIs such as id, name and coordinates. This way if other PoI data sources are added to the data model, the table only has to be related to the main PoI table using the PoI ID as foreign key.

In addition to PoI data the model must include data of users and their profiles. In the Online Aalborg Guide it should be possible for users to have more than one profile. This way the user can, for instance, create a profile for when the user is shopping, going to and from work or visiting tourist attractions.

In order to handle the time-related location-based advertisements, the data model must store these somehow. To handle this, a *POIEVENTS* table is created. This table is related to the main PoI table. This means that any advertisement is related to a PoI with a geographical position. The reason for this is, that the position is needed in order to provide advertisements that are within a specified range from the users. This is at the same time a limitation. For example, Web sites are not located at geographical positions, making it difficult to advertise for Web sites. Another problem could occur if a PoI is an event, which takes place in a city or a region. Then the PoI event would need to be related to a geographical area. Setting the geographical position to a place in Denmark and then setting the advertisement distance so large that it covers the whole of Denmark could solve the limitation of advertisements for Web pages. For an event, which takes place in a geographical area, the event can often be split into smaller events, which take place at a geographical position. For instance a carnival, which takes place at more than one location could be split into "The Parade", which starts at the town square, "The Bands Play" at the central park and "Crowning of the King of the Carnival" at the center stage in the central park. Another solution would be to create a table similar to *POIEVENTS*, which is not related to the main PoI table and hence does not have a reference to a geographical position.

The *POIEVENTS* table is able to model more than just advertisements. Other time-related information can be stored in the table. In the data from the Aalborg Tourist Bureau, some of the tourist PoIs has opening hours. These are also stored in the *POIEVENTS* table. If other time-related information is added at a later time the *POIEVENTS* table is able to handle this.

Figure 4.1 describes the complete data model. The schema contains the main table *POI*. This table contains the basic information that is needed to describe a PoI. All PoIs have a name and a position $(X,Y)$ as minimum. *UTM* and *DD* attributes represents the geographical position of the PoIs in UTM32 [USG01] measured in meters and in WGS84 [WGS03] measured in decimal degrees. In addition, a PoI can belong to a category (*POIAREA*) and a sub category (*POITYPE*). This is modeled as a many-to-none relationship between *POI* and *POIAREA* and between *POI* and *POITYPE*, since an area or type of PoI can be related to one or more PoIs and because some PoIs might not have any categorization, (*POIAREA*) or sub category (*POITYPE*). A *POIAREA* category could be "Attractions" or "Events" while a *POITYPE* sub category could be "Museums" or "Amusement parks" for category "Attractions" and "Sport" or "Music" for category "Events."

Table 4.1 shows three PoIs. Only the *UTM* attribute is shown to represent the position of the PoIs. This attribute is represented as an SDO.GEOMETRY type for use in Oracle Spatial to perform spatial operations, such as finding nearest PoIs. As shown in the table, the PoI "Tivoliland" belongs to the area with ID 2. Table 4.2 shows that this is an attraction. "Tivoliland" has the type with ID 11. Table 4.3 shows that it is

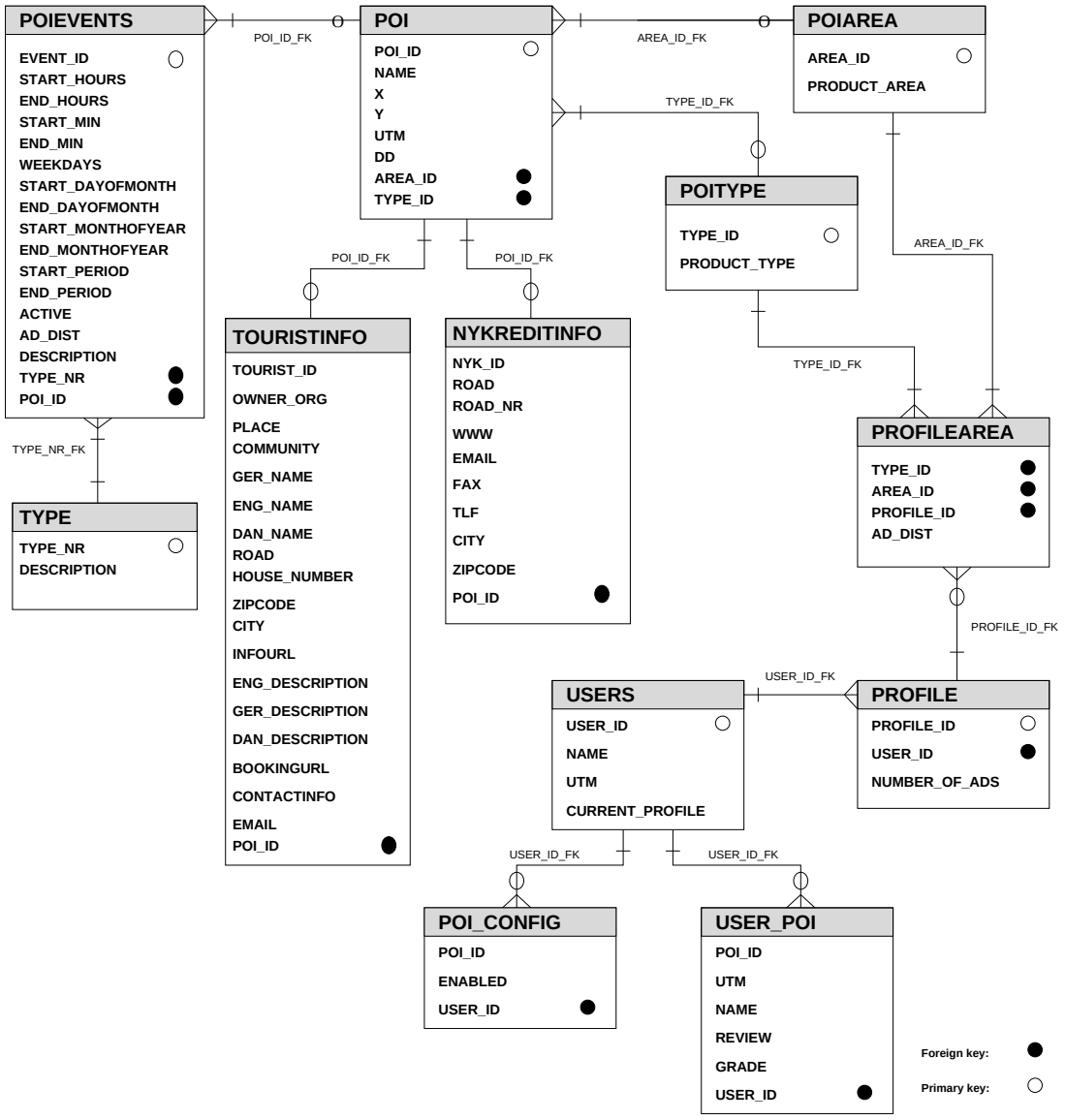


Figure 4.1: Data model

an amusement park. “Karneval i Aalborg” belongs to area 3, which is an event and has the type ID 10, which is a folk festival. “Netto” belongs to area 101, which is Nykredit and has type ID 116203, which is Supermarkets. All data from Nykredit is located in table *NYKREDITINFO*, and all these PoIs belong to the same PoI area with AreaID 101. The sub categories of the Nykredit PoIs are different and are located in the *POITYPE* table, for instance 116203 which is “Supermarkets.”

The *POI* table can be related to several sub PoI categories. This approach ensures a flexible design for the framework, since it is easy to relate additional sub PoI cate-

| **POI** | | | | |
|---|---|---|---|---|
| **POI_ID** | **NAME** | **UTM** | **AREAID** | **TYPEID** |
| 100059 | Tivoliland | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(556502, 6322690, NULL), NULL, NULL) | 2 | 11 |
| 100046 | Karneval i Aalborg | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(555323, 6322449, NULL), NULL, NULL) | 3 | 10 |
| 1566 | Netto | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(555600, 6323382, NULL), NULL, NULL) | 101 | 116203 |

Table 4.1: Sample of *POI* table.

| **POIAREA** | |
|---|---|
| **AREA_ID** | **PRODUCT_AREA** |
| 2 | Attractions |
| 3 | Events |
| 101 | Nykredit |

Table 4.2: Sample from *POIAREA* table.

| **POITYPE** | |
|---|---|
| **TYPE_ID** | **PRODUCT_TYPE** |
| 11 | Amusement parks |
| 10 | Folk festivals |
| 116203 | Supermarkets |

Table 4.3: Sample from *POITYPE* table.

gories to the main *POI* table. For use in the Online Aalborg Guide, two independent PoI data sources, Nykredit data and Aalborg Tourist Bureau data, has been related to the main *POI* table, as the *TOURISTINFO* and *NYKREDITINFO* tables. If further PoI data sources were to be integrated into the framework, they would be related in the same way. In addition it is possible to relate several categories to the sub categories. For instance a restaurant table or table of parks could be related to the *TOURISTINFO* table. These tables would then hold attributes, which are particular for the category e.g. *DogsAllowed* for parks. The *POI* table is related to the sub PoI categories *TOURISTINFO* and *NYKREDITINFO* by a one to zero-or-one relationship since each PoI is always related to one TouristInfo PoI or one NykreditInfo PoI. This means that a PoI does not have to be related to a sub category but each PoI of a sub category is related to a PoI in the main *POI* table.

Examples of PoIs from *TOURISTINFO* and *NYKREDITINFO* tables are shown in Tables 4.4 and 4.5, respectively. Table 4.4 shows some of the attributes that the table *TOURISTINFO* holds. The *TOURIST_ID* attribute is an external key that relates to the data from Aalborg Tourist Bureau. The *POI_ID* attribute is an internal key, which relates to the *POI* table. All PoIs from Aalborg Tourist Board have had 100000 added to their values, in order to avoid conflicts with other data sources e.g. Nykredit data. The *TOURISTINFO* table holds, amongst others, descriptions and names of PoIs in both Danish, German and English. For the *TOURISTINFO* PoIs the Danish name *DAN_NAME* has been used as the *NAME* attribute in the *POI* table. The *NYKREDITINFO* table does not have an attribute to describe the name of the PoI, since the data only has a Danish name and these values are placed in the *NAME* attribute of the *POI* table.

| **TOURISTINFO** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **TOURIST_ID** | **ENG_NAME** | **ENG_DESC** | **ROAD** | **HOUSE_NUM** | **ZIPCODE** | **CITY** | **POI_ID** |
| 59 | Tivoliland | Tivoliland is one of the largest amusement parks in Denmark and ... | Karolinelundsvej | 40 | 9000 | Aalborg | 100059 |
| 46 | Carnival in Aalborg | Sunday/Monday for children and Friday and Saturday for adults. During the carnival Aalborg receives about 100.000 people. | Vesterbro | 2 | 9000 | Aalborg | 100046 |

Table 4.4: Extraction from *TOURISTINFO* table.

Users are modeled in the *USERS* table, which has an user ID (*USER_ID*), name of user (*NAME*), UTM position (*UTM*) and *CURRENT_PROFILE* attribute. The *USERS* table is related to the *PROFILE* table through a one-to-many relationship since a user can have more than one profile. The profile which is currently in use is stored in the

| NYKREDITINFO | | | | | | |
|---|---|---|---|---|---|---|
| **NYK_ID** | **ROAD** | **ROAD_NR** | **CITY** | **ZIPCODE** | **TLF** | **POI_ID** |
| 1566 | Vesterbro | 99 | Aalborg | 9000 | 98167655 | 1566 |

Table 4.5: Extraction from *NYKREDITINFO* table.

*CURRENT_PROFILE* attribute.

Table 4.6 shows an example of how a user is represented in the *USERS* table. The user "Alex" is currently using the profile with ID 2 and has last updated the position (*UTM*) and at location (555003,6322500).

| USERS | | | |
|---|---|---|---|
| **USER_ID** | **NAME** | **UTM** | **CURRENT_PROFILE** |
| 1 | Alex | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(555003, 6322500, NULL), NULL, NULL) | 2 |

Table 4.6: Example of a user represented in the *USERS* table.

The *PROFILE* table has primary key *PROFILE_ID* which means that each profile has it's own unique key. The *NUMBER_OF_ADS* attribute is used by the user of the profile to state how many advertisements the user wishes to receive at one time. In Table 4.7 the user with ID 1 from before is related to profiles with Id's 2 and 3.

| PROFILE | | |
|---|---|---|
| **PROFILE_ID** | **USER_ID** | **NUMBER_OF_ADS** |
| 2 | 1 | 1 |
| 3 | 1 | 5 |

Table 4.7: Example of two profiles related to a user in the *PROFILE* table.

The *PROFILE* table is related to the *PROFILEAREA* table. The *PROFILEAREA* table contains IDs of categories of PoIs , *AREA_ID*, and sub categories of PoIs, *TYPE_ID*. Furthermore, the table contains attribute *AD_DIST*. *AD_DIST* is used when "pushing" advertisements to users. The user can specify for each category and sub category PoI, a range for each category and sub category PoI, within which a PoI must be located, in order for the user to receive an advertisement from this kind of PoI.

Table 4.8 shows that the user in this way could, for instance, specify "Attractions" (*AREA_ID* = 2) and "Amusement Parks" (*TYPE_ID* = 11) and set the Advertisement distance to 5000 meters (*AD_DIST* = 5000). The user has then subscribed to amusement parks in the profile with ID 2 and amusement parks will appear in the application,

if the user is near an amusement park. In addition the user will only receive advertisements from amusement parks if they are within 5 km. In the profile with ID 3 the user has chosen "Nykredit" as the area (*AREA_ID* = 101) and "Supermarkets" as the type (*TYPE_ID* = 116203) and the user has chosen to only receive ads from supermarkets, if they are within 500 m.

| **PROFILEAREA** | | | |
|---|---|---|---|
| **PROFILE_ID** | **AREA_ID** | **TYPE_ID** | **AD_DIST** |
| 2 | 2 | 11 | 5000 |
| 3 | 101 | 116203 | 500 |

Table 4.8: Example of a user represented in the *USERS* table.

The *PROFILE* table is related to *PROFILEAREA* through a one-to-many relationship since a user profile can subscribe to several categories and sub categories of PoIs. Finally the *PROFILEAREA* table is related to both *POIAREA* and *POITYPE* tables by a many-to-one relationship since more than one profile can have a subscription to one PoI area or PoI type and since one PoI area or one PoI type can be subscribed to in more than one profile.

The table *POI_CONFIG* is intended for use in the *Favorites* feature, where a user can save a PoI as a favorite for easy access later. Furthermore, the table is intended for use if the user wants to ban a PoI and never see it again in the application. This could be useful if the user goes past the same PoI again and again. This feature has not been implemented in the GPSOne prototype application, however, the table is ready for it to be implemented. There is a one-to-many relation between the *USERS* table and the *POI_CONFIG* table, since a user can have more than one banned or favorite PoI.

Table 4.9 shows how the user (*USER_ID* = 1) has marked the PoIs with Id's 16 and 17 as favorites (*ENABLED* = 1) and has banned the PoI with ID 20 (*ENABLED* = 0).

| **POI_CONFIG** | | |
|---|---|---|
| **POI_ID** | **ENABLED** | **USER_ID** |
| 16 | 1 | 1 |
| 17 | 1 | 1 |
| 20 | 0 | 1 |

Table 4.9: Example of the *POI_CONFIG* table.

The table *USER_POI* is used for storing user PoIs. The *User_PoI* feature has been developed on the server, but is not available in the GPSOne prototype application. A user PoI consists of the basic attributes: ID (*POI_ID*), position (*UTM*) and name (*NAME*). In addition, optional attributes are review/note and grade. These can be used to share

information among users and other users can benefit from other users' experiences. Finally the *USER_POI* table consists of a user ID, which identifies which user has submitted the PoI to the system. There is a one to many relation between *USERS* and *USER_POI* tables, since a user can be related to more than one user PoI.

Table 4.10 shows an example where the user (*USER_ID* = 1) has added three user PoIs "Home", "Work" and "Cheap Hostel".

| **USER_POI** | | | | | |
|---|---|---|---|---|---|
| **POI_ID** | **UTM** | **NAME** | **REVIEW** | **GRADE** | **USER_ID** |
| 1000001 | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(555213, 6322555, NULL), NULL, NULL) | Home | NULL | NULL | 1 |
| 1000002 | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(556003, 6323000, NULL), NULL, NULL) | Work | NULL | NULL | 1 |
| 1000003 | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(560567, 6322500, NULL), NULL, NULL) | Cheap Hostel | Nice unnamed hostel with clean beds. | 4 | 1 |

Table 4.10: Example of the *USER_POI* table.

## 4.2 Temporal Event Management

Figure 4.2 shows the part of the complete data model that is used for handling temporal data such as opening hours, events, advertisements, etc. The Aalborg City Guide contains actual data of opening hours from hundreds of PoIs in Aalborg and a smaller number of generated advertisements to use for the GPSOne prototype.

As Figure 4.2 shows the *POI* table is related to the *POIEVENTS* table by a one-to-many relation, since a PoI can have more than one event over a period of time and one event must be related to one PoI. In table 4.12 examples of data from the *POIEVENTS* table are given. The *POIEVENTS* table contains in addition to an event ID (*EVENT_ID*) a PoI ID (*POI_ID*) which holds the ID of the PoI which is related to the given event. The table also hold an attribute used to specify whether the PoI event is activated or deactivated (*ACTIVE*). The *TYPE_NR* attribute is used to distinguish different types of events. A table *TYPE* holds descriptions and ID numbers of the different types of events e.g. opening hours and advertisements. The *AD_DIST* attribute in table *POIEVENTS* can be used by commercial PoIs, such as cafe's, supermarkets, amusement parks etc. to specify the range they wish their advertisements to reach. For instance, an amusement park may want to "push" advertisements to potential customers up to 10 km away, while a small cafe might only want to advertise within a range of

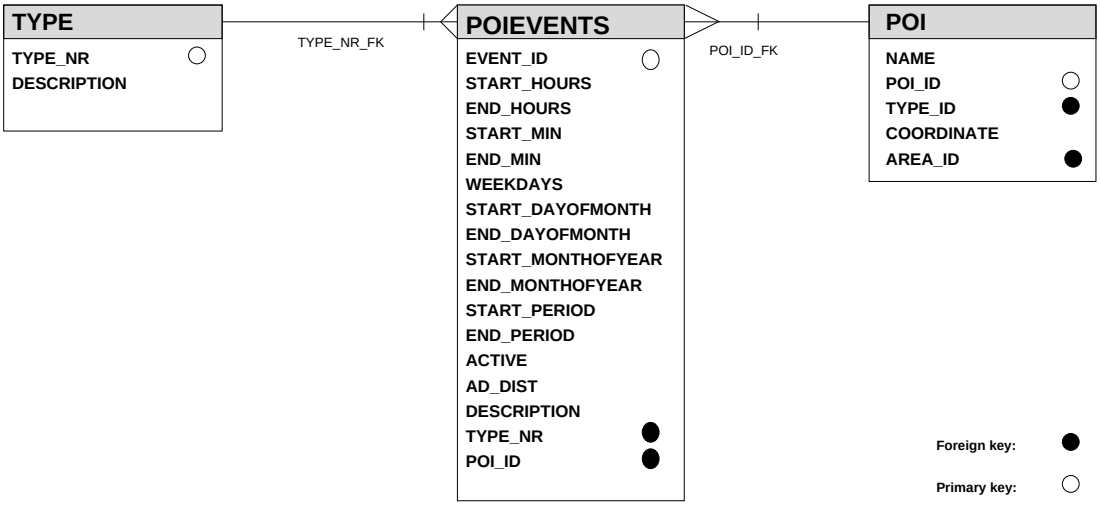


Figure 4.2: Temporal event management model

500 meters. Finally, the *DESCRIPTION* attribute holds a description of the event. This is particular interesting for ads, since this is the attribute that holds the text which is "pushed" to the users.

*POIEVENTS* table holds temporal attributes (*ST_HOURS, E_HOURS, ST_MIN, E_MIN, WEEKDAYS,ST_DAY, E_DAY, ST_MONTH, E_MONTH, ST_PERIOD, E_PERIOD*) which are explained in the following examples.

Table 4.12 on page 42 shows examples of the values in the *POIEVENTS* table and Table 4.11 displays the *TYPE* table.

| **TYPE** | |
|---|---|
| **TYPE_NR** | **DESCRIPTION** |
| 1 | Opening Hours |
| 2 | Advertisement |

Table 4.11: Table *TYPE*.

*POI_ID* = 100059 which is "Tivoliland" is related to the events with Id's 1, 2 and 3 which are all descriptions of the opening hours during the summer of 2003. Since they are not advertisements the *AD_DIST* attribute is set to NULL. Each event has a number of time-related attributes. The period attributes, *ST_PERIOD* and *E_PERIOD*, determine which dates the event should run between. the *WEEKDAYS* attribute determines which weekdays the event is valid for, with "1" being Monday and "7" being Sunday. The *ST_DAY*, *E_DAY*, *ST_MONTH* and *E_MONTH* describe a sub period in which the event runs. This method of modeling events allows for events that, for

| POIEVENTS | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **POI_ID** | **EVENT_ID** | **ST_HOURS** | **E_HOURS** | **ST_MIN** | **E_MIN** | **WEEKDAYS** | **ST_DAY** | **E_DAY** | **ST_MONTH** | **E_MONTH** | **ST_PERIOD** | **E_PERIOD** | **ACTIVE** | **AD_DIST** | **DESCRIPTION** | **TYPE_NR** |
| 100059 | 1 | 12 | 20 | 0 | 59 | 1234567 | 1 | 31 | 1 | 12 | 01.05.2003 | 30.06.2003 | Y | NULL | Opening hours | 1 |
| 100059 | 2 | 10 | 21 | 0 | 59 | 1234567 | 1 | 31 | 1 | 12 | 01.07.2003 | 31.07.2003 | Y | NULL | Opening hours | 1 |
| 100059 | 3 | 12 | 19 | 0 | 59 | 1234567 | 1 | 31 | 1 | 12 | 01.08.2003 | 31.08.2003 | Y | NULL | Opening hours | 1 |
| 100046 | 4 | 11 | 23 | 0 | 59 | 1234567 | 22 | 22 | 5 | 5 | 01.01.2003 | 01.01.2004 | Y | 50000 | Come and join the carnival parade | 2 |
| 1566 | 5 | 10 | 11 | 0 | 59 | 5 | 1 | 31 | 7 | 8 | 01.01.2003 | 01.01.2008 | Y | 1000 | Free taste samples of our new products | 2 |

Table 4.12: Example from the *POIEVENTS* table.

example, run in the first week of the month from May to June. The events for "Tivoliland" in Table 4.12 are interpreted as:

- *EVENT_ID* = 1. Applies from 12.00-20.59 each day from the from 1st of May 2003 until 30th of June 2003. Type is 'Opening Hours'.
- *EVENT_ID* = 2. Applies from 10.00-21.59 each day from the from 1st of July 2003 until 31st of July 2003. Type is 'Opening Hours'.
- *EVENT_ID* = 3. Applies from 12.00-19.59 each day from the from 1st of August 2003 until 31st of August 2003. Type is 'Opening Hours'.

The PoI with ID 100046 ("Karneval i Aalborg") is related to the event with *EVENT_ID* = 4. This event Applies from 11-24 on the 22nd of May 2003. The type is 'Advertisement' (*TYPE_NR* = 2) and the advertisement is only displayed for users which are within 50000 m and only if the PoI category is subscribed in the user's current profile.

The last event (*EVENT_ID* = 5) is in this example related to "Netto" (*POI_ID* = 1566). The event Applies from 10-12 every Friday all June and July month from 1. of January 2003 until 1. of January 2008. The type is 'Advertisement' (*TYPE_NR* = 2) and the advertisement is only displayed for users if they are within 1000 m and at the same time has subscribed to "Supermarkets".

Figure 4.3 shows how the advertisement distances work.
In Situation 1, the user will receive an advertisement from the PoI "Tivoliland" since the PoI "Tivoliland" is within the user's advertisement distance and the user is within the PoI's advertisements distance. Since both these conditions are true the user is pushed an advertisement. However, this situation must be present at the time when the GPSOne application checks the server for advertisements, which occur at certain time intervals. This means that the user is not overburdened with advertisement. But it also means that the situation as presented in Situation 1 can occur without the user being pushed an advertisement. This is the case if the conditions become true and then false in the time between two advertisements checks by the client application.

In Situation 2, the PoI advertisement distance is large enough to reach the user's position. However, the user has chosen a small advertisement distance for this type of PoI, so no advertisement is sent to the user.

In the last situation, Situation 3, the PoI "Netto" is within the user's advertisement distance. However, the PoI's advertisement distance is not large enough to include the user's position. Hence in this situation no advertisement is sent to the user.

The data model and related issues have been dealt with, and in the server application can be established. The next Chapter gives the architecture of the required services.

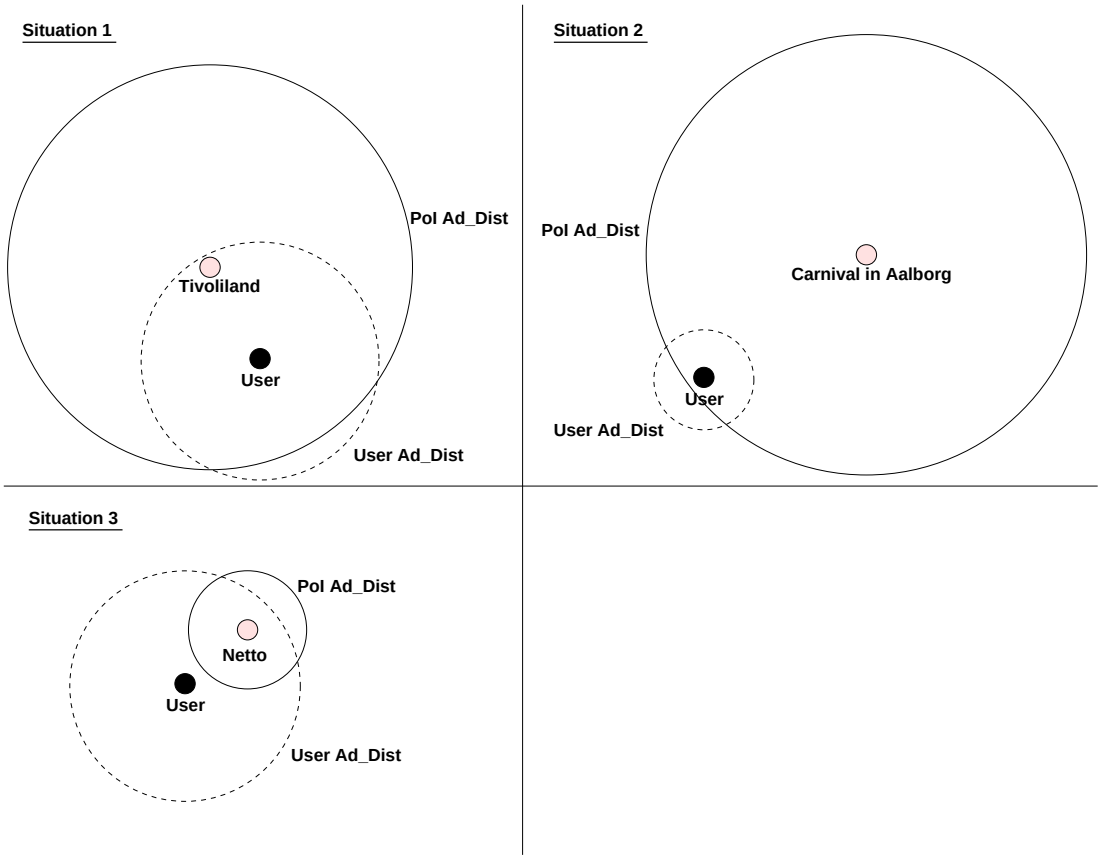


Figure 4.3: Examples of PoI and user advertisement ranges. In Situation 1 an advertisement is send to the user. In Situation 2 and 3 no advertisement is send.

# Chapter 5

# Server Application Layer

This chapter describes the design of the application level of the push-based LBS framework. The application level consists of components and services. The components can be used as building blocks, whereas services have a defined interface to client applications or web interfaces. The components are implemented as Java classes and the services are implemented as Java servlets. The framework is an extension of the LBS framework from [ACKN03], but only classes and methods used to implement the Online Aalborg Guide are included in this chapter. Section 5.1 describes the class diagram of the Server Application Layer. The next section, Section 5.2, describes the flow of the services.

## 5.1 Class Diagram

The class diagram makes use of an inheritance structure provided by an abstract superclass *LBSServlet*. This makes it easy to add additional services to the application layer by creating specializations of the *LBSServlet* class and hence inheriting the components related to *LBSServlet*.

Figure 5.1 shows the class diagram based on the UML notation. Due to the amount of methods contained in the classes, only the most important methods for understanding the application structure are included in the class diagram. The four classes at the bottom of the class diagram *PoIIndex*, *AdServlet*, *FurtherInfo* and *AdServlet* are concrete Java Servlet classes and make up the service layer since these have strictly defined interfaces to the client applications on the client layer. The service interface is described in Chapter 6. The four classes inherit from the abstract superclass *LBSServlet* which is a specialization of the standard Java2EE class *HttpServlet*. The *LBSServlet* class has relations to the three basic components of the LBS framework *Query*, *Database* and *GIS*. The four service classes inherit these relationships since the classes are specializations of *LBSServlet*. This inheritance structure makes it easy to add new services to

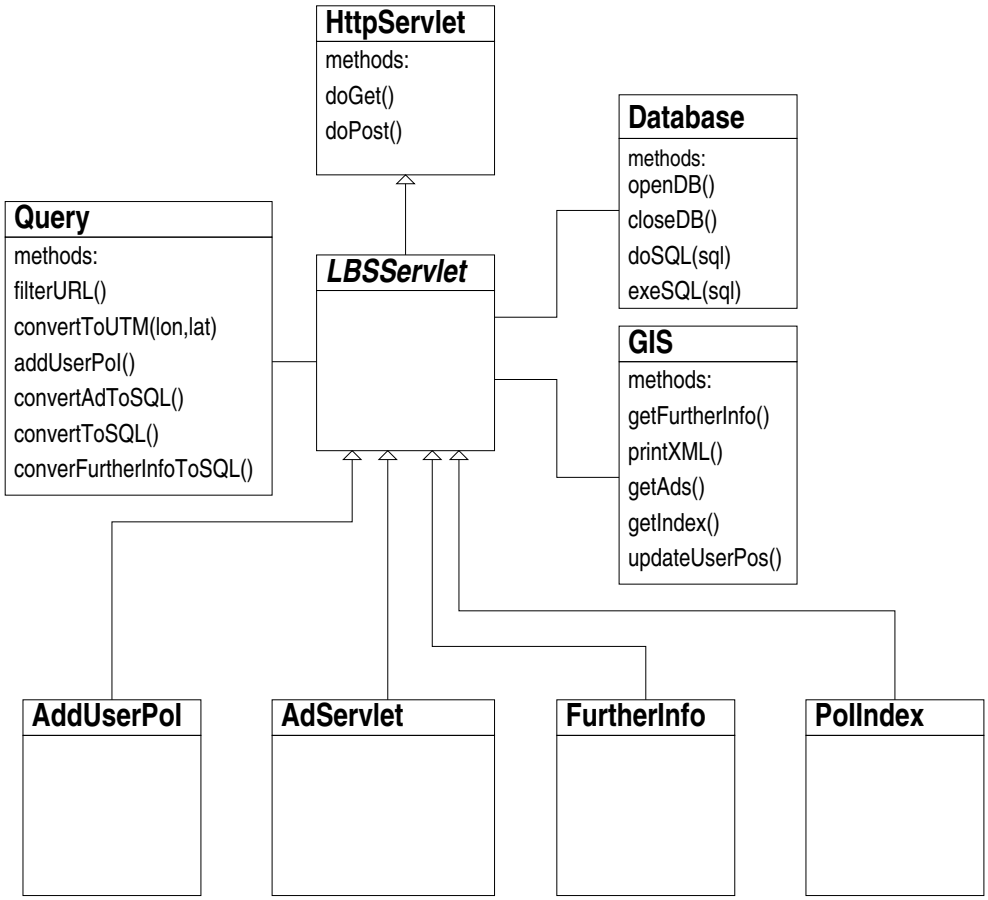


Figure 5.1: Class diagram of the server application layer.

the framework by adding a new sub class to the *LBSServlet* class and hence inherits the relations to the components needed to create services.

The component classes which are Java classes consist of *Query*, *Database* and *GIS* classes.

The *Query* class has methods to parse the input parameters received from the client application and use these parameters to form an SQL statement used to query the database. The `filterURL()` method extracts the values received from an HTTP request URL and stores the values in corresponding variables. From these variables it is possible to form SQL statements depending on which service is requested. The `convertFurtherInfoToSQL()` method forms an SQL statement based on the ID of a PoI and is called if the *FurtherInfo* service is requested. The `convertToSQL()` method is used when the *PoIIndex* service is called and forms an SQL statement, based on a user name or user ID and user coordinates. The `convertAdToSQL()` method is called when the *AdServlet* service is requested. Based on the user ID, coordinates and

system time, an SQL statement to be used for requesting advertisement is formed. The `AddUserPoI()` method is used when the *AddUserPoI* service is requested. This method opens a connection to the database, and adds the user PoI based on coordinates and user name or user ID, before closing the database connection. Finally, the `convertToUTM(lat,lon)` method can be used if requests are received in WGS84 decimal degrees format. If this is the case, the values are converted to UTM 32 measured in meters since this eases distance calculations. The GPSOne application always forms requests using the UTM 32 format. However, in the future other applications may use the WGS84 format.

The purpose of the *Database* class is to handle queries performed on the database. The class has a method for establishing a connection to the database, `openDB()`, and a method for closing connections to the database, `closeDB()`. Additionally the class contains two methods for performing queries on the database. The `exeSQL(sql)` method is used when executing insertions, updates or deletions in the database, whereas the `doSQL(sql)` method is used when querying the database. The `doSQL(sql)` returns a Java ResultSet object which is correspondent to a table, whereas the `exeSQL(sql)` method does not have any return object.

The *GIS* class is responsible for performing the queries received from *Query* object and returning the appropriate output to the client application. The `getFurtherInfo()` method is used when the *FurtherInfo* service is requested. The `getAds()` method is used in a similar way when the *AdServlet* service is requested. Similarly, `getIndex()` method is used when the *PoIIndex* service is requested. In this case the `printXML()` method is used. This method returns the output of the PoI index as either customized format used for the GPSOne application or as XML which can be used by future client applications. The customized format is specified in Chapter 6 which deals with the Service Layer of the framework. The last method included here is the `updateUserPos()` method which is called when the *PoIIndex* or *AdServlet* services are requested. When these services are requested the client application provides the position of the user. This is used by the `updateUserPoi()` method to connect to the database and update the server's knowledge of the user's position.

## 5.2 Sequence Diagrams

After having introduced the class design of the server application layer, a more specific model of the flow is given in the form of sequence diagrams. Sequence diagrams is a UML notation which gives a visualization of the actual structure and flow of a given task or function of the system [MMMNS98]. In the following the sequence diagrams of the services *PoiIndex*, *AdServlet*, *FurtherInfo* and *AdServlet* are presented.

### 5.2.1 PoIIndex

In Figure 5.2 the sequence diagram of the *PoIIndex* service is shown.

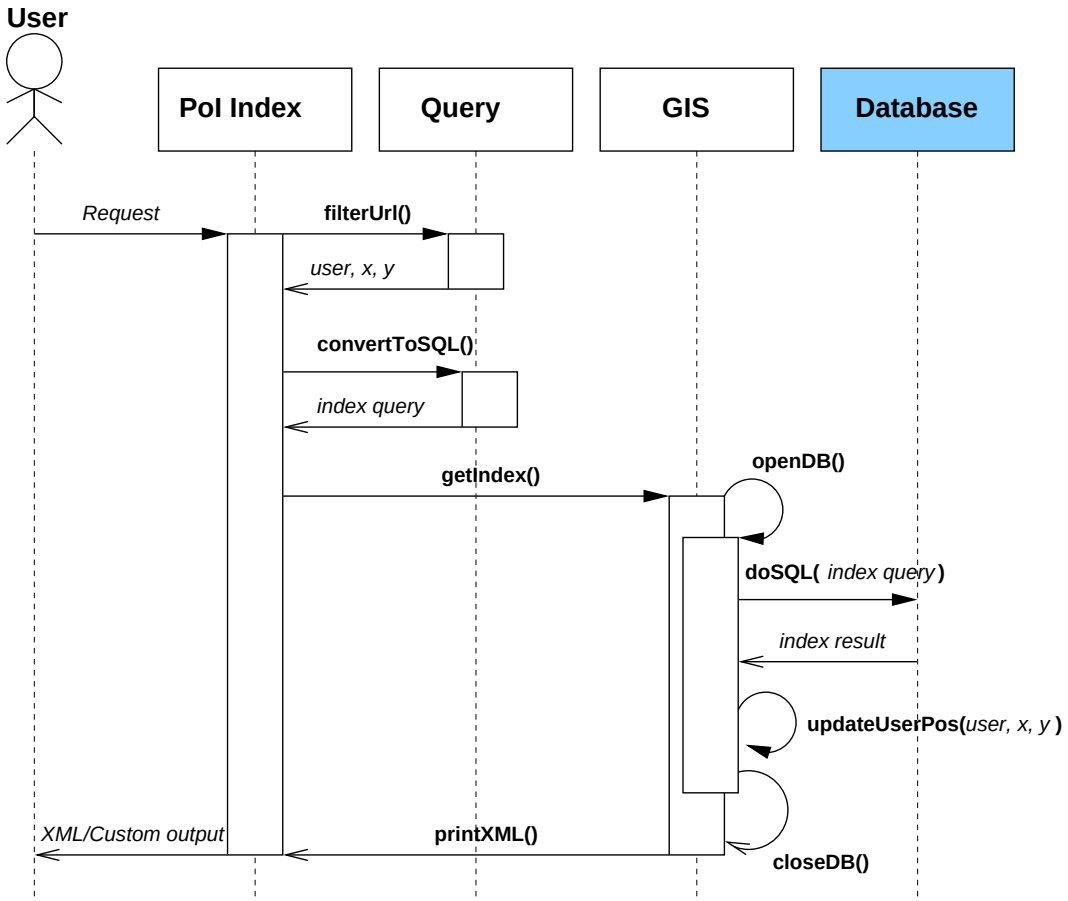


Figure 5.2: Sequence diagram for PoIIndex service.

The purpose of the servlet is to enable client applications to retrieve a PoI index, as mentioned in Section 3.3. The client makes a HTTP request for the service containing parameters containing the ID of the user, and the current coordinates of the client. These parameters are filtered from the URL and placed in corresponding variables by the `filterURL()` method. Based on these variables the SQL query for the index is formed by the `convertToSQL()` method. Next, the `getIndex()` method of the related *GIS* class is invoked. This method opens a database connection by calling the `openDB()` method from the *Database* class. The method then retrieves the SQL statement formed by the *Query* object, performs the query and receives a *ResultSet* object. Before closing the database connection the method updates the user position stored in the database by updating the *UTM* attribute of the *USERS* table. Finally the result of the query is returned to the client in the format requested. This is done by the `printXML()` method of the *GIS* class.

### 5.2.2 AdServlet

Figure 5.3 shows the sequence diagram of the *AdServlet* service. The purpose of this servlet is to deliver time-related location-based advertisements to client applications.

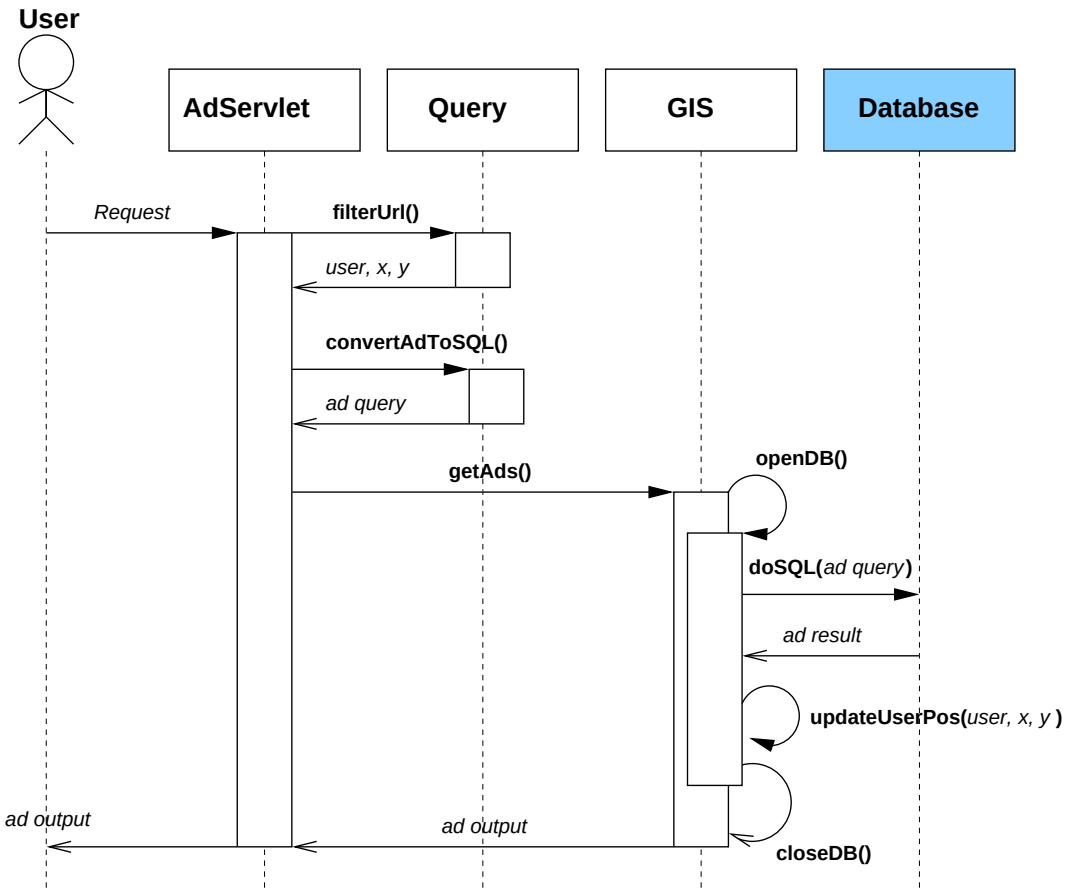


Figure 5.3: Sequence diagram for AdServlet service.

The flow of this service is almost identical to the flow of the *PoIIndex* service. The parameters for this service are also a user ID and client coordinates. The parameters are extracted from the URL and stored in variables by the `filterUrl()` method. The SQL query used to retrieve the correct advertisements is formed by the `convertAdToSQL()` method. The `getAds()` method of the *GIS* class is then called. This method opens a database connection, queries the database and receives a ResultSet object. Then the advertisement output is formed, based on a specified output format, before the method updates the user position and closes the database connection. This service does not print the output in other formats than the GPSOne application expects. This format is specified in Chapter 6.

### 5.2.3 FurtherInfo

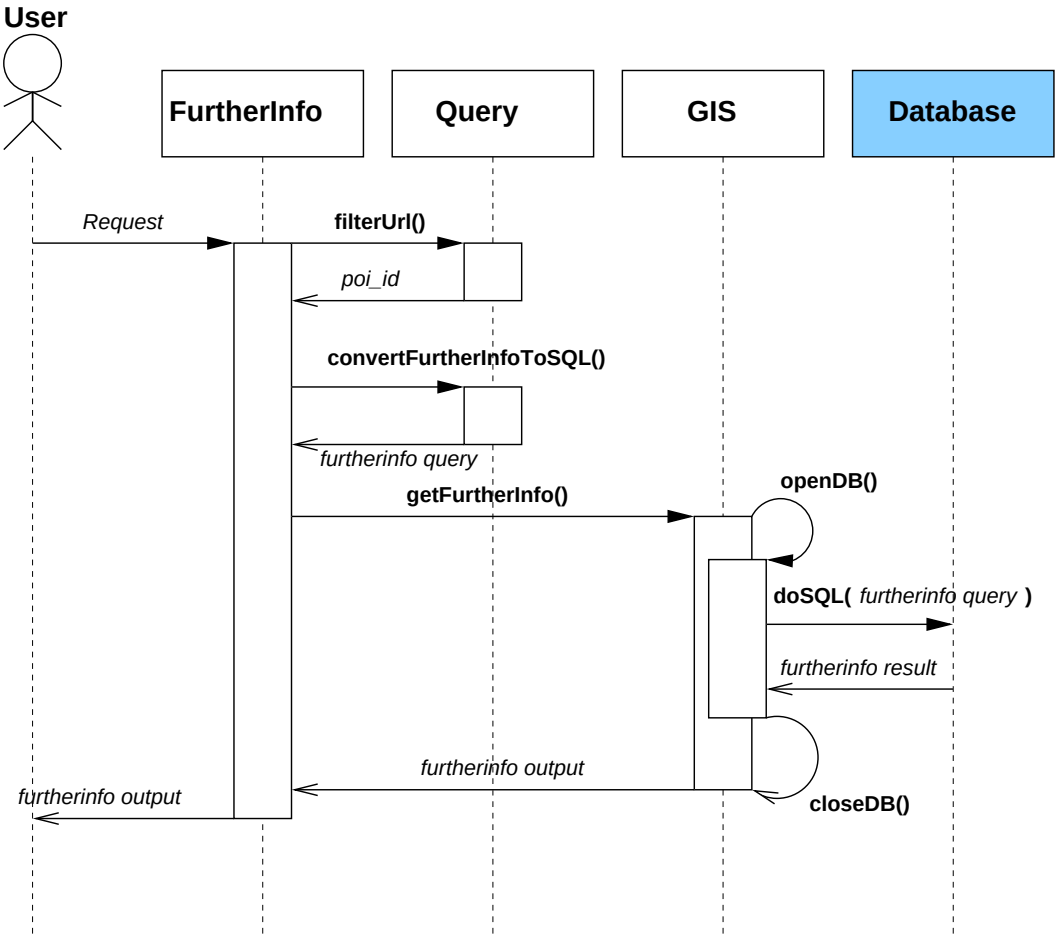


Figure 5.4: Sequence diagram for FurtherInfo service.

The flow of the *FurtherInfo* servlet is shown in Figure 5.4. The purpose of this service is to let client applications retrieve information about a particular PoI. The flow of this servlet follows the flow of the *PoIIndex* and *AdServlet* servlets. The parameter for the service is a PoI ID. This ID is parsed from the URL and stored in a variable by the `filterUrl()` method of the *Query* object. The SQL query, to be used for querying information about the PoI, is formed in the `convertFurtherInfoToSQL()` method. Then the `getFurtherInfo()` method of the *GIS* class is called. This opens a database connection, performs the query, outputs the result as specified by the further info format and closes the database connection. The main difference between the flow of *FurtherInfo* and *AdServlet* is that the position of the client is not a parameter and hence the client position in the database is not updated.

### 5.2.4 AddUserPoI

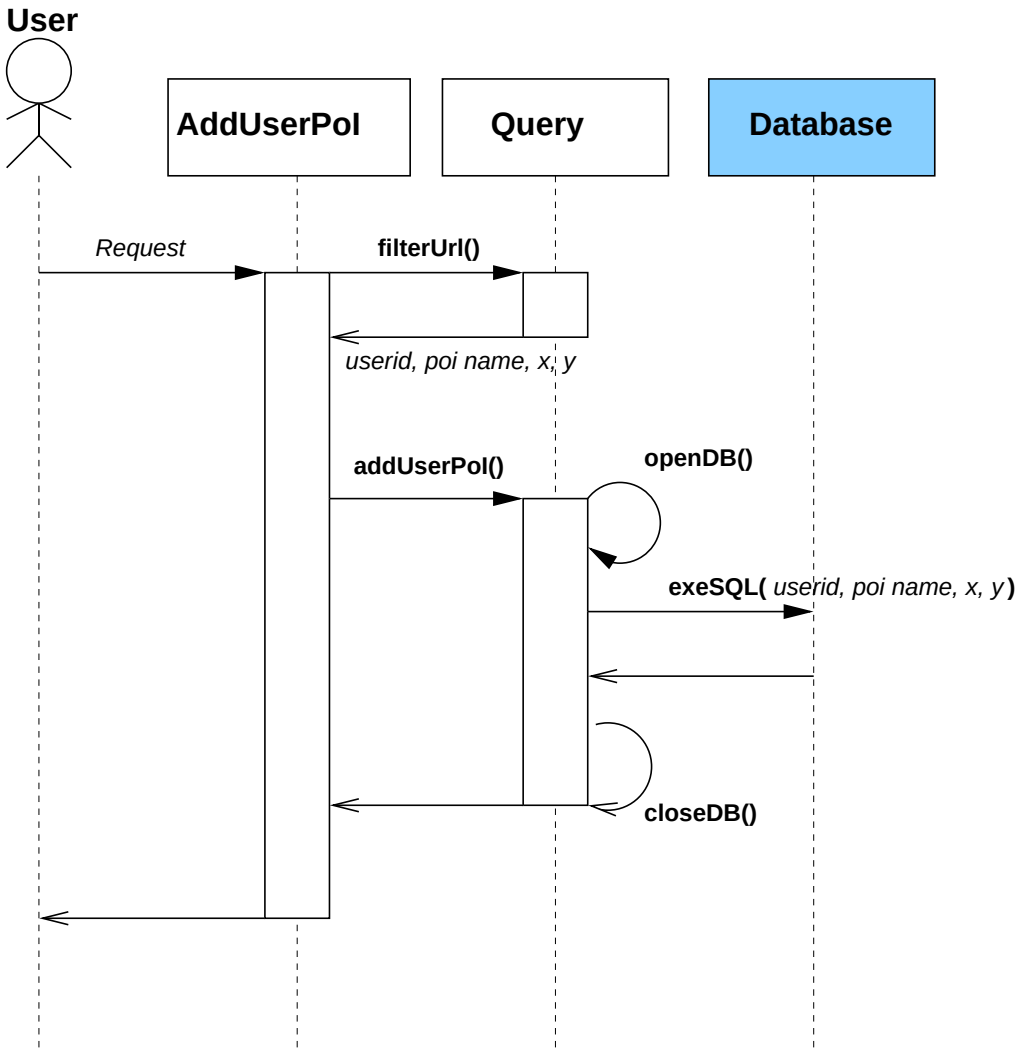


Figure 5.5: Sequence diagram for AddUserPoI service.

Figure 5.5 shows the sequence diagram of the *AddUserPoI* service. The purpose of this service is to let users add their own PoIs to the Online Aalborg Guide. The parameters for this service are a user ID, a name of the new PoI and the coordinates for the new PoI. Similar to the other services, these parameters are parsed from the URL and placed in corresponding variables by the `filterUrl()` method. After this, the `addUserPoI()` method of the *Query* class is called. This method opens a database connection, forms the SQL statement, based on the variables, and executes the SQL statement using the `exeSQL()` method of the *Database* class. The statement adds a new tuple to the *USER_POI* table. Finally, the method closes the database connection. The *AddUserPoI* service is fairly simple since the service basically forms an SQL

statement and executes this. Due to this and since no output is required the service does not need to initialize a GIS object as the other services do.

The services presented in this chapter have defined interfaces which are a part of the Service Layer. The Service Layer of the framework is described in the next chapter.

# Chapter 6

# Service Layer

This Chapter describes the interface definitions between the Service Layer and the Client Layer. In addition, examples of how the results are generated are shown. The interface definitions are used by the GPSOne application to access the services on the Service Layer. The interface between the services in the Service Layer and client applications of the Client Layer are based on HTTP requests. The next sections describe these requests based on example executions.

## 6.1 PoIIndex

The parameters for this service are:

- `userid` - The ID of the user.
- `username` - The user name of the user.
- `x` - The easting in meters.
- `y` - The northing in meters.
- `xdd` - The longitude in decimal degrees.
- `ydd` - The latitude in decimal degrees.
- `format` - The output format.

The request must contain three parameters: `userid` or `username` and (`x`, `y`) or (`xdd` , `ydd`). The `format` parameter is optional. If the `format` parameter is left out or if `format=0` then the request output is formatted as defined for the GPSOne application. `format=1` requests the output as XML.

Example:
`Http://euman-ext1.novi.dk/PoiIndex?`

```
username=alex&x=556500&y=6322700
```

This URL states that user “alex” requests a PoI index and the current location is (556500,6322700). This example extends the example in Section 4.1, where the user has subscribed to “Amusement parks”, “Folk festivals” and “Supermarkets” in the user’s current profile. The range threshold is defined to be 100 km and the PoI index threshold is defined to be 300 PoIs. Based on the inputs and the current profile the following query is formed:

```
SELECT C.DIST, C.POI_ID, C.NAME, C.X, C.Y
FROM
 (SELECT MDSYS.SDO_GEOM.SDO_DISTANCE(MDSYS.SDO_GEOMETRY(2001, 82343,
   MDSYS.SDO_POINT_TYPE(556500.0, 6322700.0, NULL),
   NULL, NULL), POI.UTM, 0.001) DIST,
   POI.POI_ID, POI.NAME, POI.X, POI.Y
  FROM
   POI, PROFILE, USERS, PROFILEAREA
  WHERE
   (PROFILE.USER_ID=0 OR USERS.NAME='alex')
   AND USERS.USER_ID = PROFILE.USER_ID
   AND PROFILE.PROFILE_ID = USERS.CURRENT_PROFILE
   AND PROFILEAREA.PROFILE_ID= PROFILE.PROFILE_ID
   AND POI.AREA_ID = PROFILEAREA.AREA_ID
   AND POI.TYPE_ID = PROFILEAREA.TYPE_ID
   AND SDO_WITHIN_DISTANCE(POI.UTM, MDSYS.SDO_GEOMETRY(2001, 82343,
   MDSYS.SDO_POINT_TYPE(556500.0, 6322700.0, NULL),
   NULL, NULL), 'DISTANCE = 100000')='TRUE'
  ORDER BY DIST ASC) C
WHERE ROWNUM <= 300
ORDER BY DIST ASC;
```

This query returns 221 PoIs. The first six results are shown in Table 6.1

| DIST | POI_ID | NAME | X | Y |
|---|---|---|---|---|
| 10.198039 | 100059 | Tivoliland | 556502 | 6322690 |
| 473.820641 | 1545 | Netto | 556035 | 6322791 |
| 571.171603 | 6741 | Superbrugsen Færø Plads | 556969 | 6322374 |
| 1001.20577 | 100058 | Las Vegas - Lasergame * Funhouse | 555767 | 6323382 |
| 1058.08979 | 100267 | Game World | 555835 | 6323523 |
| 1073.77838 | 100075 | 4. juli aftenfest - Galla middag | 555430 | 6322610 |

Table 6.1: Result of a PoI index query.

The PoI index is returned to the client application as semi-colon separated values and is presented as `POI_ID;NAME;X;Y`. In addition to the PoI index the **Update Threshold** is calculated and presented at the end of the PoI index as `¤DIST`. The output of the example execution for the first six results is:

```
100059;Tivoliland;556502;6322690
1545;Netto;556035;6322791
6741;Superbrugsen Færø Plads;556969;6322374
100058;Las Vegas - Lasergame * Funhouse;555767;6323382
100267;Game World;555835;6323523
100075;4.  juli aftenfest - Galla middag;555430;6322610
¤33329
```

If the service is requested to deliver the output in XML using the `format` parameter the URL would looks like this:

```
Http://euman-ext1.novi.dk/PoiIndex?
username=alex&x=556500&y=6322700&format=1
```

The output would then be the following:

```
<?xml version = ’1.0’ encoding = ’iso-8859-1’?>
<PoiIndex>
   <poi>
      <name>Tivoliland</name>
      <id>100059</id>
      <x>556502</x>
      <y>6322690</y>
   </poi>
   <poi>
      <name>Netto</name>
      <id>1545</id>
      <x>556035</x>
      <y>6322791</y>
   </poi>
   <poi>
      <name>Superbrugsen Færø Plads</name>
      <id>6741</id>
      <x>556969</x>
      <y>6322374</y>
   </poi>
   <poi>
      <name>Las Vegas - Lasergame * Funhouse</name>
      <id>100058</id>
      <x>555767</x>
      <y>6323382</y>
   </poi>
   <poi>
```

```
      <name>Game World</name>
      <id>100267</id>
      <x>555835</x>
      <y>6323523</y>
   </poi>
   <poi>
      <name>4. juli aftenfest - Galla middag</name>
      <id>100075</id>
      <x>555430</x>
      <y>6322610</y>
   </poi>
   <maxpoidist>33329</maxpoidist>
</PoiIndex>
```

## 6.2 AdServlet

The parameters for the AdServlet service are:

- `userid` - The ID of the user.
- `username` - The user name of the user.
- `x` - The easting in meters.
- `y` - The northing in meters.
- `xdd` - The longitude in decimal degrees.
- `ydd` - The latitude in decimal degrees.

The request must contain three parameters: `userid` or `username` and (`x`, `y`) or (`xdd` , `ydd`).

Example:
`Http://euman-ext1.novi.dk/AdServlet?`
`username=alex&x=556500&y=6322700`

The URL states that the user with user name “alex” requests the AdServlet service and has coordinates (556500,6322700).
Based on the input and the current profile a query is formed as shown in Appendix A. Depending on the system time, the parameter and the current profile in use the query returns a number of advertisements. Examples of results are shown in Table 6.2
The output from the advertisement query is returned to the client applications and presented as `NAME: DESCRIPTION`.

| DIST | POI_ID | NAME | DESCRIPTION | CURRENT_PROFILE | AREA_ID | TYPE_ID | EVENT_ID |
|---|---|---|---|---|---|---|---|
| 10.198039 | 100059 | Tivoliland | Senior citizens entry for free | 4 | 2 | 11 | 1027 |
| 473.820641 | 1545 | Netto | Free taste samples of our new products | 4 | 101 | 116203 | 1028 |
| 1203.46583 | 100046 | Karneval i Aalborg | Come and join the carnival parade | 4 | 3 | 10 | 1029 |

Table 6.2: Result of an advertisement query.

The output from the results in Table 6.2 is:

```
Tivoliland:  Senior citizens entry for free
Netto:  Free taste samples of our new products
Karneval i Aalborg:  Come and join the carnival parade
```

The client application can then use this information to display for the users in an appropriate way. The number of advertisements displayed is specified by the user in the profile setting.

## 6.3 FurtherInfo

The parameter for this service is:

- `poiid` - The ID of the PoI.

This request only needs to contain this one parameter.

Example:
`Http://euman-ext1.novi.dk/FurtherInfo?poiid=100059`

This example URL requests further information about the PoI with the ID 100059. Depending on whether the PoI originates from the Nykredit data sources or Aalborg Tourist Bureau, the query is performed on the *NYKREDIT_INFO* table or the *TOURIST_INFO* table. The queries are trivial select statements over two tables. Examples are shown in Appendix B. Since the PoIs have different attributes depending on which kind of data source they originate from, the output also depends on this. The output is formed in the following way for PoIs from the *NYKREDIT_INFO* table:

```
name¤NAME
address¤ROAD ROAD_NR
city¤ZIPCODE CITY
tlf¤TLF
```

For PoI from the *TOURIST_INFO* table the output is formed as:

```
name¤NAME
address¤ROAD HOUSENUMBER
```

```
city¤POSTALCODE CITY
DESC¤ENG_DESCRIPTION
```

The URL example requests further information about “Tivoliland” (PoI ID = 100059) which is stored in the *TOURIST_INFO* table and hence the output is:

```
name¤Tivoliland
address¤Karolinelundsvej 40
city¤9000 Aalborg
DESC¤Tivoliland is one of the largest amusement parks in
Denmark and since the start in 1946 the purpose has been
to give the guest value for money.  Tivoliland is a world
of it’s own with plenty of star attractions.  Most of them
made especially for Tivol
```

The specified output format is utilized by the GPSOne application in order to parse the values and display the information in an window for the user. The last line containing a description is always trimmed to 254 characters, since 255 is the maximum number of characters a single file line of the Symbian 6.1 OS can handle.

## 6.4 AddUserPoI

The parameters for the AddUserPoi service are:

- `userid` - The ID of the user.
- `username` - The user name of the user.
- `x` - The easting in meters.
- `y` - The northing in meters.
- `xdd` - The longitude in decimal degrees.
- `ydd` - The latitude in decimal degrees.
- `poiname` - The name of the added user PoI.

The request must contain four parameters: `userid` or `username` and (`x`, `y`) or (`xdd` , `ydd`) and finally `poiname`.

Example:
`Http://euman-ext1.novi.dk/AddUserPoi?username=alex&`
`x=556500&y=6322700&poiname=parking_lot_near_tivoli`

This URL example states that a user wishes to add a user PoI. In order to insert the values into the database, an available user PoI ID must first be found. This is done by sorting the user PoI ID's in the *USER_POI* table and selecting the last. If the request contains a user name the corresponding user ID must be found since the *USER_POI* table stores user ID's. This is done by a simple lookup in the *USERS* table. For this example the highest user PoI ID is 1000004 (*POI_ID* = 1000004) and the user ID of "alex" is 1 (*USER_ID* = 1). The SQL statement to insert a user PoI into *USER_POI* table is then:

```
INSERT INTO USER_POI VALUES(
1000004+1, MDSYS.SDO_GEOMETRY(2001, 82343, MDSYS.SDO_POINT_TYPE(556500,
6322700, NULL), NULL, NULL), 'parking_lot_near_tivoli', NULL, NULL, 1);
```

Table 6.3 shows the *USER_POI* table after insertion of the new PoI.

| **USER_POI** | | | | | |
|---|---|---|---|---|---|
| **POI_ID** | **UTM** | **NAME** | **REVIEW** | **GRADE** | **USER_ID** |
| 1000004 | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(555213, 6322555, NULL), NULL, NULL) | Home | NULL | NULL | 1 |
| 1000005 | SDO_GEOMETRY(2001, 82343, SDO_POINT_TYPE(556500, 6322700, NULL), NULL, NULL) | parking_lot_near_tivoli | NULL | NULL | 1 |

Table 6.3: The *USER_POI* table after insertion of the user PoI with ID 1000005.

# Chapter 7

# Client Application Layer

This chapter describes the design and implementation of the client application. The general architectural design is presented in Section 7.1, and the implementation of each part of the application is described in detail. Section 7.2 describes the *Core* class, Section 7.3 describes the *Update* class and the implementation of the update algorithm. In Section 7.4 the *FileHandler* class is described and in Section 7.5 the class handling the local PoI cache is described. Section 7.6 describes the implementation of the map display in the GPSOne application, and 7.7 details how the classes operate together.

## 7.1 Class Diagram

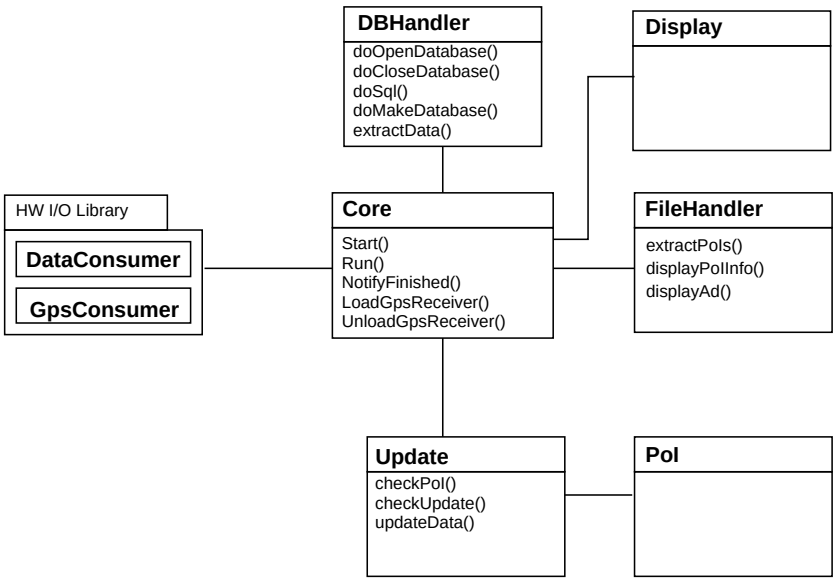


Figure 7.1: The Class Diagram of the client application

The Class Diagram for the client application is illustrated in Figure 7.1. As illustrated in the figure, the client has eight classes. The main class in the Class Diagram is the *Core* class. The *Core* class is responsible for handling GUI events and delegating tasks to the other classes.

The *Update* class is responsible for keeping track of PoIs and for requesting updates when needed. The *PoI* class contains information on PoIs. The *FileHandler* class handles file operations, such as opening retrieved data and images files. The *DBHandler* class handles the small-scale database on the client. The *Display* class is responsible for updating the user interface, including map and PoI display.

The *DataConsumer* and *GpsConsumer* classes provide interfaces to the GPS and to the Internet. These classes are a part of an external set of libraries, HW I/O library, that implement the protocols NMEA and HTTP respectively. These protocols are not described in further details. The *Core*, *Update*, *Display DBHandler* and *FileHandler* classes are described in further detail in the following sections.

## 7.2 The Core Class

The *Core* class acts as a central task unit. When a new set of GPS coordinates is ready from the *GpsConsumer*, the *Core* class invokes the *Update* class. The *Update* class then checks to see if an update is needed. If an update is needed, the *Update* class calls the `getPoIIndex()` method of the *Core* class. The `getPoIIndex()` method calls the `Get()` method of a *DataRetriever* object from the *DataConsumer* class. The `Get()` method is called with an URL to the service. The *DataRetriever* then retrieves the appropriate file and calls the `NotifyFinished()` method in the *Core* class. The `NotifyFinished()` method uses a state variable to determine which type of data the *DataRetriever* is retrieving. This variable can have one of five values:

| Value | Meaning |
|---|---|
| **EGetNone** | The *Dataretriever* is idle, and is ready for new requests. |
| **EGetPoIIndex** | The *Dataretriever* is asked to get the Index of PoIs. |
| **EGetPoIInfo** | The *Dataretriever* is retrieving information about the currently nearest PoI. |
| **EGetImage** | The *Dataretriever* is retrieving an image. |
| **EGetAd** | An advertisement text is requested through the *Dataretriever*. |

Depending on the state, an appropriate data handling is called. For the PoI index the *FileHandler* method `extractPoIs()` is called, followed by a call to the *DBHandler* method `extractData()`. For further information about a PoI, denoted *PoI Info*, the *FileHandler* method `displayPoIInfo()` is called. If an Image is being loaded, the

*Display* class method `getImageDone()` is called to notify that the image is ready. For Ads, the `displayAd()` method of the *FileHandler* is called. The flowchart in Figure 7.2 shows the flow of the *Core* class. The "Update Needed" choice is handled in the `checkUpdate()` method in the *Update* class. The "DataRetriever Idle" choice is checked whenever any kind of data download is requested. If the DataRetriever is idle, the request is fulfilled, but if not, the request fails, and the request will need to wait for the next GPS update, which is performed every second. This potentially leads to starvation, especially when images are requested. Since images are cached locally, the problem is only present when the user enters a not previously visited area.

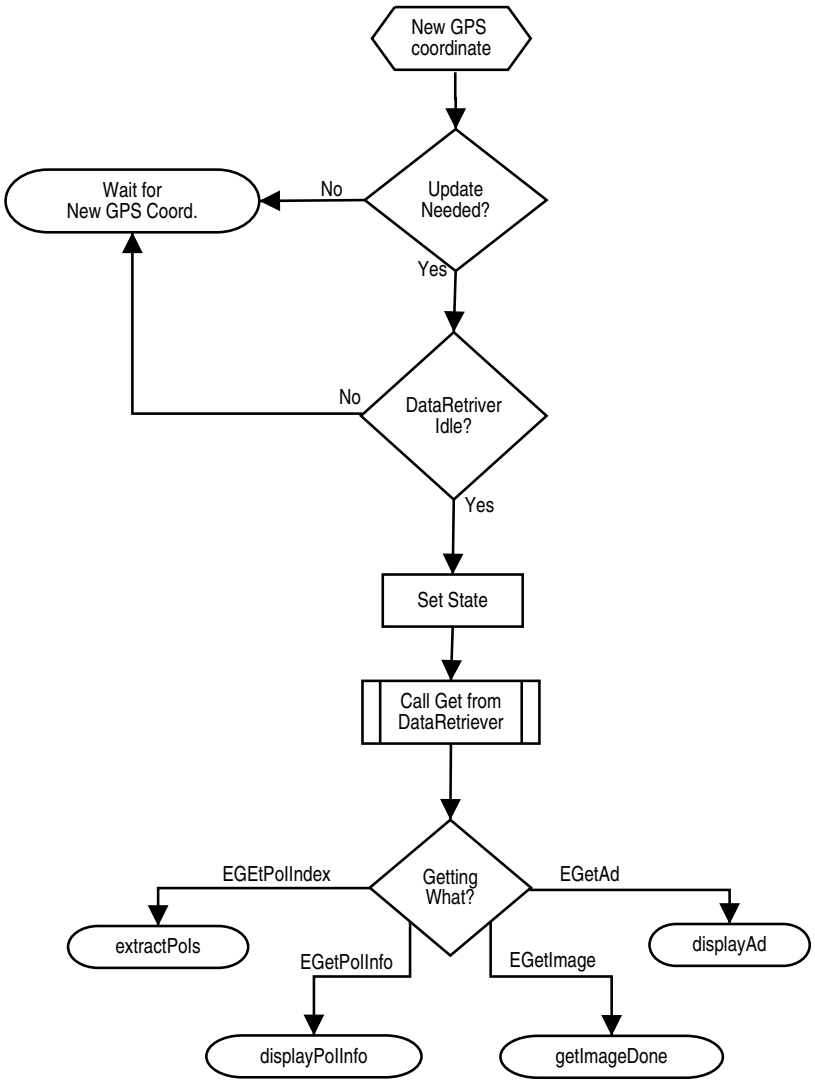


Figure 7.2: The Flowchart of the Core class.

## 7.3 The Update Class

The *Update* class implements the algorithm in Section 3.3.2. The algorithm forms the basis for determining whether an update is necessary or not. The method `checkUpdate()` of the *Update* class is called by the *Core* class with the current GPS position as parameter whenever the GPS produces a valid position.

The position is checked against the last update position and the update threshold, to see whether an update is needed. If there is a need to update, the method `getPoIIndex()` of the *Core* class is called, and retrieves a PoI index from the LBS Application server. If an update is not required, the PoIs currently stored in memory, namely the ones which are within the update threshold, are checked to find which is the nearest PoI. The code for the `checkUpdate()` method is presented below:

```
Function checkUpdate(){

  if (boundThreshold()){
    updateData();
  }
  else{
    checkPoI();
  }
};
```

The `boundThreshold()` method is the update threshold mentioned in Section 3.3. If the user's movement is within the (0.3 $\times$ Threshold Dist) distance the method returns false and the `checkPoI()` algorithm begins. Otherwise, the PoI array in the main memory is no longer valid, and an update request to the LBS Application server is initialized by the `updateData()` method.

The `checkPoI()` method is responsible for finding the nearest PoI within the user's defined threshold. The code for the algorithm is described below:

```
 1 Function checkPoI(){
 2 if (!isUpdating){

 3   for ( i=0; i < ArrayOfPoI.EndOfIndex; i++ ){

 4    ArrayOfPoI[i].PoIDist =
 5    calcDist(currentLocationX,CurrentLocationY,
 6    PoILocationX, PoILocationY) ;

 7    if (ArrayOfPoI[i].PoIDist  < UserThreshold){

 8      if (ArrayOfPoI[i].PoIDist  < NearestPoI.PoIDist){
 9       NearestPoI = ArrayOfPoI[i];
10       isUpdating = true;
        }
```

```
      }
11     else {
12      isUpdating = false;
       }
     }
   }
```

The array *ArrayOfPoI* is the PoI index. Each element in the array contains a PoI object. The *PoIDist*, an attribute in the PoI object, is used by the algorithm, and contains the distance between the user's position and the PoI's position. The distance is calculated by the `calcDist()` method. *NearestPoI* is an attribute that contains the nearest PoI object.

The algorithm works as follows: For each PoI in the PoI index the distance between the user's position and the PoI is calculated by the `calcDist()` method (lines 1-6). The calculated distance is compared with the user's predefined Threshold (line 7). If the distance is less than the user's predefined threshold, the PoI is a candidate to be the nearest PoI in relation to the user's current position.

The candidate PoI's distance is compared with the nearest PoI object, so far. If the candidate PoI is nearer than the nearest PoI object so far, the candidate PoI object is saved as the nearest PoI (lines 7-10). When the algorithm reaches the end of the array, the nearest PoI is found. This is a linear time algorithm, which means that the larger the number of PoIs in the array, the longer it takes to complete the algorithm. Hence, the size of the array is balanced according to the size of the main memory and the number of PoIs in the index, in order to have the lowest possible response time. Low response time is crucial in order to provide precise data to the user. This depends on the GPS receiver's update frequency and the user's speed.

If a PoI is found within the user's threshold, the information about the PoI is pushed to the user. This algorithm runs every second, if valid coordinates are received from the GPS receiver. The *isUpdating* attribute is used to ensure that the algorithm has finished execution before it is executed again with new coordinates as input.

## 7.4 FileHandler Class

The *FileHandler* is responsible for file related operations. Any time a text file is to be displayed or in other ways manipulated, it is done in the *FileHandler* class. Although the *FileHandler* class handles files, the image files needed for the map, are not handled in this class. The reason for this is, that the image decoding is an integral part of the *Display* class.

The *FileHandler* class handles three types of files; PoI index files, PoI Info files and advertisement files. Each of these three file types have a unique form as described in Chapter 6, and each needs to be handled differently.

The PoI index stored in the PoI index files needs to be extracted and inserted into the PoI database on the client. For each PoI it is necessary to construct an SQL statement which can be executed by the *DBHandler* method `doSql()`. Furthermore, the server calculates a new update threshold, which is also extracted and updated in the *Update* class.

PoI info files contain further information about a PoI. In the method `displayPoIInfo()`, the *Filehandler* extracts the formal data, i.e. name, telephone number and address of the PoI, from the file and presents it in list form. If a description of the PoI is available, the list contains an extra entry, “Show Further Info”, which allows the user to view a description of the PoI.

The advertisement files contain the text of the advertisements, so the *FileHandler* only needs to display the text, this is done in the `displayAd()` method.

## 7.5 DBHandler Class

The *DBHandler* class is used to handle the PoI database on the client. The database stores PoIs from the PoI index in permanent storage. The reason for storing the PoIs in the database, is to minimize the need for updating the PoI index from the LBS Application Server. Especially if a user has turned off the application it would be preferable to store the PoI index, so the user does not have to update the PoI index every time the application is started. When storing the PoI index, a PoI index update is only necessary if the update threshold is exceeded.

The *DBHandler* class acts as a kind of wrapper class for the DBMS of the Symbian OS. The `doSql()` method is used for executing SQL statements created by the *FileHandler*. The DBMS available on the Symbian platform supports a subset of the SQL standard. For the purpose of the client application, only the `SELECT`, `DELETE` and `CREATE TABLE` statements are needed.

The PoI database consists of a single table *PoI* which has four attributes:

**id** The PoI ID from the server database.

**name** The name of the PoI, for displaying to the user.

**x** The UTM longitude of the PoI.

**y** The UTM latitude of the PoI.

The *DBHandler* creates the table using the SQL statement:

```
CREATE TABLE PoI
(id INTEGER, name CHAR(127),x INTEGER, y INTEGER)
```

The small scale DBMS is sufficient to provide long term storage of a PoI Index. Each time the *Update* class needs to find the nearest PoI, the PoI are retrieved from the database. This is done by the `extractData()` method, which extracts data from the database and uses the data to create an array of PoI objects.

## 7.6 Display Class

This section describes the *Display* class of the Aalborg Guide. The main functionality of the *Display* class lies in the map display. The main purpose of the map is to give the user a visual representation of the surroundings. With text-based descriptions, the user may be mislead by the description. With a map, the users can see their positions in relation to the PoIs, helping them find a PoI. The following describes the map display design and implementation.

### 7.6.1 Design

In this project, the maps are acquired online from KMS. The maps from KMS are raster maps. The KMS 'Kortforsyningen' can output a map of a selected area of Denmark as a JPEG or PNG formatted image. The request are HTTP request and is based on the OpenGIS Web Map Server Interface Implementation Specification [Ope03b]. This service is used to retrieve the maps needed in the application.

One way of getting a map to present on the client terminal, is to retrieve the entire area covered by the Aalborg Guide. This is infeasible, as an image with a sufficient level of detail would take up more storage space than available on the client. The solution to this is to partition the entire image into smaller parts that will fit on the client terminal. This allows the client to only store relevant parts of a map, that is, the parts the user visits.

This method for using several small images to create one large is called tiling, and is widely used for computer graphics, most notably in video games [McI96]. The principle behind this is illustrated in Figure 7.3. The entire map is divided into tiles of equal size. Each of these tiles are then assigned a global set of coordinates which specifies the position of each tile. When drawing the map to the screen, only the tiles that overlap with the visible area are drawn. The method can be used to minimize the number of images stored in main memory. If the number of images in memory can cover the

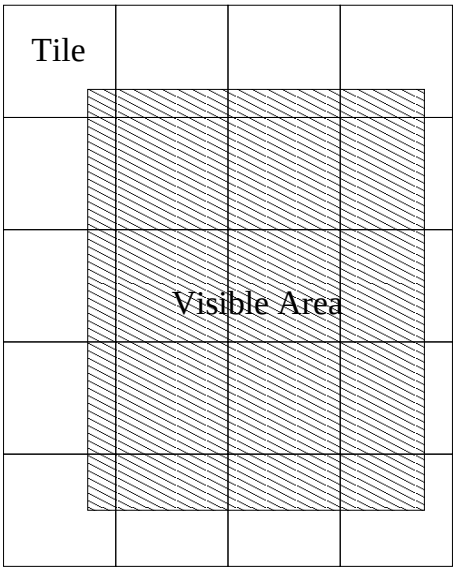


Figure 7.3: An image drawn from several individual tiles.

screen, a smooth scrolling effect can be achieved. This is especially important when showing a map, since the user will need to continuously follow the road on the map. If the map is not moving smoothly, the users could loose track of their position.

Smooth animation of tile based graphics is achieved by the method shown in Figure 7.4. When the map moves, the visible tiles are drawn. Each tile is represented by a bitmap that can be drawn on the screen. In video games, when the tiles move off the visible screen, they are removed and new tiles are loaded and added on the opposite side of the screen. This does not make sense when dealing with geographical maps, however, the principle of tiling images is used. The tile bitmaps are retrieved from KMS, but are cached in local storage.

Some limitations arise when designing applications for mobile terminals. These limitations have an influence on the design of the application. The terminal used, Nokia 7650, has a maximum 4MB of storage, meaning that a limited amount of map data can be stored. The maximum data rate of the GPRS phone used is 40.2kbps [Nok03a], which is slow for updating maps from the Internet. These limitations affect the map display of the application in these ways:

- The number of map pieces (tiles) that are available at any given time is limited by the memory.
- The size of individual tiles affects the download time of the tiles.
- Overhead from files (i.e. image file headers and storage fragmentation) may

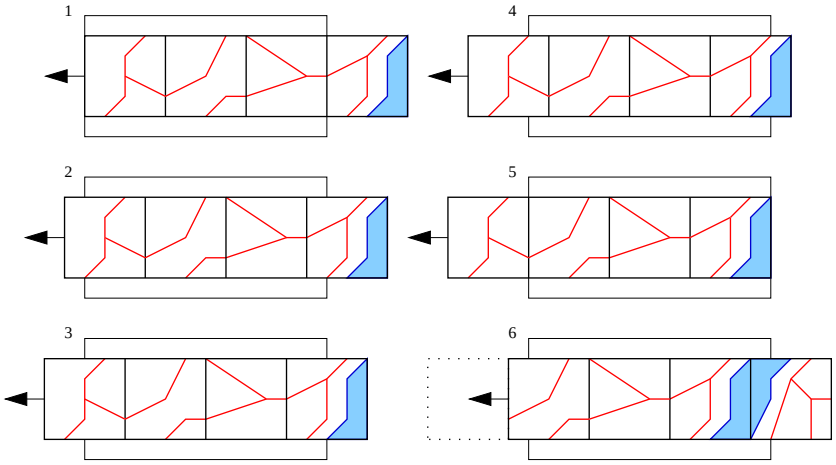


Figure 7.4: Tiled animation. When tiles move outside the visible area, they are not drawn.

waste precious space.

The map display is designed to make a compromise between these limitations. The design has the following features:

- Each tile is 256×256 pixels
- Only four tiles in memory
- Size of the map in the tile is 2000×2000 meters

The size of 256×256 pixels is chosen, as it is slightly bigger than the screen, meaning that when a user moves around, it is only necessary to load a few images. This means that fewer requests to the KMS map server will free up bandwidth for other online purposes, i.e. downloading advertisements and PoI index. A test of sizes of JPEG images shows that smaller images tend to use a large amount of data for the file structure. The test involved a square single-color image, saved as JPEG with the worst quality setting. As seen in Figure 7.5 and Table 7.1, images of size 16×16 pixels and below are equal in byte size. A possible explanation lies in the JPEG compression, which is further explained in [JPE03]. This means that small images have too little actual image information compared to their size, although images with actual detail may have a better ratio.
The image size of set to 256x256. This gives a file overhead of 670 bytes, which is acceptable.

The size of the map in each tile is 2000x2000 meters. This gives a clear, uncluttered map, which is informative without being confusing. This leads to an implementation of the map display as described in the following.

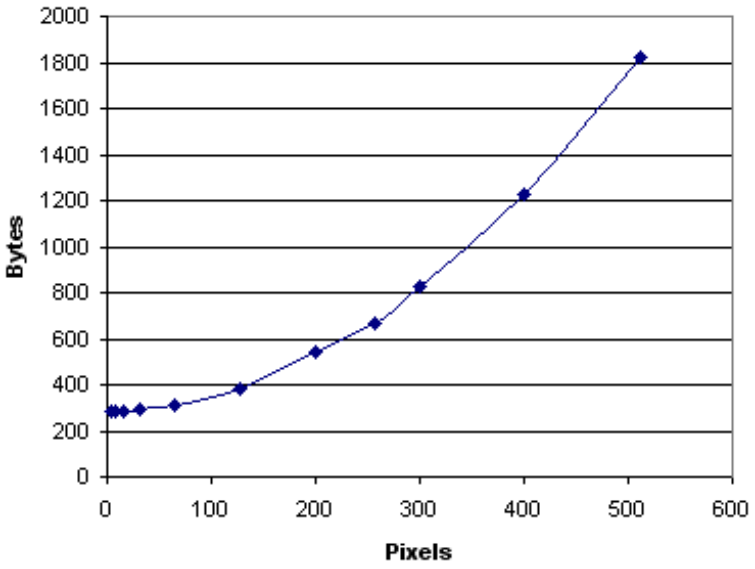


Figure 7.5: Graph of JPEG image size vs. byte size. Images are square.

| **width height** | 4 | 8 | 16 | 32 | 64 | 128 | 200 | 256 | 300 | 400 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **bytes** | 286 | 286 | 286 | 292 | 310 | 382 | 539 | 670 | 827 | 1223 | 1822 |

Table 7.1: JPEG image size vs. byte size

### 7.6.2 Map Implementation

Each tile is 2000m×2000m, with 256 pixels in each direction, giving a resolution of 7.8125 m/pixel. The map display uses four tiles, which are displayed as shown in Figure 7.6. At all times, the tile which the user is located in, is designated as the center tile. Depending on which square of the center tile, the user is located in, the surrounding tiles are designated *Vertical*, *Horizontal* and *Diagonal*. If the user is located in the lower left square of the center tile, the surrounding tiles to the user's left, down and lower left are loaded (picture 1 and 2 of Figure 7.6). If the user is located in the upper right square of the center tile, the surrounding tiles to the right, up and upper left are loaded (picture 3 of Figure 7.6). The UTM coordinates of this tile is used to place the tiles in relation to the center point on the screen.

To calculate the screen coordinates ($X_{pixel}$, $Y_{pixel}$) to draw the image on, the current GPS coordinates of the user is used. Since each tile is 2000m in each direction, the tiles can be seen as being squares in a grid, beginning at (0,0), in order to calculate the corner coordinates of the tile which the user is currently on, the UTM coordinates are rounded down to the nearest 2000m. This gives the corner coordinates for the bot-

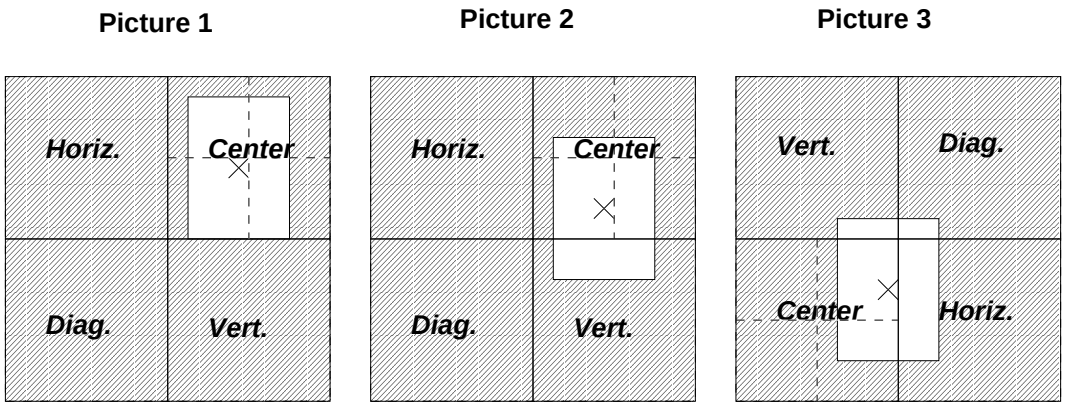


Figure 7.6: The implementation with the four tiles. The white area is the viewable screen area.

tom left corner of the tile[1]. To calculate the pixel offset from the center point of the screen, the difference between the user coordinates ($X_{user}, Y_{user}$) and the tile coordinates ($X_{tile}, Y_{tile}$) is divided by the map resolution $Res_{map}$ (7.8125m/pixel). This is then subtracted from the center pixel point ($Cp_{\mathrm{x}}, Cp_{\mathrm{y}}$), given in these formulas:

$$X_{pixel} = Cp_x - \frac{X_{user} - X_{tile}}{Res_{map}} \tag{7.1}$$

$$Y_{pixel} = Cp_y - Height_{Image} - \frac{Y_{user} - Y_{tile}}{Res_{map}} \tag{7.2}$$

The $Height_{Image}$ is the height of the image, and is subtracted since the UTM coordinates start in the bottom left and the terminals display starts in the top left corner of the screen.

As an example, the position of Fredrik Bajers Vej 7E (FVB 7E) is approximately (560013,6319413). By first rounding down to the nearest 2000m this becomes (560000,6318000). This is the corner of the tile that FBV 7E is on. The map piece for this is shown in Figure 7.7. To determine on which pixel position the map should be drawn, the numbers are inserted into Equations 7.1 and 7.2. The screen size is 176×208 pixels, so the center point is (88,104). This gives the following calculation:

$$X_{pixel} = 88 - \frac{560013 - 560000}{7.8125} = 86 \text{ pixels.} \tag{7.3}$$

$$Y_{pixel} = 104 - 256 - \frac{6319413 - 6318000}{7.8125} = 28 \text{ pixels.} \tag{7.4}$$

The map is then drawn at the position (86,28) on the screen.

[1]Only in the Eastern part of the Northern hemisphere.


Figure 7.7: The map piece for the position of Fredrik Bajers Vej 7E (shown by the arrow).

The three other tiles are designated as *Vertical*, *Horizontal* and *Diagonal*. Depending on which quarter of the center tile the user is on, the three secondary tiles are filled with the appropriate map pieces, as shown in Figure 7.6. Since the tiles are 2000m in each direction, this value is added to or subtracted from the center tile UTM coordinates to give the UTM coordinates of the horizontal, vertical and diagonal tiles. Table 7.2 shows the rule for determining the coordinates for the secondary tiles. In the

| **Quarter** | **Horizontal** | **Vertical** | **Diagonal** |
|---|---|---|---|
| **Upper Left** | (-2000,0) | (0,2000) | (-2000,2000) |
| **Upper Right** | (2000,0) | (0,2000) | (2000,2000) |
| **Lower Left** | (-2000,0) | (0,-2000) | (-2000,-2000) |
| **Lower Right** | (2000,0) | (0,-2000) | (2000,-2000) |

Table 7.2: The (x,y) values to add to the Center Tile corner coordinates.

example above, the user's position is in the upper left corner of the map. This gives the corner coordinates for the other three secondary map images as shown in Table 7.3

The map pieces retrieved from KMS are stored in local storage. This enables the client to cache the images. The client does this by trying to first open the image from local storage, if this is not successful, the image is requested from KMS for later use.

| Image | UTM X | UTM Y |
|---|---|---|
| **Horizontal** | 558000 | 6318000 |
| **Vertical** | 560000 | 6320000 |
| **Diagonal** | 558000 | 6320000 |

Table 7.3: The positions of the three secondary maps.

To make this caching possible, each of the four tiles has associated corner coordinates in UTM. This corner coordinates are used to keep track of which image is loaded into each tile. Each image file in local storage is named as follows:

```
map_XXXXXX_YYYYYYY.jpg
```

Where the `XXXXXX` is the `x` coordinate and `YYYYYYY` is the `y` coordinate. The four images in the example have the filenames:

```
map_558000_6318000.jpg
map_558000_6320000.jpg
map_560000_6318000.jpg
map_560000_6320000.jpg
```

If any of the four tiles are to be updated, e.g. the user has moved off the center tile, the coordinates is updated and a new image is loaded into the tile.

The tile approach allows for a fluid display of maps, as opposed to showing a new map when the user moves off the screen.

## 7.7 Sequence Diagrams

The sequence diagram for updating PoIs is seen in Figure 7.8. The *Core* class is thread based, with the *GpsConsumer* as the controller. When a user starts the system, the *GpsConsumer* starts receiving GPS data. As soon as a usable coordinates become available, the *GpsConsumer* calls the `Run()` method of the *Core* class. The *Core* class then calls the *Update* class which checks whether an update is necessary.

Figure 7.8 shows the sequence if an update is needed. When this is the case, the *Update* class calls the `Get()` method in the *DataConsumer* to get a new updated PoI index. When finished, the *DataConsumer* calls the `NotifyFinished()`method in the *Core* class, which then calls the `extractPoIs()` method in the *FileHandler*. This method iterates through the received file, inserting each PoI into the local PoI database. This is done by the *DBHandler* class with the `OpenDatabase()`,

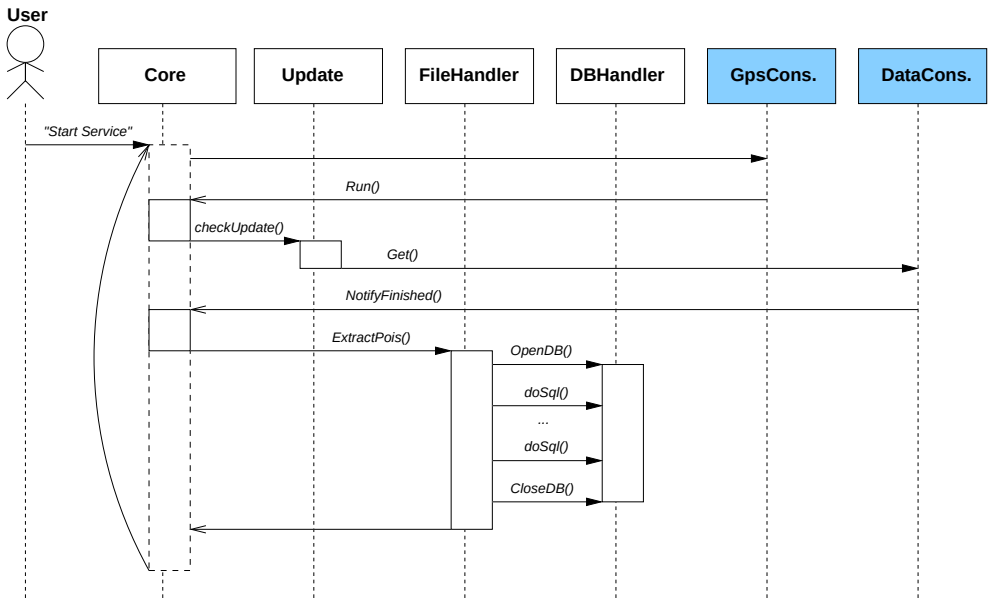


Figure 7.8: The Sequence Diagram of the client, when an update is needed.

`doSql()` and `CloseDatabase()` methods.

After this, the client is put in a waiting position until the *GpsConsumer* presents another set of coordinates to the *Core* via the `Run()` method. The sequence diagram in Figure 7.9 shows what happens when an update is not required. The difference between updating and not updating is that the Update class, instead of receiving a file from the Internet, uses the method `checkPoI()` to iterate through the local list of PoIs to find the nearest PoI, which is then presented to the user.

For advertisements and PoI info, the sequence diagram is presented in Figure 7.10. The *DataConsumer* is called to get the information, and when received, the *Core* class uses the *FileHandler* to display the information in dialog boxes.

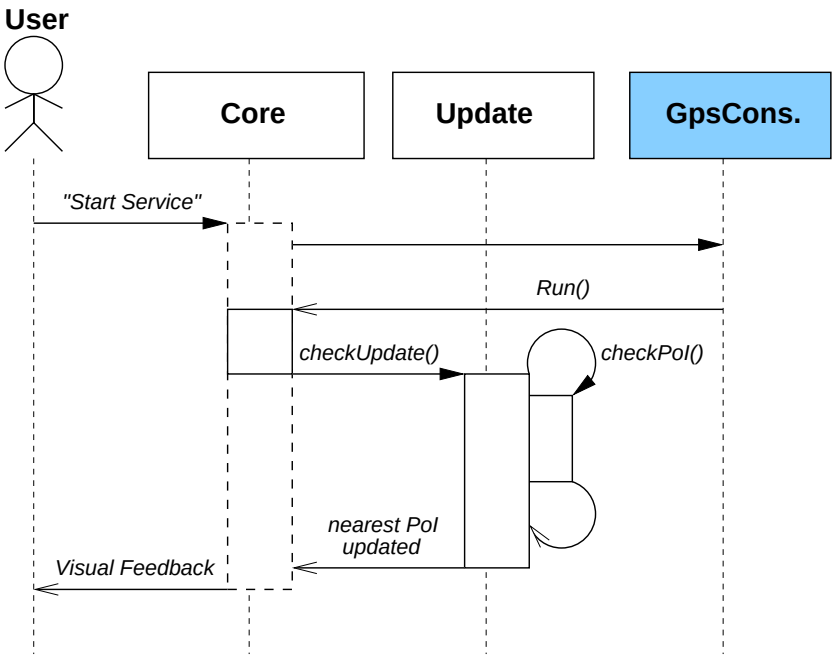


Figure 7.9: The Sequence Diagram of the client, when no update is needed.

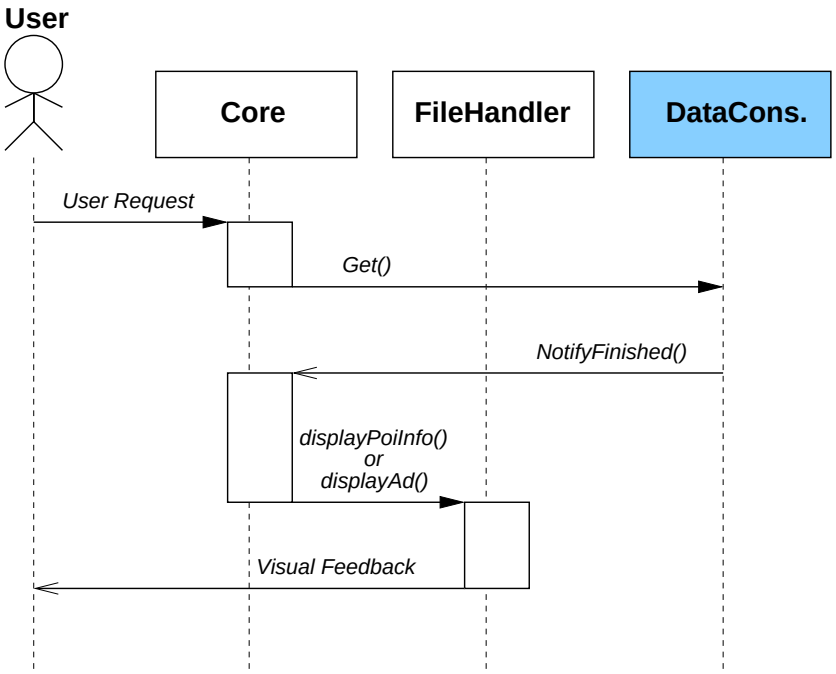


Figure 7.10: The Sequence Diagram for advertisements and PoI info.

# Chapter 8

# Scenario

This chapter exemplifies the use of GPSOne, through use cases. The use cases are examples of how a user would interact with the system. Each example will consist of a task the user wants to perform. With every choice the user takes, the reactions from the client and server are described.

The user is a person called Alex[1]. Alex is traveling to Aalborg for the first time, and owns a Series 60 mobile phone. Alex has installed GPSOne on the phone and has bought or rented a Bluetooth GPS.

## 8.1 Scenario: Setting Up

Alex needs to inform the system of Alex' preferences. This requires Alex to log onto the Web site that accompanies the software and create a new user account. Figure 8.1 shows this.
After Alex is created as a new user, with the username "alex", Alex logs into the system and must now make a profile. This is displayed in Figure 8.2.

The user can now add and remove categories of PoIs to the profile. The web page for this is displayed in Figure 8.3.

Alex chooses these categories for the profile:

- Amusement parks
- Folk Festivals
- Supermarkets
- Buildings

[1] Short for Alexandra or Alexander

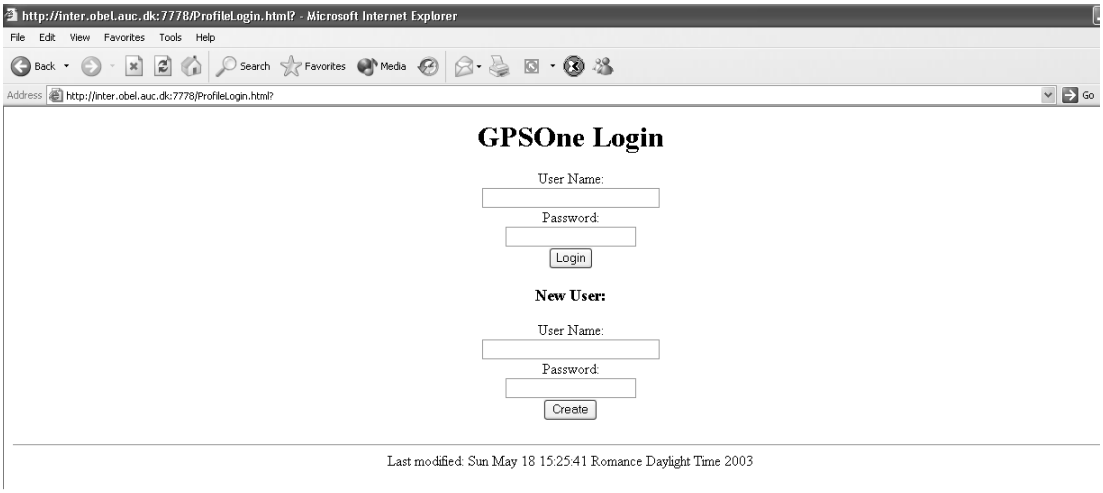


Figure 8.1: The logon Web page of the Online Aalborg guide.

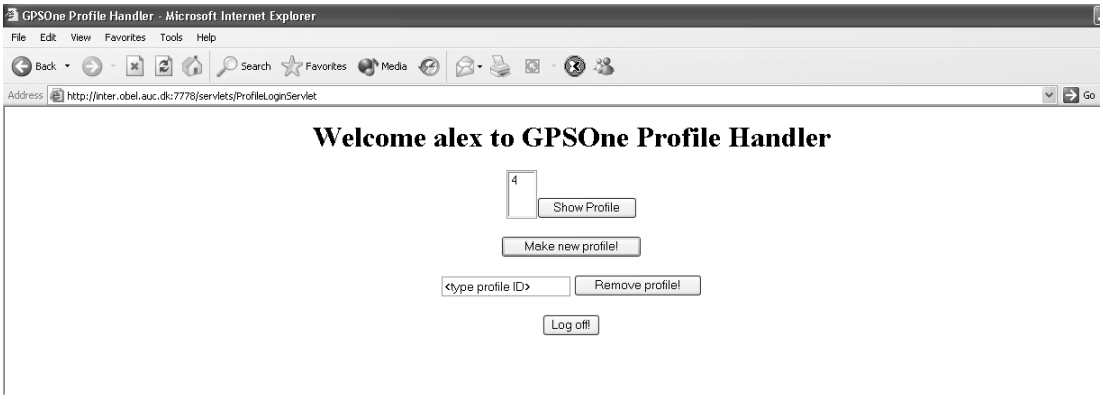


Figure 8.2: The user creates a profile.

By adding PoI categories to the system, the *ProfileArea* table is updated with the categories. This allows the system to use the categories in order to find information relevant for the user.

## 8.2 Scenario: Starting Service

After Alex has set up a profile, GPSOne is now ready to be used. On the phone Alex starts up the application and is prompted to “select start on the menu”. This brings up a box where Alex can type in the username associated with the profile. Alex types in

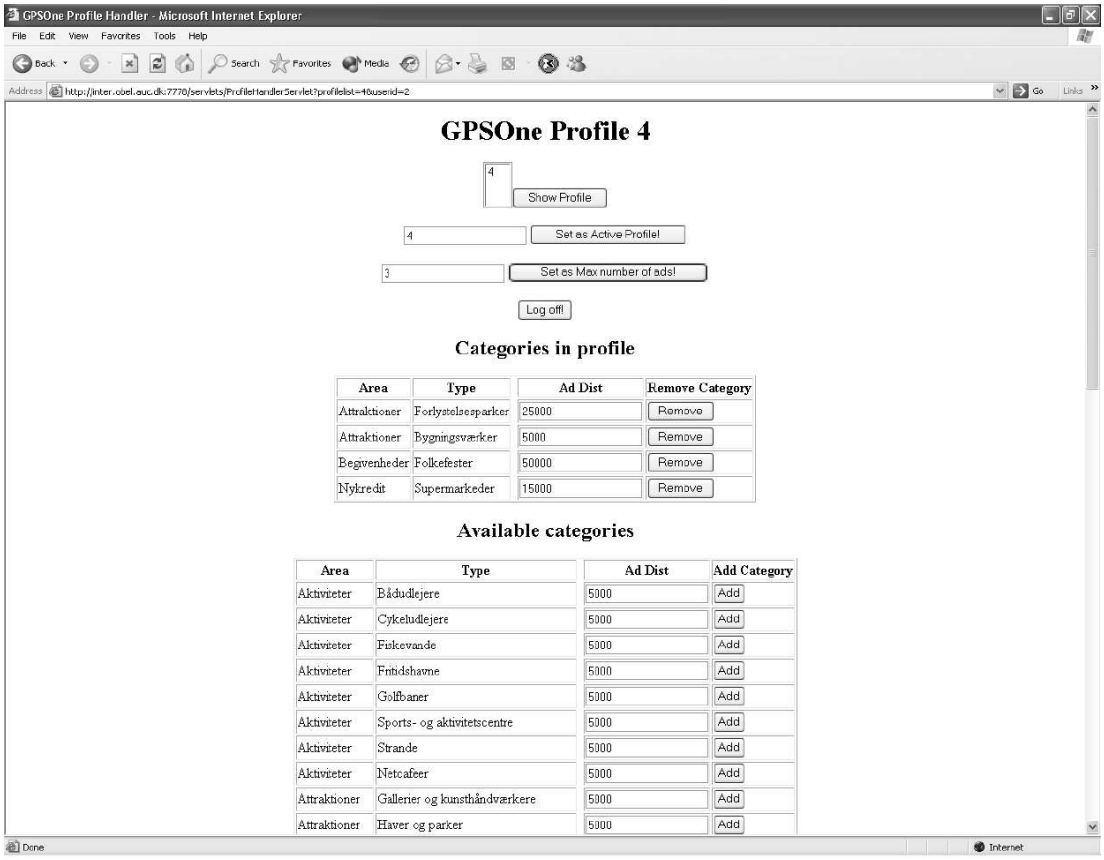


Figure 8.3: The setup of GPSOne profile.

"alex" and presses OK. This starts up the GPS receiver and the application begins to check for nearest PoIs. This is displayed in Figure 8.4

Since it is the first time Alex uses the system, the data and maps are downloaded from the Internet. After a short while, the area around Alex shows up on the screen and a text box with info on the nearest PoI is shown (Figure 8.5). Alex goes for a walk. This makes the client application look for new nearest PoIs.

Suddenly, as Alex is walking around in the center of Aalborg, "Jens Bang's House" is displayed on the screen. Alex has chosen "Buildings" in the profile, so Alex wants more information. By choosing "More info" from the menu, Alex gets information on the PoI. In the case of "Jens Bang's House", the information is:

**Name:** Jens Bang's House

**Address:** Østerågade 9


Figure 8.4: The system asks for user name.


Figure 8.5: GPSOne running.

**Postnr:** 9000

**City:** Aalborg

**More Info...** Choose to view

Figure 8.6 shows the information as it is presented in the system. The information was retrieved from the server by calling the *FurtherInfo* servlet on the server. The servlet takes a PoI ID number as parameter and returns the information from the database. The information is shown as a menu where the user can scroll through the information at will. The "`More Info...`" menu item allows the user to see a description of the PoI, as shown in Figure 8.7. The description is sent along with the information. Figure 8.6 shows the information as it is presented in the system.


Figure 8.6: The information is shown in list form.

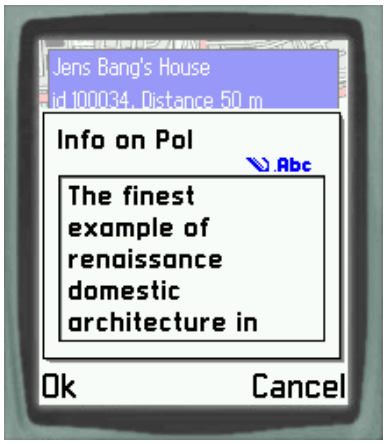


Figure 8.7: The further information is shown.

## 8.3 Scenario: Changing Profiles

Alex is meeting a relative in Aalborg, and wants to talk over dinner. Alex needs to find a restaurant that is nearby. Alex switches the profile from the one being used to a new profile featuring restaurants. When the profile is changed, the nearest restaurant pops up on the mobile phone, and an indicator of the direction is shown .

## 8.4 Scenario: Advertisements

The restaurant “Perlen” has just placed an advertisement in the system, which is valid from 12-14 the current day, and has chosen to advertise within a distance of 500 meters. Alex wanders around in Aalborg and comes within 500 meters of the restaurant at 12

o'clock. Alex receives the ad "Special lunch deal. 50 kr,-. Today ONLY!" on the phone (see Figure 8.8). Alex decides to try out the restaurant.


Figure 8.8: Advertisement in GPSOne.

# Chapter 9

# Evaluation

The GPSOne application is a push-based LBS prototype based on the *Combined Approach*. In order to minimize the continuous updates to the LBS Application Server, the *Update Threshold Approach* and the *Update Algorithm*, mentioned in Section 3.3, are implemented. This means that the number of updates from the client application to the server is reduced considerately. The update performance depends on the user's preferences, the number of PoIs that are nearby the user, and the maximum number of PoIs allowed in the Index. Compared to sending an update of the user's position each second, the implemented approaches reduces the number of updates considerately, since a user most likely will be able to travel within a range of several kilometers, before an update is necessary.

The *Update Algorithm* ensures that GPSOne monitors the nearest PoI available within the user's desired distance. However, the *Update Algorithm* is not able to monitor more than one PoI. In order to monitor more than one PoI the algorithm has to sort the PoI index, after the distance between the user and the PoIs has been calculated. Generic sorting algorithms described in [CLRS01] can be used to sort the PoI index. Since the sorting algorithm has to be applied each time the GPS coordinates are extracted, the sorting routine may decrease the performance. This depends on the size of the PoI index.

The implementation of the *Raster Map Caching Approach* reduces the number of requests to the KMS raster map server. The downloaded raster maps are stored in the mobile phone and reused if possible. This leads to a significant reduction of network traffic in GPSOne. However, the LRU caching algorithm of the *Raster Map Caching Approach* has not been implemented in GPSOne. This can result in filled storage, if GPSOne is used over a long period of time. When GPSOne is used for the first time, a number of raster maps have to be downloaded before it is possible to draw a map on the screen. When the first maps have been downloaded, the *Raster Map Caching Approach* allows the maps to be displayed faster to the user.

The download time of the maps might affect the performance. The average download time for a raster map, from KMS using a GPRS connection on a Nokia 7650, is about 5 seconds. GPSOne downloads at most four raster maps at a time, hence at least 20 seconds are required when maps are downloaded. Assume the user's position is mapped to one of the downloaded raster map illustrated in the left picture of Figure 9.1. In order to issue a map request, the user has to move halfway up the image size. Since the image size corresponds to 2000 meters, the user needs to move at least 1000 meters. If the user moves more than 1000 meters within 20 seconds, the drawing of the maps will not be able to keep up with the movement of the user. This means that the user has to move with a speed of more than 180 km/h for this to occur. However, the user might move around the center of the four downloaded raster maps as illustrated in the right picture of Figure 9.1. This situation triggers map requests frequently enough to clutter the map drawing, since 16 maps must be downloaded. The network stability also affects the map drawing performance.

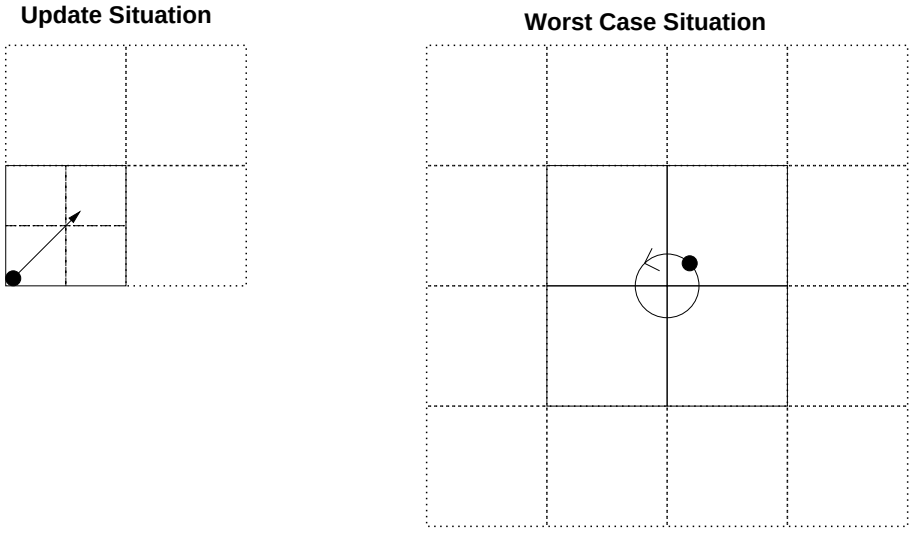


Figure 9.1: The right picture illustrates the worst-case situation, which will trigger 16 map requests. The left picture illustrates, that the user needs to move at least half of the image size, in order to issue a map request.

The GPSOne application depends on the network access to the LBS Application Server and the KMS raster map server. However, if the user stays permanently in an area for a longer period of time, the PoI index will still be valid and the updates to the LBS Application is reduced. Without the information of the user's interests in the user profile, it is not possible to reduce the updates as much as achieved.

The profiles are subscribed to at a Web site located at the LBS Application Server. If the user could create and modify the profiles using the GPSOne application, it would be more flexible and convenient for the user.

In order to use GPSOne, the Bluetooth GPS receiver has to establish satellite connection. In best case, the satellite connection can be achieved within a few seconds, and in the worst case, it can take several minutes. The satellite connection depends on the surroundings and the weather [DB03]. Once the satellite connection is established, the Bluetooth GPS receiver can recalculate the location, even if the connection is temporarily broken. However, a Bluetooth GPS receiver is fairly expensive, since the Emtac GPS is the only Bluetooth GPS on the market at current time. Other Bluetooth GPS receivers will soon enter the market [NAV03], and will most likely make Bluetooth GPS receivers cheaper.

Extension of the LBS framework has enabled the framework to support both push and pull-based services. The use of Java servlets enables the possibility to support many users concurrently. Furthermore, the component-based design provides easy component extension.

A few organizations have defined standards regarding mobile services. The W3C has defined a Point Of Interest eXchange Language Specification (POIX) [W3C03]. The POIX specification defines how to specify PoIs using the XML format. The PoI data format described in Section 4.1 does not follow the POIX convention. The reason is, that the POIX specification defines geo-referenced objects, such as lines and arcs, which are not relevant in this project. Furthermore, the POIX has a comprehensive PoI location definition. In the POIX specification, a PoI can have more than one position related. A PoI can have a position for the entrance denoted as the *Terminal Point*, a position for the actual mapping in a road database *Introductory Point*, and a series of points to specify the route between the two points. This PoI specification is too comprehensive to use in this project, since the data available on this project only includes one geographical point for each PoI. However, the PoI index has the options to be sent to the client in XML format. Modifications can adapt the PoI data to the POIX specification.

Another consortium, PARLAY Group, defines specifications and API definitions for developing mobile services [PAR03]. The LBS framework does not follow the PARLAY convention. The LBS framework's component based architecture and the use of Java technology enables the framework the option to adapt such specification if necessary in the future. In relation to geographic standards, a Geography Markup Language (GML) has been defined by the openGIS consortium [Ope03a]. GML is also based on XML specifications, and can be used to define geographic objects, surfaces, lines and points. The GML specification is an alternative to POIX. These specifications can be used in order to adapt the PoI data, used in the project, and thereby making it possible integrate with other LBS systems. The next chapter concludes the project.

# Chapter 10

# Conclusion

Pervasive computing has received great attention in recent years. A type of pervasive computing is highly personalized services such as LBSs. The research within the field of LBSs is important, and could bring future users closer to a pervasive computing environment.

The main purpose of the project is to extend the already developed LBS framework to support push-based features, and use this to develop a prototypic LBS system, the Online Aalborg Guide. As a part of the Online Aalborg Guide, a client application GPSOne, for Symbian OS mobile phones has been developed. The push-based LBS framework, which is an extension of the LBS framework, has been implemented and organized in various layers and components, such that modularity and transparency are provided for developers.

One of the design criteria of the Online Aalborg Guide is to minimize the number of position updates from the client, and thus reducing the computational load of the server, and minimize the network transmission between the client and the server. This will also reduce the cost for users to access the service. In order to meet this design criterion, the architecture is based on a combined approach. The combined approach distributes the logic between the client and the server, and part of the data is stored at the client. The Online Aalborg Guide is implemented such that the client application monitors the nearest PoI and displays this. The nearest PoI is found in a PoI index stored at the client. The PoI index is formed based on the user's position and interests. This means, that the PoI index only needs to be updated when the user changes interests or moves outside a PoI index range where the nearest PoI cannot be found.

Storage is another design issue that the Online Aalborg Guide must handle. Most client terminals have limited storage capacity, including mobile phones. This must be taken into consideration when designing a LBS such as the Online Aalborg Guide. Hence the PoI index only contains the most basic information about a PoI. If the user wishes further information about a PoI, this information is retrieved from the server. The PoI

index for the Nokia 7650 includes only a limited number of PoIs, based on the user's position and interests. This means that the storage space in the client terminal can be used for other purposes, such as storing images of maps. The client application caches images of maps used, so maps don't have to be fetched from the Internet every time. This reduces the network usage and increases the performance of the client application.

The Online Aalborg Guide includes push-based advertisements from PoIs, that are nearby the user. The advertisements are based on the user's position, interests and the system time. This means that commercial PoIs, such as restaurants or amusement parks, can specify a time interval in which their advertisement is sent to users. However, in order to target the advertisements, in the interest of both the PoI and the user, the user must, for instance, have restaurants or amusements parks as interests. Furthermore, the PoI must be within the user's specified advertisement range, and the user must be within the PoI's specified advertisement range. This means that the Online Aalborg Guide and the push-based LBS framework is able to deliver information based on time and location.

The realization of the Online Aalborg Guide demonstrates the applicability of the push-based LBS framework, and highlights some of the issues involved when construction LBSs.

# Chapter 11

# Future Work

This chapter discusses future work on the Online Aalborg Guide, the GPSOne application, and the LBS Framework. Initially, the services that are left out of this project could be implemented. These are; the *Buddy Finder*, *Favorites*, *User PoIs*, *Reviews*, *Pictures* and *Route-Planner*.

The Favorites can be implemented mainly at the server, as a service that adds the PoI to the users favorites. Another solution would be to maintain the favorites on the client terminal, this would lower the load on the server, but the user would not be able to easily transfer favorites between different devices.

Reviews and Pictures could supplement the user PoIs, as well as the already present PoIs. The users would be able to submit pictures and reviews of PoIs, thus adding content to the system. Reviews, Pictures and User PoIs demand a server that can store potentially large amounts of data. Depending on whether the user wants to publish the user PoIs, reviews, favorites and pictures, the service can be distributed between the client and server.

The Buddy Finder requires implementation of user tracking on the server, such that the server can maintain a model of all users positions and movements. To maintain a model that is detailed enough to let users find each other with reasonable accuracy, a large number of updates are needed. The current update method only updates the user's position in the server, when a PoI index or further information about a PoI is requested. To maintain the advantages of a low update rate, the server will need to be able to predict the users' positions. This requires that prediction algorithms are added to the server, and that they can affect the update policy of the client.

As an additional technical feature, the client could be extended to support other LDTs than GPS, for example by using cell-based positioning, or by allowing the user to enter an address. The application currently uses a dynamic library, or DLL, to provide GPS access. Cell-based positioning should be implemented in the same manner. This

would enable the user to choose the relevant LDT at will. The cell-based LDT only provides precision that is good enough for services such as weather forecasts, but not precise enough for guiding a user between two addresses. Address-based positioning would require the user to enter a nearby address, which would then be translated into a set of coordinates. This would require the server to have a set of road data with addresses mapped onto. The address-based LDT does not support push services, as the user takes an active role in the LDT.

Even though the two mentioned LDTs are not suitable for precise push-based services, they can be used when a GPS receiver is not available. Since they rely solely on the mobile phone, cell-based positioning uses the radio in the phone, which is available at all times. Using other LDTs than GPS also allows users with mobile devices that cannot be connected with a GPS to use the services, only at a degraded level.

Another technical improvement would be to implement the services using data exchange standards. The current implementation only uses the WMS standard to retrieve JPEG images from KMS, and textual data is transferred in a proprietary format. The services running on the server need only small adjustments to use XML for data transfers, whereas the client application will need to support XML. This requires an XML parser to be implemented for the client platform.

If an XML parser is present, it is possible to implement SVG (Scalable Vector Graphics), which is based on XML. The SVG format can be used to transfer and display maps and geographic information in a vector based image format. The advantage of using SVG instead of JPEG images is, that the vector format takes up less space than raster images. Additionally, vector-based graphics can be scaled without loosing detail. This will allow for smooth zoom functionality in the application. Furthermore, the image cache could be replaced by a vector database, where maps are stored as shapes.

To realize maps in this way will require a vector-based representation of the maps. Likewise, if only geographic features are transferred in SVG, it would be possible to overlay the SVG graphic onto the existing raster maps, a functionality that could be used for showing routes on the map. These kinds of vector graphics would then need to be generated by the server, which in [ACKN03] is nearly implemented.

At the client side, additional services can be implemented. These services may even become distributed, such that clients may exchange data between each other. Such a distributed service could appeal to tourist groups. Tourists could exchange pictures and information about sights, favorites and reviews. This would give an extra dimension to sightseeing.

Another application is to implement warning applications, such as those described in Section 1.5. Senior citizens could either be equipped with similar phones, or custom

devices. The warning applications would be similar to the buddy finder, where concerned parents or relatives could locate their children or elders on their own devices, or from a computer.

# Bibliography


[ACKN03] Kristian V.B. Andersen, Michael Cheng, and Rasmus Klitgaard-Nielsen. Framework for building location based services. Technical report, Aalborg Universitet, 2003.

[Blu03] The Official Bluetooth Wireless Info Site. http://www.bluetooth.com, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[Dan03] Danish Tourist Board. Visit denmark - the official guide to denmark. http://www.visitdenmark.com, 2003.

[DB03] Alastair Davies and Jenni Barclay. Silva gps and electronic compass. http://www.gly.bris.ac.uk/WWW/TerraNova/silvagps/gps.html, 2003.

[DR02] Goran M. Djuknic and Robert E. Richton. Geo-location and assisted gps. http://www.lucent.com/livelink/090094038000e51f_White_paper.pdf, 2002.

[Fly02] Peter Flynn. Xml faq. http://www.ucc.ie/xml/faq.xml, 2002.

[Gag03] Jevgenij Gagach. Copilot documentation. http://www.cs.auc.dk/~jevgenij/copilot/docs, 2003.

[Gil02] Chuck Gilbert. How is the accuracy of a gps receiver described. http://www.romdas.com/technical/gps/gps-acc.htm, 2002.

[GPS03] GPSWorld. Gps development timeline. http://www.gpsworld.com/gpsworld/static/staticHtml.jsp?id=7956, 2003.

[Int03] IntelliWhere. Intelliwhere trackforce: for tracking the location of field crews and mobile assets. http://www.intelliwhere.com/PRO/TRA/PRO041.asp, 2003.

[Jan01] Elina Janssen. Gis online - location based services. http://www.geog.uno.edu/~ejanssen/GIS Online.ppt, 2001.

[JPE03] JPEGFAQ. Jpeg image compression faq. http://www.faqs.org/faqs/jpeg-faq/part1/preamble.html, 2003.

[KMS03] Kort og Matrikelstyrelsen. http://www.kms.dk, 2003.

[Lon03] Lonely Planet Publications. Lonely planet citysync. http://www.citysync.com, 2003.

[Mar03] Margareth Evangelista Marmori. Location-based services on the mobile internet. http://www.kommunikationsforum.dk/log/Locationbased.pdf, 2003.

[McI96] Jason McIntosh. Jason's article on tile graphics. http://www-cs-students.stanford.edu/~amitp/Articles/Tiletech.html, 1996.

[Met03] Metoffice. Metoffice weather forecast. http://www.metoffice.com, 2003.

[Min03] Ministeriet for Videnskab Teknologi og Udvikling. Teknologisk fremsyn om pervasive computing (computere i alt). http://www.teknologiskfremsyn.dk/html/ikt_about.html, 2003.

[MMMNS98] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt Orienteret Analyse og Design*. Marko, 2 edition, 1998.

[NAV03] NAVMAN. NAVMAN. http://www.navman.com, 2003.

[NME03] NMEA. The National Marine Electronics Association. http://www.nmea.org, 2003.

[Nok03a] Nokia. Nokia - 7650 phone features. http://www.nokia.com/nokia/0,8764,815,00.html, 2003.

[Nok03b] Nokia. Nokia - nokia on the web. http://www.nokia.com, 2003.

[Nok03c] Nokia. Nokia mposition solution description v1.0. http://www.forum.nokia.com/main/1,,040,00.html?fsrParam=1-3&fileID=2648, 2003.

[Nyk02] Nykredit. Nykredit. http://www.nykredit.dk, 2002.

[Ope03a] Open GIS Consortium Inc. Opengis geography markup language (gml) implementation specification. http://www.opengis.org/techno/documents/02-023r4.pdf, 2003.

[Ope03b] Open GIS Consortium Inc. Opengis web map server interface implementation specification revision 1.0.0. http://www.opengis.org/techno/specs/00-028.pdf, 2003.

[Ora02] Oracle. Oracle spatial user's guide and reference release 9.2. http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/appdev.920/a96630/toc.htm, 2002.

[Ora03] Oracle. Oracle corporation. http://www.oracle.com, 2003.

[PAR03] The PARLAY Group. http://www.parlay.org, 2003.

[Por03] Timo Poropudas. Nokia 7650 puts symbian into drivers seat. http://www.nordicwirelesswatch.com/wireless/story.html?story_id=2347, 2003.

[SKS96] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 3 edition, 1996.

[Sta97] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice-Hall, 3 edition, 1997.

[Sun02] Sun Microsystems. Java 2 enterprise edition features. http://java.sun.com/j2ee/, 2002.

[USG01] USGS. The universal transverse mercator (utm) grid. http://mac.usgs.gov/mac/isb/pubs/factsheets/fs07701.html, 2001.

[Vin02] Vindigo. Vindigo. http://www.vindigo.com, 2002.

[Vis03] VisuAide. A gps system for the blind and visually impaired. http://www.visuaide.com/gpssol.html, 2003.

[W3C03] W3C. Point of interest exchange language specification. http://www.w3.org/TR/poix/, 2003.

[WGS03] WGS84. Wgs 84 - world geodetic system 1984. http://www.wgs84.com, 2003.

[Wik03a] Wikipedia. Global system for mobile communcations. http://www.wikipedia.org/wiki/GSM, 2003.

[Wik03b] Wikipedia. Universal Mobile Telephone System. http://www.wikipedia.org/wiki/UMTS, 2003.

# Appendix A

# Advertisement Query

---

This advertisement query is performed on monday the 16-05-2003 at 22.25. The user is “alex” and the user’s position is (556500,6322700).

```
select c.dist, c.poi_id, c.name, c.description,c.current_profile, c.area_id, c.type_id, c.event_id
from
 (select
    mdsys.sdo_geom.sdo_distance(mdsys.SDO_GEOMETRY(2001, 82343,
    mdsys.SDO_POINT_TYPE(556500.0,6322700.0, NULL), NULL, NULL), poi.utm,0.001) dist,
    poi.utm, poi.poi_id, poi.name, description, users.current_profile,profilearea.area_id,
    profilearea.type_id, event_id
   from
    poi, profile, users, profilearea,poievents
   where (profile.user_id=0 or users.name='alex')                                    10
    and users.user_id=profile.user_id
    and profile.profile_id=users.current_profile
    and profilearea.profile_id=profile.profile_id
    and poi.area_id=profilearea.area_id
    and poi.type_id=profilearea.type_id
    and poi.poi_id=poievents.poi_id Í
    and poievents.type_nr=2
    and active =1
    and start_hours <= 22
    and end_hours >= 22                                                              20
    and start_min <=25
    and end_min >=25
    and weekdays like '%1%'
    and start_dayofmonth <= 16
    and end_dayofmonth >= 16
    and start_month_of_year <= 5
    and end_month_of_year >=5
    and start_period <= to_date('16-5-2003','dd-mm-yyyy')
    and end_period >=to_date('16-5-2003','dd-mm-yyyy')
   order by dist asc) c, profilearea, poievents                                      30
where c.current_profile=profilearea.profile_id
 and c.area_id=profilearea.area_id
```

```
  and c.type_id=profilearea.type_id
  and c.dist <= profilearea.ad_dist
  and poievents.event_id = c.event_id
  and c.dist <= poievents.ad_dist
order by dist asc;
```

---

# Appendix B

# Further Info Queries

---

Further Info query for the Nykredit PoI “Netto” with PoI ID 1545.

```
select name, gadenavn, gadenr, postnr, bynavn, tlf
from poi, nykreditinfo
where nykreditinfo.poi_id=poi.poi_id
and poi.poi_id=1545; -- Nykredit PoI: Netto
```

Further Info query for the Tourist Info PoI “Tivoliland” with PoI ID 100059.

```
select name, road, housenumber, postalcode,city, textUK
from poi, touristinfo
where touristinfo.poi_id=poi.poi_id
and poi.poi_id=100059;    -- Tourist PoI: Tivoliland
```

---

# Appendix C

# Client Method Summary

This Apendix gives an overview of the methods available in each of the Client Classes.
The following methods are available in the *Core* class:

**LoadGpsReceiver()** Loads the GPS component library.

**UnloadGpsReceiver()** Unloads the GPS component library.

**Start()** Initializes the GPS component.

**Run(aGpsHealth)** Called by the GPS component when a position is available. The value of `aGpsHealth` determines if the GPS point is valid or not.

**NotifyFinished(aError)** Called by the **Dataretriever** when it has finished. The variable `aError` contains the status of the initial request, which is either success or failure.

The `Start()` method is called from the GUI when the user selects ***Start Service*** from the menu. The `Load` and `UnloadGpsReceiver` methods are called when constructing or destroying an *Core* object.
The *Update* class has the following methods:

**checkPoI()** Finds the nearest PoI of those loaded into memory. Only PoIs which are within the user-defined threshold are considered candidates.

**checkUpdate()** Checks if the update threshold has been exceeded. If it has, `updateData()` is called to get a new set of PoIs from the server, otherwise `checkPoi()` is called. Is called by the `Run()` method in the *Core* class.

**updateData()** Requests a new set of PoIs by calling `getPoiIndex()` in the *Core* class.

The *FileHandler* class has the following methods:

**extractPoIs()** Opens the file that was downloaded by `getPoIIndex` and inserts the PoIs into the database.

**displayPoIInfo()** Opens and displays the info that was downloaded after the user requests to see more information on a PoI.

**displayAd()** Opens and displays an advertisement file that was downloaded.

The *DBHandler* class has these methods:

**doOpenDatabase()** Opens a connection via the DBMS to the database file.

**doCloseDatabase()** Commits any changes done to the database and closes the DBMS connection.

**doSql()** Executes a given SQL command on the currently open database.

**doMakeDatabase()** Creates a fresh database file store.

**extractData()** Extracts data from the database and puts it into a PoI array in the *Update* class.