Supporting Seamful Development of Positioning Applications through Model Based Translucent Middleware

Jakob Langdal Jensen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Abstract

Positioning technologies are becoming ever more pervasive, and they are used for a growing number of applications in a broad range of fields. We aim to support software developers who create position based applications. More specifically, how support can be provided through the use of specialized middleware, and how that middleware can provide developers with methods for controlling application qualities that are related to the positioning process.

One key challenge is to understand how to support application development in a heterogeneous domain like that of positioning. Recent trends in application design for context aware applications in general are advocating that seams (problem areas caused by technologies interconnecting) can be exploited by end-users *if* they are made available to them. A system allowing this kind of interaction is said to be *seamfully designed* as opposed to the traditional goal of ubiquitous computing where seamlessness is advocated.

Another challenge is to provide middleware designers a set of tools that allow them to build translucent middleware, i.e., middleware where the level of openness can be differentiated. Such middleware should provide application developers with sufficient control over the positioning technologies while maximizing the number of responsibilities that can be delegated to the middleware.

We address these challenges by proposing *model based translucency* as a technique for building middleware that supports run-time inspection, is open to adaptation and extension, and can be used to realize a seamfully designed middleware.

For position based applications, overall application quality often depends on properties that are orthogonal to the core positioning functionality; therefore, quality management tend to cross-cut various abstractions of the positioning middleware used to support application development.

We transfer the concept of *tactics* from the field of software architecture and apply it to specific qualities related to position based applications. We further argue that many of these tactics can be implemented as *policies* that can be enforced by a translucent middleware.

Acknowledgements

I would like to acknowledge everyone who have contributed to the completion of this dissertation. I wish to thank my supervisor, Kaj Grønbæk, for providing guidance, especially during the writing phases of the process. Furthermore, I would like to thank my, now former, boss, Andy Drysdale, and my colleagues at the Alexandra Institute who have made sure that I never felt disconnected or isolated.

In addition, I would also like to acknowledge all of my colleagues working on the "Galileo" project for contributing to a great working environment and a supportive team spirit. I would especially like to thank the people with whom I have written papers: Mikkel Kjærgaard, Torben Godsk, Thomas Toftkjær and Kari Schougaard. A special thank goes to Kari for the countless interesting discussions we have had, on almost every topic imaginable. Furthermore, I would like to thank Torben and Thomas for putting up with sharing an office with me for over three years.

Finally, I wish to thank my wonderful wife, Vivi, for being supportive throughout the entire project

Structure

This dissertation is made up of two main parts. Part I introduces the topic of the dissertation, motivates the research and presents the overall results. Part II contains six papers which support the approach of model based translucency. Two of these papers are unpublished and still in preparation. Besides these papers, the middleware produced during this research project is reported upon in [17] which is not included in Part II.

Within Part I, references to the included papers are on the form "Paper [I,II,III,IV,V,VI]", and a short overview of the papers is listed here:

Paper I Model-based Translucency in Middleware: Supporting Seamful Development *Kari Rye Schougaard and Jakob Langdal Jensen. Proceedings of the 2nd International Workshop on Middleware for Pervasive Mobile and Embedded Computing, 2010 (M-MPAC'10).* This paper presents a middleware construction approach where a domain specific layered meta-model of the base system is used to convey understanding of core middleware behavior to application developers.

Paper II PerPos: A Translucent Positioning Middleware Supporting Adaptation of Internal Positioning Processes. *Jakob Langdal Jensen, Kari Rye Schougaard, Mikkel Baun Kjærgaard and Thomas Toftkjær. Proceedings of Middleware 2010 ACM/IFIP/USENIX 11th International Middleware Conference, Bangalore, India, November 29 - December 3, 2010.* This paper suggests an architecture and an implementation of a middleware for developing position based applications. The middleware uses a reflective meta-model to achieve a level of "translucency", i.e., to allow application developers to look inside the middleware and adapt it according to specific details of the application context.

Paper III EnTracked: Energy-Efficient Robust Position Tracking for Mobile Devices. *Mikkel Baun Kjærgaard, Jakob Langdal Jensen, Torben Godsk and Thomas Toftkjær. Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services, 2009.* This paper proposes and evaluates a power saving strategy for updating

GPS positions transmitted from a mobile device to a server by taking into account a device model representing sensor characteristics.

Paper IV Exposing Position Uncertainty in Middleware. *Jakob Langdal Jensen, Kari Rye Schougaard, Mikkel Baun Kjærgaard and Thomas Toftkjær. Proceedings of the 2nd International Workshop on Middleware for Pervasive Mobile and Embedded Computing, 2010 (M-MPAC'10).* This paper introduces the concept of seamful design for developers as a way to enable application developers to manage "seams" within a middleware.

Paper V Implementing Tactics for Positioning Quality Improvements in Lightweight Extensions to Positioning Middleware. *Jakob Langdal Jensen and Kari Rye Schougaard. Unpublished, in preparation.* This paper presents examples of how model based translucency can be used as a framework for implementing tactics for improving application qualities.

Paper VI Quality trade-offs in position based applications: A survey and taxonomy of tactics for improving positioning related qualities. *Jakob Langdal Jensen and Kari Rye Schougaard. Unpublished, in preparation.* This paper provides a taxonomy of application qualities of particular interrest in postion based applications. In addition, the paper presents a range of tactics for improving these qualities and analyses how the tactics influence each other.

Contents

Part I

Overview

Chapter 1

Introduction

The main area of study in this dissertation is the support of software developers who create position based applications. More specifically, how development of applications that use positioning technologies can be supported through the use of specialized middleware, and how that middleware can provide developers with methods for controlling application qualities that are related to the positioning process.

Positioning technologies are becoming ever more pervasive, and they are used for a growing number of applications in a broad range of fields. As the use of positioning grows the need for reliability and dependability increases. Because positioning ultimately relies on computers and sensors interpreting some phenomena of the physical world, position based applications are often susceptible to local variations in the environment specific to the actual deployment situation. These variations often have a significant influence on the overall quality of applications; therefore, developers need methods that allow them to react to the variations and somehow mitigate the effects they cause.

Traditional middleware for positioning hides some of the complexity of the domain behind high-level abstractions in order to simplify application development. However, some of the elements that are hidden using such an approach can be quite useful in actual application situations. Furthermore, the overall quality of position based applications are often directly linked to specific properties of the underlying technology. Therefore, developers need ways of adjusting the level of abstraction presented by the positioning middleware.

The research presented in this dissertation is addressing the challenge of using middleware to support development of position based applications while allowing for manipulation of application qualities. The goal is to establish design principles and identify middleware properties that allow middleware to aid development of positioning based applications by providing developer support in two distinct ways. First, the heterogenous aspects of positioning technologies should be managed primarily by middleware. Second, positioning

middleware must provide enough openness to support the application developer in exploiting both technical and context details relating to the actual situation in which the application is deployed.

The structure of this chapter is as follows. In Section 1.1 the research is motivated by a presentation of key challenges related to designing a middleware that supports development of position based applications. In Section 1.2 the research contributions of the dissertation are presented. In Section 1.3 and 1.4 the applied research method and the project context is presented. Finally, in Section 1.5 the chapter is concluded with an overview of the remaining chapters.

1.1 The challenge of supporting development of position based applications

The main challenges when generalizing development support for position based applications can, in large, be attributed to two main properties of the domain: the heterogeneity and amount of positioning technologies available, and the inherent difficulty of creating accurate digital representations of dynamic physical phenomena. This combination is manifested by the growing number of ways to determine the position of an entity, all of which may have different operational properties. For instance, properties they may differ on are:

- The representation of location information. One method could model location as coordinates, while other methods use symbolic representations, e.g., meeting room on 2nd floor, or a combination of these.

- Position data quality. Accuracy and precision[1] characteristics varies significantly between various technologies. For instance, standard consumer Global Positioning System (GPS) devices provide an accuracy of approximately 5-10 meters in good conditions and a precision of approximately 95%.

- Effect on overall system quality. Some examples are: positioning methods that require high amounts of power compared to the system as a whole, methods that use communication channels which may influence latency, and methods that disclose information which might be seen as private to the user.

- The availability in a specific environment. For instance, GPS works best in an outdoor environment while WiFi or ultra sonic positioning is more appropriate for indoor use. Furthermore, availability can vary over the

[1] Accuracy refers to the closeness of several position fixes to the true, but unknown position of a target. Precision refers to the closeness of a number of position fixes to their mean values [60]

course of a single application execution, e.g., relying on one technology while outside and switching to another when entering buildings.

These properties make the task of providing high-quality position based applications to end-users both complex and challenging. Using a middleware is a great way to generalize the heterogeneity problems, however, with it comes the risk of reducing expressiveness for the application developer. In the following sections both the benefits and the challenges of using middleware to mediate access to positioning technologies are presented.

Middleware for positioning

One way to manage the technological heterogeneity of the positioning domain is to use a middleware which can provide: a common Application Programming Interface (API), standard protocols, and infrastructure services that support the development of position based applications [60].

Using a positioning middleware is useful for supporting both application developers as well as system developers. Application developers gain simplicity. System developers (people who extend the middleware with new sensors) gets a streamlined way of providing complex functionality.

Generally, positioning middlewares have high-level responsibilities, e.g., navigation and coordinate system transformations; and low-level responsibilities, e.g., querying and reporting positions. Moreover, the traditional goal of a positioning middleware is to enable position transparency, i.e., hiding all aspects of positioning from the application developer to provide a transparent experience when working with heterogeneous technologies [60].

Sensing as a process of refinement

Positions are, generally, inferred from sensor readings obtained by sensors that observe properties of the physical world. The nature of determining a position based on input from sensors can be logically represented as a refinement process where data is processed at an increasing level of abstraction [42].

An illustration of a refinement process is shown in Figure 1.1. The figure shows a typical example of how the readings from a GPS sensor can be processed step-by-step before they reach the application as an abstract representation of a position. First, the communication with the sensor is handled by an adapter or driver. Then, sensor readings are parsed from their native format into a data format which is understood by the middleware. After being parsed, the data is interpreted according to the middleware's spatial understanding, represented in the figure by a location model component. After this step the data is representing a position, and, depending on the purpose of the specific middleware, further processing steps could involve various forms of filtering and high-level reasoning before the position is delivered to the application.

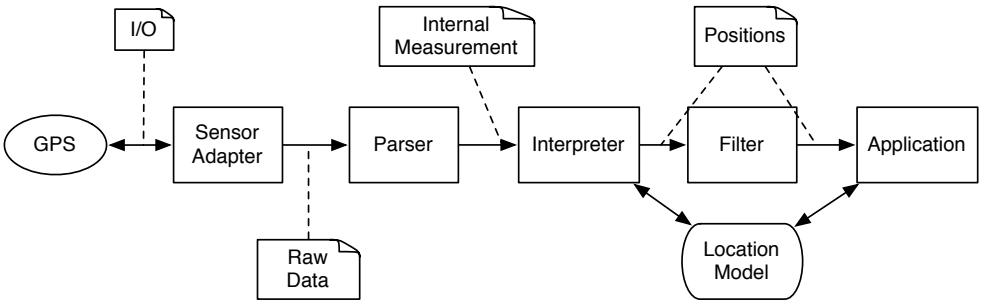


Figure 1.1: Illustration of the processing steps involved when determining a position based on sensor readings.

Layered representation

The sequential view on the positioning process leads to the natural assumption that a middleware for positioning should be implemented using a traditional layered architecture where layer boundaries represent the transition from one processing step to the next to provide transparent access to positioning devices as done, among others, by [42] and [88].

The typical positioning middleware has an architecture as illustrated in Figure 1.2. Access to physical sensors is handled through a hardware abstraction layer and represented as abstract sensors through a clean API. Typically, this API provides developers with transparent access to a position delivery system, i.e., no knowledge of the specific sensor device is needed in order to obtain a position. This is the architecture used in the two dominant device-centric positioning middlewares for consumer devices, the Java Location API [70] found on Android devices and the Core Location Framework [3] used on iOS devices. Furthermore, the architecture is also found in several research middlewares [9, 42, 62, 88]

Cross-layer properties

However, in practice, truly transparent positioning is not necessarily a desirable strategy when developing position based applications. Positioning technologies suffer from a host of inherent imperfections that impact the positioning process. For instance, positioning technologies do not provide pervasive coverage because buildings, humans, and walls might block signals used for positioning. Furthermore, positions delivered by sensors can be erroneous due to signal noise, delays or faulty system calibration etc. Also, positioning systems are of varying quality, accuracy, resolution and representation [11, 18, 52, 56].

In a system that is structured in layers, like previously described, handling these problems will often require actions that cross the layer boundaries. For example, a high-level distance calculation needs to take the accuracy of sensor

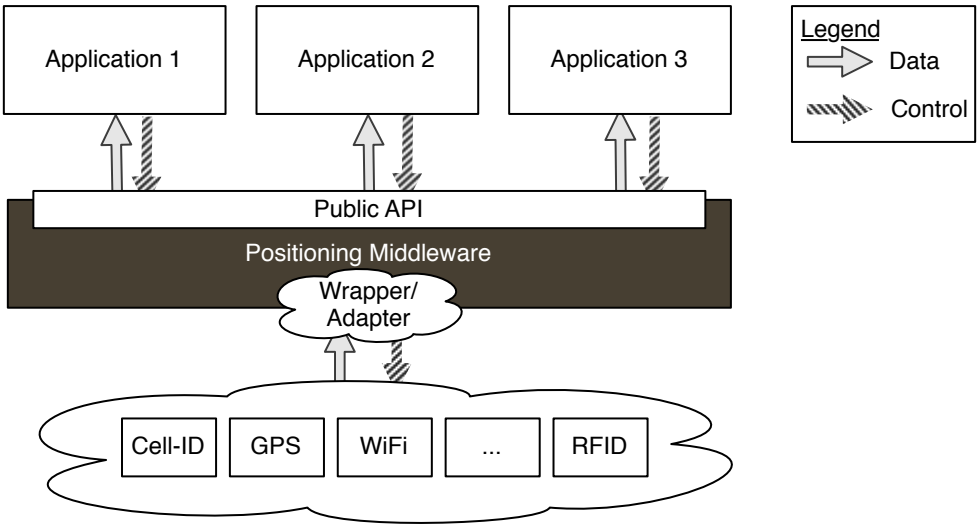


Figure 1.2: The typical architecture of a positioning middleware wraps physical sensors through a hardware abstraction layer and presents an API which provides developers with transparent access to the sensors.

readings into account. This accuracy information is typically only available to layers near the sensors while the distance calculation is placed in a layer with more generic functionality. Therefore, it is sometimes necessary to compromise strict layering in favor of flexibility and transfer low-level information across layer boundaries [38].

A typical solution for bringing information from low-level to high-level layers is to provide abstract representations of sensor qualities that are common to various positioning technologies, e.g., accuracy and precision, and then squeeze as much information into them as possible and ignore that which does not fit [70, 88]. Another solution is to provide high-level layers with generic access to sensors by creating a taxonomy for sensors and reflect that in abstract sensor representations [9, 60, 42]. However, such taxonomies quickly become inadequate, in part, because common abstractions and terms have not yet settled completely in the field of positioning, but, more importantly, because positioning methods and sensors often combine several technologies, thus blurring the relations in the taxonomy. Even with similar technologies there are subtle differences that may have great impact on the resulting quality.

Designing the API

While strongly encouraged, following the principle of *information hiding* [84] does have some negative consequences when designing middleware. In some situations it can lead to a system that hides too much information, in particular, the wrong information. At first, it might seem obvious to support

positioning through middleware services that simply provide application developers with high-level primitives that capture key concepts of the positioning domain, e.g., coordinates, distance, speed etc. But, by introducing these high-level abstraction the details of the encapsulated process also become hidden and inaccessible.

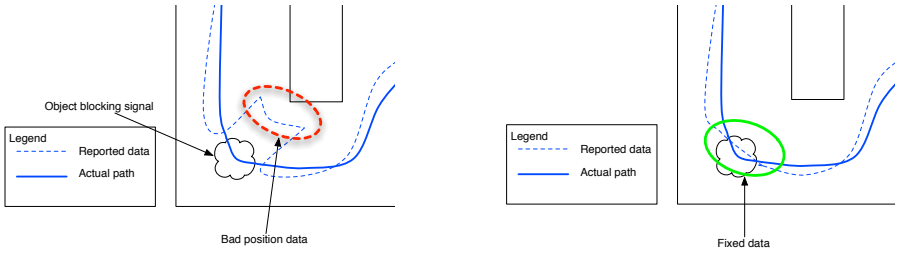
Assume that positioning is provided as a middleware service. That is, the middleware takes care of communicating with the sensors, it interprets the data stream and determines a position. In order to provide a common API to users of the middleware the positions are represented as simple coordinates in a common coordinate system. This is very convenient from a developer perspective as applications can be written in a technology agnostic way. However, in situations where the positioning technology provides added information it could be very useful for developers to be able to exploit this information.

Several positioning technologies provide useful information besides that which can be represented by pure coordinates. For example, GPS sensors collect information about visible satellites, signal strength and similar technology specific details; or radio based technologies, like WiFi positioning, provide information about available access points etc.

This kind of information, although not generic, can be very useful in application specific situations to improve on quality and user experience as the following scenario demonstrates.

> Assume that we are developing an application that uses GPS to track users in order to provide location specific information. After deploying the application we see that in certain areas we seem to trigger the wrong information because the position reported to the application deviates from the actual position of the user. By observing the system, we find that the deviating positions are always off by approximately the same distance as illustrated in Figure 1.3(a).
>
> To mitigate the situation, we choose to use the following simple algorithm: 1) detect the situation automatically, and 2) correct it by shifting the position into place before using it in the application.
>
> GPS devices usually provide an estimate of the accuracy of the current position, and they often provide information about satellite constellations and received signal strengths. This information can be used to detect the suspicious situation; the inaccuracy suggests displaced data, and satellite information indicate in which direction the signal is blocked. Correcting the position only requires us to substitute the current position with one that is shifted in the direction of the obstacle as shown in Figure 1.3(b).

The scenario presented above shows how details of a particular technology directly affect application quality and how it can be used to improve upon the



(a) The positions reported to the application are skewed because of an obstacle in the environment which is blocking the signal.

(b) Using knowledge about the specific environment the reported positions are corrected before reaching the application.

Figure 1.3: Situation where specific information about the environment is used to correct positions reported to the application.

quality of reported positions. This kind of details can also be used to improve on other kinds of qualities. Several application level qualities are directly dependent upon how the details of the technology are handled. For example, some sensors have significant power requirements, others incur latency to the system, and some can be used to change application level behavior to provide better user experiences.

The problem of providing application developers with the right amount of information can seemingly be solved by a properly designed middleware. Therefore, the better the middleware designer is at identifying the right set of details to expose to developers, the greater the chances are that the middleware is applicable for the developer's tasks. However, balancing the set of exposed middleware functionality is a non-trivial task since different applications pose different requirements for domain specific functionalities.

The challenge of finding the right design for interacting with developers is further elaborated with offset in the scenario presented above. There are three apparent ways for developers to realize the solution of the scenario: 1) using a high-level middleware API only; requiring developers to circumvent the middleware to acquire sensor information. 2) adding access to low-level details to the middleware API; potentially leading to information overload. 3) combine a high-level API with the possibility to "open up" the middleware and accessing the low-level details in a middleware supported manner.

High-level API only

Normally, the process of generating a position is hidden from developers or implemented in a generic way. This design strategy will only support the scenarios pre-envisioned by middleware designers.

Concrete applications that utilize positions will likely have at least some requirements that are not fully satisfied by a generic middleware. For many

types of applications the only viable option is to implement a custom positioning management system because successful operation requires access to specific low-level properties that are hidden by the positioning middleware, in effect rendering the middleware useless.

Because of the high cost of supporting all possible combinations of application requirements, it becomes harder to include high-level positioning reasoning into the middleware. As a consequence, the most common tools used by application developers provide only basic hardware abstraction and little or no high-level functionality [3, 70].

Add low-level details to API

Even if the needed information were to be reported alongside high-level abstractions like coordinates etc., the corrective functionality of the scenario would have to be implemented as part of the application, and the developer implementing the correction would need access to the application source code.

Furthermore, the added information might be specific to the technology applied, as with the GPS sensors in the case of the scenario above, and adding information for all kinds of sensors and input sources would clutter the interfaces of the high-level data types. Add to this that it is very hard to anticipate what the exact set of available low-level functionality should be.

Combine high- and low-level details

Generally, providing high-level abstractions while providing detailed control of low-level properties is a challenge when developing middleware, especially for domains that, like positioning, relies on sensing the physical world. To mitigate the problems of either hiding low-level details or risk cluttering APIs the middleware could combine the benefits of both approaches in a way that allows application developers to differentiate the amount of low-level details made available. We claim that this property can be achieved by making the middleware translucent, i.e., letting developers switch between working with the middleware as a "black box" and having access to the internal mechanisms on-demand.

Providing translucency in a middleware poses two key challenges for the middleware developer. First, they must identify which parts of the domain needs to be manipulatable by application developers and which that can safely be encapsulated. Second, they must design an API that allows developers to adjust its provided functionality.

Challenges

In summary, supporting development of position based applications can be achieved by using a middleware approach. The following is a summary of the

key challenges involved when creating middleware for position based application:

- The heterogeneity of positioning technologies and varying characteristics of sensors.
- Low-level technical properties greatly influence application level qualities.
- Determining the functionality to provide to application developers depend on both positioning technologies and application context.

The next section contains an overview of the contributions of this dissertation and how they relate to these challenges.

1.2 Contributions

The primary goal of the research presented in this dissertation is to support developers, especially those who develop position based applications. The overall strategy for achieving such support is to use a middleware that mediates communication between application developers and positioning technologies. In particular, we advocate the use of *translucent* middleware, i.e., middleware which provide a certain degree of openness, or semi-transparency, allowing developers to better adapt the middleware to their specific needs. This presents a number of challenges which are addressed by the contributions outlined here.

Seamful design for developers

One key challenge is to understand how to support application development in a heterogeneous domain like that of positioning. Recent trends in application design for context aware applications in general are advocating that seams (problem areas caused by technologies interconnecting) can be exploited by end-users *if* they are made available to them. A system allowing this kind of interaction is said to be *seamfully designed* as opposed to the traditional goal of ubiquitous computing where seamlessness is advocated. The main proponents for this design approach are Benford et al. [10, 11, 12, 81] and Chalmers et al. [25, 26, 37].

The contributions in this dissertation goes beyond this approach and applies it to design of software that is targeted developers, not end-users. We propose a method called *seamful design for developers* which is a design method that guides developers to expose appropriate parts of the domain. The method is conceptualized and presented in Paper II and Paper IV; and the method is further clarified and elaborated in Chapter 2. The contributions related to the method of seamful design for developers are:

- Guidelines for identification, conceptualization and internalization of *seams* which allow developers to take informed decisions when designing software for domains that, like that of positioning, rely on sensing the physical world (Paper II, Paper IV and Chapter 2).
- Application and analysis of seamful design for developers with respect to the uncertainty inherently related to positioning, resulting in an architecture for uncertainty-aware positioning middleware (Paper IV and Chapter 2).
- A collection of uncertainty management functionality that is relevant to the domain of position based applications (Chapter 2).
- An architecture supporting integrating uncertainty handling into a positioning middleware as a module (Paper IV and Chapter 2).
- Analysis of various API design paradigmes (request, event, and annotation based) with regard to their ability to support flexible openness of middleware across abstraction layers (Paper IV).

Seamful design is a high-level design philosophy that can be realized in numerous ways, however, in general, it indicates a need for some level of openness or translucency, especially in relation to middleware design. It is this aspect which is the focus of this dissertation.

Model based translucency

The challenge is to provide middleware designers a set of tools that allow them to build translucent middleware, i.e., middleware where the level of openness can be differentiated. Such middleware should provide application developers with sufficient control over the positioning technologies while maximizing the number of responsibilities that can be delegated to the middleware. We address this challenge by proposing *model based translucency* as a technique for building middleware that supports run-time inspection and is open to adaptation and extension.

In general, openness and adaptation of middleware can be achieved through various means; two main approaches are reflective middleware and architectural reflection. Reflective middleware is a generic method for dynamically exposing the internal behavior of middleware mediated domains [58], and architectural reflection involves exposing a structural model of a system to enable reasoning about high-level architectural properties [35, 45, 91].

Translucency is sometimes interpreted as providing the same openness as generic reflective middleware, however, we advocate that a more mediated openness with focus on certain aspects of the middleware is easier to understand and use, and that it provides a safer, although potentially less powerful, development model. With model based translucency we propose to combine

the generic openness provided by reflective middleware with suport for structural self-representation as supported by architectural reflection. Furthermore, model based translucency allows middleware designers to combine "best practice" design principles for modularization and abstractions with the power of reflection and introspection.

The fundamental ideas of model based translucency are presented in Paper I while Paper II presents PerPos, a concrete implementation of a middleware for positioning realized using the technique. The following contributions are made based on these papers:

- Model based translucency is proposed as a technique for building middleware that supports run-time inspection and adaptation differentiated according to situation specific needs (Paper I and Chapter 3).

- A concrete translucent middleware, PerPos, that supports an iterative development cycle: run, inspect, adapt (Paper II, Chapter 3 and Chapter 4).

Furthermore, the PerPos middleware has been used to explore how model based translucency supports application developers in managing and improving on application qualities, e.g., power consumption, position accuracy, sensor availability etc.

Using tactics to improve application quality

For position based applications, overall application quality often depends on properties that are orthogonal to the core positioning functionality; therefore, quality management tend to cross-cut various abstractions of the positioning middleware used to support application development. The research presented in this dissertation approaches application qualities for positioning based applications from two angles. Firstly, through the exploration of a single key quality, power consumption; and secondly, through methods for achieving generic support for managing qualities.

The first approach uses power consumption as an example of a quality with a cross-cutting nature; to reduce power consumption of an application many different aspects must be taken into account: the power used by the sensor, communication costs, frequency of updates required by the user etc. In Paper III we present a novel method for optimizing power consumption of mobile devices with the following contributions:

- A device model that can account for the power consumption of a device based on profiling concrete devices (Paper III).

- Position tracking protocols that take changing system conditions into account, e.g., positioning delays and position accuracy (Paper III).

- A method (implemented by dynamic programming) that can minimize power consumption and satisfy robustness by calculating the optimal plan for when to power on and off features of the mobile device such as the GPS module or the UMTS radio (Paper III).

The method optimizes the power consumption of a position tracking system by combining a motion model of user behavior with overall system requirements for position quality. The method has been realized and evaluated in a system called EnTracked. This system was developed prior to the PerPos middleware and has beed used for generating requirements for the translucent properties of the PerPos middleware.

The second approach to application qualities is allowing application developers to manage and improve specific application qualities in a modular way. This is a challenge for middleware designers which is addressed in Paper V, Paper VI and Chapter 4. We transfer the concept of *tactics* from the field of software architecture and apply it to specific qualities related to position based applications resulting in the following contributions:

- A collection of key application qualities that are particular interesting in relation to positioning (Paper V and Paper VI).

- A taxonomy of qualities relating to position based applications. The taxonomy follows the ISO-9126 [46] classification for generic software qualities where possible (Paper VI).

- A catalogue of tactics for improving on qualities of positing based application. These tactics are inspired by the field of software architecture where tactics are fundamental design decisions and proven means to achieve specific qualities (Paper VI).

- Analysis of how seamful design for developers and model based translucency enable middleware support for implementation of tactics and management of qualities (Paper V, Paper VI and Chapter 4).

The tactics collected can be used as recipes to improve the quality of position based applications. We find that many application level qualities are affected by internal parts of the positioning process and they are usually spread across several abstraction levels and the tactics reflect this. In Paper V and in Chapter 4 we investigate how tactics can be modularized using the translucent middleware, PerPos, which is presented in Paper II and Chapter 3. We further argue that many of these tactics can be implemented as *policies* that can be enforced by a translucent middleware.

	Build	Evaluate	Theorize	Justify
Construct				
Model			I,VI	I,VI
Method	I, III, IV, VI	I, III, IV, VI	I	I
Instantiation	II, III, IV, V	II, III, IV, V	II,	II

Table 1.1: Overview of how the papers of Part II fit into the research approach framework.

1.3 Research approach

March and Smith's framework for research in information technology [72] is used as a basis for describing the research approach of this dissertation. In their paper, the terms *information technology research* and *computer science* are used interchangeably. First, the framework is briefly presented, then it is related to the research reported on in the papers of Part II.

According to March and Smith, research in computer science can seen as a combination of the approaches used in natural and design sciences. The distinction between the two being defined as follows: "natural science aims at understanding and explaining phenomena", whereas "design sciences aims at developing ways to achieve human goals". In design sciences, the basic activities are creating/building and evaluating, whereas in natural sciences the basic activities are mainly posing theories and justifying them. Next, these activities are related to the field of computer science.

In computer science, objects subject to investigation are typically artefacts **created** or **built** by humans, as opposed to a field like physics where natural phenomena are studied. The framework suggests that to determine the performance of an artifact it must be **evaluated** by using appropriate metrics. Then, to explain the observed performance **theories** are developed that describe the artifact and its relation to the environment. Finally, the theories must be **justified** through analysis or experimentation, depending on the nature of the artifact. According to the framework, these four research activities – building, evaluating, theorizing and justifying – can be applied to four kinds of research outputs that March and Smith call: constructs, models, methods and instances. Constructs (or concepts) describe a domain, models express relationship between constructs, methods describe processes (e.g., algorithms) and instances are the realization of artifacts.

In Table 1.1 the framework is used to classify the research documented by the papers in Part II. The table shows how the research activities outputs of the papers and the research activities relate to the framework. From the table we see that most of the research effort has been focused on building and evaluation of methods and instances. This indicates an engineering oriented research approach where methods from design science is primarily used. Some of the research involves theorizing and justification, however,

these theories are mainly about artefacts generated within the context of this dissertation. This reflects the original intention that the research of the PhD project should be carried out experimentally bottom-up based on constructive reasoning. This means that the research hypotheses are sought validated primarily through implementing and evaluating prototypes. Being part of a larger research project, the prototypes and experiments are integrated into other activities of the project. Thereby gaining some synergetic effects and valuable feedback from other people actually using the prototypes. In addition, some of the evaluations have been integrated into an assignment for students participating in a project course in pervasive computing.

1.4 Research context

The PhD project is a part of a research project called "A platform for Galileo based pervasive positioning" funded by the Danish National Advanced Technology Foundation [44]. The project is structured as a consortium of the following partners: Terma A/S, Danish Agricultural Advisory Service, Systematic A/S, Alexandra Institute A/S, Aarhus University, and Aalborg University. The overall goal for that project is to produce a toolbox to support the development of applications based on the European satellite positioning system, Galileo, expected to be operational around 2014. One of the products of the project is a software platform that will support the development of professional position based applications, e.g., mission-critical systems where it is important that the end-user can rely on the output of the application.

1.5 Chapter overview

The remainder of this Part I is structured as follows. In Chapter 2, the method of seamful design for developers is presented. Then, in Chapter 3, model based translucency is presented. And, in Chapter 4 model based translucency is evaluated with regard to its ability to support management of qualities related to the positioning domain. Following this, in Chapter 5 the contributions are related to other approaches to designing positioning middleware. Finally, in Chapter 6 we conclude and look at future directions and challenges.

Chapter 2

Seamful design for developers

In this chapter the concept of *seamful design for developers* is presented as a method to allow application developers to exploit details of a system that are normally hidden in an effort to achieve a seamless experience. The chapter is based primarily on Paper IV. The terminology used in the chapter revolves around the concept of *seams* which is used as a means for describing the internal conflicts of a system. The term seam refers to the notion of seamless operation where users interact with a system unaware of any heterogeneity of the technology used for its realization.

In Weiser's seminal paper: The computer for the 21st century [94], seamless design is presented as a goal for how computers should be integrated into the world. A seamless design will allow computers to temporarily disappear from our awareness. This will in turn enable us to focus on the goal for which we use the computer, instead of focusing on the computer itself. In the positioning domain, this means that the concrete positioning systems and their characteristics are hidden for the user who utilize the position information.

Motivated by the imperfection of sensing technologies used in ubiquitous computing, several authors such as Chalmers et al. [26, 25], and Benford et al. [10, 11] have argued for contrasting the goal of *seamless* design with one of *seamful* design. Where seamless design argues that certain technical aspects of a system should be hidden from the user, seamful design advocates that some of these aspects can be greatly beneficial for the overall user experience. Chalmers and Galani define that the goal of seamful design should be to make the seams available in a designed manner, but not in focus at all times, so that

> "one can selectively focus on and reveal [seams] when the task is to understand or even change the infrastructure." [26, p. 251]

In short, a seamfully designed system exposes some characteristic properties of the underlying technology. At first, this approach might seem to

be in conflict with the thought of seamlessness usually pursued in ubiquitous computing systems; however, "seamfulness" is not advocated as the primary way to interact with a system and seamlessness is still a desired quality of a system. Instead, it should be seen as a quality in the event of a breakdown of seamlessness and in certain very application specific situations where the otherwise hidden details are useful.

The seamful design approach proposed by Chalmers et al., and Benford et al. is originally oriented exclusively at end users. We propose to use the same mindset when designing software for *developers*. This has resulted in the design approach presented as *seamful design for developers* in Paper II and Paper IV. In the following sections this design approach is outlined (Section 2.1) and exemplified by analyzing how the uncertainty inherent to positioning can be exposed to application developers through a middleware for developing position based applications (Section 2.2).

2.1 The seamful design process

As argued in Chapter 1, there is a risk of hiding interesting information from application developers when using a straight-forward approach for designing middleware for positioning. For the common use case this is fine, but, when the information is required it should be possible for application developers to access it.

When applied to middleware design, seamful design for developers should results in a middleware that makes the seams of a system available to application developers when needed and otherwise leave them hidden. Based on this overall goal, we define seams to be **the places in a system where two or more non-equivalent parts interconnect**. Examples of seams found in typical positioning systems are:

- Sensor limitations manifested as a mismatch between properties of the physical world and their digital counterparts, e.g., when reported positions deviate from the actual position (accuracy), or when a series of reported positions differ when the actual position is static (precision).
- Information loss caused by generalized data structures, e.g., when translating raw sensor readings into abstract terms of location and coordinates sensor specific details are often discarded.
- Unforeseen side-effects from internal optimizations, e.g., update frequency drops caused by an otherwise well-intended internal power conservation scheme.

Middleware developers who want to apply seamful design for developers first have to identify under which circumstances the seamless design approach of traditional middleware is problematic. Furthermore, they must posses some

knowledge of the seams of the underlying technologies, and they must know how to use information about seams to improve their application domain. This can all be described as an iterative process going through the following three primary activities:

1. Identification of component relations that constitute a seam, and the key aspects which capture that seam.

2. Conceptualization of the aspects found in the previous step.

3. Internalization of the concepts, making them integral parts of the middleware.

In the following sections these three activities are further unfolded in the context of designing a middleware for position based applications. Furthermore, the discussion of the activities are demonstrated through examples based on how the uncertainty related to positioning can be seen as a seam in a positioning system.

Identification

The first activity is all about identification of properties of the application domain that constitute seams. This identification is typically based on experiences working with technologies of the application domain.

For position based applications, it is common for users to experience discrepancies between what the application tells them and what they know to be true based on their own senses, e.g., when a car navigation application insists that the car is driving on the main road even though the driver is certain that he has turned off-road. This particular situation can be caused by numerous sources of errors in the system; it could be the GPS sensor reception quality, it might be the "snap-to-road" algorithm, or outdated map data etc. These error sources are seams in the positioning system and they represent the same aspect of the positioning process, namely uncertainty.

In our work with positioning, we have found that *uncertainty* is an aspect that generally captures a large set of seams in the positioning domain, especially when access to positioning technologies is mediated by a middleware. Many of the processes related to inferring positions based on sensor input exhibit inaccuracies that are manifested as uncertainty in reported position. Properties like sensor characteristics, sensor and communication latency, and conversion from analogue to digital representation all cause users and developers to attribute some level of uncertainty to reported positions.

Identifying seams in any domain is a somewhat subjective task. The seams identified depend largely on the experience of the developers and of the use cases of the target application. However, in general, seams are likely to be caused by analogue phenomena that cannot be fully digitalized and they are

likely to be sound at in the components of the system where the digitalization is occurring.

Conceptualization

When seams have been identified they need to be conceptualized into something that is part of the system interfaces. This is needed in order to allow developers some means for manipulating the system in response to effects caused by the seams. In the case of the uncertainty aspect of positioning this means a middleware that supports development of position based applications must incorporate some understanding of uncertainty.

Finding these concepts is not fundamentally different from finding concepts in the primary domain; therefore, developers should employ all the standard design methods of traditional software analysis when designing concepts to represent seams.

From our experiences working with position based applications we have found a number of concepts that developers can use to cope with uncertainty in the positioning domain. We organize them into three major groups, sensing, models and high-level domain functionality. A short outline of these groupings are presented here, for more details see Paper IV.

Sensing Positioning is ultimately dependent upon sensors measuring some physical phenomena which is turned into a digital representation. During the sensing process uncertainty is introduced by both noise in the physical world and by the sensors limited digital representation.

The primary concepts for uncertainty that we logically relate to measurements are: *accuracy*, *precision* and *staleness* (age of a given measurement).

Models Models of how certain positioning technologies perform in a given environment can be used to reduce uncertainty or provide applications with means to make informed decisions. Models can have both historic and predictive nature.

Concepts that we find to be generic in connection to modeling the positioning technology are: *predicted accuracy/precision*, *estimated sensor availability* (Probability that the position sensor is going to deliver a new position fix within X seconds) and *estimated sensor trust* (Probability that the position sensor fixes have been falsified).

Two examples of systems where models are used to manage uncertainty are: PosQ, a system using maps of historic accuracy and precision readings to predict the quality of future measurements [56]; and EnTracked presented in Paper III, which uses device and motion models to reduce power consumption.

High-level domain functionality Uncertainty can also be conceptualized as part of high-level positioning functionality. This is functionality where operations are performed on positions in a spatial context, e.g., proximity detection or target destination prediction.

For high-level functionality uncertainty is not as easily separated into isolated concepts. We find that uncertainty is best represented as a probability or confidence level associated with the high-level operation performed. These levels and probabilities can then be used as trigger levels and thresholds at position request at the application level.

The exercise of conceptualizing the seam is of cause very subjective, depending on the experiences of the actual people doing the design. The choices for how seams are conceptualized are directly dependent upon the exact overall focus of the middleware under design. However, as a field matures, the community is likely to converge on at least some key concepts. Also, the grouping of concepts presented here for uncertainty can probably be generalized to other domains where sensing and interpreting physical phenomena is involved.

Internalization

The internalization activity is about making the concepts identified earlier an integral part of the middleware. As shown in the previous section, concepts relating to a seam fall into different groups with varying levels of complexity and interdependency to the main domain functionality. This is reflected in the way those concepts can be internalized by the middleware. The internalization spans from providing means for simple inspection of concepts to deep integration with functionality of the primary domain.

During our work with positioning uncertainty we identified at least two distinct ways of working with uncertainty from an application developer's perspective.

- Dynamically changing positions delivered to the application based on some criteria. In many situations, uncertainty can be dealt with by just adjusting the positions before they are reported to applications. If the middleware internalizes uncertainty in a way that allows developers to express threshold criteria it can be used to trigger modification of the position values before they are reported to the application. If internalized as an extendable policy mechanism in the middleware, this way of working with uncertainty has the benefit of being usable after application deployment without requiring changes to either application nor middleware.

- Dynamically changing the control flow within application level code. Sometimes, the application is required to change behavior if uncertainty

properties changes. For example, in a mapping application where positions with low uncertainty is plotted as a single point and when the uncertainty rises, the position is plotted with the confidence range shown in addition. Support for this kind of use requires that the middleware internalizes uncertainty as a quantifiable measure which developers can monitor and react to. Internalized as events or monitorable values, this way of working with uncertainty requires application developers to integrate some understanding of uncertainty into their applications.

When deciding how to internalize concepts relating to a seam, both of these types of interaction should be supported. Furthermore, application developers should be able to ignore the seam and should not be forced to define criteria and coping functionality at deployment time.

In the following section the architecture for a middleware providing support for both ways of working with uncertainty is presented. Furthermore, we show that the concept of uncertainty can be implemented as a complement, orthogonal to the basic positioning functionality.

2.2 Integrating uncertainty in a positioning middleware

In Paper IV we propose a middleware architecture that allows uncertainty to be internalized. We argue that using this architecture uncertainty management can be structurally separated from core positioning functionality as illustrated in Figure 2.1. The architecture consists of two main components, Position management and Uncertainty management, which share a common hardware abstraction component through which sensors are controlled.

An important design goal for the architecture is that the Position management component should be realizable by an existing positioning middleware with minimal source code changes, e.g., the Java Location API [70]. To support this goal we argue that the following features must be available in the system. First, the hardware abstraction component must allow interception and modification of data. When sensors are being queried, the hardware abstraction component communicates with the sensor and delivers its readings to the client. It is these readings that must be interceptable and they must be modifiable. This allows the Uncertainty component to monitor the sensor readings and extract uncertainty related information from them. Furthermore, it allows the Uncertainty component indirect control of the Position component by manipulating the readings before they are received by that component. Second, the Uncertainty component must be able to control the Position component on behalf of the application. This will, for example, allow the Uncertainty component to implement functionality which changes reporting thresholds based on uncertainty. Aside from these two points of

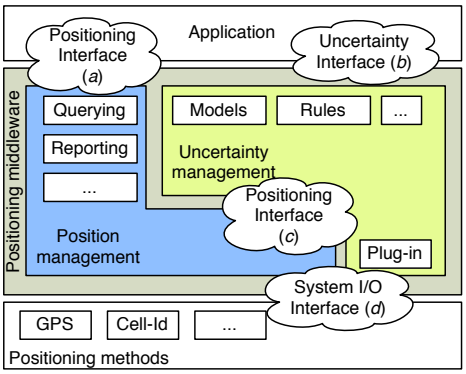


Figure 2.1: An architecture for a middleware that supports position based applications where uncertainty is internalized as a 1st class entity on par with the positioning functionality. The figure is replicated from Paper IV.

interception the Position component can remain oblivious to the existence of the Uncertainty component. The downside of this one-way integration leaves the Uncertainty component with all the book keeping responsibility. That is, because the Position component knows nothing about uncertainty it has no way of utilizing its internal data structures to represent uncertainty data; thus, potentially, causing double representation in the Uncertainty component.

Based on this architecture, we have evaluated the suitability of three different API design paradigmes for operating the uncertainty functionality. They are: a simple parameterized query API, an event based API and a declarative API. The APIs are evaluated with regard to both of the interaction patterns described in the previous section, changing delivered positions and changing application control flow. The following are the uncertainty related functionalities that where evaluated.

- Inspection and monitoring of uncertainty related properties of positions delivered to the application level, e.g., accuracy and precision.

- API access to concepts including defining constraints for positioning reporting, e.g., report positions when within 100 m of (x, y) with an accuracy of 10 m.

- Reacting to changes in levels of uncertainty, e.g., changing application control flow when positions no longer satisfy specified uncertainty requirements.

- Filtering based on uncertainty concepts, e.g., only report positions with an accuracy less that 5 m.

Table 2.1: Summary of evaluation of three different API design paradigmes.

Property	Naive API	Event	Declarative
Localizes changes to existing code	no	yes	yes
Separates uncertainty and position code	no	no	yes
Preserves existing code	no	yes	yes
Separates configuration from handling	no	yes	yes
Scales well with number of concerns	no	no	yes

- Provide hooks for supplying coping strategies. This includes augmentation of measurement objects.

For each API paradigme the following properties are evaluated: localization of changes to existing code, separation of uncertainty and positioning code, preservation of existing code, separation of configuration and handling, and scalability in respect to number of uncertainty concepts. The results are summarized in Table 2.1 and the conclusion is that a declarative approach is preferable for interaction between application and uncertainty functionality.

Using a declarative API has the advantage that it allows different uncertainty concepts to be handled independently. Furthermore, application developers can focus only on the concerns relevant to the actual application context. This in turn means that even for large number of uncertainty concepts the middleware API can be kept clean and uncluttered. Furthermore, as suggested in Figure 2.1 the handling of uncertainty should be implemented as a feature orthogonal to the main positioning functionality. While this is indeed possible with the appropriate extension mechanism, a declarative API greatly reduces the development effort for the middleware developers. Integrating concepts for a seam into a middleware leads to some generalization of the concepts. This in turn means that information is potentially lost in the process. An example, accuracy is a generic property of positioning technologies. However, the underlying effect causing the accuracy is entirely dependent upon the actual technology in use.

2.3 Generic support for seamful design requires translucency

In summary, with seamful design for developers we propose a method that supports a mindset where tool developers are encouraged to introduce a certain level of openness into their products.

For middleware developers this becomes something of a challenge. Realizing seamful design either requires middleware designers to be very thorough, and even then they miss something, or the middleware should itself be well suited for adaptation, allowing application developers to do the seamful de-

sign themselves. The latter can be achieved by using a translucent middleware which provide differentiated access to low-level details and provide differentiated control over internal processing elements of the middleware. The next chapter presents a middleware design approach providing these capabilities using a technique called *model based translucency*.

Chapter 3

Model based translucency

A middleware is defined as a piece of software that mediates access to one part of a system to another, typically providing high-level representation of a complex and heterogenous subsystem. A middleware is often described as a "black box" and its internal modus of operation need not be known to its surroundings. However, sometimes it is useful to be able to look inside a middleware in order to inspect and manipulate it. This can be achieved by making the middleware reflective. Reflective middleware is middleware that incorporates structures that represent various aspects of the middleware itself, a *self-representation* [71]. The self-representation of a middleware is causally connected with the middleware such that change in either the middleware or the self-representation will cause corresponding change in the other. This self-representation allows middleware designers to provide a differentiated view of the middleware functionality.

In the reflective middleware community it is common to refer to a dichotomy of transparent and translucent middleware with respect to the users of the middleware. For example in this quote:

> "A desirable middleware model provides transparency to the applications that want it and translucency and fine-grain control to the applications that need it" [58, p. 37]

Here, transparency should be interpreted as the middleware completely hiding the system it mediates access to, while translucency means that the middleware allows some level of access to the mediated system.

In the domain of positioning, middleware designed for the traditional goal of transparency, normally, follows the principle of information hiding [84] and hides all aspects of positioning from the application developer to provide a transparent experience when working with heterogeneous technologies. This means that it abstracts away imperfections in technologies and hides uncertainty from the developer [70, 62].

While some may interpret translucency to mean that a middleware should provide a generic openness with full access to inspect and change functionality [58, 31], we advocate that an interpretation where translucency means **a mediated openness with focus only on certain aspects of the middleware** may be easier to understand and use; and it may provide for a safer, although less powerful, development model.

In this chapter *model based translucency* is presented as a novel technique for building middleware, and the PerPos middleware is presented as an example realizing the technique. The chapter is based primarily on Paper I and Paper II.

Normally, a middleware is implemented based on some carefully chosen architecture which describes how individual parts of the middleware should interact. However, this architecture is usually not directly observable by application developers. The basic thought behind model based translucency is to make this internal structure as well as the internal behavior of the middleware accessible through a programmatically available *model* and thereby make it tangible by application developers.

This is similar to architectural reflection [35, 45, 91] where the internal structure is manipulated through a model, except that behavior and semantics must also be represented. Therefore, we claim that there is a need for a model that is specifically tailored for the actual application domain, and that translucency can be achieved by designing a model that captures the dynamic behavior of the middleware, in particular, the domain specific behavior. That is, not the generic component structure of the middleware, but, the domain specific components that are involved in serving the application. To produce this model the middleware itself can be implemented on top of a generic reflective middleware which provide the model with appropriate hooks into the base system. This is achieved by combining various methods that in unison contribute to overall middleware translucency: reflective middleware for flexibility, a well structured API for differentiated control of middleware functionality, and an internal architecture of the middleware which provides a supporting structure that helps application developers understand how the middleware provides its functionality.

The remainder of this chapter presents the overall design guidelines for model based translucency, an overview of the PerPos middleware and a reflection on how it relates to model based translucency and seamful design for developers. For further details about the actual model exposed in PerPos see Paper II.

3.1 Model design guidelines

As argued in Chapter 1, typical position based applications must operate in highly dynamic environments. This means that many important run-time

characteristics of the application are somewhat unpredictable by developers and must be identified and handled after deployment. For example, the propagation of radio signals from GPS satellites through obstacles like walls of buildings has a significant effect on position quality and will cause severe variations depending on the deployment environment. One approach to this problem is to use a simple *test*, *fix*, *redeploy* scheme to bring the application to a state where all environmental and deployment specific effects are handled. For a single deployment it might be possible to get satisfactory application quality by using this scheme, however, such a solution is most likely not generalizable. This is illustrated in the following. An application developer observes a problem or an effect at the application level for which he cannot immediately identify the cause. High-level APIs are designed to provide abstract access to the system. This means that even though the application developer uses a best-effort approach and tries to take into account all edge conditions supported by the middleware he will at some point hit the proverbial wall and be left with no middleware supported way of solving the problem.

The application logic at the point where the middleware is used might seem totally sound and the observed effects at the application level simply cannot be explained using the API provided by the middleware. When the positioning middleware is entirely opaque the developer must look for the problem either in whatever data he can observe from the *outside* using the public API of the middleware; or by circumventing it and monitoring the actual sensing hardware directly.

If, on the other hand, the middleware is translucent the developer is supported in looking for the cause *inside* the middleware guided by the various components of the middleware. When the problem is identified, the next step should be to instruct the middleware to take care of it. This could either be through reconfiguration of the middleware, or it could require new functionality that must be provided as some kind of extension to the middleware.

It is therefore reasonable to require a positioning middleware to provide some level of inspection of run-time system state. Combined with the ability to adapt middleware behavior at run-time, it will allow for an *inspect*, *analyze*, *adapt* development cycle that is better suited for highly dynamic deployments.

Generic reflective middleware does supports this goal, however, the generic openness usually provided by such systems does not provide guidance for application developers as to how they are supposed to understand the dynamic behavior of the middleware.

In model based translucency we propose that the middleware exposes a *domain specific* reflective model of the runtime processes of the middleware. We claim that this will allow application developers to appropriate the strengths of the middleware and adjust its internal ways of operation to accommodate specific application needs. The model should encode the middleware developer's understanding of the dynamic behavior of the middleware; and it should be seen as a medium to communicate this understanding to the application

developers who use the middleware. In order to achieve this, we claim that the terms and constructs used in the model should be directly related to the application domain supported by the middleware.

In Paper I we argue that a model adhering to the the following guidelines will make the middleware translucent in a way that promotes understandability.

Inspectability The model must act as an observable internal state of the middleware. That is, there must be a direct correlation between model entities and the actual state of the computational elements of the middleware.

Manipulatability The model must support changes, and changes made to the model must be absorbed by the middleware. The middleware must ensure that all legal modifications to the model are also legal in the middleware.

Tangibility The model must be an explicit and operational representation of the central concepts and functionality of the middleware. That is, by inspecting and manipulating the model the developer must be able to learn *how* the middleware realizes its functionality. This in turn means that concepts reified in the model should be humanly understandable.

Adaptability The model must contain points of extension allowing application developers to extend the middleware functionality through extensions to the model. Furthermore, the model must be domain specific in relation to the mediated technology so that new elements of that domain can be integrated and added at a later point in time.

Differentiated abstractions The model must provide several perspectives on the middleware behavior, preferably, organized in levels of abstractions.

Inspectability, manipulatability and adaptability can usually be achieved through application of techniques for reflective middleware, interception and language level reflection, while tangibility and differentiated abstractions rely more on the actual design effort of middleware designers. Tangibility is in large determined by the software architecture applied to the middleware and how that architecture is captured by the model. Finally, realizing differentiated abstractions requires thorough API design.

We propose a number of guidelines for designing the major design artefacts of a model based translucent middleware. These artefacts are depicted in Figure 3.1, 1) is the internal architecture of the base system of the middleware, 2) is the API of the meta-model, and 3) is the design of the reflection mechanism for hosting the meta-model. The primary principle guiding the design of these artefacts should be that of seamful design for developers as

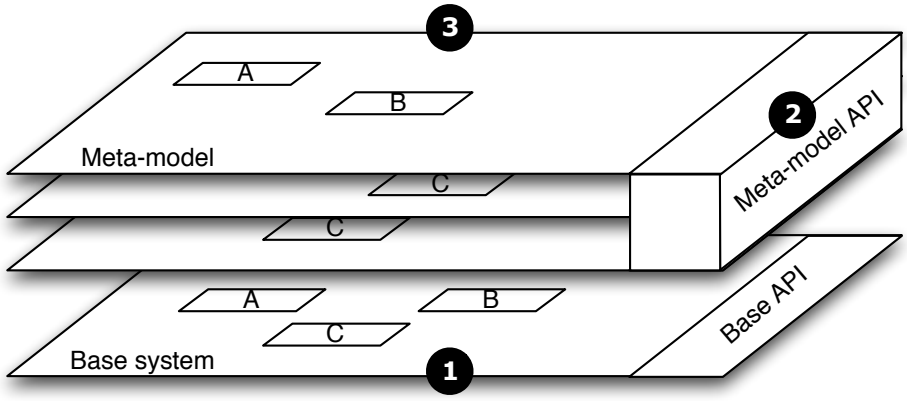


Figure 3.1: Primary design artefacts relevant to model based translucency. 1) design of the base system with core functionality, 2) design of the meta-model API and 3) design of the meta-model and its layered structure.

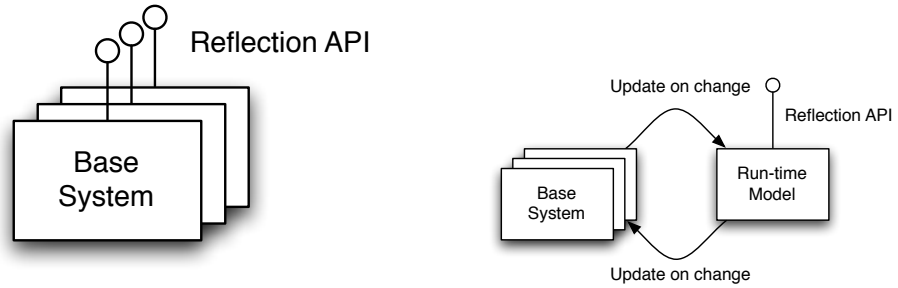
presented in Chapter 2. We argue that this principle is used when selecting what to expose to developers and how to shape the API of the middleware. First, the seams of the domain must be identified and conceptualized. Then, the middleware designer must design an internal architecture that allows these seams to be manipulated through the elements of the architecture.

Internal middleware architecture

Designing the internal architecture of a middleware is a very domain specific task. However, in order to stimulate interaction and management of seams in the domain, some basic properties are beneficial to include in the design. It is important that each time information is somehow removed from the system it is represented by some artefact, e.g., when evaluating a non-reversible function on data, the parameters and the data sent to the function should be interceptable. Furthermore, to facilitate tangibility a component based architecture is preferable. This stimulates developers to encapsulate functionality, it promotes separation of concerns, and it makes for a straight forward representation of the structure of the middleware in the meta-model. We suggest that the components of the model should correspond as directly as possible to domain concepts, e.g., sensors for a positioning middleware, or network protocols for a distribution middleware.

Meta-model API structure

To allow application developers control of concepts that relate to the seams of the domain, the API should support opportunistic use, i.e., the API should be



(a) Causal connection ensured by shared architecture.

(b) Causal connection ensured by model translation component.

Figure 3.2: Two different ways of ensuring causal connection between base system and the reflective self-representation or meta-model of the system.

unobtrusive when not in use and allow for features to be picked and activated independently.

As argued in Chapter 2 we find that a declarative API ensures that different aspects and seams can be handled independently by application developers. When designing the declarative API for a seam, it is important to ensure that there is correspondence between base system properties of that seam and the API elements representing it. To realize the primitives for the declarative API, there is a need for the middleware to provide access and control of internal parts of the middleware at several levels of abstraction. Furthermore, the meta-model API should support meta-level programming, allowing the declarative API to be implemented using meta-level programming itself, thereby allowing indirect changes to the API.

Using a declarative API to provide interaction with application developers enables application-level meta-programming. This in turn enables programmatic enforcement of policies. Policies are meta-level elements that manipulate the middleware through the meta-model API in order to enforce themselves. Policies can be used to realize tactics for managing various quality properties of the middleware. For the positioning domain, we present a collection of qualities and associated tactics for managing them in Paper VI.

Reflection mechanism

The design of the reflection mechanism plays a significant role towards achieving translucency. There are at least two very distinct ways of realizing a reflective run-time model for any given system, traditional reflection and mirror reflection [19], this is illustrated in Figure 3.2.

In traditional reflection, Figure 3.2(a), the run-time model and the base system are tightly integrated such that elements belonging to the reflective

run-time model are members of the base system components. For component based middleware, as an example, this means that component containers are aware that they are components within a component system, and the meta-data, representing that knowledge, is co-located with the component functionality.

In mirror based reflection, Figure 3.2(b), a dedicated component is responsible for translating events of the base system into appropriate elements in the model, and for translating model changes back into the base system. This is typically done through hooks in the base system which allows the translation component to inspect and adapt the base system. This approach has the benefit that the base system is not "polluted" by additional meta-information, and mirroring also promotes clear separation between base and meta levels. This clear separation enables meta-programming to be independent of the base, in many cases. Furthermore, using mirror based reflection, the meta-level is occupied by mirror objects matching the base-level objects. This means that, in general, mirror based reflection is very well suited for maintaining a meta-model with high *structural correspondence*, i.e., the structure of the meta-model corresponds to the structure of the middleware components [19]. However, while it is possible to design the meta-model with low structural correspondence, for our purpose of enabling translucency, high correspondence is important as it enhances the tangibility of the middleware.

In the following section the PerPos positioning middleware, presented in Paper II, is used as a vessel to investigate how these goals can be satisfied in order to realize a model based translucent middleware.

3.2 Model based translucency in the PerPos middleware

PerPos is a translucent positioning middleware that is built based on the ideas of seamful design for developers. The fundamental idea behind PerPos is to provide transparent positioning while still allowing the middleware to be "opened up" by the developer, thereby gaining access to internal mechanisms of the positioning process. This section gives an overview of the middleware and relates it to the goals of model-based translucency. First there is an overview of how the positioning functionality is structured in the middleware. Then there is an analysis of how the meta-model used in PerPos is satisfying the goals stated above.

Positioning functionality

The positioning specific functionality of PerPos is organized in layers inspired by the Location Stack [42] which have the following seven layers with increasing levels of abstraction: Sensors, Measurements, Fusion, Arrangements,

Contextual Fusion, Activities and Intentions. In PerPos we have collapsed some of these layers so that we get only three layers in total: Sensing, Target and Reasoning. The Sensing and Target layer of PerPos roughly correspond to the first four layers in the Location Stack; and the Reasoning layer corresponds to the remaining three. The Sensors and Measurements layers of the Location Stack correspond to the PerPos Sensing layer and Fusion and Arrangements correspond to the PerPos Target layer.

The Sensing layer is responsible for managing the low-level steps of the positioning process. This includes providing hardware abstractions, encapsulating physical sensors and providing systems for emulation and logging of data. The primary elements of the Sensing layer is hardware drivers/wrappers, common measurement classes, data parsers, emulators and sensor configurations. The sensing layer contains no explicit conceptualization of positions, all measurements are seen as pure data entities not undergoing any form of interpretation.

The Target layer is responsible for interpreting the measurements and data provided by the Sensing layer. In the Target layer measurements are converted into positions based on appropriate location models. High-level manipulation of the positions like fusion of several data sources also belongs to the Target layer. In addition, the Target layer is responsible for mapping target identities to concrete sensors. The primary elements of the Target layer is configurations, coordinate system functionality, location models and sensor fusion mechanisms.

The main responsibility of the Reasoning layer is to provide high-level reasoning about positions and targets. This includes functionality like activity recognition, context sensitivity, clustering detection algorithms, proximity alarms, distance based notification etc.

In addition to an API for the positioning functionality, the PerPos middleware provides a self-representation in the form of a meta-model also accessible by application developers.

Meta-model

In PerPos, we generalize position based applications into stream-based processing of data. In this way the positioning activity can be expressed as a graph consisting of data processing nodes that incrementally refine the position calculation. At the core of the middleware there is a reflection mechanism that maintains a causally connected meta-model of the positioning process which provides methods for manipulating the actual positioning system. This model is presented in details in Paper II and briefly summarized in here.

The meta-model is divided into three levels which represent the middleware at different levels of abstraction. These levels are illustrated in Figure 3.3, which is replicated from Paper II. The three layers ordered by level of abstrac-

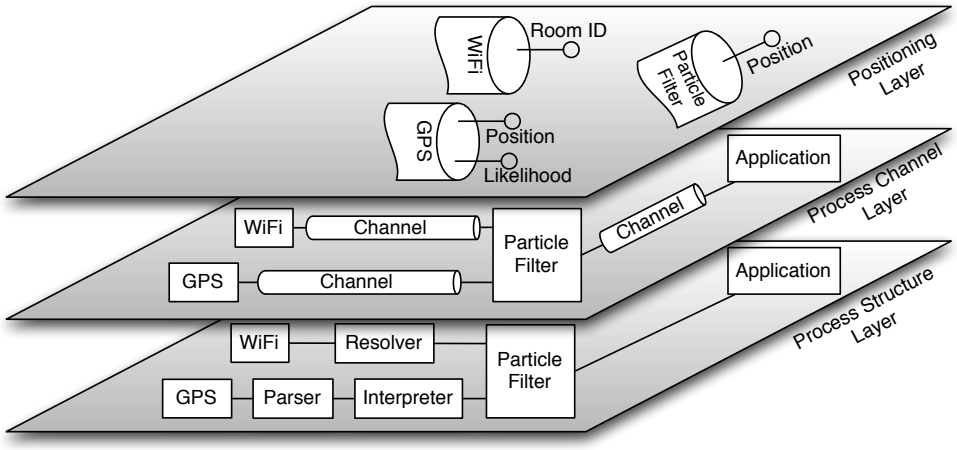


Figure 3.3: PerPos exposes the internal positioning process in a three layered model. The illustration shows the model representation of one instance of the base system. The base system is set up to provide positions from a GPS and a WiFi system and fuse them together to one through the use of a particle filter. This state is then represented in the three layers of the model. The lower layer reflects the structural composition of the base system, the middle layer provides an abstract representation of the data flow, and the top layer provides application level positioning support.

tion, starting from the most abstract are: Positioning Layer, Process Channel Layer and Process Structure Layer.

The model contains nodes representing each individual processing step. These steps are then grouped into channels that represent a larger part of the flow. For each data element produced by a channel, the channel maintains a hierarchal data structure containing the output of all sub-nodes that have contributed to that output. From an application perspective, channels represent the connection between the application and a positioning system.

Applications are able to access the channels at their endpoints through the public API of the PerPos middleware. The functionality exposed through the channel endpoints are usable in applications as declarative constructs. This functionality is exposed to applications as annotations that can be attached to the code tying application and position providers together. For example, a channel could implement functionality that maintains a certain level of power consumption of providing a consistent level of accuracy.

PerPos provides a programming model for manipulating the model and the nodes, thereby controlling quality properties of the application. The meta-model provides composition functionality, e.g., add/remove node etc. Application developers have the ability to extended the functionality of individual nodes and of entire channels by adding custom functionality packaged

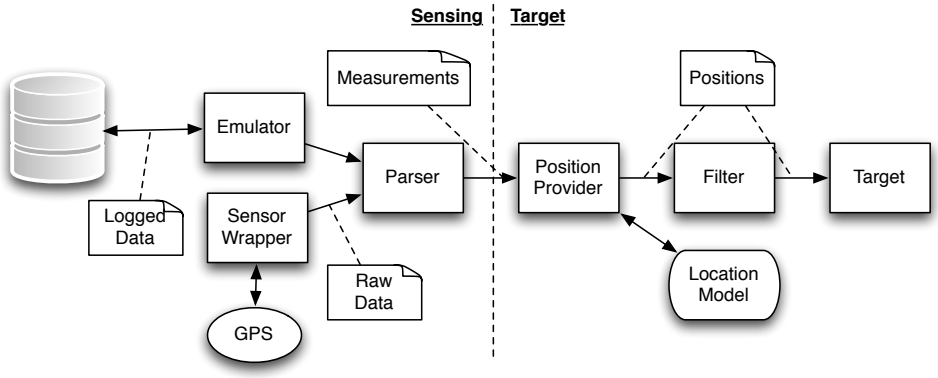


Figure 3.4: Example instance of a data processing graph. Data from different sources, emulator and GPS, are incrementally refined, left to right, into position data that can be correlated with a tracked target and delivered to applications.

in components called Feature and Channel Feature respectively.

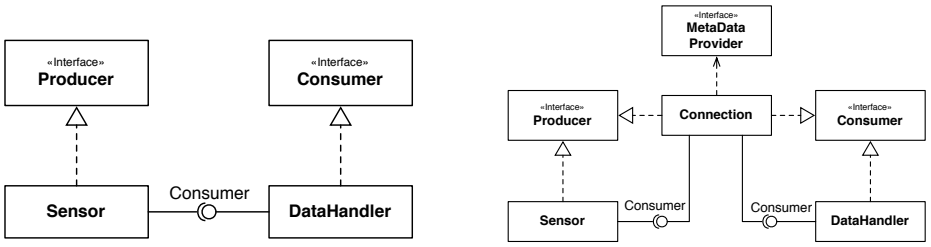
In the following sections we see how the meta-model of the PerPos middleware satisfies the five goals for model based translucency stated earlier.

Inspectability

The base system of the PerPos middleware is centered around a data processing graph representing the positioning process. This graph is composed of components that are either data producers, data consumers or both, illustrated in Figure 3.4. The middleware can dynamically create proxies for all the components in this processing graph and thereby creating hooks for intercepting component input and output. Application developers can programmatically, and through a visualization, get information about the components at runtime. They can, for example, retrieve the names of the sensors, get information on how components are connected to each other, output the data that are sent between components, etc. These abilities are how PerPos satisfy the demand for inspectability.

Manipulatability

For the middleware designer, the explicit representation of the processing structure provides a way of encapsulating internal positioning functionality into small well defined components that are flexible and enable clear separation of concerns. Moreover, the structure makes the task of creating a reflective model of the process somewhat easier as long as the relationship is one-to-one between base system and model. By keeping this relation while allowing com-



(a) Before interception. (b) After interception.

Figure 3.5: UML diagram showing how the positioning chain is extended by interceptors.

ponents to be added, removed, connected and disconnected, the model is able to support the *manipulatability* goal, adding to the overall translucency.

Tangibility

In the base system, the individual components of the processing graph implement either the `Consumer` interface or the `Producer` interface or both, see Figure 3.5(a). The reflection component that builds the meta-model of the process exploits this design by dynamically adding proxy objects in between all consumer/producer connections as shown in Figure 3.5(b). The producer-consumer structure enables the meta-model to support the goal of *tangibility* because it allows the model to mirror the actual architecture of the middleware. Furthermore, the components of the meta-model all correspond to actual components of the position refinement process in the base system.

Adaptability

The *adaptability* goal is realized through an extension mechanism where nodes of the model can be extended by what we call a Feature. Features can intercept input and output of a component and they can access the implementation of components through language level reflection. For example, a node representing a sensor can contain a Feature called `state` that exposes a property which can be set to either `on` or `off`, resulting in powering on or off the actual sensor respectively. Also, a Feature can provide additional data representing various state information, e.g., current position, accuracy etc. In addition, the data flowing between nodes can be accessed and monitored.

Differentiated abstractions

The goal of *differentiated abstractions* is realized by the layered model of PerPos. The API for manipulating the model is briefly presented here, for

Listing 3.1: Accessing the API of the Process Structure Layer and the Process Channel Layer

```
1 ...
2 gps = Model.getProcessStructure().getComponent("gps");
3 Model.getProcessStructure().addFeature(gps, Features.logger);
4 ...
```

a more detailed description see Paper II, Section 2. The API is divided into three groups based on the three layers of the model as they are shown in Figure 3.3.

The Positioning Layer corresponds to the standard positioning API described above with the three abstractions: Sensing, Target and Reasoning. It is used as the primary way for applications to interact with the middleware. Basic position querying and reporting is accessed through the API for this layer. The details of this layer is further discussed in Chapter 4.

The Process Channel Layer provides control over the channel concept briefly described earlier. The layer provides the possibility for adding functionality to the middleware that spans several producer-consumer components. The components of meta-model are grouped into so called channels which represent the single-strained flow between two components. For each output produced by the sink component of a channel, the channel is capable of collating the input and output of all intermediate steps. It is possible to attach features to these channels, called Channel Features. These Channel Features can react to the output produced by the channel and inspect the output of all the components within the channel wich contributed to that output.

The Process Structure Layer provides direct access to the individual components of the meta-model, and to its structure. Through the API it is possible to change middleware behavior by adding and removing Features, and by connecting and disconnecting individual components.

An example of how the model is accessed from applications is shown in Listing 3.1. The example shows how the API is used to add a logging feature to the GPS component by using the Process Structure Layer. In the actual Java implementation of PerPos, each layer of functionality is accessible through a static class called `Model` and developers can chose to interact with the middleware at any abstraction level.

3.3 Benefits of model based translucency

The PerPos middleware supports both seamless and seamful interaction in that is delivers technology independent positions at a high-level layer while allowing for structured inspection and adaptation of the internal processing that lead to the high-level positions. Thus, to the extend that sensors and

processing elements contains information that may be used to deduce for example, current coverage, accuracy, and signal noise, this information, which is usually hidden for the sake of transparency, can be used to expose the seams. Furthermore, because all the sensor specific details are accesible from the application level, applications can dynamically adapt the middleware depending on application requirements, user preferences, device characteristics, and overall system state. Furthermore, using a dynamically adaptable model of the middleware allows flexibility at run-time instead of being forced to do up-front configuration during service provisioning time.

The benefit of model based translucency is that extensions and adaptations can be written in the context of a model with a limited set of primitives, thereby providing a simpler programming model. The simplifications of the reflective meta-model ease the task of getting an understanding of the middleware functionality and help provide developers with a better overview of complex systems. Furthermore, it allows extensions to be modularized and provides support for separation of concerns. This allows developers to manage application level concerns that are orthogonal to the primary domain supported by the middleware. As we shall see in Chapter 4 overall application quality of position based application are often dependent upon orthogonal properties and need interaction with the middleware at many levels.

Chapter 4

Quality management using tactics and policies

As previously stated, the primary goal of the research presented in this dissertation is to support developers of position based applications in their efforts to deliver high-quality position based applications. This chapter provides an overview of software qualities that are of particular interrest for position based applications, and argues that these qualities can be managed by the application of tactics targeted specifically at this particular domain. These qualities and tactics are thoroughly analysed in Paper VI. Furthermore, the usefulness of translucent middleware is analysed in relation to supporting developers in the task of improving these qualities. Also, it is shown that the flexibility provided by model based translucency allows the middleware to be continuously augmented with support for new quality improving tactics. This is done by showing how tactics are implementable as declarative policies in the PerPos middleware and by letting a group of students use the middleware to develop a number of position based applications.

4.1 Qualities in position based applications

In general, the quality of position based applications are affected in much the same way as any other software product, and developers need to be aware of how to achieve satisfactory quality levels. In addition, there are a number of qualities that need special attention when using positioning in an application and the way various aspects of the positioning process influences application quality poses a significant challenge to developers.

Like in traditional applications, qualities affecting position based applications involve parts at all levels of the system. For example, the functionality quality *accuracy* of positions reported to an application using a GPS receiver is dependent upon the quality of the sensor, the strength of signals received

Quality	Description
Functionality	
Accuracy	The maximum distance between the reported and the true position.
Precision	The maximum distribution of a series of reported positions corresponding to the same true position.
Security	The level of access to the location data generated by the positioning system.
Reliability	
Integrity	A measure of the validity/credibility of readings.
Availability	The ability of a system to perform the required functionality a given point in time.
Efficiency	
Response time	The capability of the positioning system to provide appropriate response and processing times.
Freshness	The age of reported positions from the time they are requested.
Time variance	Providing a stable and stream of reported positions and reducing jitter in the stream.
Power efficiency	Reducing the amount of power used in order to deliver positions of a specified quality.
Bandwidth efficiency	Reducing the amount of communication bandwidth used to provide a specified level of positioning quality.

Table 4.1: Quality characteristics relating to position based applications.

and interpretation of coordinates in respect to application level geographical data. When implemented in a middleware, these properties are usually implemented at different layers of abstraction, e.g., the sensor is accessed in a hardware-near layer, signals are interpreted in a support layer with libraries for generic GPS functionality and the geographic interpretation of coordinates is handled closer to the application layer. This means that, in order to improve on a quality like accuracy, access to several layers is necessary.

In relation to position based applications, some qualities are of particular interrest. These qualities are primarily those that are somehow affected by technical limitations of the positioning technologies involved. The main quality characteristics we find relevant are: functionality, reliability and efficiency. Within each of these categories we have chosen a set of sub characteristics that are of particular relevance to position based applications. In Table 4.1 these quality characteristics are summarized and they are adapted to fit the ISO 9126 [46] standard for software product quality in Paper VI.

4.2 Using tactics to manage qualities

We propose to make these qualities manageable by defining a set of *tactics* that each describe a proven method for controlling a specific quality characteristic. The concept of quality tactics is adapted from the field of software architecture where they are used as a form of recipes for achieving improvements to particular quality attributes [7]. For each of the relevant qualities we have formulated a number of different tactics based on solutions found in related research papers. Based on the literature, our experience working with positioning technologies and experimentation on prototypes build using the PerPos middleware we have analysed the tactics in relation to the various qualities. Table 4.2, which is replicated from Paper VI, shows the influence that each individual tactic has on the various quality characteristics. Each row in the table represents one of the tactics described in Paper VI along with an indication of which qualities are affected by the tactic.

One example of a tactic that improves *efficiency* by reducing power consumption is "Dynamic using motion information". This tactic controls the sampling of the positioning sensor based on the output from a secondary sensor that determines whether the user is moving or stationary, e.g., output from an accelerometer [53]. Using this tactic it is possible to reduce the *power consumption* significantly in scenarios where the tracked target is mostly stationary. However, if the tracked target is always moving, using this tactic might in fact increase the *power consumption* due to the added accelerometer. Furthermore, because lowering the sample rate result in ignoring some samples, *accuracy* is potentially affected negatively. In addition, the unpredictability of the sampling rate means that *time variance* is affected negatively, i.e., samples are not guaranteed to be delivered steadily and uninterrupted.

Another example is "Sensor fusion". This tactic combines the output from several sensors – preferably of different types – for the same target and combines it into one aggregated position while taking into account an error model for the sensors [42, 88]. While sensor fusion is intended to increase accuracy, it will also often improve precision. However, as multiple sensors are involved, *power consumption* is typically higher than relying on a single sensor. Furthermore, as the aggregation process imposes a level of latency to the position calculation, *freshness* is potentially reduced.

The tactics presented in Paper VI are all specifically targeted for the middleware layer, i.e., only in the middleware is all the information needed to realize the tactics available. Therefore, it would be only logically to implement tactics as part of a positioning middleware. Many application qualities, however, have a *cross-cutting* nature, because the properties of the system that they depend upon are spread across conceptual boundaries of the positioning process. This in turn means that in order to implement a tactic to control such a quality, access to several levels of the middleware is required.

Tactic name	Quality characteristics and sub-characteristics									
	Functionality			Reliability		Efficiency				
						Time Behavior			Resource Utilization	
	Accuracy	Precision	Security	Integrity	Availability	Response time	Freshness	Time variance	Power	Bandwidth
Sensor Fusion	+	+					–		–	
Use Environment Information	+									
Constrain by Motion Model	+									
Statistical Temporal Filtering	±	+					±			
Use Terminal Based Positioning			+							
Anonymize Data										
- Frequently Change Pseudonym			+							
- Report k-anonymity Region	–		+				–	–		
Hide True Position										
- Append Position Dummies	–		+						–	–
- Degrade Data Accuracy	–		+		(+)					(+)
- Use Application Specific Queries			+		(+)	(+)			(+)	(+)
Manage Information Receivers										
- Prompt User			+			–	–	–		
- Use Policies			+							
Fault Detection				+					–	–
Error Estimation				+					–	–
Graceful Degradation	–			+	±					
Expand Coverage					+					
Combine Coverage Areas					+				–	
Use Fast Positioning Technique	±					+				
Constraint Data Age							+	–		
Extrapolate for Missing Values	±	±						+		
Sensor Selection	–							–	+	(+)
Duty Cycling										
–Static	–						–	–	+	(+)
–Dynamic Using Motion Information	–							–	+	(+)
–Dynamic Using Application Logic	–					–		–	+	(+)
–Dynamic Using Motion Prediction	–							–	+	(+)
Decrease Number of Messages										
–Cache Position Data	–							–	(+)	+
–Periodic Querying or Reporting	–							–	(+)	+
–Only Communicate Deviations						(+)	(+)		(+)	+
–Distance Based Reporting	–								(+)	+
–Movement Based Reporting	–								(+)	+
–Aggregate Data						–	–		(+)	+
–Local Caching and Processing				–					(+)	+
Decrease Size of Messages										
–Asymmetrical Compression									(+)	+
Schedule Data Communication							–	–	(+)	+
Use Application Specific Protocols									(+)	+

Table 4.2: Overview of tactics and their quality implications. Legend: "+": increase of the quality. "(+)": derived increase of the quality. "–": decrease of the quality. "±" : may have an effect on the quality. Please refer to the tactic descriptions in Paper VI for further explanations of the quality implications.

4.3 Realizing tactics using policies

The meta-model made available using model based translucency can be used as a "skeleton" to which sub elements of a tactic can be attached. To prevent potential fragmentation, tactics implementations can be modularized through the use of a policy mechanism. Exposing this policy mechanism to developers through a declarative API allows developers to ignore the policies that are not relevant for addressing a particular quality. Furthermore, the flexibility provided by model based translucency allows the middleware to be continuously augmented with support for new quality improving tactics. In the following sections, all examples using a meta-model and extensible middleware are based on the use of the PerPos middleware and the meta-model described in Chapter 3 and Paper II.

Using a meta-model to implement tactics

Since tactics are not generally applicable in all situations, it is necessary to be able to enable and disable tactics depending on the actual deployment situation. Moreover, although tactics can be described in generic terms, actual implementations differ greatly; different implementations of the same tactic can have entirely different side effects or system requirements. This property of tactics makes it very difficult to integrate other than the most generic of tactics into a middleware at design time.

The meta-model exposed in PerPos can be used as a framework for implementing tactics that require detailed control and inspection of internal processes of the middleware. Through the combination of having the positioning process explicitly represented as a flow-graph and the extensibility provided by Features, a great deal of quality improving tactics can be implemented in PerPos without a priori knowledge encoded into the middleware. A Feature is a piece of code that lives in the base system of the middleware and is attached to a component in a way that is controlled by the meta-model. A Feature must implement a simple interface which specifies a number of callbacks that the components of the base system use to activate the Feature when it is appropriate. The callback hooks available to a Feature are listed in Table 4.3.

Figure 4.1 shows an example of how a tactic can be supported by the meta-model. The tactic is shown in a generalized form to illustrate the basic mechanisms involved. At the center of the figure is the meta-model, here consisting of a `Sensor` component, some arbitrary processing component and an `Application` component. The purpose of this hypothetical tactic is to modify the behavior of the sensor whenever position data received by the application satisfy certain criteria, e.g., raise or lower GPS sampling rate in relation to traveling speed reported to the application. To implement this tactic, the middleware must first be extended to allow changing the behavior of the sen-

Table 4.3: Hooks available to features in PerPos

Name	Description
`processOutput`	Process data produced by a component or a feature attached to it.
`processInput`	Process data received by a component.
`cancel`	Allows a feature attached to a component to cancel further processing of data done by the component.
`produce`	Allows a feature to produce data on behalf of a component.
`reflect`	Allows a feature access to the implementing artefacts of a component through language based reflection.

sor and provide inspection of the appropriate values of the positioning data. In PerPos, this is achieved by attaching Features to the meta-model. Each component in the meta-model can contain arbitrary values stored in the component meta-data. These values can be read and written by Features and the meta-model provides a notification mechanism allowing tactics implementations subscribe to changes in specific values. For the generic tactic of this scenario, two Features are attached to the model: one `Control` Feature providing state modification behavior to the `Sensor` component, and one `Detector` Feature inspecting the data being delivered to the application and setting appropriate meta-data values. The dynamics of the scenario is shown by the numbered steps in the figure. When the system is running, 1) the `Detector` Feature continuously monitors input to the `Application` component, 2) appropriate meta-data values are written to the `Application` component, 3) the code implementing the tactic is notified of the change in meta-data, and 4) the behavior of the `Sensor` component is changed through the `Control` Feature.

The scenario illustrated here requires the ability to inspect internal values transmitted within the middleware and it requires the ability to change the state of certain components. More generally, to use a meta-model as a framework or skeleton for attaching the realization of a tactic, the middleware must have an internal component representation of the positioning process and the middleware must provide the following basic abilities:

1. Allow access to the input values for a component method.
2. Allow access to the output value of a component method.
3. Allow cancelation of the output of a component method.
4. Allow custom code to replace the output of a component method with another value.
5. Allow custom code to initiate data output on behalf of a component method.

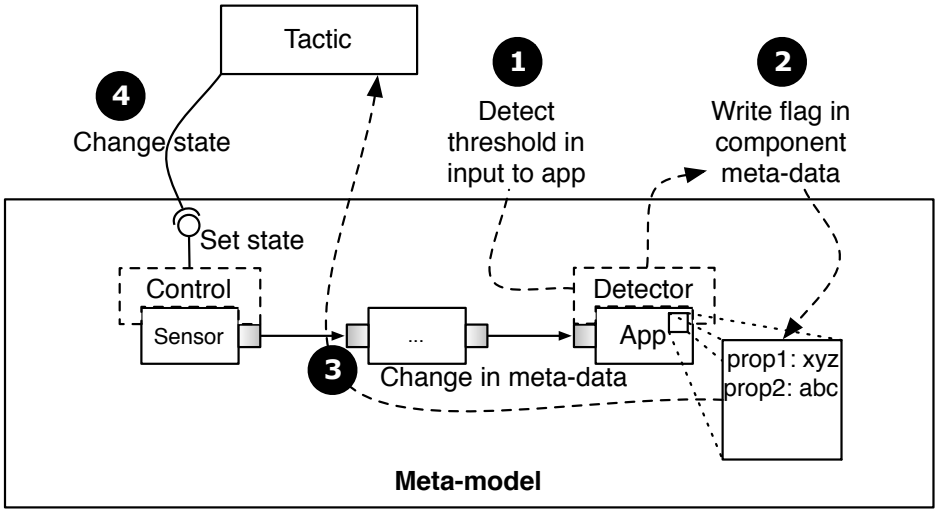
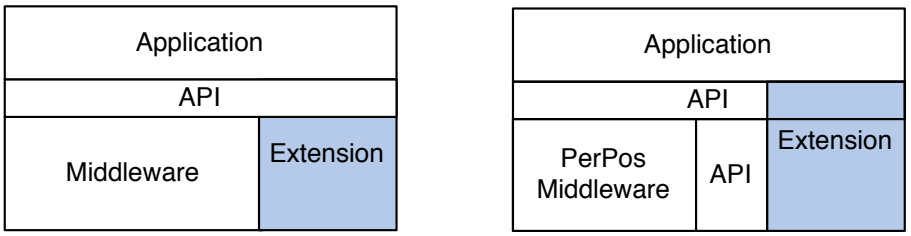


Figure 4.1: Using the meta-model to implement a tactic. The meta-model is augmented with two Features, `Control` and `Detector`. 1) The detector compares produced data with threshold values and 2) writes flags as meta-data in `Application` component 3) which are registered by the tactic 4) which in turn changes behavior of the `Sensor` .

6. Allow custom code access to the objects realizing a component and ability to call methods on them.

The first 4 abilities are, generally, satisfied by `before`, `around` and `after` constructs in languages like Lisp and by the corresponding aspect oriented programming constructs [50]. In Fractal [21] they are supported by redefining the bindings between the components to allow interception or by using component controllers to intercept requests. Adding data to the output of a component method can be implemented by adding a parallel call that logically happens at the same time as the original call. The last ability is satisfiable with generic language reflection.

More generally, the feasibility of implementing tactics as *post hoc* extensions depends on the implementation of the middleware. It is simpler to develop an alternative behavior for a component method that carries out a well-defined step in a chain of processing steps than for a complex method that takes raw sensor data as input and outputs a high-level position. Moreover, when accessing the implementing class of a component, the implementation of this class determines the kind of information that can be extracted and to what extend the component can be controlled. The extension developer needs semantic understanding of the implementing class; and middleware designers must take this into account and make sure that the meta-model and the



(a) Extending a transparent middleware only changes the behavior of the middleware, not the API.

(b) Extending a translucent middleware allows the API to be extended in addition to middleware behavior.

Figure 4.2: The difference between extending a transparent and a translucent middleware.

extension mechanisms helps to convey this understanding.

Policies as a modularization of tactics

Besides ensuring that application developers understand the middleware, using the meta-model to realize tactics may lead to a fragmented implementation with implementation artefacts scattered across multiple parts of the system, i.e., Features attached all around the model, semantics defined by the meta-data properties and the logic controlling the tactic itself located at the application level. This can be observed from the example shown in Figure 4.1 and the examples of tactics presented in Paper V. This scattering or fragmentation motivates the need for some level of encapsulation or modularization supporting implementation of tactics.

In addition to the problem of scattered implementations, tactics often require very application specific customizations, e.g., critical threshold values, geographical boundaries or similar parameters available only in the final deployment. Therefore, it is unlikely for a generic positioning middleware to include implementations of specific tactics. To overcome this problem of lacking inclusion of tactics, middleware designers can delegate the implementation of tactics to application developers or specialized domain experts.

There are several ways to extend the functionality of a middleware, especially when comparing middleware providing transparent access to a domain to one providing translucent access. For the former, typical extensions can change functionality at predefined areas or provide support for new devices, leaving the API unchanged, see Figure 4.2(a). For the latter, extensions include those possible in a transparent middleware, and, in addition, the external behavior is likely to be extensible as well, shown in Figure 4.2(b).

One method for supporting modularized extensions that can be created by third-party developers is to let developers write and inject *policies* into the

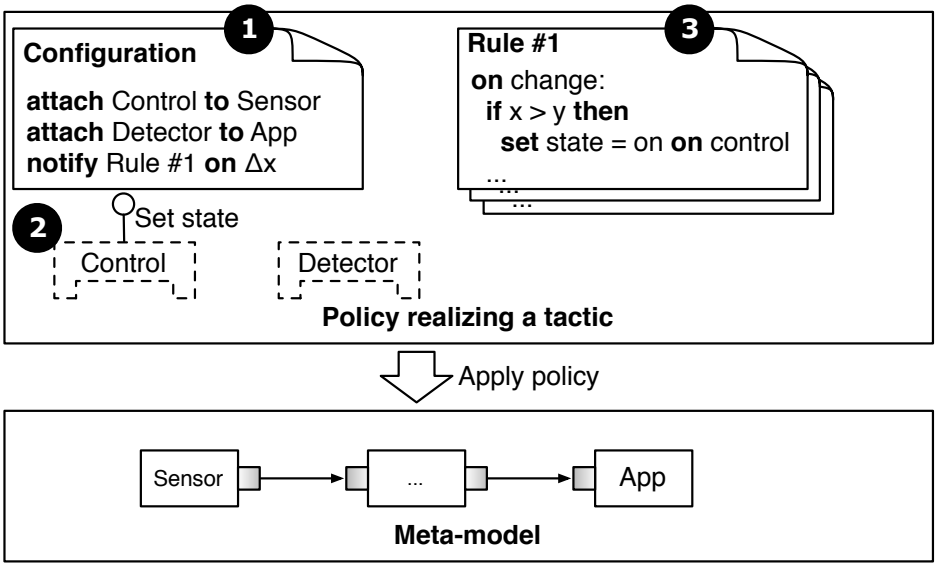


Figure 4.3: Encapsulate all the artefacts needed to apply a tactic to the meta-model in a policy.

middleware. In this context, a policy is a modularization of the elements realizing a tactic through the use of a meta-model as presented earlier. Basically, a policy consists of the three primary artefacts illustrated in Figure 4.3: 1) a configuration describing how the policy should be attached to the meta-model, 2) a set of Features used for realizing the support structure for the tactic, and 3) a trigger-action rule containing code for monitoring meta-model state and executing changes to the model upon triggering. Policies are interpreted and enforced by a policy engine implemented as part of the meta-layer.

The use of policies for encapsulating extensions to the middleware can be used to extend not only the middleware functionality, but also the API used by application developers. When a new policy is created, it can be registered with the middleware; if the set of registered policies are made available to application developers, they can effectively be seen as extensions of the API. In the next section we see how the API of PerPos is extended in this way, by exposing policies through declarative annotations. The idea is to introduce a module concept for encapsulating quality properties. These modules will be required to provide entries to add to the API. For example a module for handling accuracy limits could expose the following entries: max/min allowed accuracy and notify on accuracy threshold.

Listing 4.1: Simple position logger

```
1 import perpos.position.PositionHandler;
2 import perpos.power.MinimizePowerConsumption;
3 ...
4 @PositionHandler
5 @MinimizePowerConsumption
6 public void onNewPosition(Position p) {
7   db.insert(p);
8 }
9 ...
```

The declarative API of PerPos

Based on the analysis of seamful design for developers presented in Chapter 2, we have designed a declarative API for binding the PerPos middleware to applications. The standard API contains roughly the same basic functionality as the Java Location API for J2ME (JSR-179), i.e., applications can request positions through either push, pull or periodic update schemes. The main entry point for an application is the configuration of notification and querying. The application registers an event handler that should be notified when positioning events are generated by annotating a method with the `@PositionHandler` annotation, illustrated in Listing 4.1. This binding between the application and the middleware can be further annotated with parameters declaring constraints to be satisfied by the binding. A few examples of basic annotations are: `@Target([targetID])` for receiving positions of a specific target, `@UpdateRate([updates/second])` for specifying the position update rate, and `@MaxDistance([distance in meters])` for indicating the maximum distance between two consecutive position updates.

The declarative nature of this API is especially useful for policies that represent tactics. By allowing policies to be associated with an annotation that can be applied to the binding of the position handler, these policies can provide declarative extensions to the API. For example, assume a policy that handles power efficiency using a set of tactics all relating to the power efficiency sub quality category. This policy continuously monitors the meta-model, and applies tactics dynamically to improve power efficiency. By exposing an annotation named `@PowerEfficient` and associating it with the policy, application developers can declare that the delivery of positions must be power efficient which is the ensured by the policy.

In the PerPos policy model, a policy consists of a Java annotation, a class implementing an interface, `Policy`, and, possibly, a set of Features to augment the meta-model. The annotations used in the PerPos API are not linked to a specific policy, instead the connection between annotations and policies are configurable. This means that the *what* (annotation) is separated from the *how* (policy).

Listing 4.2: Simple rule for preventing stale measurements.

```
1  ...
2  public class PreventStale extends ActivationPolicy {
3  ...
4  protected Condition getCondition() {
5    return new Condition() {
6      public boolean satisfied() {
7        boolean isGpsOn = (boolean) gps.getProperty("state");
8        long time = (long) gps.getProperty("timeSinceUpdate");
9        return !isGpsOn && time > staleThreshold;
10     }
11   };
12 }
13 protected Action getActivationAction() {
14   return new Action() {
15     public void execute() {
16      Model.getProcessStructure().invoke("start").on(gps);
17    }
18   };
19 }}
```

Listing 4.2 shows an example implementation of a simple policy that is intended to prevent stale measurements. The policy contains two methods: a condition method for triggering the policy, and an action method for effectuating the policy. The policy continuously checks how much time has passed since the last position was received from the GPS sensor. If this time is greater that a specified threshold it ensures that the GPS is powered on. This policy is associated with an annotation called `@PreventStale([threshold in seconds])`.

This kind of extension opens up for arbitrary ways of constraining the position delivery. Users of the middleware can keep private libraries policies tailored specifically to their deployment scenarios and they can use publicly available policies produced by third parties. For optimal reusability and collaboration the annotations exposed should be organized according to a commonly agreed upon ontology, e.g., based on the quality taxonomy presented in Paper VI.

Policies for managing application qualities can be implemented at several levels of abstraction. Some policies are high-level and are associated with very generic annotations, e.g., `@HighAccuracy` to ensure as high accuracy as possible, `@LowLatency` to minimize the time from the position changes until it is reported, or `@Secure` to indicate encryption of position communication. Other policies are more low-level and provide specific annotations, e.g., `@UseFallback` to allow the middleware to select secondary positioning methods if needed, `@RequireAccuracy(150)` for accepting a certain level of inaccuracy, `@FitToModel(name=BuildingA)` to force positions to fit a model,

e.g., snap positions to hallways of buildings, and `@PromptUser` to ask the user for permission before initiating positioning.

The down-side of using very generic high-level annotations is that the implementors are limited in their choice of tactics. On the other hand, the approach allows applications to remain oblivious to the details of the policy and rely on the policy to do the optimization. Because annotations - policy associations are configurable, high-level annotations can be associated with low-level policies. This allows applications to be developed for generic deployment scenarios and the quality management to be tailored the specific deployment. Furthermore, using high-level annotations enables the implementations of individual quality handling tactics to be separated, enabling these generic policies to encapsulate handling of a single quality. Within the implementations of these policies several tactics targeting that quality can then be implemented and the collaboration between them can be tightly controlled.

By allowing application developers to choose which policies they want active, they are effectively declaring what qualities they want improved. The possibility to create policies combined with translucency allow developers to extend the middleware with new functionality and make it available to application developers through the normal middleware API. Interdependency between tactics motivates the need for using the taxonomy proposed in order to describe to the policy engine how the policies affect each other and allow it to prioritize qualities. The following section presents an evaluation of the concepts presented here based on prototype development and student projects.

4.4 Developer experience

The PerPos middleware has been used as an experimental platform in a masters courses held by Aarhus University. The course contained a thorough introduction to both the model as well as the quality control mechanisms, i.e., tactics, policies and Features. The students were asked to implement a position based application for an Android smartphone. They were also asked to choose one or more qualities to improve by implementing tactics as policies using the PerPos middleware and then evaluate how much they were able to improve on the chosen qualities. Furthermore, the students were asked to reflect upon their experiences working with the PerPos middleware. The course projects and the reports provided by the students presents a great opportunity for evaluating the programming model and generic developer appeal of the various API elements of the PerPos middleware, and the underlying design philosophy. During the course, the students were provided with the following tools:

- Lectures on qualities and the use of tactics and policies in relation to position based application development.

- An implementation of the PerPos middleware capable of running on the Android platform.
- Introduction to the API and programming model provided by the PerPos middleware.
- An inspection tool capable of connecting to running instances of the application and visualizing the components and connections of the meta-model.

The applications produced by the students are all relatively simple in respect to use of the positions. While some of the applications are just collecting generic traces in a best-effort manner and presenting them on a map, other applications pose subtle variations in the intended use cases which directly affected the choice of quality improvement tactics. One application is targeted runners doing interval training workouts. For this application, the most important information for the user is the average speed in each of the individual "sprints" of a workout session. In another application, positions are only interesting when the user is stationary, e.g., for detecting points of interest. For the second application, the stationary points should be positioned with the highest precision possible, while intermediate positions when traveling are of no importance.

All the students chose power consumption as the quality they wanted to improve, and they all went for reducing power consumption by optimizing on the use of the GPS receiver. The primary reason for this choice was that battery power is such a limited resource on mobile devices and that the GPS receivers of modern smart phones are, by far, the most resource demanding sensor available. Also, the students had limited experience with the positioning domain and battery consumption is very easily quantifiable. Although all projects were, generally, focusing on the same quality, very different tactics and policies were applied. In the following, three representative policies are described.

One example of quality improvement from the student applications is a policy that realizes an instance of the "Duty Cycling" tactic by combining two sensors, a GPS receiver and an accelerometer based pedometer. The tactic relies on the less power demanding, but more inaccurate, pedometer until overall accuracy reaches an unsatisfactory level and then temporarily powers up the GPS until a "good" position fix is obtained. This tactic reduces power consumption considerably, but at the cost of accuracy.

A second example is a tactic specifically targeted the interval training workout application. The tactic frequency of the user's steps to determine when to power down the GPS receiver. The faster the user is moving, the more samples are required from the GPS receiver, thus power consumption scales with user speed. This tactic is realized by having a set of policies that each activate at certain predefined step frequencies and then adjusts the

sampling rate of the GPS receiver according to a motion model that has been experimentally determined.

A third, and very simple, policy for reducing power for the "points of interrest" application uses the step count from the pedometer to determine whether the user is moving or not. Depending on the state of the user, the GPS receiver is turned off when the user changes state from stationary to moving, and it is turned back on for the opposite transition.

In addition to letting students develop applications, we have also been creating several vertical prototypes during the development of the PerPos middleware. One example of a prototype used illustrating the utility of policies is a very basic "sports tracker" application capable of tracking jogging routes and measuring the distance traveled. The application is implemented using a naive approach; it simply collects every position from the built-in GPS and summarize the distance between each consecutive position. However, this approach is very inaccurate because of the relatively poor precision of GPS receivers. When traveling at low-speed (walking/running) with the GPS sampling positions at a rate of 1 Hz, positions are distributed around the true position, i.e., summarizing the distance between all reported positions will result in the total distance being significantly greater than the one actually traveled.

There are several solutions to the problem of summarizing distances. One being to reduce the sampling rate, another is to continuously calculate the average position. The point here being that with dynamically adjustable policies/tactics, several solutions can be tried and the one best suited the actual deployment can be chosen.

In the post course evaluation, the students were interviewed about their experiences working with the PerPos middleware and the concepts of the programming model. In general, these interviews showed that the students were very pleased that they were able to structure the code using Features and policies allowing them to separate quality control from application code. They pointed out that encapsulation of quality handling is a useful property to have available when developing position based applications and they were generally successful in using policies and Features to implement tactics.

While most of the groups used Features and policies as intended, there were a single group who circumvented intended flow of data from Features through component properties to policy or application, and interfaced directly with the Features. The primary reason for this choice was that the group judged that they did not need to make their solution generically applicable and were satisfied with having Features that were tailored to their particular application instance.

Another observation from the interviews is that even though the students had been introduced to the visual inspection tool, most groups resorted to adding custom inspection Features in which they could print specific values to a log or to the console. They defended this choice by arguing that the Feature

extension mechanism provided very easy access to the parts of the system they needed to inspect. However, in hindsight, some groups did admit that they would probably have avoided some obstacles had they used the inspection tool.

We also learned, based on the interviews, that because of the clear separation of concerns provided by the policy mechanism and the meta-model extension model, all the students were able to keep their applications very simple at the points where they interface with the PerPos middleware. Furthermore, it was confirmed that it is possible to use the exposed meta-model of the positioning process as a medium for manipulating the middleware. Also, it confirmed that quality improvement can indeed be achieved by implementing tactics and policies as described in this chapter.

In conclusion, although conducting experiments using students with little training in the positioning domain may not be the optimal evaluation scenario, we do believe that if a group of students are able to grasp the concepts presented here and produce significant quality improvements during a rather limited time, it is reasonably to expect that a group of trained professionals could do at least the same.

4.5 Discussion

As argued in Section 4.2 improving on an application quality most likely impose some level of trade-off for other qualities. Balancing the trade-offs when improving a quality requires some level of trial-and-error. The use of the PerPos meta-model enable an explorative development style where tactics can be developed iteratively and be easily evaluated and compared without changing the main application. In this section, the objective is to discuss how explorative development and separation of concerns are supported by the approach for managing application qualities presented in this chapter.

A core challenge developing position based applications is how to deal with the highly dynamic nature of the execution environment. The main reasons being that no two executions are exactly the same and that translating physical phenomena to digital information is inherently inaccurate. As a consequence, developers need tools that can be used to allow applications to adapt dynamically to changing environments. Although developing such adaptation behavior can be complex, being able to inspect and change applications running in actual deployments can ease the process. By using PerPos, the extendable meta-model enables developers to approach the development task in an exploratory manner, i.e., gradually adapting applications to the deployment environment. This form of development prompts for a more iterative process than normally favored. Based on the student projects and other prototypes developed using the PerPos middleware, we observed that developers used the flexibility of the meta-model and Features to “tweak” applications

to fit specific environments, e.g., take into account special signal reception conditions caused by buildings etc. In general, specific context information can often be exploited to improve the overall application quality when the deployment setting for an application is known.

Formulating how the information is used in the form of tactics provides a basis for organizing ad hoc tweaking. Using the policy mechanism in combination with the concept of tactics, tweaks can be encapsulated and parameterized promoting some level of reuse. Furthermore, as the PerPos middleware allows policies to be dynamically added and removed, several solutions can be compared and combined using the same basic application. Using policies for encapsulating tactics as argued in Section 4.3, the actual code artefacts realizing tactics can be effectively localized in modules which in turn decreases the conceptual overhead of managing qualities. This is supported by the experiences reported by the students who had little trouble creating and applying tactics using policies.

Developer support has been an important design goal for the PerPos middleware. It is important that the middleware is actually helping developers and not imposing artificial restrictions and limiting the expressiveness of the developer. A key factor in supporting developers is the complexity of the programming model. The declarative API described in Section 4.3 allows applications to be augmented with annotations that are interpreted by the actual component responsible for handling individual aspects of a quality. In general, using a declarative API has the clear advantage that developers only need to specify *what* the system should do and not be concerned with *how* it is actually done. This delegation of responsibility further motivate a high degree of localization which in turn is allows the PerPos middleware to delegate the decision of how properties of certain qualities are handled to policies created by to those who are experts on precisely those properties.

There is, however, a risk of introducing a new level of complexity by trying to isolate tactics into individual modules. As argued in Section 4.2, tactics may have significant influence on each other, this may be either intentionally or unintentionally. Some tactics are totally incompatible, while others can reinforce each others effects. These relationships cannot be statically determined because subtle variations within each tactic can change its side-effects completely. In general, though, being aware of the potential impact tactics can have on each other allow developers to take more informed decisions about how to apply them. Furthermore, if implementations of policies use an agreed upon ontology, a common network of policies with well-known side-effects may be built.

In conclusion, the flexibility provided by the PerPos meta-model and its extension model, allows dynamic control at a fine-grained level, while tactics, as a conceptual tool, enable developers to structure and organize specializations into modules using the policy mechanism provided by the PerPos middleware. In relation to explorative development there are a number of factors that en-

able developers to tailor position based applications to particular deployment scenarios. Functionality can be dynamically extended and changed by attaching Features to the exposed meta-model. As argued in Chapter 3, this extension mechanism allows developers to interact directly with specific subsystems of the positioning middleware and exploit subtle details to improve overall application quality.

Chapter 5

Relation to other positioning middleware

In this chapter the PerPos middleware and the concept of model based translucency is related to existing positioning middleware solutions and to reflective middleware in general. Using a middleware is by far the most common way of building position based applications, and there exists several successful middleware solutions within the positioning domain, both production ready commercial solutions and experimental research ones. In this chapter four canonical representatives is related to the research presented in this dissertation and to the PerPos middleware in particular.

5.1 The Java Location API

The de-facto standard for working with positioning on mobile devices is the JSR-179 [70], although, some device manufactures use their own variants[1]. The JSR-179 is not a positioning middleware *per se*, but it suggets a common API for implementations of middleware designed to be used on a single device. The JSR-179 provides a standardized API to perform integration and control of the positioning systems available to the device. This API is heavily inspired by the capabilities of common GPS devices. The JSR-179 uses a simple terminology based on the concept of *location providers*. A location provider represents a direct connection to an actual positioning system, e.g., a GPS receiver. Applications built on top of JSR-179 must explicitly request a location provider through the *location manager*.

Location providers are capable of delivering positions to the application either as on-demand or through event notification. However, the possible types of events are limited to periodic updates and simple proximity events

[1]Nokia's Symbian OS and Google's Android OS both implement the JSR-179 while Apple uses Core Location Framework [3] which bears a close resemblance to the JSR-179.

where the application can request to be notified if the distance to a specific location is below a specified distance threshold.

The JSR-179 is designed to handle positioning technologies in a device centric way. The design assumes that only the device on which the API is implemented should be positioned. However, there is no inherent limitation on the choice of positioning technology as the JSR-179 only specifies the external API. Therefore, developers implementing the JSR-179 have to manage all communication with the actual sensors. From the view of the application developer, the JSR-179 provides an abstract representation of positioning data that is modeled to support only the most common features. The API does not allow for custom data types or other kind of extension, making JSR-179 somewhat limiting with regards to handling heterogeneous positioning technologies. However, positions are represented as high-level data structures representing either simple coordinates or a symbolic location modeled as postal addresses. Therefore, as long as the underlying positioning data can be represented in this data structure limited support can be achieved.

The primary API provided by the PerPos middleware is capable of delivering the same functionality as the JSR-179. We have used the JSR-179 as a baseline to ensure that application developers were not limited when developing applications based on the PerPos middleware, however, the names of methods differ, and the use and provisioning of location providers is hidden by the declarative API as illustrated in Chapter 4.

A major difference between JSR-179 and PerPos is that, in the former, the access to low-level details of the underlying technologies is statically modeled in the API. For example, accuracy, precision and similar properties are explicitly included in the data structure representing a position. Furthermore, in the JSR-179, the exposed data structures contains explicit representations for a range of commonly used properties of GPS receivers, e.g., Horizontal Dilution of Precision (HDOP) and number of visible satellites. In comparison to the PerPos API, the JSR-179 provides no means for extending the functionality or the set of exposed properties.

Regarding control of internal behavior, the JSR-179 provides very limited support in the form of a criteria based provisioning mechanism, similar in design to the declarative binding provided by PerPos. When an application is to bind to a location provider in JSR-179, a number of statically defined criteria can be supplied. Examples of the criteria that can be set are: rough power consumption levels (low, medium or high), cost of service, maximum response time and horizontal/vertical accuracy limits. Implementations of the JSR-179 API are not required to guarantee that these criteria are satisfied. These criteria correspond directly to the way bindings in the PerPos middleware are created, only that in PerPos the criteria are specified as annotations instead of passed as parameters. However, while the JSR-179 is limited to exactly

eight criteria[2], the PerPos API is completely extensible through addition of new policies.

Because the JSR-179 is designed for complete positioning transparency, i.e., applications should require no knowledge of the underlying positioning systems, implementing tactics for managing position related qualities requires direct source code access whenever the tactic is not realizable in the application itself. If an implementation of the JSR-179 were to be extended with tactics, the API provides no mechanism for exposing any potential parameters that could be used to adjust the tactic's behavior, thereby prohibiting implementation of all but a few very generic tactics.

5.2 The Location Stack

Location Stack by Hightower et al. [42] is a generic software engineering model for location based systems in ubiquitous computing. The model is intended to be both a conceptual framework as well as a high-level layered architecture for implementing location based systems.

The architecture suggested by the Location Stack consist of six layers: Sensors, Measurements, Fusion, Arrangements, Contextual Fusion, Activities and Intentions. The layers provide increasing levels of abstraction, ranging from basic hardware wrappers in the Sensors layer to representation of cognitive desires of users in the Intentions layer.

The Location Stack assumes that its layers can be completely separated from each other by strict interfaces describing the data moving between them. This assumption is based on five design principles identified by the authors of the system. First, all measurements can be composed of a finite set of fundamental types. Second, measurements can be combined in standard ways. Third, object relationships can be expressed in standard queries. Fourth, uncertainty of sensor readings are preserved through the entire positioning process. Fifth, activities are the basic building blocks of applications. However, the only true implementation of the Location Stack, the Universal Location Framework (ULF), has shown that the fundamental principles of the model cannot be followed in practice [38]. According to the report on the ULF, actual location based applications tend to require some level of access to low-level details of the positioning process. In the Location Stack this translates to creating cross-layer functionality which breaks the fundamental assumptions of the model.

In [38], the authors of the ULF describe how they extend their middleware to be able to account for a number of technical details relating to specific sensors. The Location Stack does not promote any form of translucency, and,

[2]The exact criteria are: horizontal accuracy, vertical accuracy, preferred response time, power consumption, cost allowed, speed and course required, altitude required and address info required.

for example, integrating the error model of a GPS receiver requires full source code access and a number of changes to internal components. The approach the authors of ULF describe for integrating new sensors contains the same actions that would be needed extending the PerPos middleware, however, in PerPos, the extension is possible without direct access to middleware source code. With full access to the ULF source code, it provides a well structured framework for implementing complex quality improvements. Even very specific tactics that require information from several sensors and control over fusion processes is realizable. However, these tactics might break the conceptual encapsulation imposed by the Location Stack architecture. In general, the ULF extendable in many of the same ways that PerPos is, however, only by changing the middleware implementation. The extensions described by the ULF authors indicate that they employed an explorative development style that could very well have been supported by introducing a meta-model on top of the ULF. Because the internal structure of the ULF represents the position processing behavior, the ULF could probably be augmented with a meta-model and thereby provide its users with the flexibility of model based translucency.

5.3 Middlewhere

Middlewhere by Ranganathan et al. [88] is a general purpose middleware for building location based applications. The primary purpose of Middlewhere is to provide location information to applications in a technology agnostic way. The system provides access to locations through an abstract module named the Location Services Module, which supports merging of location information and various low-level information like resolution, confidence and freshness.

Middlewhere has a number of key features that are briefly described here. Firstly, Middlewhere supports multiple location sensing technologies which can be fused into an aggregated report for the application. The system takes into account the error profiles and location sensing resolution of the underlying technologies when fusing the observations. The report contains a spatial probability distribution of the tracked target. Secondly, Middlewhere associates a number of quality properties to each location sensor observation. Specifically, the temporal state of the observations are represented as a time-degradation function called freshness, where the overall quality of an observation is assumed to degrade over time. This information can be relayed all the way through the middleware and presented to the application layer through a query interface. Thirdly, Middlewhere can use both coordinate based as well as symbolic location models. However, these models are statically implemented into the system and cannot be extended by application developers. Fourthly, both push and pull modes of interaction is supported. The notification mechanism is more advanced than in the JSR-179 and applications can be notified based

on custom specified location based conditions, e.g., based on relations between spatial entities.

Although Middlewhere is not designed specifically for translucency, it does provide limited access to some internal information, namely uncertainty related to sensor readings. Sensors in Middlewhere are integrated into the middleware by implementing an adapter interface to encapsulate the sensor functionality. This provides Middlewhere with a homogeneous interface towards the hardware level and allows all sensors to be treated exactly the same internally. However, these adapters are required to specify an error model of the sensor that is used in all internal calculations to take uncertainty into account. This is a very static approach to managing a seam (sensor uncertainty) and it is limited to only *one* model of the uncertainty, however, the information is taken into account in all internal calculations and in that respect, Middlewhere exhibits characteristics of a seamful design. In comparison to PerPos, the basic positioning functionality does not include uncertainty internally when determining positions, however, if information about uncertainty can be extracted or calculated somewhere along the positioning process, it can be exposed by a Feature and be injected into appropriate intermediate calculations.

Looking at the potential for quality management, unlike PerPos, Middlewhere does not implement the possibility for dynamically changing quality optimizations. For developers with access to the source code of Middlewhere the internal architecture of the system provides structures for statically extending quality management functionality but these structures are not externally available. Middlewhere does, however, implement some static quality optimizations like taking accuracy and latency into account when calculating positions. In addition, spatial models are used in Middlewhere to eliminate improbable positions. With regard to quality management using context information, Middlewhere provides support only for positioning sensors and, therefore, other context sources are not easily integrated like they are in PerPos. Furthermore, manually implementing new tactics for managing qualities are not possible in Middlewhere. Although sensors sampling rate can be controlled from the application layer, elaborate sampling schemes like the one realized by EnTracked as presented in Paper III are not supported.

5.4 The PoSIM Middleware

To the best of our knowledge, PoSIM by Bellavista et al. [9] is the only other middleware for position based applications that is specifically designed based on an idea of translucency. The middleware is designed to provide application developers with different levels of visibility into the internal workings of the underlying positioning systems. The interpretation of translucency used in PoSIM is that it should be possible to bring properties associated with sensors up to the public API available to application developers.

The fundamental thought behind PoSIM is to provide two different APIs: a simple API that resembles the JSR-179 which the authors calls transparent, i.e., it provides transparent positioning; a smart API that the authors calls a visible API, i.e., it provides acces to internal features of underlying positioning system. PoSIM converts all position measurements into a generic XML format called a position report. These reports contain the aggregated position of all active positioning devices and they can be filtered by the application.

The PoSIM middleware is made up of three layers: a sensor wrapping layer providing hardware access, a policy and data management layer and an interface layer providing an API for application developers. PoSIM allows for extension of both the sensor and policy layers. Extension of the sensor layer consists of the possibility to write adapter classes for new sensors and defining sensor capabilities according to a pre-defined ontology. All adapters are controlled through two primitives, *features* and *infos*. Features are actionable methods, e.g., state changes, power switching etc. Infos are read-only values associated with the sensor. The policy layer can be extended by adding new policies written in a declarative language consisting of relational conditions that activate corresponding actions. The set of operations for conditions consists of simple comparison of data values provided by infos while actions are limited to passing values to features exposed by sensor adapters.

Unlike PerPos, sensor adapters are statically added to the middleware and they cannot be extended in any way. Where PerPos's Features can be used to expose internal behavior of a sensor using reflection, only the predefined functionality provided by adapters is available in PoSIM. Furthermore, PoSIM has no construct equivalent to the representation of refinement of the raw sensor values that PerPos has, meaning that the internal interpretation of sensor values cannot be manipulated in any way. All values generated by a PoSIM sensor is collected in a single report to be delivered to applications and only simple filtering is supported.

The policy framework provided in PoSIM can be used to implement tactics to manage qualities. As long as the appropriate information from sensors are made available through infos and sensors can be controlled, many of the tactics involving controlling power state and sampling rate can be implemented. However, unlike in PerPos, tactics that fuse values from several sensors together to create new aggregate values are not realizable using the PoSIM policy framework.

In general, the PoSIM programming model is kept very simple, while still allowing a great deal of flexibility. However, compared to PerPos, the expressiveness is limited, especially for extensions that exploit structural properties of the middleware, e.g., ad hoc sensor fusion, controlling one sensor based on output of another etc. Furthermore, PoSIM is designed strictly for single device applications, whereas PerPos supports integrating sensors from several devices, distributing sensor values across devices and between several applications.

Chapter 6

Summary

We use the meta-model of PerPos as a base to implement a mechanism for encapsulating tactics as generic policies that provide a convenient way of applying and adjusting tactics to a position based application. The policy mechanism allows application developers to activate policies that can react to changes in properties of the meta-model and apply specific functionality to realize an appropriate tactic. Furthermore, policies provide a method for parameterizing tactics.

Throughout this dissertation we have investigated methods for helping developers of position based applications through specialized middleware. By combining the benefits of a middleware and the power of openness provided by reflection, development of position based applications can be effectively supported.

6.1 Conclusion

Model based translucency and seamful design effectively combines the benefits of having a middleware mediate a heterogeneous and complex domain with the control of having system level access to the individual parts of the mediated system. The experiences gathered by developing prototypes based on the PerPos middleware show that using model based translucency, a heterogeneous domain like positioning can be mediated by a middleware solution without limiting the ability to specialize applications to particular deployment environments.

The combination of model based translucency and declarative policies provide a controlled environment for exposing low-level properties in through a high-level API. Shifting the mindset behind seamful design from end-users to developers have resulted in a number of design guideline realized by the PerPos middleware that allow application developers to tailor the middleware according to their specific needs. Furthermore, analyzing the positioning do-

main using a seamful approach, we have found that by allowing application level interfaces to be extended with declarative annotations code for handling individual seams can be separated into separate components. In addition, adaptation of the PerPos middleware can be created using only extension primitives of the meta-model, thereby making adaptations transferrable and independent of the base system.

By analyzing positioning specific qualities it is shown that the concept of tactics can be successfully transferred from generic software architecture to a specialized domain like positioning.

The combination of tactics and policies provide a framework for conceptualizing quality management and provide a high-level understanding of positioning related quality. Exposing a meta-model of the domain specific processes within a middleware is shown to provide a framework that enable third-party developers implement quality improving tactics in individual policy modules. By isolating tactics in modules, the complexity of the application level code can be lowered because applications can delegate specific customization and quality management to the middleware.

The evaluation of the PerPos middleware by a group of students show that, with a minimum of training, complex tactics could be implemented without causing changes in application code. The experiences of the students imply that the programming model of the PerPos middleware is not overly complex, and can indeed be used to produce high-quality position based applications. The exposed meta-model provides great flexibility which can be used to apply an exploratory development style. By being able to dynamically inspecting and adapting applications, development and customization of the positioning process can be conducted iteratively and applications can be "tweaked" in respect to specific details encountered during deployment.

In short, exploiting the heterogeneity of positioning technologies can be highly beneficial to end-users if application developers can adapt their application based on deployment specific context.

The strength of solutions presented in the dissertation lies in just declaring intentions with regard to what qualities should be optimized. In this way, realization of quality management can be delegated to the middleware. Furthermore, allowing application developers to configure these declarations enables more specialized quality tactics to be integrated into the middleware. A set of predefined declarations and policies can be made available. Because they can be dynamically added and removed, they do not need to be generic. Furthermore, the declarative api and the flexibility provided by translucency allows these tactics to be parameterized which allow for semi generic tactics, targeting different groups of environments, to be integrated into the middleware.

6.2 Future work

The contributions presented in this dissertation open up a number of interesting directions for future investigation.

With regard to the investigation of tactics and qualities in the positioning domain, more studies on how individual tactics and qualities affect each other can be conducted. Tool support for automatically managing interdependency between tactics is another area where there is a lot of potential providing even better development support. Moreover, further tuning of parameters in the current tactics is likewise relevant in order to gain more quantifiable parameters for the tactics. In addition, the list of tactics presented in this dissertation is by no means exhaustive and collecting more tactics and developing new ones is an obvious extension.

Another natural extension of the tactics investigation is to show how tactics behave in other specialized domains and determine how tactics behave as a generally transferrable technique. Still related to tactics, another possibility is to do further studies into the usefulness of model based translucency for improving "normal" software qualities in position based applications, e.g., performance, redundancy, reliability etc.

Distribution of the meta-model is another area that has had very little focus in the current implementation of PerPos. In its current state, the meta-model exposed in PerPos allows for rudimentary distribution. Communication between multiple model instances is supported using a simple custom protocol. This could be greatly improved by basing the distribution on an overlay that supports automatic discovery and management of heterogeneous networks.

In this dissertation, model based translucency has been applied specifically to the positioning domain. A natural next step is the application of the guidelines for model based translucency on other domains, e.g., virtualization environments or distributed systems.

Part II

Papers

Paper I

Model-based Translucency in Middleware: Supporting Seamful Development

Kari Rye Schougaard Jakob Langdal Jensen

Abstract

Traditionally, extensibility and adaptability in middleware is achieved through thorough design of the problem domain. The key variability points are modeled at design time to allow plugin of new functionality at different points in the system. Unfortunately, it is historically shown that the actual adaptability needs at some point differs from the ones predicted.

We argue that the seams, which are hidden in middleware with seamless design, should be accessible for pervasive computing application developers. It cannot be foreseen which aspects of a seam the domain or application need. We suggest that in addition to modeling the domain variability points the middleware should expose a model of the internal processing mechanisms of the middleware itself. The middleware should support modification of this model. Furthermore, at all places where the

Published as: Schougaard, K. R. and Langdal, J., Model-based Translucency in Middleware: Supporting Seamful Development. In *Proceedings of the 2nd International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, 2010 (M-MPAC'10)

middleware achieves the normal seamless use of the domain, the model should be adaptable and adaptations should be absorbed into the middleware.

We present model-based translucency as a middleware construction goal. The supporting arguments are given and examples of use of models in middleware are compared to the requirements for which we argue.

1 Introduction

No sensor is accurate and error-free and the connectivity of mobile devices is heterogeneous and at times lacking. These are the conditions for development of pervasive applications. Middleware for pervasive computing aims at hiding these difficulties, supporting the application developers in concentrating on the application logic. Unfortunately, inaccuracies, errors, and connectivity deficiencies will sometimes have consequences that cannot be hidden.

In this paper we propose model-based translucency as a middleware design method. In this we aim to extend the adaptability of state of the art of middleware with plug-in architecture. At the same time we do not endorse the full adaptability of generic reflective middleware. While powerfull, it is also difficult to understand and make correct adaptions in generic reflective middleware.

Model-based translucency might be taken to mean that a visualization is used for providing insight instead of code. Often graphical tools pride themselves of being model-based. However, we do not mean to imply that it is necessarily a graphical tool that supports the opening the middleware, but rather an explicit and operational representation of the central concepts and functionality of the middleware. This representation may then also be rendered in a graphical tool.

1.1 Contributions

The main contribution of this paper is a formulation of model-based translucency as a middleware construction method (Section 3). Although, arguments for using models have already been given as diverse places as in the early time of object-orientation, and for the use of architectural documentation, these have not been formulated explicitly for middleware for pervasive computing.

The proposition of model-based translucency rests on a formulation and argumentation for the challenge of supporting seamful design for developers in middleware for pervasive computing (Section 2).

Furthermore, we explain how middleware with model-based translucency supports seamful design for developers (Section 3.6). This has also been argued in [63], but the arguments in this paper are significantly more substantial.

Lastly, we point out existing examples of model-based translucent middleware (Section 4). By explaining how these examples fulfill the requirements for

model-based translucency we concretize the abstractly formulated proposition of model-based translucency.

2 Middleware for Pervasive Computing should be designed for seamful interaction

In Weiser's seminal paper: The computer for the 21st century [94], seamless design is presented as a goal for how computers should be integrated into the world. A seamless design will allow computers to disappear from our awareness. This will enable us to focus on the goal for which we use the computer, instead of focusing on the computer. Such seamless designed systems should make the computerized infrastructure components they depend on disappear from the focus of the user.

Unfortunately, the sensing technologies used for translating context to digital information are never perfect. These imperfections motivated several authors such as Chalmers and Galani [26], and Benford et al. [11] to argue for contrasting the goal of *seamless* design with one of *seamful* design. Seams are defined as "[...] the places where [components and technologies] may imperfectly connect to one another or to the physical environment." [11, p.126]. The goal of seamful design is to make the seams available in a designed manner, but not in focus at all times. Seamful design enables the user to "selectively focus on and reveal [seams] when the task is to understand or even change the infrastructure." [26, p. 251].

The seamful design may also be directed at developers [63], when they are the users, as is the case for middleware. Middleware for pervasive computing traditionally enables the developer to seamlessly access high-level information based on error prone sensor input and/or to (spontaneously) communicate with other devices, in the face of heterogenous networks. Therefore it is often argued that middleware for pervasive computing needs to be extensible. For example, enabling the application developer to define application specific contexts and how the detection of this context depends on the available sensors, or to specify application specific spontaneous communication patterns or communication protocols.

In continuation of this line of arguments, we argue that the middleware developer cannot foresee which aspects of a seam the application developers need to access to solve their problems. Domain or application specific demands for specific details cannot be uniquely determined at middleware development time. We therefore put forward seamful design for developers as a challenge for middleware for pervasive computing.

3 Expose A Model of the Middleware

Developers will need to switch between seamless and seamful interaction. We assume that, in general, middleware developers cannot foresee which seams or which aspects of the seams are needed in a certain application, this means that the middleware must be adaptable. The developers must be able to change the middleware and expose a seam that is usually seamlessly handled by the middleware.

It is generally agreed that absorption of changes to the meta model in generic reflection is a powerful tool and sometimes too powerful. Generic reflection allows a developer to change what is not fully understood and thereby create havoc in the middleware.

Therefore, we argue that although generic reflection will allow a developer to work with seams we need to be able to change the middleware to access seams in a more controlled fashion. Using controlled interaction means that the developer will be restricted in the possibilities for adaption. On the other hand it is easier to understand what is changed and this will result in fewer errors.

We propose to expose a model of the runtime processes of the middleware as a method for constructing middleware for pervasive computing that supports seamful interaction for developers using the middleware. In the following we will present and argue for requirements for the the middleware and the model it exposes:

- The middleware should make the design of the model tangible.
- The middleware should support inspection of the model and the model should incorporate levels of details.
- The middleware should support manipulation of the model.
- The model should contain points for extension and adaptation.

When these requirements are fulfilled we argue that

- An exposed model eases adaptation of the middleware compared to adaptable middleware based on generic reflection.
- The middleware supports seamful interaction

3.1 Tangible Design

The central idea of model-based translucency is that an explicit and operational representation of the central concepts and functionality of the middleware is provided. This model should not be a clever user interface that hides the complexity of the middleware, but a true representation of the central structures of the middleware. The obvious way to do this, is to base

the model on reification of the central components and processes in the middleware. This resembles the approach of Balz et al. [5], who embed a state machine model in the base system code. When being a state machine is the primary structure of how the middleware carries out its tasks, using state machine tools to support the user in understanding the middleware is a good example of a tangible design.

Reification of all components or objects in the middleware would also provide a model of the middleware. Component based reification is known from the middlewares based on Fractal [21] or open ORB [15], for example GREEN [92], and for pervasive computing applications in general [45]. Other approaches to reflect all of the structure and the details of a pervasive application include the hierarchical graphs of [36]. However, not all components or objects of a middleware will be part of the primary runtime process in the middleware.

We argue for the importance of the developers' reflection and the retention of their understanding of what the middleware does. A parallel is the desirability of having an architectural model view of a system when modifying the system, compared to making the same modification only equipped with an automatically generated class or component diagram.

3.2 Inspectability

We envision that developers of pervasive computing applications based on middleware for pervasive computing will wonder how the seamless use of sensor input works and how the currently available communication means influence the described application. If building on a generically reflective middleware, model-based inspectability can be implemented by interfacing with the provided introspection facilites. In this way relevant information from the base system can be propagated to the model and made available for inspection by the developer.

Cases will arise where the application logic seams sound, but the concrete running application instance behaves unexpectedly. For an example based on an actual need to adapt a positioning middleware reported in [38], many GPS devices continue to provide location reports, by extrapolating on the last known movements, even when all satellite tracking has been lost. These reports are often erroneous, especially when a new direction has been taken - as is often the case, when entering a building. A developer with an application that switches to WiFi based positioning when indoors, based on GPS measurements being unavailable here, may experience the inaccurate positioning at times when a building has just been entered. In this case the developer will want to examine what goes on in the processing of the sensor readings. If the middleware processes are inspectable at runtime, it is easy to discover that the GPS keeps providing seemingly valid positions and reason that it might do so in spite of having lost sight of most or all satellites.

Because the capacity of the human mind is limited, we advocate that several layers of abstraction are used when possible. A very abstract view of the ongoing processing in the middleware, will allow the application developers to get an overview of the full system. If this abstract layer is inspectable, the developer may then retrieve more details of certain parts.

3.3 Manipulate-ability

The middleware should provide handles for manipulation of the model. The manipulations should be absorbed by the running middleware. For example when the middleware allows for variation in functionality, the user should be able to substitute one processing entity with another. Manipulate-ability is provided by the ability to reconfigure that many reflective systems provide. Model-based manipulate-ability can be achieved similarly to self-healing restricted by a description of the architectural style of the system [29].

Concretely, a distribution middleware could provide various protocols for message delivery. The model of the middleware should support the user in exchanging one protocol with another. The middleware should then apply the substitution on the running software. The manipulation of the model is envisioned to happen in a predesigned-way. This is parallel to providing a plug-in architecture, where the middleware developers beforehand have developed a list of plug-ins and allow developers to supply more.

Another kind of manipulation could be to in- or exclude input from available sensors. This would allow a developer to explore application functionality with or without specific context sensors. For example, do the detection of the "Entering the kitchen" fail when the input from microphones in the room is excluded.

3.4 Adaptability

We define seams as *the places where components and technologies may imperfectly connect to one another or to the physical environment.* It is at these imperfect connections that a middleware aiming for seamless use will make improvements and hide information. And, the middleware components providing seamlessness should be points for extension and adaptation in the model.

Pervasive computing applications may be characterized as requiring either context awareness, spontaneous ad hoc communication, or both [98]. In a middleware that provides high-level context information based on processing of sensor input, it is specifically the connections between components and the physical environment that is of concern. It is thus the middleware components that extract high-level information from sensor input, which should be adaptable.

In a middleware that provides spontaneous ad hoc communication components that search for devices and initiate communication, and components that repair communication errors should be adaptable.

3.5 Ease of Adaptation

The exposed model will guide the developers' understanding of how the middleware works and thereby improve their ability to extend the middleware. By exposing a model of the processing in the middleware, the developers' understanding is supported by giving insight into the core structure of the runtime processing of the middleware. This insight is limited. A general reflection mechanism would allow full insight, but also improve the possibility that the developer would be sidetracked into unimportant details when exploring how the middleware works on a running application. When exposing a model, the middleware developers guide the application developers' exploration. The middleware developers, who supposedly are experts in the domain, thus explicitly points out where in the middleware the normal seamless use is achieved. These are the points where it would be conceivable that a future user of the middleware would need to adapt the middleware to reveal aspects of the seam.

3.6 Support of Seamful Development

When the goal is to support seamful interaction by developers, it is not the core functionality of the middleware that should be adaptable - a middleware for pervasive computing should not be changed to a distribution middleware. It is the hidden information that should be made accessible; knowledge of what details are abstracted away and ability to make specific details available in the middleware API.

When the model is constructed as proposed above, it is exactly in the places where the seamlessness of normal middleware use is achieved that extension points are provided. It is thus almost a tautology that model-based translucency supports seamful development. Nonetheless, we will also argue for this with reference to an example of seamful development.

An example of support for seamful development (based on a real example of an adaptation of a middleware reported by [38]) can be illustrated through an application relying on positions from a GPS device when traveling outside and positions from a WiFi system when indoor. The application might exhibit odd behavior shortly after the tracked entity enters a building because of technological limitations of the positioning devices. Often GPS devices provide additional information of the circumstances for obtaining the current position. A typical design decision in positioning middlewares is to hide these details behind high-level abstractions. However, if the component which extracts the position from the messages from the GPS device is adaptable extra information can be extracted and provided at a higher-level. The application developer

may then access additional information of the seam between the GPS sensor component and the high-level positioning provider, thus working seamfully with the GPS sensor.

4 Model-based translucency in existing middlewares

We hope the arguments above have convinced the reader of the case for model-based translucency in middleware. However, this is all very abstract and in the following we will concretize how model-based translucency may be carried out by explaining how existing middleware solutions exhibit some of the characteristics of model-based translucency.

A middleware for controlling the Media Access Control (MAC) layer protocols in wireless sensor networks is presented in [57]. The middleware provides developers with a model of the MAC layer in which both the structure and the functionality of the protocol can be controlled. The middleware is not designed explicitly for model-based translucency. Nonetheless, some of the requirements for model-based translucency are fulfilled.

A tangible design is achieved in that core process details are exposed for the protocol the middleware is designed to carry out. By giving a visualization of the model, some degree of inspectability is also achieved. Moreover, the model can be manipulated by the user, alternative steps in the protocol can be specified and the structure can also be redefined.

In the PerPos positioning middleware presented in [63] a key design element is a model of how positions are calculated internally based on sensor readings. This model is exposed to application developers with handles for adding new functionality to the processing at several different levels of detail.

The middleware is designed as lines of processing components that carry out steps in the processing of position sensor input to high-level positions. A model of these processing components and their connections is exposed, thus making the middleware design tangible. The model can be inspected in a graph visualization, and levels of details is supported in that the processing steps in one source to sink process is abstracted to a channel in a higher level of the model. All processing steps can be adapted by methods supporting code injection before and after the processing component carries out its functionality.

5 Discussion with State of the Art

In the discussion of how seamful design for developers may be solved using existing middleware, we present two categories of work: generic reflective middleware for pervasive computing and other configurable middleware for pervasive computing.

Seamful interaction will be possible in a generic reflective middleware for pervasive computing. As all parts of a generic reflective middleware are adaptable, there will also be access to adapting the places in the middleware where the usual seamless interaction is supported.

We have only found one example of a generic reflective middleware for pervasive computing: GREEN [92]. GREEN is a configurable publish-subscribe middleware that features the ability to configure and reconfigure the middleware for the heterogeneous and changing environments of pervasive computing. Especially, for working with diverse network types. Concretely, the publish-subscribe interaction and event broker overlay can be configured by plug-ins, e.g., topic based in the first case and probabilistic multicast in the second.

GREEN employes the notion of component frameworks to manage and constrain the scope of reconfiguration operations. The GREEN architecture: hierarchically composed component frameworks, is a model of the middleware. Even a model that features layers of details, because of the hierarchical composition. That is, the arguments for using hierarchically composed component frameworks may be taken to advocate using a specific way of modeling the structure of the middleware.

GREEN is build with a component model that features generic reflection. For structural reflection this means that it features "a complete reification of the program currently executing" [58, p.34]. A developer may thus inspect or change the functionality at all points. Model-based translucency advocates that only specific places, where the seamless normal use of the middleware is achieved, are open for modifications. This will limit the possibilities of adaptation, from being able to adapt all aspects of the middleware to only being able to change how the middleware handles seams. This limitation comes with improved simplicity of making these changes.

In the second category, configurable middleware for pervasive computing there are many examples, e.g. [9, 23, 95, 98]. Most middleware for pervasive computing allows the user or administrator to enter rules, event definitions, or the like in order to specify context relevant for the application in question. In this discussion, we have chosen to concentrate on one example in order to giver a fuller explanation.

Cascadia [95] is a challenging example. Cascadia is a middleware that allows specifying, extracting and managing high-level events from raw RFID data. In the sense that most relevant seams are accessible in Cascadia, it does provide seamful design for developers, however the quality of this support does not compare to model-based translucency.

Cascadia is simple in terms of only depending on one type of sensor, thus there is only one seam between middleware components and technologies giving input about the physical world. Furthermore, it is to a high degree an open middleware. It allows the user or administrator to define new high-level events, but it also allows an administrator to discretize the building into

places, to tweak or replace the motion model and the sensor model in the particle filter used for representation of the distribution over possible positions of a tag, and finally to define a confidence table for controlling the probability of an event instead of letting it be a composition of the probabilities of the low-level events it is composed of.

Apart from the configurable parts, the seamless use of the place of tags also depends on the positions of the particles in the particle filter being hidden - only the defined places are visible - and that it is a particle filter Cascadia use for representing the uncertainty of the place of a tag. However, it may be a minor point that these places for obtaining seamlessness is not open for adaptation.

Still, there is a qualitative difference in how Cascadia supports seamful design for developers and how it is done in model-based translucency. Cascadia does not - to our knowledge - enable a user to explore how the current sensor data results in the the current high-level events. It lacks an exposure of how the middleware provides the high-level events. Because of this lack of support the developer might not understand how the middleware works internally, and therefore not be able to use the right adaption point. Cascadia has been carefully designed with extensibility points for adjusting the behavior. Nonetheless, it would be valuable to expose a model of the internal processing mechanisms of the middleware, because it cannot be foreseen with which seams a developer will want to work seamfully.

Apart from the categories explored above, it may be argued that relevant seams should be exposed in the middleware as a standard instead of depending on developers (middleware tailors) to adapt the middleware. However, we argue that exposing all relevant seams will not be achievable in a relatively new application area as middleware for pervasive computing. The area is still under rapid development and new kinds of sensors and types of context aware-applications are emerging.

In an older and more mature area most wanted seams may be well known. However, pervasive computing is a heterogeneous area and the different application areas depends on different kind of contexts - and thereby different technologies and sensors. This is in line with the arguments for configurable or extensible middleware for pervasive computing [98, 95]. Therefore it varies with which seams a developer wants to interact seamlessly and seamfully. Exposing all possible seams as a standard in the middleware, may make the middleware too heavy and clutter the API. This would be a seamful middleware, and as the common goal for middleware is to support seamless or transparent interaction we know of no examples of seamful middleware. We do not advocate for seamful middleware, but that the middleware should support the developer in changing between seamless and seamful interaction.

6 Conclusion

We have proposed a middleware design philosophy aiming at supporting application developers in switching between seamless and seamful development. Model-based translucency continues and extends the line af configurable middleware for pervasive computing. In addition to the extension points related to the domain of the application, which are often available in middleware for pervasive computing, we argue that the internal process in the middeleware should be modeled and available for extension and adaptation.

Seamful development would also be possible in a generic reflective middleware, but we argue that adaptation points in a model of the middleware are easier to understand than when all objects or components of the middleware are adaptable. The improved understanding means that less errors will result from concrete adaptations. In spite of the restrictions in adaptability in comparison to generic reflective middleware, we argue that model-based translucency in middleware do support the developer in unforeseen access to details related to seams between components or between components and the physical world.

6.1 Research Challenges

The first and foremost research challenge emerging from the proposition of model-based translucence is, indeed, to develop a middleware for pervasive computing that follows this design philosophy and evaluate how well it suports seamful development and also how its qualities compare to other middlewares for pervasive computing.

A pervasive application will often be distributed over several devices. A challenge for model-based translucency is how to handle for example timing when the model contains information from distributed sources.

There is a performance penalty in maintaining a model of the middleware. Research need to be directed at how to manage this performance degradation. A possibility may be only to generate the model on demand. This would mean that the performance loss would only occur when a developer is inspecting or managing the running system.

Acknowledgements

We thank Tony Gjerulfsen and Jesper Wolf Olsen for interesting discussions of the importance of supporting inspectability and adaptability of software. The authors acknowledge the financial support granted by the Danish National Advanced Technology Foundation under J.nr. 009-2007-2.

Paper II

PerPos: A Translucent Positioning Middleware Supporting Adaptation of Internal Positioning Processes

Jakob Langdal Jensen Kari Rye Schougaard
Mikkel Baun Kjærgaard Thomas Toftkjær

Abstract

A positioning middleware benefits the development of location aware applications. Traditionally, positioning middleware provides position transparency in the sense that it hides low-level details. However, many applications require access to specific details of the usually hidden positioning process. To address this problem this paper proposes a positioning middleware named PerPos that is translucent and adaptable, i.e., it supports both high- and low-level interaction. The PerPos middleware provides translucency with respect to the positioning process and allows programmatic definition of application specific features that can be applied to the internal position processing of the middleware. To evaluate these capabilities we extend the internal position processing of the

Published as: Langdal, J., Schougaard, K. R., Kjærgaard, M. B. and Toftkjær, T., PerPos: A Translucent Positioning Middleware Supporting Adaptation of Internal Positioning Processes. In *Proceedings of Middleware 2010 ACM/IFIP/USENIX 11th International Middleware Conference*, Bangalore, India, November 29 - December 3, 2010.

middleware with functionality supporting probabilistic position tracking and strategies for minimization of the energy consumption. The result of the evaluation is that using only the proposed capabilities we can, in a structured manner, extend the internal positioning processing.

1 Introduction

The development of location aware applications benefit from a positioning middleware. A number of these exist. However, existing positioning middleware has shortcomings in their support for extending the middleware functionality and inspecting the positioning mechanisms. The problem is that although location-aware applications often need a neat position, with all technological details and sensing uncertainties hidden away, often access to these details are needed. For instance, for improving positioning using probabilistic tracking [41], visualizing the positioning infrastructure [81], minimizing energy consumption of location-aware applications [53] or adding high-level reasoning based on machine learning [100]. Therefore, a positioning middleware that gives a structured cross-level access to the positioning mechanisms is needed.

Imagine a simple location aware application that shows the current position as a point on a map when outdoor and highlights the currently occupied room when within a building. It may be implemented on a mobile phone, using the internal GPS receiver and WiFi-signal strength measurements, and interfacing with a server containing an indoor WiFi positioning system [53] and a location model service for translation of the position to a room number. A positioning middleware is used to encapsulate the positioning systems, the location model service and the conversion between various coordinate systems. The positioning process for the example is shown in Figure 1.1. Now, maybe it turns out that the positioning is not accurate enough. The developer wants to improve the positioning by probabilistic tracking implemented as a particle filter [41] that takes into account the likely user movement specific for the application, and location models to impose restrictions on possible movements in the environment. The following requirements for a positioning middleware can be derived from this example.

- Adding a new kind of positioning mechanism and use this in the middleware, without changing the interface, on which the application using the positioning middleware relies.
- Allowing low-level access to the currently employed positioning mechanism and inspection of the process behind.
- Allowing extension of the provided functionality at steps in the process that leads to the production of a position.

Given a middleware that fulfills these requirements a particle filter can be inserted as a new kind of positioning mechanism, without affecting the

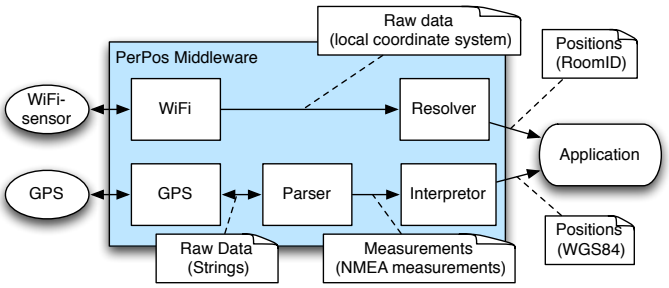


Figure 1.1: Concrete positioning processes for the example Room Number Application.

high-level functionality and API of the middleware. Necessary functions of the particle filter, e.g., for calculating the likelihood of sensor readings can be implemented by accessing low-level sensor information and exposing it at the correct step in the positioning process. This also enables developers to address many of the timing issues associated with combining multiple sensor readings to one measurement, which is further complicated by sensors with different output frequency.

Particle filters are only one example of applications that require a middleware that fulfills these three requirements. Generally, access to low-level information and the ability of inspection and extension is needed to visualize the positioning infrastructure when authoring location-aware applications [81], manage sensors to minimizing energy consumption [53] or to structure the reasoning process when determining transportation mode of a target by segmentation, feature extraction, decision tree classification and hidden-markov model post processing [100]. When designing a middleware it is virtually impossible to foresee all the features that will be useful in the future. Therefore, it is desirable to be able to extend core middleware functionality.

The first requirement of adding a new kind of positioning mechanism and using this in the middleware, without changing the interface has been fulfilled by existing positioning middleware: MiddleWhere [88], the Location Stack [42], and PoSIM [9]. Although some architectural issues remain, e.g., in the Location Stack the particle filter may be plugged in as a new kind of sensor. However, this positioning middleware use a layered architecture, with sensors in the first layer, adaptation of sensor input to a common representation of measurements in the second, and a fixed reasoning engine for multi-sensor fusion in the third. This means that the new complex sensor, which incorporates sensor fusion, will violate the architecture of the middleware as also argued by Graumann et al. [38].

In connection with the second requirement positioning middleware, such as MiddleWhere [88] and the Location Stack [42], expose a common representation of position information and uncertainty for all kinds of sensors. This

means that they do not support accessing information that is not part of the interface or the process behind. PoSIM [9] allows the user to specify control mechanisms and information that the positioning technologies must or may expose. This enables access to low-level information, however, it does not provide the application developer with access to the positioning process.

With regards to the third requirement, the layered architecture of the Location Stack inappropriately restricts possible extension points and has architectural issues as argued for above. MiddleWhere [88] and PoSIM [9] do support that new functionality is specified and implemented in sensor wrappers but not that new features are attached to the position information at a higher level or a later stage in the processing. In order to do this, the process that lies behind the construction of a high-level position must be exposed by the middleware as provided by the PerPos middleware.

The PerPos platform is a middleware for pervasive positioning that can be leveraged when building indoor and outdoor location-aware applications. The services provided by the middleware range from specific utility services to application components that can be deployed in several ways. To provide translucency and adaptation the PerPos middleware is designed around the central idea of representing the steps of the actual positioning process explicitly as a graph based on the flow of information from sensors to application code. This representation constitutes a reflection mechanism [58] that allows application developers to control and extend the positioning process and for the design to fulfill the three requirements stated above. We do not provide the functionality of a generic reflective middleware, and in Section 4 we argue that careful design of what is exposed through reflection decreases the conceptual overhead involved when developers perform adaptations.

1.1 Contributions

In this paper we present our positioning middleware: PerPos. We concentrate on how the middleware fulfills the three requirements stated above, in short, supporting plug-in of complex positioning mechanisms and allowing structured access to and adaptation of the actual internal positioning process.

- We present our multi-level abstraction of the position processing and explain the programming model it provides for location-based application developers (Section 2).

- For three examples: detecting unreliable GPS readings, a particle filter for position improvement, and a power reduction scheme, we explain how we have implemented them by using the adaptation programming model of PerPos (Section 3). These examples are provided as proof of concept for the proposed positioning processing abstractions and programming model.

- In order to compare our solution with others, we analyze what would be needed to implement the examples in existing positioning middleware (Section 3). In comparison with existing middleware designed for transparent use, PerPos allows adaptation of the positioning process without access to the code. In comparison with existing translucent middleware PerPos supports timing information and control of the positioning process itself.

- We introduce the concept of seamful design for developers (Section 4). We explain the needs for a translucent and adaptable middleware for positioning and how the supported programming model make PerPos fulfill these needs. We discuss how this relates to the concept of seamful design, and argue that the seamful metaphor is usefull for developers of translucent sensing middleware.

2 Design of Layered Reification and Adaptation of Position Processes

Generally, positioning middleware encapsulates the processing of sensor measurements that is necessary to obtain a position in a technology independent format. The PerPos middleware is designed around the central idea of representing individual steps of the actual positioning process explicitly as a directed acyclic graph based on the flow of information from sensors to application code. Nodes in this graph represent the implementation of processing steps and are called Processing Components. Edges in the graph represent the data that flows between components. The notion of explicit representation of processing graphs has previously been applied in a number of other domains, e.g., Solar [27] for generic context fusion, PAQ [87] which supports generic queries over temporal-spatial data and PCOM [8] which uses a component graph to compose behavior.

The PerPos positioning middleware uses the graph representation to support inspection and adaptation of the positioning process by exposing the processing graph to developers. The graph is exposed as a tree where data is traveling from leaf nodes toward the root. The root node represents the application that is receiving position data and the leaf nodes represent actual sensors. Internal nodes represent discrete processing steps. Branching (or merging if viewed in the processing direction) in the tree occurs when position data from several sources are combined. Usually, combinations of data from several sources take place in special sensor fusion components which often is a part of positioning middlewares [88, 42]. However, it may also take place in other high-level data reasoning components that also take into account other kinds of information, e.g., context information, user input or physical constraints based on building models. In Figure 1.1 we see an example of two

linear trees connected to the same application providing it with WiFi data and GPS positions.

The PerPos API exposes the processing tree to developers through three levels of abstraction providing increasing control of the positioning process. The levels constitute three different views on the positioning process as it is implemented internally in the middleware. The first is the positioning layer providing the abstractions of a traditional positioning middleware. The next two layers provide inspection through reflection on two different levels. The reason for splitting this functionality into two layers is to minimize the complexity involved when using a general reflective programming style. Therefore, the second layer provides access to an abstract structure of the underlying positioning process. In many cases the information in this layer will be sufficient to understand, e.g., the component composition that produced a position, thereby avoiding the added complexity of the more detailed layer. The third layer is responsible for reifying the actual positioning process as a tree structure and maintaining a causal connection between the positioning system and the tree. In Figure 2.1 we see how a processing graph is represented at the three levels. The configuration shown in the figure is from an application which incorporates a particle filter aggregating measurements from a GPS and a WiFi sensor.

The primary interaction with the PerPos middleware is through a traditional positioning API associated with the top-most layer in Figure 2.1. It supports both push and pull semantics for retrieving position-based data as derived based on input from connected sensors. The structure of the API resembles the JSR-179 [70] where applications can request a location provider which matches a set of criteria. Position-based data can then be obtained through this location provider in a technology transparent way. The API provides operations for specifying functional requirements for the location provider, retrieving position-based data, e.g., a WGS84 position, a room number or the k-nearest targets and setting up location related notifications, e.g., based on proximity to a point or target etc.

At the second layer the API provides access to an abstract structure of the underlying positioning process presented as a tree consisting of three basic node types and the processing channels connecting them. The nodes are either: data sources, components that merges data sources, or the root node representing the application. The API provides operations for inspecting the data flowing through the processing channels as well as handles for changing the functionality of the channels. The data processing channels provide a high-level extension model that allow application developers to implement algorithms that reason about the data delivered to the application.

At the third and most detailed layer the application has access to a detailed processing graph representing each processing step of the positioning system. At this level the API supports fine-grained control of both the structure of the positioning process and its internal behavior.

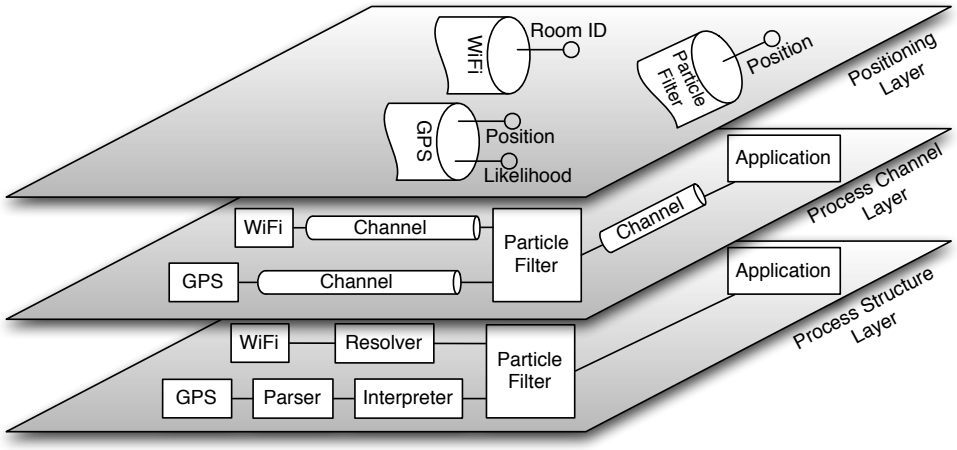


Figure 2.1: The three levels of abstraction on the positioning process provided by the PerPos middleware.

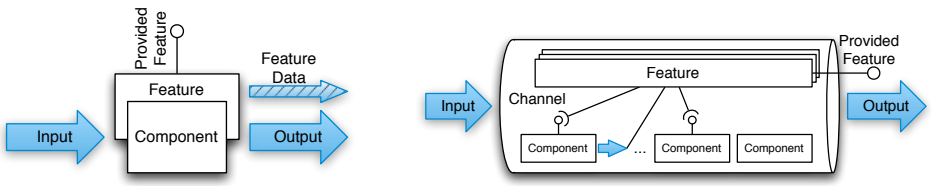
In the following sections each layer exposed by the PerPos API is presented in more detail. The layers are presented in increasing order of abstraction level of the provided concepts, starting with the most detailed layer.

2.1 Process Structure Layer

The layer exposing the structure of the positioning process, the bottom layer in Figure 2.1, is called the Process Structure Layer (PSL) and represents the most detailed level of interaction provided by the PerPos middleware. This layer is responsible for reifying the actual positioning process as a tree structure and maintaining a causal connection between the positioning system and the tree. Each node in the tree is a Processing Component that acts as either a producer or a consumer of data contributing to the positioning process, or both.

Applications can manipulate the composition of components in the tree through the API of the PSL, e.g., `insert`, `delete` and `connect`. Furthermore, the API allows applications to extend the tree with new components and augment existing components with new functionality.

Processing Components consist of three main elements: input ports, output port and implementation of functionality. A Processing Component has a single output port and may have multiple input ports. Input ports are connected to output ports of other components. These connections are established either by direct calls to the graph manipulation API, based on explicitly defined system level configurations or through dynamic resolution of dependencies between components. To make sure that port connections are realizable Processing Components must declare requirements for input ports



(a) Component feature. (b) Channel feature.

Figure 2.2: The two kinds of extension features in the PerPos middleware.

and define a set of provided capabilities for output ports. When extending a processing tree with new components developers must specifically declare these requirements and capabilities. As custom components are added to the PerPos middleware the dependencies are resolved and when satisfied the components are added to the processing graph appropriately and the classes implementing the Processing Component functionality is instantiated. The PerPos middleware provides the concrete implementation with access to a set of input ports as well as a reference to the output port to which it should deliver data.

The PSL API supports inspection of the reified processing graph including access to all methods available on the implementing classes of the Processing Components. Both the behavior of the Processing Components and the set of available methods can be modified by attaching what we call Component Features to them. Component Features are small code modules that can hook into a component and augment it in three ways. Firstly, data can be manipulated when flowing into or out of the component. Secondly, additional data can be associated with the data flowing out of the component. Thirdly, component state can be read, exposed and manipulated. Figure 2.2(a) illustrates a Processing Component with a Component Feature attached. The input requirements of Processing Components also include a listing of any Component Feature that the component is dependent upon. In the following we explore the dynamics of each of these augmentation types.

Changing Produced Data

A Component Feature can intercept the flow of data before and after it enters the component to which it is attached. This allows the Component Feature to effectively control the external behavior of the component. Whenever data is sent to a component the middleware calls the `consume` method on every Component Feature attached to the component which allows the Component Feature to alter the data before it is delivered to the component. The same process is repeated for outgoing data where the `produce` method of the Component Feature is called allowing for alteration of data. Note that this type of extension cannot change the data type of the data produced.

Adding Data

In addition to altering the data produced by the component, a Component Feature is able to provide new data that may be based on both input and output of the component. A Component Feature can call the method `produce(data)` on the component to which it is attached. This will result in the data passed to the method being propagated through the processing tree as if it were produced by the component itself. When adding data the capabilities of the output port is changed to include the new type of data. The generated data is only propagated through the processing graph if the next component in the graph explicitly declares that it accepts input from the Component Feature.

Changing Component State

Lastly, a Component Feature can add state manipulation and inspection functionality to individual Processing Components. When doing this, the component will to its surroundings appear to implement the functionality provided by the feature. Examples of this kind of extension are features that expose internal state of a component like various threshold levels used or provide access to changing parameters of component implementations. The application developer can create complex high-level functionality by combining the ability to traverse the nodes of the processing tree with this kind of state manipulation features.

2.2 Process Channel Layer

The middle layer is called the Process Channel Layer (PCL) and it is a view of the position processing where only data sources and merging processing components and the data-flow between them are represented. Thus, the PCL allows inspection of the positioning process in terms of the major processing components. In many cases the information in this layer will be sufficient to understand the component composition that produced the position, thus avoiding the added complexity of the PSL. The process is presented as a tree structure where the application is the root and the nodes are Processing Components representing either the originating data source or components that merge input from two or more data sources, effectively becoming a data source itself. The connection between components in the PSL are called Channels and encapsulates the positioning process taking place between its end points. An example of the channel abstraction is visualized in the middle layer of Figure 2.1. Channels are dynamically created when the PerPos middleware assembles the Processing Components involved in the positioning process. The PerPos API supports inspection of the Channels and the methods they provide and Channels can be extended through the use of what we call Channel Features similarly to the way that Processing Components are extended through Component Features. Channel Features are used to add

functionality to a Channel that requires access to data at different stages in the positioning process, especially, functionality that cannot be achieved by connecting to a single Processing Component. Figure 2.2(b) shows how a Channel Feature can depend on several internal elements of a Channel. The functionality of a Channel Feature is often partly decomposed into Component Features which the Channel Feature then depends on. A Channel Feature declares its input requirements and output capabilities. Input requirements may include Component Features, Channel Features, and Processing Components. Output capabilities may relate to the data produced by the Channel or to the Channel itself. From the perspective of a Processing Component or from the application a Channel Feature is semantically equivalent to a Component Feature attached to the last Processing Component of the Channel.

To support extension a Channel groups the output of every internal processing step into logically coherent groups. For each data element produced by a Channel it collects all intermediate data elements that logically contributed to that element and places them in a hierarchical data structure. This grouping is achieved by having a notion of logical time that relates to the data process of an entire Channel. Data will always flow from the source through the processing graph until the Channel produces a result. Therefore, it is possible for the Channel to assign a logical time unit to every layer of the processing tree that can be used to identify which processing steps are contributing to the final output of the Channel. For each logical time step the Channels registers all corresponding data produced by the Processing Components in the Channel in a tree structure representing the logical chronology. An example of a data tree for the GPS-channel is presented in Figure 2.3. In the figure the data is presented as tuples with three elements: the data, the logical time of the current layer, the time range of the data used to generate the element. The example shows data produced by the GPS sensor, the Parser and the Interpretor components respectively. In the example several strings from the GPS sensor is needed to produce one NMEA[1] sentence, and the first NMEA sentence did not contain a valid position, therefore another is needed before the Interpretor produces a WGS84[2] position.

A Channel Feature is required to implement the `apply(dataTree)` method and update its internal state when it is called. The method is called by the middleware every time the Channel delivers a data element. Through this method the Channel Feature has access to the concrete data tree that was used to produce the Channel output. The exact structure of Processing Components in the Channel is not known at implementation time. Therefore, the feature must handle the complexity of not knowing for example the number of layers in the data tree or the number of data chunks of each kind. For

[1] NMEA or National Marine Electronics Association is a data format for data produced by GPS receivers.

[2] World Geodetic System dating from 1984 is the predominant coordinate system for encoding global coordinates.

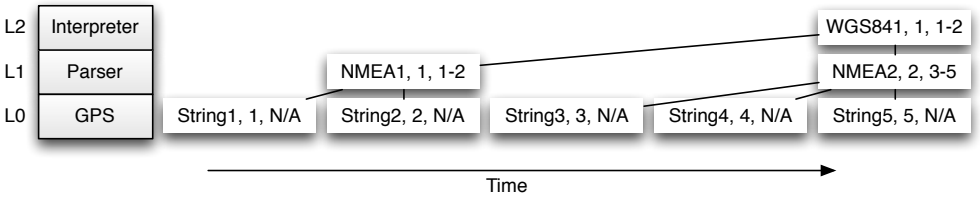


Figure 2.3: An example data tree for the GPS Channel.

example, when implementing a Likelihood feature for the GPS Channel, the feature specifies that it depends on a Processing Component that provides the Component Feature which can access HDOP information. Because, the Interpretor is implemented so that it only returns a value when a valid position is produced several NMEA sentences will be available in the data tree related to one output of the Channel. Components that filter according to certain rules may be inserted in the Channel, and the Channel Feature must implement strategies to cope with this fact.

In summary, the PCL contains a representation of the major flow of data in the position data process. The Channel tree exposes how single strained source-to-sink-flows connect the components as well as the features they provide. Furthermore, it supports adaptation of functionality that depends on several steps in the process, by allowing definition of a Channel Feature.

2.3 Positioning Layer

The top layer of the PerPos middleware exposes high-level position data and we call this the Positioning Layer. It presents a view of the position data processing that contains the Channel end-points including their features. All the features originally implemented in the PerPos middleware are visible as well as all available Channel Features. It is especially in the ability to access middleware adaptations in the high-level interaction, where details are abstracted away, that the PerPos middleware distinguishes itself from existing positioning middlewares. We consider the logical timing functionality of the Channels to be an important part of this ability. Even though, interactions with features take place at this abstract level, the middleware takes care of the coupling to the details that were actually a part of the high-level position in question.

At this level the PerPos API exposes the middleware functionality that is also part of a traditional closed positioning middleware. This includes push and pull semantics for retrieving positions from currently available sensors; definition of tracked targets, which may have several sensors attached to them; and a selection of services that can be leveraged for the development of location-aware applications [17]. To summarize, the combined effect of provid-

ing the specific extension mechanisms, presented here, is that the high-level API of the PerPos middleware can be effectively extended without requiring changes to the middleware itself.

3 Middleware Adaptations Enable Development of Detail Demanding Applications

In this section we support the utility of our design by explaining a number of concrete use-case examples that exploit the flexible API of the PerPos middleware. The examples are based on our own work with positioning technologies and particle filters. We will flesh out the details of the examples and include code snippets. After each example we will muse over how the example would be implemented in other positioning middleware.

We have for this evaluation realized the PerPos middleware in the Java language and built it on top of the OSGi service platform [82]. The components of the PerPos layers are mapped into the OSGi platform as service components and the dynamic composition mechanisms of OSGi is used for connecting the components.

3.1 Detecting Unreliable Readings by Adding Component Feature

The quality of GPS readings are greatly affected by atmospheric conditions and satellite constellation properties. GPS devices usually continue to produce measurements even if they loose sight of the satellites. Therefore, as argued in [38], filtering positions delivered by a GPS receiver according to the number of satellites available for the measurement can be used as a technique for increasing the reliability of readings. We have implemented this functionality by creating a new filtering Processing Component and inserting it into the processing tree. The Processing Component depends on a Component Feature named `NumberOfSatellites` which provides access to the concrete number of satellites available in each measurement. We insert the filter component after the Parser component. `NumberOfSatellites` is implemented as a Component Feature that is attached to the Parser component and adds a new data element to its output. The filter component extracts the number of satellites and forwards only measurements based on a satisfactory number.

In the ULF, an implementation of The Location Stack [38], the problem is solved by adding the satellite information to the position format used by the middleware. This means that satellite data is part of the position information for other kinds of positioning technologies as well. A resembling solution would be needed for MiddleWhere. Implementation of the extension of the position format requires access to the code for the middleware. In PoSIM [9] an `info` could be specified and implemented in the Sensor Wrapper in order to obtain

the number of satellites. However, PoSIM does not focus on filtering and it is unclear how a policy that tested the number of satellites would be used to delete an already obtained position from the system.

3.2 Integrating a Particle Filter Using Channel Feature

The particle filter we have implemented requires access to a number of low-level properties of the positions used in its calculations. In particular, the implementation of the filter depends on functionality that can provide a value indicating how likely it is that the current sensed position represents the actual true position.

Using the PerPos middleware we have implemented this likelihood functionality as a Channel Feature that calculates the probability based on HDOP values associated with the raw GPS reading. The HDOP values are extracted by a Component Feature from an intermediate parsing components in the positioning tree. This construction is visualized in Figure 3.1 and involves three different code artifacts, labeled by numbers in the figure. 1) Shows the key input handling parts of the Particle Filter implementation. Upon reception of a new position the Channel Feature called `Likelihood` is retrieved from the current input port and applied to each particle. 2) Shows how the `Likelihood` feature is implemented. The `apply(dataTree)` method is called by the middleware each time the Channel produces data. The method implementation collects the HDOP values from the data tree and uses it to update the internal state of the feature. When the method `getLikelihood(particle)` is called by the Particle Filter it calculates the likelihood estimate based on the collected HDOP values. 3) shows how the HDOP value is extracted and added to the output of the Parser component.

For testing this approach, we used some previously recorded sensor data and fed it into our PerPos middleware implementation of the particle filter. This was done using an emulator component that reads sensor data from a file and presents itself as a sensor. The emulator was plugged into the processing graph, taking the place of the sensors. Using this approach we were able to produce a refined trace as shown in Figure 3.2.

In the Location Stack [42] or MiddleWhere [88] the HDOP information is not available through a public API. The information is, therefore, not accessible to application developers. It may of course be accessed by circumventing the middleware, but then the timing functionality that connects the information to the correct position must be implemented as well. Another possibility is to extend the middleware's representation of a position with the information. This, however, requires access to the source code of the middleware. Furthermore, it would mean that this information is propagated up to highler levels always, even though most middleware uses does not need it. In PoSIM [9] a HDOP `info` may be specified and a wrapper for the GPS that extracts the information and make it available for higher levels may be writ-

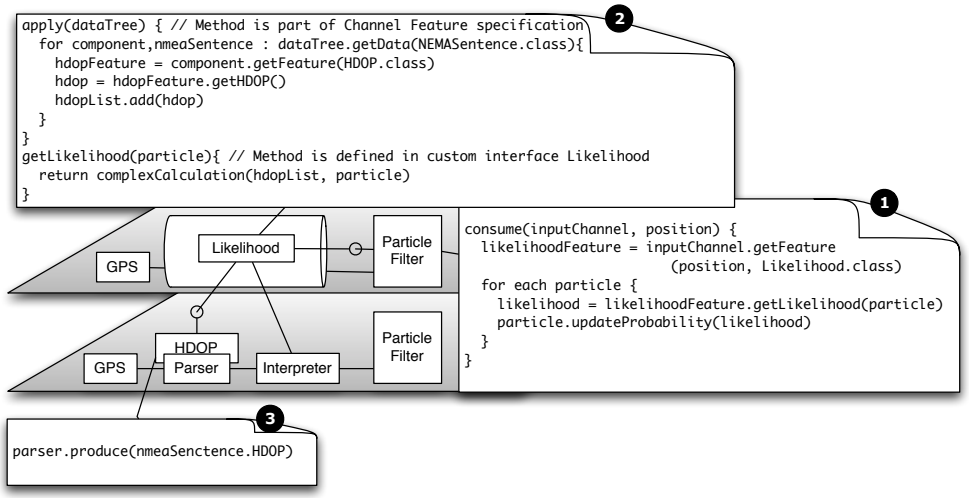


Figure 3.1: Code snippets used to provide the particle filter with a likelihood estimate based on HDOP values. The code is shown as pseudo code for clarity, the actual code is Java.

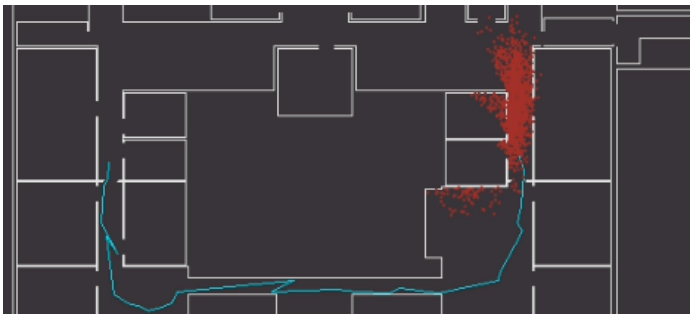


Figure 3.2: Example run of a particle filter implemented using the PerPos middleware. Red dots indicate particle positions, the blue line indicates the evaluated trace, and white lines indicate walls.

ten. However, when questioned it will always return the latest HDOP value, which may correspond to a new position.

3.3 Power Efficiency

In previous work we have created a power efficient solution for tracking mobile targets called EnTracked [53]. The EnTracked system targets mobile clients that report positions to a server for further processing. In short, the system minimizes the amount of data sampled at the mobile device according to a motion model and thereby reducing the number of power consuming data transmissions made to the server. As part of the validation of the PerPos

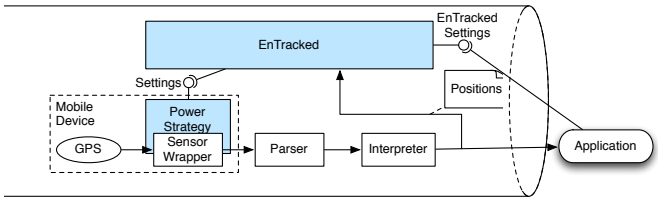


Figure 3.3: Processing graph for the implementation of EnTracked using the extensible PerPos API.

middleware design we have reimplemented key parts of the EnTracked system using the processing graph abstractions.

The processing graph of the reimplemented version of EnTracked is shown in Figure 3.3. The actual GPS sensor is located on a mobile device along with an instance of the PerPos middleware. The Processing Component called Sensor Wrapper in the figure is running on the mobile device while the Parser and Interpreter components run on a server. The application is supplied a position provider that delivers positions provided by the Channel with end-point after the Interpreter component. This channel is illustrated as the "tube" wrapping the components.

The original EnTracked system contains a client-side updating scheme that dynamically determines when to activate and deactivate the GPS device. The operation mode of this scheme is controlled by a server-side component. To obtain the same behavior using the graph abstractions we have implemented this updating scheme as a Component Feature, called Power Strategy, attached to the Sensor Wrapper component. The Power Strategy feature provides methods for controlling the operation mode of the updating scheme. In the EnTracked system the server-side component is controlling the updating scheme based on threshold levels for the maximum distance between two consecutive position updates. This behavior is implemented in the Channel Feature labeled EnTracked in the figure. This Channel Feature continuously monitors the output of the Interpreter component and calls the appropriate methods on the Power Strategy feature.

As stated earlier the PerPos middleware is realized in the Java language and built on top of the OSGi service platform [82]. Because, OSGi supports transparent distribution of services through the D-OSGi specification the processing graph can span several hosts with little added configuration overhead.

As MiddleWhere [88] provides a Position middleware containing a World Model, where all available position information is stored, this scenario does not apply to their domain. Configuration of sensors is not discussed. Likewise, The Location Stack [42] places obtained Measurements in a database and does not discuss sensor configuration. How to implement a power consumption scheme

using PoSIM is discussed in [9]. They suggest to define a PowerConsumption PoSIM control feature and allow it to be set to for example low and high. Again, a Sensor Wrapper that implements the feature must be defined. A policy of when to invoke the feature can be written. It will then be evaluated along with other policies in order to reason on the dynamic management of the positioning.

3.4 Concluding on the Examples

We have seen that in traditional positioning middleware as The Location Stack [42] and MiddleWhere [88] we need access to the code in order to propagate extra information up by extending the position data format. This solution does not scale well; if there is a large variance in the needed information for different applications and positioning technologies, as we expect, this is problematic.

For translucent positioning middleware as PoSIM [9] extra information may be accessed and devices controlled. Nonetheless, PerPos is superior in its retainment of timing information connecting low-level and high-level information and in the ability to controll the positioning process itself.

4 Translucent Middleware Guided by the Notion of Seamful Design for Developers

In this section we discuss the need for a positioning middleware that provides both transparency and translucency. Moreover, we introduce the notion of seamful design for developers and argue that designing for seamful use is a usefull metaphor for developers of translucent middleware.

In the reflection community it is common to refer to the dichotomy of transparent and translucent middleware. For example in this quote: "A desirable middleware model provides transparency to the applications that want it and translucency and fine-grain control to the applications that need it"[58, p. 37]. Positioning middleware designed for the traditional goal of transparency aims for the widely recognized principle of information hiding [84] and hides all aspects of positioning from the application developer to provide a transparent experience when working with heterogeneous technologies. This means that it abstracts away imperfections in technologies and hides uncertainty for the developer [70, 62]. In some cases, as in the examples presented in Section 3, this leaves the application developers at a loss, because the middleware they employ does not provide adequate support for handling imperfections of the underlying technologies.

The concept of translucency may advocate a generally open middleware with full access to change functionality. However, a designed reification that focus on certain aspects of the middleware may be easier to understand and

use than a full and only allowing specific adaptations gives a safer although less powerful development model. In our work we have been inspired by the notion of seamful design for developers, which we will now introduce.

In Weiser's seminal paper: The computer for the 21st century [94], seamless design is presented as a goal for how computers should be integrated into the world. A seamless design will allow computers to disappear from our awareness. This will enable us to focus on the goal for which we use the computer, instead of focusing on the computer itself. Such seamless designed systems should make the computerized infrastructure components they depend on disappear from the focus of the user. In the positioning domain, this means that the concrete positioning systems and their characteristics are hidden for the user who employ the position information.

However, due to the inherent imperfection of sensing technologies, in practice, it is hard to hide the characteristics of positioning in order to provide transparency. For instance, positioning technologies do not provide pervasive coverage because buildings, humans, and walls might block signals used for positioning. The positions delivered can be erroneous due to signal noise, delays, or faulty system calibration. Motivated by the imperfection of sensing technologies used in ubiquitous computing, several authors such as Chalmers and Galani [26], and Benford et al. [11] have argued for contrasting the goal of *seamless* design with one of *seamful* design. They define seams as "[...] the places where [components and technologies] may imperfectly connect to one another or to the physical environment." [11, p.126]. The goal of seamful design is to make the seams available in a designed manner, but not in focus at all times, so "one can selectively focus on and reveal [seams] when the task is to understand or even change the infrastructure." [26, p. 251].

Previously, seamful design has been directed towards the end user. Nonetheless, our focus is on the developer which has to face the same technology imperfections, only they occur during application development. For the developer of position based applications, imperfect connections might both occur within software components and between the positioning technology and the physical environment.

Instead of only focusing on position transparency, a seamful positioning middleware should expose and internalize key aspects of the positioning process in a designed manner. Thus, the developer can access both the imperfections of sensing technology to capture reality and the "imperfections" in the processing of position data, namely the design of the encapsulation and the abstracting away details. The set of seams a developer might be interested in cannot be determined uniquely. Therefore, a seamful middleware must provide means for the developer to extend the set of exposed seams.

To apply seamful design to the domain of positioning developers have to identify that the seamless design of the middleware is problematic under specific circumstances. Furthermore, they must posses some knowledge of the seams of the positioning technologies or in the position calculation process,

and they must know how to use information about seams to improve their application. It is clear that seamful use of a positioning middleware requires expert domain knowledge. Therefore, a positioning middleware should be designed for both seamless and seamful use, with concepts for both seamless and seamful positioning. The seamful middleware should not only support one type of developers, but developers with different skills, short and long schedules, and different types of applications.

The PerPos middleware supports both seamless and seamful interaction in that is delivers technology independent positions at a high-level layer while allowing for structured inspection and adaptation of the internal processing that lead to the high-level positions. Thus, to the extend that sensors and processing elements contains information that may be used to deduce for example, current coverage, accuracy, and signal noise, this information, which is usually hidden for the sake of transparency, can be used to expose the seams.

Concretely, in PerPos we reify the actual processing in a graph of processing components and flows of data and allow adaptation of processing components. Moreover, the processing is reified in a position source view, where the pipeline from one source to either a merge or the application is abstracted to a data channel. Through this view we allow adaptations that depend on data produced at several intermediate steps of the positioning process.

Our experience in developing services for positioning based on the PerPos middleware shows that it is the seams that calls for extra functionality in the middleware. This is concordant with the experience reported by Graumann et al. [38]. The inherent position uncertainty called for the development of a likelihood feature. The poor performance of GPS in indoor environments coupled with the device strategy of continuing to send positions called for the number of satellites feature. The limited battery capacity called for the power conservation feature. The approach of exposing and allowing adaptation of the processing components and the positioning process is especially suited to support the developer to choose when to access and possibly propagate to a higher level the information that is abstracted away in a seamless approach.

The concept of seamful design for developers has inspired the design of PerPos. It is a powerful metaphor when designing middleware for sensing domains, because it focuses the design of how to develop the handles available in a translucent and adaptive middleware to allow representation and improvement of the imperfections.

5 Related Work

In this section we will cover related work with respect to existing positioning middleware and to reflective middleware.

MiddleWhere by Ranganathan et al. [88] is a general purpose middleware for building location based applications. The primary purpose of MiddleWhere

is to provide location information to applications in a technology agnostic way. The Location Operating REference model (LORE) [28] focuses on providing high-level location data together with sensor fusion and intelligent notification. Cascadia by Welbourne et al. [95] is a middleware for detecting location events from RFID-based events. The middleware implements probabilistic fusion for detecting location events from raw RFID events. It provides both a declarative approach and an API that facilitates the development of applications which rely on location events. However, in all three systems the functionality (e.g., location models or sensor fusion) are statically implemented into the system and cannot be extended by application developers. Furthermore, there is no support for allowing application developers to extend the systems with functionality for handling cross-cutting concerns.

Location Stack by Hightower et al. [42] is a generic software engineering model for location in ubiquitous computing. The model is intended to be both a conceptual framework as well as a high-level layered architecture for implementing location based systems. However, the only true implementation of the Location Stack, the Unified Location Framework (ULF), has shown that the fundamental principles of the model cannot be followed in practice [38]. According to the report on the ULF, actual location based applications tend to require some level of access to low-level details of the positioning process. In the Location Stack this translates to creating cross-layer functionality which breaks the fundamental assumptions of the model.

PoSIM by Bellavista et al. [9] is a middleware for positioning applications designed to mediate access to heterogeneous positioning systems. The middleware is designed to provide application developers with some level of visibility into the internal workings of the underlying positioning systems. Operations for handling cross-cutting concerns are executed by adding or removing behaviors to the system expressed as declarative policies. The policies are written in a declarative language and the set of operations for conditions consists of simple comparison of data values while actions are limited to passing values to operations of the sensor wrapper.

There also exist more general context provision middlewares. An example is Contory proposed by Riva [90] that based on a query abstraction allow applications to request context information including spatial information.

In comparison, PerPos is a positioning middleware that supports a designed inspection and adaptation of the internal position processing. The middleware facilitates both seamless use of high-level positions and seamful use of details in the form of extraction of low-level information and adaptation of the position data processing, along with exposure of seams in the high-level interaction.

Traditional middleware are not well suited for dealing with dynamic aspects such as device-sizes and network availability. Given information about device types or network infrastructures the handling of such dynamic aspects can be optimized, e.g., by selecting protocols that better fit the underlying net-

work infrastructure. To address this problem, reflective middleware has been proposed, as described by Kon et al. [58], to provide traditional transparency coupled with translucency and fine-grain control. Reflective middleware provides inspection of their internal state using reflective meta interfaces. In a mobile context Carisma is a middleware proposed by Capra et al. [24] that support reflection by allowing programmers using policies to specific how the middleware should handle context changes for the provided services. PCOM proposed by Becker et al. [8] provides adaptation for pervasive component-based systems by contracts that specify dependencies between components and resources. PAQ [87] supports adaptive persistent queries over temporal-spatial data in dynamic networks. The system provides reflective programming abstractions to support the construction of applications that dynamically evaluate the cost of executing a query in the current environment and adjust the query's processing according to the application's needs. In comparison, PerPos is also a reflective middleware but provides a designed inspection and adaptation of the internal positioning process.

6 Conclusion and Future Work

In this paper we presented the design of PerPos a middleware for pervasive positioning that supports a designed inspection and adaptation of the internal position processing. The middleware facilitates both seamless use of high-level positions and seamful use of details in the form of extraction of low-level information and adaptation of the position data processing, along with exposure of seams in the high-level interaction. We have demonstrated the utility of the design by demonstrating how three example applications that all required access to internal details of the positioning process can be implemented using the adaptability of the middleware. Furthermore, we have argued for the potential of an adaptable positioning middleware. Finally, we have introduced the concept of seamful design for developers and discussed how the concept may focus the notion of translucent middleware.

In the future, we plan to research how traditional software qualities can be supported by the model based approach to translucency, e.g., reliability, scalability and performance. Furthermore, we will conduct user studies to validate the concept of translucency provided through seamful design.

Acknowledgements

We thank the rest of the PerPos group for help in implementing the middleware or applications that use the middleware. The authors acknowledge the financial support granted by the Danish National Advanced Technology Foundation under J.nr. 009-2007-2.

Paper III

EnTracked: Energy-Efficient Robust Position Tracking for Mobile Devices

Mikkel Baun Kjærgaard Jakob Langdal Jensen
Torben Godsk Thomas Toftkjær

Abstract

An important feature of a modern mobile device is that it can position itself. Not only for use on the device but also for remote applications that require tracking of the device. To be useful, such position tracking has to be energy-efficient to avoid having a major impact on the battery life of the mobile device. Furthermore, tracking has to robustly deliver position updates when faced with changing conditions such as delays due to positioning and communication, and changing positioning accuracy.

This work proposes EnTracked — a system that, based on the estimation and prediction of system conditions and mobility, schedules position updates to both minimize energy consumption and optimize robustness. The realized system tracks pedestrian targets equipped with GPS-enabled devices. The system is configurable to realize different trade-offs between energy consumption and robustness.

We provide extensive experimental results by profiling how devices consume power, by emulation on collected data and by validation in

Published as: Kjærgaard, M. B., Langdal, J., Godsk, T. and Thomas Toftkjær, T., Entracked: Energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, 2009.

several real-world deployments. Results from this profiling show how a device consumes power while tracking its position. Results from the emulation indicate that the system can estimate and predict system conditions and mobility. Furthermore they provide evidence for that the system can lower the energy consumption considerably and remain robust when faced with changing system conditions. By validation in several real-world deployments we provide evidence that the real system works as predicted by the emulation.

1 Introduction

An important feature of a modern mobile device is that it can position itself. Not only for use locally on the device but also for remote applications that require tracking of the device. Examples of such applications are geo-based information applications [22] or proximity and separation detection for social networking applications [61] just to mention a few. To be useful, such position tracking has to be energy-efficient to avoid having a major impact on the power consumption of the mobile device. Optimizing the operation of mobile devices for energy efficiency is an important issue and research is trying to address it from many angles, for instance, by trying to lower the impact of network protocols on power consumption [68] or by optimizing the execution at the operating system level [2]. Furthermore, tracking has to be robust in order to deliver position updates within limits when faced with changing conditions such as delays due to positioning and communication, and changing positioning accuracy.

To quantify the impact of position tracking on power consumption, we measured the power consumption of a Nokia N95 phone for 30 minutes, while the phone periodically positioned itself using the built-in GPS receiver and then send the position data using UMTS to a remote service hosted on an internet-connected server. The measurements were repeated with different time intervals between the periodic position updates. The average power consumption measured is plotted in Figure 1.1. The results highlight that even for moderate time intervals between position updates such as sixty seconds the power consumption is as high as 0.6 watt, which is twelve times more consumed power than when the phone is idle and the double amount of power compared to when the phone is used with the screen turned on.

From Figure 1.1 one might propose to minimize the power consumption by using large intervals between position updates, but then maintaining position accuracy becomes a problem, as a pedestrian target can walk or run quite far during two to five minutes. To address this problem previous research such as [67, 93] has proposed dynamic tracking to try to minimize the frequency of needed position updates by only requiring updates after the target has moved more than a specified distance or has moved out of a specified area.

The systems proposed in previous research for such dynamic tracking have

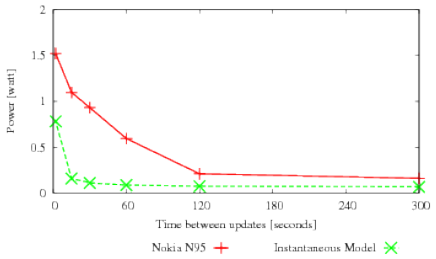


Figure 1.1: Average power consumption for periodic position updating measured on a N95.

several drawbacks. First, the research assumes that power consumption for positioning and sending is instantaneous meaning that the power consumption per position sensing and sending is a constant and that you can calculate the total power consumption by just multiplying this constant with the number of position updates. For instance, for the Nokia N95 this constant could be set to 1.523 joules [1]. Using this model to calculate the power consumption with different periodic intervals gives the results in Figure 1.1, which we denote by 'instantaneous model'. From the results we can see, that this model is not at all able to account for the real power consumption of a device. The reason is that this model does not take into account the delays involved when either the GPS or the GSM transceiver is initializing, nor the delays of it powering off. Second, previous research assumes that positioning takes constant time. This is not true, for instance, for sensing position with a GPS module the positioning time depends on how well the GPS module knows the frequencies of visible satellites, the current satellite constellation and several other parameters. Third, they considered the accuracy of positioning to be constant. This is also incorrect: the accuracy of GPS positioning depends on the number of visible satellites, signal-blocking structures near the receiver and several other factors. Fourth, they did not deploy the systems on actual hardware. Systems were evaluated either by simulation or emulation.

This paper proposes EnTracked - a system that, based on the estimation and prediction of system conditions and mobility, schedules position-updates to both minimize energy consumption and optimize robustness. The realized system tracks pedestrian targets equipped with GPS-enabled devices. The system is configurable to realize different trade-offs between energy consumption and robustness.

We make the following contributions in this work: First of all, we present a system that uses the estimation and prediction of system conditions and mobility to build a robust energy-efficient tracking system. Secondly, we pro-

[1] We have based this value on the average energy consumption for updating position every second, assuming that one update takes one second.

file how devices consume power for tracking and propose a device model that can account for the real power consumption of a device. Thirdly, we propose methods for position tracking that take the changing system conditions into account, e.g., positioning delays and position accuracy. Fourthly, we propose a method (implemented by dynamic programming) that can minimize power consumption and satisfy robustness by calculating the optimal plan for when to power on and off features of the mobile devices such as the GPS module or the UMTS radio. Fifthly, we provide a deep investigation by means of emulation to show that the system can: estimate and predict system conditions and mobility; lower the energy consumption considerably; and remain robust when faced with changing system conditions. Finally, we implement EnTracked and use this prototype in a real deployment to gather results showing that the real system works as predicted by the emulation.

The remainder of this paper is structured as follows: In Section 2, we present the relevant related work. Subsequently, we present our profiling results and propose a device model that can account for the real power consumption of a device. Then we introduce the EnTracked System in Section 4. The results from evaluating our system by means of emulation are presented in Section 5. In Section 6 we discuss our implementation and the results of our real-world deployment. Finally, in Section 7 we provide a summarizing discussion and Section 8 concludes the paper and provides directions for future work.

2 Related Work

As mentioned earlier, existing research have proposed dynamic tracking for updating position information about a target. Previous works such as [30, 66, 67] study dynamic tracking to minimize communication and to minimize the load on server nodes by lowering the number of position updates. Leonhardi et al. [67] study time-based and distance-based tracking that takes a constant positioning accuracy and target speed into account. They study by simulation the number of updates each tracking technique produces and the average and maximum uncertainty of the server-known position. They have later extended this work for tracking based on dead-reckoning [66]. Systems that tries to minimize the number of position updates for a specific application such as GeoPages have also been proposed [22].

A later work focusing on both energy efficiency and GPS positioning is Farrell et al. [33]. They propose methods that take into account a constant positioning delay, target speed, and stress the importance of the fact, that it is not energy-free to use the GPS constantly as assumed by earlier work. The methods have been evaluated by simulation, where they can save around 50% energy in the evaluated scenarios assuming an instantaneous power model. They have later extended this work for area-based tracking where they also

take constant position accuracy and communication delays into account.

For sensor networks, Tilak et al. [93] propose dynamic tracking techniques that take into account target speed and heading. They assume that the positioning uncertainty is negligible. They evaluate the methods by simulation for proximity-based sensor network positioning. For an indoor sensor network setting, You et al. [96] propose dynamic tracking techniques that take into account a constant positioning accuracy and delay, target speed and acceleration to detect if the target is moving or not. They evaluate the techniques by emulation for IEEE 802.15.4 signal-strength-based indoor positioning and one of their results is that considerable energy savings can be gained from the use of an accelerometer to detect if the target is stationary or not.

In comparison, our work proposes techniques that take into account dynamically estimated position accuracy and delays, communication delays, power constraints, target speed and acceleration (to detect if the target is moving or not). Furthermore we evaluate our techniques both by emulation and in real-world deployments.

3 Device Model

To be able to dynamically track a mobile device robustly and energy-efficiently we require a device model that can account for the delays and power consumption of the device. If we do not have such a model we cannot make decisions that will minimize the power consumption and make position updates within required limits. As motivated in the introduction previous research have assumed a simple, instantaneous power model. When using a more detailed model, one drawback is that it depends on more device dependent parameters, therefore we discuss how such parameters automatically can be estimated and the generality of the model in Section 7.

The proposed model is based on profiling the delays and power consumption of a Nokia N95 8GB[2] mobile phone. The N95 is a 3G phone with an internal GPS module and a triaxial accelerometer, both of unspecified brand, and a 1200 mAh battery. The main contributors to the delays and power consumption of the phone are the individual components used in the N95 and as such we hypothesize that the proposed model is generalizable to a wider range of phones than just the N95. The phone runs the Symbian 60 operating system version F1. We measured the power consumption of the phone by using the Nokia-developed tool Nokia Energy Profiler [80] version 1.1. The Nokia Energy Profiler tool has been built by Nokia to enable developers to analyze the power consumption of mobile applications and it supports a power-sampling rate of 4Hz. To measure the delays and power consumption of different features, several Python scripts have been developed that enable and disable

[2]For the remainder of this paper the Nokia N95 8GB phone is abbreviated as the N95 phone or simply N95

features and measure various delays. The Python scripts run on the N95 with the aid of the Python Interpreter for S60, version 1.4.4 [86] and the included library that provides access to phone features such as the internal GPS and the triaxial accelerometer. The internal GPS supports a sampling rate of 1Hz and the triaxial accelerometer a sampling rate of 30Hz. For the measurements involving sending data using the phone's UMTS radio, a TCP/IP server was implemented in Java and deployed on a server connected to the internet and with a public IP address that the phone was able to connect to over UMTS.

The proposed device model consists of two parts: a power model that describes the power usage of the phone; and a delay model that, for instance, describes the delays when requesting a phone feature, e.g., the time it takes for the GPS to return a position. In both models we consider the following phone features:

- accelerometer (a)
- GPS (g)
- radio idle (r)
- radio active (s)
- idle (include Python and Nokia Energy Profiler) (i)

These are the features that we find relevant for phone tracking, 'idle' is not strictly a feature, but is included in the power model for completeness. For interactive user applications on the device, one would also need to take into account the power usage of features such as the computations of the application logic, the key strokes, camera use, and screen use.

The power model consists of two functions defined in Equation 6.1: the power function *power* and the consumption function $c_{d,p}$ where d is a feature's power-off delay and p it's power consumption.

$$\begin{aligned} power(T) &= \sum_{t=1}^{T} i_p + c_{a_d,a_p}(a_t) + c_{g_d,g_p}(g_t) \\ &\quad + c_{r_d,r_p}(r_t) + c_{s_d,s_p}(s_t) \end{aligned} \tag{3.1}$$

$$c_{d,p}(x) = \begin{cases} p & \text{if } x <= d \\ 0 & \text{if } x > d \end{cases}$$

The equation uses the variables a_t, g_t, r_t, s_t for the different features listed in the feature list above. Each variable denotes, at time step t, the number of seconds since the feature was last powered off (a variable is zero if the feature is in use in the current time step t). Since the idle power consumption is constant no variable i_t is introduced. Furthermore the parameters i_p, a_p, g_p, r_p, s_p

denote the power consumption of a feature, e.g., 0.324 watt for the N95 internal GPS. The parameters a_d, g_d, s_d, r_d denote the number of seconds a feature takes to power off after last use, e.g., 30 seconds for the N95 internal GPS. The values of the parameters will be determined in Section 6.1 and Section 6.2 from traces collected from a N95 phone.

The delay model is given as two functions req_g, req_s that describe the request delays for the GPS and for activating the radio for sending. For the other features the request delays are negligible in our case, because they are below 100 milliseconds. The functions are defined and their values determined in Section 6.2.

3.1 Power Consumption

To determine the power parameters i_p, a_p, g_p, r_p, s_p we have collected a number of power consumption traces with a N95 phone with different features enabled and disabled. Before each trace collection and before all other of our experiments, the phone was fully charged to counter the influence of the non-linear voltage decrease of batteries [20]. First, the Nokia Energy Profiler application was started. Then the python interpreter was started with a Python script that enabled or disabled certain features for a specific amount of time. The total script running time was five minutes for these measurements. Then the Python interpreter was closed and the Nokia energy profiler was stopped. The power consumption trace collected with the Nokia energy profiler was exported to a file. These traces were trimmed to remove the consumption logged while the Python script was not running and when the screen was powered on. The average feature consumptions were calculated from the trimmed traces and is listed in Table 6.1. In the model we use the average values for the parameters. Table 6.1 also lists the standard deviations. As these values are rather small, using the average value in our model is a reasonable choice. Just for reference, we also measured the power consumption of the screen, which is around 0.2 watt. However, as discussed earlier we do not use this value in our device model.

Table 3.1: N95 features' power consumption.

Feature	Avg. Power [watt]	Std. Dev. [watt]
Idle (i_p)	0.0621	0.0173
Idle (i_p) + Logging	0.0647	0.0197
Accelerometer (a_p)	0.0503	0.035
GPS (g_p)	0.324	0.0435
Radio idle (r_p)	0.466	0.0324
Radio active (s_p)	0.645	0.0470

3.2 Delays

The delay model includes two types of delays as introduced earlier. The first is request delays, which is the time a feature uses to get operational. The second type is delays when powering off, which is the time a feature takes to power off after the last usage.

Request Delays

The request delays have been measured using the same experimental setup as described in Section 6.1, but with two changes. First, for GPS request delays the Python scripts logged the time difference in the internal clock between requesting a GPS measurement and the moment when a position was returned. Second, for the radio request delays the Python script logged the timestamp provided by the GPS for each position. This timestamp is taken at nearly the same time as when the Python script starts to request a TCP/IP connection to the server and on the server the Java application logged the time of data reception. The server was configured to synchronize its time using the Network Time Protocol [75] which can synchronize the clock to a precision of tens of milliseconds over the internet. The radio request delay was then measured as the difference between the GPS timestamp and the reception timestamp on the server. This difference includes both the time to activate the UMTS radio and the transmission time of the packet data.

A trace lasting fourteen hours was collected in order to measure the GPS request delays for varying periodic intervals between requests. For periodic intervals shorter than 85 seconds, the interval was increased one second for every five repetitions, while for periods equal to or longer than 85 seconds the increase step was five seconds. The collected trace is plotted in Figure 6.1(a) and shows that the request delays depend on the periodic interval between requests.

When a GPS device starts up it has to acquire the carrier frequencies on which the satellites send their signals and get data about the satellites' orbits, also known as ephemeris data [49]. The used N95 had Assisted GPS enabled, which means that the GPS receives the approximate signal frequencies, the ephemeris data, and other relevant data through the cell network, which speeds up the acquisition process. The GPS still has to tune to and synchronize with the actual signals. This process can take several seconds depending on the GPS device in use. The reason why the GPS has to tune into the actual carrier frequencies is that, due to effects such as the Doppler effect, the signal frequencies are shifted on their way from the satellites. From Figure 6.1(a) one can note that with periods above 30 seconds the GPS has to use at least five seconds to lock-on to the signals and estimate it's position. With periods below 30 seconds it only takes around one second. The reason for this is that the 30 seconds period matches the power-off delay for the GPS

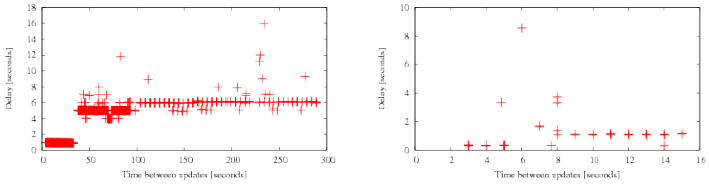
module (this value is determined later in this Section). So for 30 seconds after the last GPS request the GPS power is kept on and the GPS is locked on to the signals and therefore it does not have to re-acquire the signals. When the GPS powers off, it looses the signal locks and has to start over again. From the results we notice that with higher periods a longer acquisition time is needed to re-acquire a signal lock. Based on these results we have chosen to model the GPS request delays as the function req_g given in Equation 6.2 which depends on whether the periods being below or above the 30 second limit. As can be noticed from Figure 6.1(a), sometimes it takes even longer than 6 seconds, so to favour robustness over energy-efficiency, one could set this value higher. In comparison, previous work [34] uses a constant value of 0.5 seconds for the GPS request delay.

$$req_g(x) = \begin{cases} 1 & \text{if } x <= 30 \\ 6 & \text{if } x > 30 \end{cases}$$
$$req_r(x) = \begin{cases} 0.3 & \text{if } x <= 6 \\ 1.1 & \text{if } x > 6 \end{cases} \tag{3.2}$$

To determine the UMTS radio request delay a trace lasting 30 minutes was collected. During this trace data was sent from a N95 phone to a server over UMTS with different periods. The collected data cover periods from one second to fifteen seconds and is shown in Figure 6.1(b). From Figure 6.1(b) we can observe that with a period of less than 6 seconds, the radio request delay is very short, around 300 milliseconds. Above 6 seconds it is around 1100 milliseconds with some fluctuations. The limit of 6 seconds matches the power-off delay for the radio from active to idle, determined later in this section. The reason is that when the radio powers off it can take the radio up to several seconds to be activated again as stated in the technical report published by Nokia [79]. The fluctuations can be explained by variations in the communication path from the phone to the server over both the cellular network and the Internet, e.g., lost packets and delays. Based on these results we have chosen to model the radio request delay as the function req_r given in Equation 6.2. For both the function req_g and req_r we have focused on the average case which (because of the existence of higher delays) trades energy-efficiency over robustness.

Power-Off Delays

The power-off delay which is the time a feature takes to power off after the last usage has also been measured using the same experimental setup as described in Section 6.1. To determine the power-off delays, thirty minutes of data was collected where each minute, a GPS position was requested, then when a position was returned the position data was sent to a server and when the data



(a) GPS request delays. (b) Radio request delays.

Figure 3.1: Request delays for GPS and the radio for different periods between requests on a N95.

Table 3.2: N95 power-off delays for features.

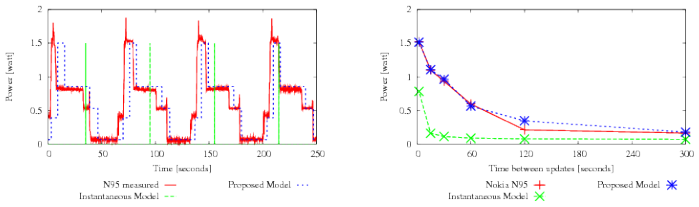
Feature	Avg. [second]	Std. Dev [second]
GPS	30.0	0.735
Radio idle	31.3	0.337
Radio active	5.45	0.774

was sent, the connection was closed. To determine the power-off delays, the power consumption traces were manually inspected and timestamps for the enabling and disabling of different features were entered into a trace file. The enabling and disabling of a feature could be determined from knowledge about the collection procedure and from knowledge about the power consumption of each feature, as determined by the experiments presented in Section 6.1. From the entries in the trace file the values listed in Table 6.2 were calculated. The results indicate that the power-off delay for the GPS and for radio idle is around thirty seconds and a little below six seconds for radio active. The power-off delay for radio idle is relative to when radio active has powered off to idle mode. The reason no power-off delay is listed for the accelerometer is that the accelerometer did not power off when requested to by our Python code. Only when the interpreter was stopped the power consumption dropped for the accelerometer. Furthermore the accelerometer uses an order of magnitude less power than the radio and the GPS. Therefore the accelerometer is treated in the same manner as the idle consumption, i.e., as a constant power usage.

3.3 Device Model Validation

To validate the proposed device model, we now compare the average power consumption for the periodic sampling calculated with this new model to the power consumption traces collected on a N95 phone. These traces were mentioned earlier in Section 1 and have not been used in the above sections to derive the model. Therefore they can be used to validate that the model can

explain the power consumption of a real phone with high precision. Figure 6.2(a) plots data from the collected trace for 60 seconds periodic tracking, overlaid with the predicted power consumption of the two other models. We can see how the proposed model closely matches the real power consumption, whereas the instantaneous model does not. Average values have also been calculated for a 30-minute scenario, which match the length of the traces collected on the N95 phone. The results of the collected traces and the two models have been plotted in Figure 6.2(b). The results show that the new model can describe the power consumption of a real phone with a much higher precision. Therefore this model can be used to inform the design of our tracking techniques towards minimizing the power consumption, whereas the instantaneous model has misinformed earlier research.



(a) Over time for 60 seconds periodic tracking.

(b) Average values.

Figure 3.2: Power consumption on a Nokia N95, with the instantaneous model and the proposed model.

4 EnTracked

The goal of EnTracked is to dynamically track mobile devices in both an energy-efficient and robust manner. Thus, robust, position updates have to be delivered to applications within application-specified error limits, where error refers to the distance between the application-known position and the real position of the device. Given that in this paper we focus on pedestrian targets we can assume that there is an upper threshold on target speed v_{max} which we assume to be $10m/s$. Furthermore, the focus on pedestrian targets guides us how to detect whether the target is moving or not by using an accelerometer.

To use EnTracked, position-based applications have to provide error limits that they want targets tracked with. But one might ask if it is the case that position-based applications always want the highest possible accuracy. In practice application providers will be motivated to minimize their applications' power consumption by providing limits because users will stop using their applications if they experience that they quickly drain their device's battery.

Privacy restrictions might also provide error limits if users specify lower limits for the granularity of which an application is allowed to track them with. Another option is that users themselves can decide how they want to trade application experience with energy efficiency by setting the limits themselves.

For a lot of applications it is also possible to calculate relevant error limits. A map application, that shows the positions of a number of mobile devices, can use the zoom level to determine relevant error limits (such as 25 meters for street-level view, 100 meter for a suburb, and 200 meter for a city-wide view). Another example is the many types of social networking applications that focus on relationships between the positions of devices, for instance, to detect when people come into proximity or when they separate. Methods have been proposed to efficiently track devices to reveal relationships, such as the ones proposed by Küpper et al. [61]. The methods work by dynamically assigning tracking jobs with changing error limits that they calculate based on the distance between the targets. Such methods produce tracking error limits ranging from 10 meters to several kilometres, depending on the distance between the devices.

When a position-based application requests to use EnTracked the steps illustrated in Figure 4.1 are carried out. Firstly, an application issues a request for the tracking of a device with an error limit (1). Secondly, the server propagates the request to the client-side part of EnTracked (2). Thirdly, the client finds a start position and returns it through the server to the application (3)+(4). Fourthly, the EnTracked client logic schedules features to deliver the next position within the error limit (5). Fifthly, at some point later EnTracked determines that a new position has to be delivered to the client through the server (6)+(7). If several applications requests tracking for the same device, EnTracked configures the device for tracking with the smallest requested error limit to fulfil both of the applications' limits.

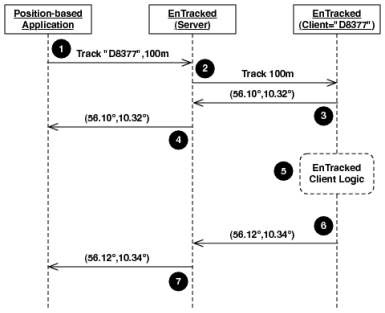


Figure 4.1: The steps of EnTracked when used by a position-based application.

When a request has been received by the EnTracked client, the client handles the request following the steps illustrated in Figure 4.2. To return the initial position to the server a GPS position is requested (1) and reported

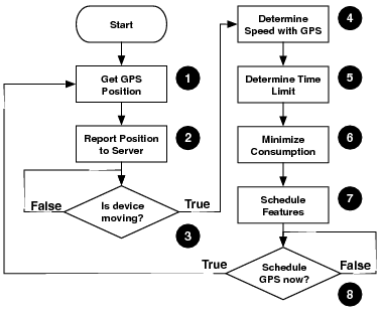


Figure 4.2: Flow chart of the EnTracked client logic.

to the server (2). Then, using the accelerometer-based method presented in Section 4.1, it is determined if the device is moving or not (3). If not, the logic waits for movement. When moved, the speed of the device is determined using GPS measurements as described in Section 4.2 (4). Then, using the error model presented in Section 4.3, a time for the next GPS position reading is calculated (5). This time limit is then given to a dynamic programming algorithm proposed in Section 4.4, that — based on the current power state of device features — finds the optimal strategy for minimizing the power consumption for this time limit and how to schedule features to satisfy the limit considering both possible GPS and radio delays (6). Then, the logic follows the scheduling plan calculated by the dynamic algorithm (7), restarts the process when appropriate (8), and returns the next GPS position to the server.

4.1 Detecting Movement

Movement can be detected by using accelerometers and has been proposed previously for indoor dynamic tracking by You et al. [96]. The Nokia N95 — as many other mobile devices — has a triaxial accelerometer which is used by EnTracked for movement detection. EnTracked only detects two states of movement, i.e. standing still and moving. As we have a robustness requirement stating that we have to position within a certain limit, we are interested in a detection scheme that has a low tolerance for movement, which will ensure that we detect movement very well. We have used the following scheme for movement detection: first, an accelerometer measurement is collected for each of the three axes; next, for each axis the variance of the last 30 measurements is calculated and the three variance values are summed. In Figure 4.3 we have plotted such summed variance values for a trace of accelerometer readings collected for a person that, at first is standing still, then starts walking and then stops again. Figure 4.3 also plots a manually collected ground truth for when the person was moving. From Figure 4.3 we can see that there

is a noticeable difference in variance between standing still and moving. To detect movement, we use a threshold which was selected to be 1000 based on the receiver-operating-characteristic curve plotted in Figure 4.4 to have an equal tradeoff between detecting still (true positive rate) and not detecting movement (false positive rate).

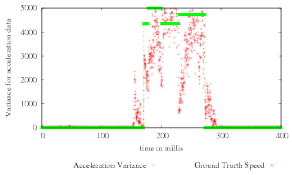


Figure 4.3: Summed variance for each of the axes over accelerometer data for 30 measurements.

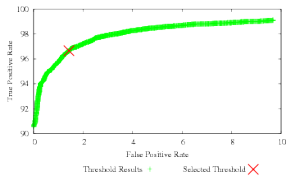


Figure 4.4: Receiver-operating-characteristic curve with positive as detection of the device being still.

As the device running EnTracked could be carried and handled in many different ways, this might cause false detections. If a person is stationary, but gesturing with the device in hand the accelerometer will detect this movement, which will increase power consumption. If a person is walking with the device in hand, and keeping the device steady, there might not be enough acceleration in any direction for the variance to reach the threshold for movement detection. This poses a problem and can only be solved by using a more clever movement detection scheme such as the ones proposed by Reddy et al. [89]. However, for EnTracked only one sudden move with the device will make the state change and then speed estimation based on GPS measurements will kick in.

4.2 Estimating Speed

The movement speed of a mobile device v_{est} can be estimated from GPS measurements v_{gps}; however, we need to analyse whether or not it is reliable. GPS modules normally implement a Kalman filter to estimate the speed of the GPS [49] from GPS positions and measured Doppler shifts of the satellite signals. Therefore we choose to use the GPS module estimated speed because it gives more accurate speed estimates compared to the method used by earlier work

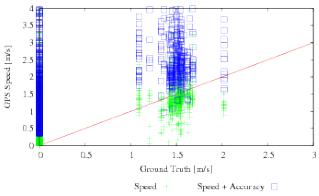


Figure 4.5: GPS speed versus ground truth speed.

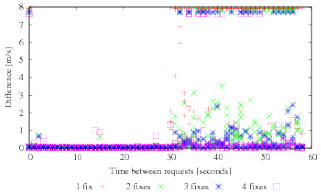


Figure 4.6: Wake-up speed measurements.

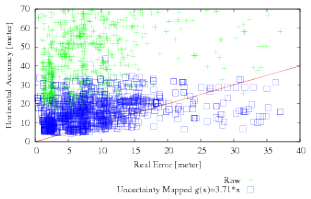


Figure 4.7: Real error versus horizontal accuracy

such as [33], that only takes into consideration the time difference and distance between the two last GPS positions received from the GPS. In order to evaluate the accuracy of the GPS-estimated speed v_{gps}, we compare it with the ground truth speed $v_{groundtruth}$. The ground truth speed is calculated by interpolating between manually collected timestamps for the visiting of well-known reference spots while collecting GPS measurements. Using a subset of our trace data collected for emulation, we have plotted v_{est} compared to $v_{groundtruth}$ in Figure 4.5. From the figure we see that v_{gps} tends to correlate with $v_{groundtruth}$. As v_{gps} in many situations is underestimated, we have plotted $v_{gps} + v_{accuracy}$ in Figure 4.5. $v_{accuracy}$ is the estimated accuracy of the estimated GPS speed v_{gps} provided by the GPS module. $v_{gps} + v_{accuracy}$ is generally overestimated, which improves the robustness of the system. However, at very low speeds the GPS speed is largely overestimated because the value of the estimated accuracy is high. Therefore, for detecting stationary situations we rely on movement detection with an accelerometer as presented in Section 4.1.

In order to save power, we are interested in turning off the GPS, so that it doesn't consume unnecessary power. When the GPS has been put to sleep for some time, we need to know at what point we are able to trust the speed measurements provided by the GPS. Though we want to know the speed as quickly as possible, we don't want to use the speed measurements until they give us the required accuracy, which may vary depending on how long the GPS has been sleeping and the number of measurements v_{gps} is based upon. We

have investigated the number of necessary GPS measurements needed in order to get a v_{gps} with sufficient accuracy. Figure 4.6 plots measurements where the GPS is put to sleep for a varying period of time and restarted. After restart, v_{gps} based on one through four measurements have been collected, and the difference between v_{gps} and $v_{groundtruth}$ is calculated. During this test $v_{groundtruth}$ is equal to zero meters per second. For reasons that we will not investigate as a part of this work, there are some speed measurements that the used Python library returns as "not a number". In order for such cases not to be neglected, they are placed at the top of the graph (from 7.6 m/s for 4 measurements and up to 7.9 m/s for 1 measurement), but must **not** be considered valid, inaccurate measurements.

According to Figure 4.6, it seems that one GPS measurement is sufficient as long as the GPS doesn't sleep more than 30 seconds between measurements. However, as the sleeping period rises to 30 seconds or more, one, two and three measurements become insufficient. From 30 seconds onwards, it seems that four measurements provide the best speed measurements with some inaccuracy.

4.3 Error Model

For calculating the time limit for when to deliver the next GPS measurement to an application, we use the following error model inspired by the error model proposed by Ferrell et al. [34]. This error model takes into account the estimated uncertainty of the last GPS position delivered to the application u_{gps}, the time since the last GPS position t_{gps}, and the estimated speed v_{est} (as described in Section 4.2). The error model then calculates the current error e_{model} with respect to the last delivered GPS position as defined in Equation 4.1.

$$e_{model} = u_{gps} + (t - t_{gps}) * v_{est} \tag{4.1}$$

As an estimate for the uncertainty of a GPS position, we use the horizontal accuracy outputted by the GPS in the N95 phone. Most GPS modules are able to output such information calculated based on the quality of the satellite signals and the quality of the signal processing. The horizontal accuracy outputted in Symbian OS is specified to be the 68% error quartile [69], which means that in 68% of the time the error should be less than this value. For one of the traces collected for our emulation, we have plotted the real error of a GPS position versus the horizontal accuracy for the GPS position in Figure 4.7. The real error was calculated with respect to a manually entered ground truth. From the plot, we can see that the real error is largely overestimated, and to get an on-average more precise estimate, we choose to use a linear model $g(x) = a * x$ to map the horizontal accuracy outputted by the GPS. The parameter a was found by using linear regression on the trace entries for the following equation $g(x) = a * x - 2$; the -2 was added to (on-average)

optimize the error to be overestimated with 2 meters. This value can be changed to trade power efficiency (with smallest possible uncertainty values) and robustness (with largely overestimating uncertainty values). Figure 4.7 also plots the values that have been mapped by the linear model. Even tough the values are mapped, most still overestimate the error to keep favouring robustness.

The calculation of the time limit for the next GPS position t_{limit} is based on the application-defined error limit d_{limit}, the current error e_{model} (using Equation 4.1) and the estimated speed v_{est}. The limit is found, using Equation 4.2, as the time it will take the device to move beyond the application limit considering the current error with respect to the last delivered GPS position.

$$t_{limit} = \frac{d_{limit} - e_{model}}{v_{est}} \tag{4.2}$$

4.4 Minimize Consumption

To minimize the power consumption and to be robust based on the time limit, we need to calculate when to power features on and off. This calculation has to take into account the delays for powering off features and the request delays when powering features on again. To calculate when to power features on and off, we have formalized the problem as a minimization problem for a set of recursive functions as defined in Equation 4.3. The problem is defined for the variables g, r, s that denote the number of seconds since the GPS, radio idle, and radio active, respectively was requested to start powering off. The problem is to find g'', r'', s'' (that denote the variable states when the features should be powered on again) in comparison to g_0, r_0, s_0 (the feature states when calculating a solution to the minimization problem). The problem also takes into account the upper time limit t_{limit} as defined in Equation 4.2.

$$req_g(x) = \begin{cases} 1 & \text{if } x <= 30 \\ 6 & \text{if } x > 30 \end{cases}$$

$$req_s(x) = 1$$

$$C_{u,p}(t, x, x_0) = \begin{cases} 0 & \text{if } t = 0, x = x_0 \\ p + C_{u,p}(t-1, x) & \text{if } t > 0, x = 0 \\ C_{u,p}(t-1, x-1) & \text{if } t > 0, x > u \\ p + C_{u,p}(t-1, x-1) & \text{if } t > 0, 0 < x <= u \\ \text{Undefined} & \text{else} \end{cases} \tag{4.3}$$

$$(g'', r'', s'') = \arg\min_{g,r,s}\{C_{30,0.324}(t - req_g(g), g, g_0) \\ + C_{31,0.466}(t, r, r_0) + C_{6,0.645}(t, s, s_0) | t = t_{limit} - req_s(s)\}$$

The solution is found by finding the variable values that minimize the sum of the three instances of the power consumption function $C_{u,p}(t, x, x_0)$, where u is the power-off delay, p the power consumption of the feature, and t the time step. To simplify the power consumption function, it is assumed that a solution to the minimization problem will either choose to keep features powered on or to power them off, then later power them on again. Therefore, the function does not model the possibility to turn a feature on again, except as a solution to the problem. The solution for the minimization problem is calculated by an algorithm based on dynamic programming which means that the running time of the algorithm is $O(t_{limit}^2)$. Therefore, the running time depends on the representation of t_{limit}, and — to keep the needed number of computations as low as possible — the functions and values in Equation 4.3 are restricted to represent time in seconds.

5 Emulation

In this section we present the results of our emulation to study if EnTracked can lower the energy consumption and remain robust when faced with changing system conditions.

We have implemented the EnTracked system based on a layered architecture as depicted in Figure 5.1. From a high-level view there are two layers: a *platform layer* and a *client logic layer*. The platform layer represents a device that is capable of providing GPS positions and acceleration measurements. The client logic is implemented such that it can run on top of any platform that can provide it with position and acceleration data.

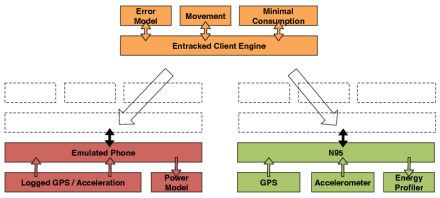


Figure 5.1: System architecture for emulation and deployment

For the emulation we exploit this layered design, in that we can exchange the N95 phone with an emulation engine. The emulation engine reads data files of the same format as the ones used for the analysis of the phone characteristics. The output of running the EnTracked client on top of the emulation engine contains the same information as running it on the actual N95 phone. Furthermore the emulation engine outputs information on the state of the phone at all emulation time steps. This state information is used in combination with the normal output to calculate power consumption and robustness for various runs of the EnTracked client.

To evaluate the power consumption of EnTracked, we selected a number of parameter variations to emulate. The data used for emulation was collected using a N95 phone that continuously sampled the GPS position and the accelerometers while walking a predetermined route. During this data collection, the actual position was recorded manually on a small laptop. Based on the collected data we correlated the phone's actual positions with the positions reported by the GPS. We made three data collection runs on the route shown in Figure 5.2, each lasting approximately 35 minutes and the percentage of stationary time is 67.1%. The route runs through a parking lot and has a length of 700 meters. We kept the phone stationary for several minutes at three locations, marked in the figure. During the run we made sure only to walk in straight lines and we registered the actual position every time the direction was changed. Furthermore we made sure to maintain a steady pace between these points, which allows us to calculate a reasonable estimate of the ground truth speed.

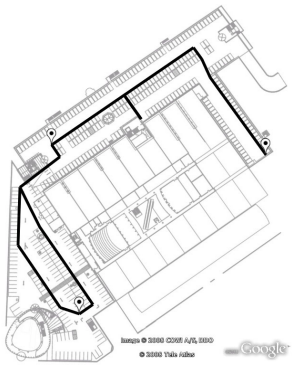


Figure 5.2: Route used for collection of emulation data. (0.7km)

We evaluated EnTracked as described in Section 4 and a periodic sampling algorithm in the emulation system. Here we present three emulations: one for which the goal for the positioning precision was defined to be 25 meter, i.e., the system should never report a position that is more than 25 meter from the actual position of the phone; one for which the goal was 100 meter; and one for which the goal was 200 meter. These limits corresponds to the value d_{limit} in Section 4.3.

A naive approach to obtain the required precision, while still conserving some power compared to continuous sampling, is to sample the position at an interval small enough that it would be impossible to move outside the area bounded by the requested precision between GPS fixes. Assuming that the GPS device is carried by a pedestrian these intervals is every 2.5th second, every 10th second and every 20th second for 25 meter, 100 meter and 200 meter respectively[3]. We denote this sampling strategy by the name the *periodic*

[3]The interval is based on the assumption that it is very unlikely that a pedestrian moves

strategy. For the emulation, the EnTracked algorithm was limited to the desired precision and in the case that the GPS estimated speed was returned as "not a number" we set the speed to the maximum possible speed of 10 m/s.

5.1 Robustness

Our robustness goal is that the real error must remain below a given application limit. We have analyzed the robustness in our emulations for the different strategies by comparing the last GPS position sent to the server with the ground truth known position. The real error is calculated as the difference between these two positions. In Figure 5.3 and Figure 5.4 we compare the real error for 10 and a 20 seconds periodic strategies with EnTracked configured with a limit of 100 meters and 200 meters, respectively. From the figures we see that when EnTracked detects the device is stationary, it does not update the position and when it detects that the device is moving, it schedules sleeps depending on the movement speed. During the sleep periods the error increases until the sleep ends and a new GPS position is provided. The longer sleeps for EnTracked with a 200 meter limit can also be seen.

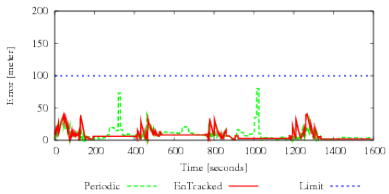


Figure 5.3: Real error in emulation with a limit of 100 meter and a periodic strategy.

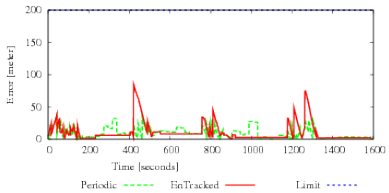


Figure 5.4: Real error in emulation with a limit of 200 meter and a periodic strategy.

The robustness results for both the periodic strategies and EnTracked and EnTracked(β) with the limits 25 meter, 100 meter and 200 meter are summarized in Table 5.1 as the average real error and the percentage of time the real error went above the threshold limit. EnTracked(β) is our system but

faster than the world record for 100 meter dash

Table 5.1: Robustness results as average error and percentage of time above the error limit.

		Periodic	EnTracked(β)	EnTracked
Avg.[m]	25m	8.2	8.4	7.8
	100m	8.6	16.4	10.6
	200m	9.6	24.4	14.3
Limit	25m	2.0%	3.2%	2.2%
	100m	0.0%	0.0%	0.0%
	200m	0.0%	0.0%	0.0%

without movement detection. The average error is greater for EnTracked(β) because it only schedules sleeps. The reason why the average error is greater for EnTracked with a 200 meter limit than for 25 and 100 meter is that with a limit of 200 meter, longer sleeping periods are scheduled compared to the 25 and 100 meter limits. This also enables EnTracked with the 200 meter limit to provide better energy saving. The reason that the limit is crossed when configured to 25 meter is that a single bad GPS measurement can have an error above the limit.

5.2 Energy Efficiency

Based on the emulations we have calculated the power consumption of different periodic strategies and EnTracked using the device model described in Section 3. In Figure 5.5 and Figure 5.6 we compare the power consumption for: a 10 and a 20 seconds periodic strategy; EnTracked configured with limits of 100 meters and 200 meters. From the figures we see that when EnTracked detects the device to be still, no position updates are produced, thus the power consumption drops significantly because the GPS and the radio can be switched off. During movement, sleeps are scheduled and depending on their length, the power usage drops significantly. The periodic strategy on the other hand has a repeating static power consumption pattern and never drops significantly because the GPS and radio can never be switched off.

Table 5.2 summarize the power usage of the periodic strategies and EnTracked and EnTracked(β) for limits of 25m, 100m and 200m. Furthermore the power savings as the percent savings in comparison to the power use of continuous sampling which in our case is similar to a one second periodic strategy. From the table we can see that EnTracked saves considerably more power than the periodic strategy. The fact that the EnTracked strategy uses motion detection allows it to save power proportional to the time the device is stationary. Even without using the motion detection we can see that the EnTracked client should be able to conserve a fair amount of power, compared to

Table 5.2: Power consumption

		Periodic	EnTracked(β)	EnTracked
Avg.[W]	25m	1.468	1.351	0.781
	100m	1.331	0.851	0.710
	200m	1.264	0.608	0.600
Savings	25m	0.0%	6.89%	43.36%
	100m	9.19%	40.55%	48.73%
	200m	13.58%	56.05%	56.20%

the periodic strategy. With a limit of 200 meter EnTracked(β) nearly saves as much power as EnTracked because it is able to schedule long sleeps on low estimated speeds. However, if the device is suddenly moved the EnTracked(β) might sleep while the error limit is crossed, because a long sleep has been scheduled. This makes EnTracked preferable over EnTracked(β).

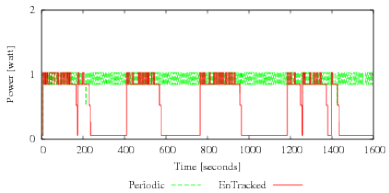


Figure 5.5: Emulated power consumption with a limit of 100 meter and a periodic strategy.

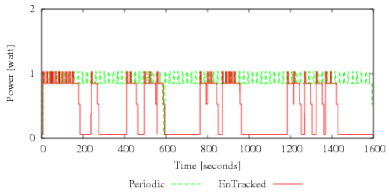


Figure 5.6: Emulated power consumption with a limit of 200 meter and a periodic strategy.

6 Real-World Validation

To be able to dynamically track a mobile device robustly and energy-efficiently we require a device model that can account for the delays and power consumption of the device. If we do not have such a model we cannot make decisions

that will minimize the power consumption and make position updates within required limits. As motivated in the introduction previous research have assumed a simple, instantaneous power model. When using a more detailed model, one drawback is that it depends on more device dependent parameters, therefore we discuss how such parameters automatically can be estimated and the generality of the model in Section 7.

The proposed model is based on profiling the delays and power consumption of a Nokia N95 8GB[4] mobile phone. The N95 is a 3G phone with an internal GPS module and a triaxial accelerometer, both of unspecified brand, and a 1200 mAh battery. The main contributors to the delays and power consumption of the phone are the individual components used in the N95 and as such we hypothesize that the proposed model is generalizable to a wider range of phones than just the N95. The phone runs the Symbian 60 operating system version F1. We measured the power consumption of the phone by using the Nokia-developed tool Nokia Energy Profiler [80] version 1.1. The Nokia Energy Profiler tool has been built by Nokia to enable developers to analyze the power consumption of mobile applications and it supports a power-sampling rate of 4Hz. To measure the delays and power consumption of different features, several Python scripts have been developed that enable and disable features and measure various delays. The Python scripts run on the N95 with the aid of the Python Interpreter for S60, version 1.4.4 [86] and the included library that provides access to phone features such as the internal GPS and the triaxial accelerometer. The internal GPS supports a sampling rate of 1Hz and the triaxial accelerometer a sampling rate of 30Hz. For the measurements involving sending data using the phone's UMTS radio, a TCP/IP server was implemented in Java and deployed on a server connected to the internet and with a public IP address that the phone was able to connect to over UMTS.

The proposed device model consists of two parts: a power model that describes the power usage of the phone; and a delay model that, for instance, describes the delays when requesting a phone feature, e.g., the time it takes for the GPS to return a position. In both models we consider the following phone features:

- accelerometer (a)
- GPS (g)
- radio idle (r)
- radio active (s)
- idle (include Python and Nokia Energy Profiler) (i)

[4]For the remainder of this paper the Nokia N95 8GB phone is abbreviated as the N95 phone or simply N95

These are the features that we find relevant for phone tracking, 'idle' is not strictly a feature, but is included in the power model for completeness. For interactive user applications on the device, one would also need to take into account the power usage of features such as the computations of the application logic, the key strokes, camera use, and screen use.

The power model consists of two functions defined in Equation 6.1: the power function *power* and the consumption function $c_{d,p}$ where d is a feature's power-off delay and p it's power consumption.

$$\begin{aligned} power(T) &= \sum_{t=1}^{T} i_p + c_{a_d,a_p}(a_t) + c_{g_d,g_p}(g_t) \\ &+ c_{r_d,r_p}(r_t) + c_{s_d,s_p}(s_t) \end{aligned} \tag{6.1}$$

$$c_{d,p}(x) = \begin{cases} p & \text{if } x <= d \\ 0 & \text{if } x > d \end{cases}$$

The equation uses the variables a_t, g_t, r_t, s_t for the different features listed in the feature list above. Each variable denotes, at time step t, the number of seconds since the feature was last powered off (a variable is zero if the feature is in use in the current time step t). Since the idle power consumption is constant no variable i_t is introduced. Furthermore the parameters i_p, a_p, g_p, r_p, s_p denote the power consumption of a feature, e.g., 0.324 watt for the N95 internal GPS. The parameters a_d, g_d, s_d, r_d denote the number of seconds a feature takes to power off after last use, e.g., 30 seconds for the N95 internal GPS. The values of the parameters will be determined in Section 6.1 and Section 6.2 from traces collected from a N95 phone.

The delay model is given as two functions req_g, req_s that describe the request delays for the GPS and for activating the radio for sending. For the other features the request delays are negligible in our case, because they are below 100 milliseconds. The functions are defined and their values determined in Section 6.2.

6.1 Power Consumption

To determine the power parameters i_p, a_p, g_p, r_p, s_p we have collected a number of power consumption traces with a N95 phone with different features enabled and disabled. Before each trace collection and before all other of our experiments, the phone was fully charged to counter the influence of the non-linear voltage decrease of batteries [20]. First, the Nokia Energy Profiler application was started. Then the python interpreter was started with a Python script that enabled or disabled certain features for a specific amount of time. The total script running time was five minutes for these measurements. Then the Python interpreter was closed and the Nokia energy profiler was stopped. The

power consumption trace collected with the Nokia energy profiler was exported to a file. These traces were trimmed to remove the consumption logged while the Python script was not running and when the screen was powered on. The average feature consumptions were calculated from the trimmed traces and is listed in Table 6.1. In the model we use the average values for the parameters. Table 6.1 also lists the standard deviations. As these values are rather small, using the average value in our model is a reasonable choice. Just for reference, we also measured the power consumption of the screen, which is around 0.2 watt. However, as discussed earlier we do not use this value in our device model.

Table 6.1: N95 features' power consumption.

Feature	Avg. Power [watt]	Std. Dev. [watt]
Idle (i_p)	0.0621	0.0173
Idle (i_p) + Logging	0.0647	0.0197
Accelerometer (a_p)	0.0503	0.035
GPS (g_p)	0.324	0.0435
Radio idle (r_p)	0.466	0.0324
Radio active (s_p)	0.645	0.0470

6.2 Delays

The delay model includes two types of delays as introduced earlier. The first is request delays, which is the time a feature uses to get operational. The second type is delays when powering off, which is the time a feature takes to power off after the last usage.

Request Delays

The request delays have been measured using the same experimental setup as described in Section 6.1, but with two changes. First, for GPS request delays the Python scripts logged the time difference in the internal clock between requesting a GPS measurement and the moment when a position was returned. Second, for the radio request delays the Python script logged the timestamp provided by the GPS for each position. This timestamp is taken at nearly the same time as when the Python script starts to request a TCP/IP connection to the server and on the server the Java application logged the time of data reception. The server was configured to synchronize its time using the Network Time Protocol [75] which can synchronize the clock to a precision of tens of milliseconds over the internet. The radio request delay was then measured as the difference between the GPS timestamp and the reception timestamp on the server. This difference includes both the time to activate the UMTS radio and the transmission time of the packet data.

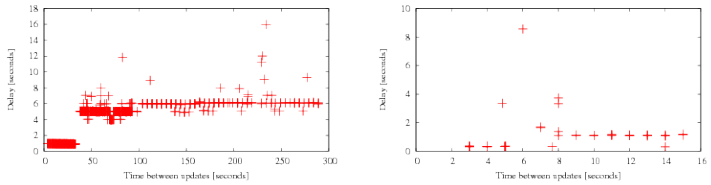
A trace lasting fourteen hours was collected in order to measure the GPS request delays for varying periodic intervals between requests. For periodic intervals shorter than 85 seconds, the interval was increased one second for every five repetitions, while for periods equal to or longer than 85 seconds the increase step was five seconds. The collected trace is plotted in Figure 6.1(a) and shows that the request delays depend on the periodic interval between requests.

When a GPS device starts up it has to acquire the carrier frequencies on which the satellites send their signals and get data about the satellites' orbits, also known as ephemeris data [49]. The used N95 had Assisted GPS enabled, which means that the GPS receives the approximate signal frequencies, the ephemeris data, and other relevant data through the cell network, which speeds up the acquisition process. The GPS still has to tune to and synchronize with the actual signals. This process can take several seconds depending on the GPS device in use. The reason why the GPS has to tune into the actual carrier frequencies is that, due to effects such as the Doppler effect, the signal frequencies are shifted on their way from the satellites. From Figure 6.1(a) one can note that with periods above 30 seconds the GPS has to use at least five seconds to lock-on to the signals and estimate it's position. With periods below 30 seconds it only takes around one second. The reason for this is that the 30 seconds period matches the power-off delay for the GPS module (this value is determined later in this Section). So for 30 seconds after the last GPS request the GPS power is kept on and the GPS is locked on to the signals and therefore it does not have to re-acquire the signals. When the GPS powers off, it looses the signal locks and has to start over again. From the results we notice that with higher periods a longer acquisition time is needed to re-acquire a signal lock. Based on these results we have chosen to model the GPS request delays as the function req_g given in Equation 6.2 which depends on whether the periods being below or above the 30 second limit. As can be noticed from Figure 6.1(a), sometimes it takes even longer than 6 seconds, so to favour robustness over energy-efficiency, one could set this value higher. In comparison, previous work [34] uses a constant value of 0.5 seconds for the GPS request delay.

$$
\begin{aligned}
req_g(x) &= \begin{cases} 1 & \text{if } x <= 30 \\ 6 & \text{if } x > 30 \end{cases} \\
req_r(x) &= \begin{cases} 0.3 & \text{if } x <= 6 \\ 1.1 & \text{if } x > 6 \end{cases}
\end{aligned}
\tag{6.2}
$$

To determine the UMTS radio request delay a trace lasting 30 minutes was collected. During this trace data was sent from a N95 phone to a server over UMTS with different periods. The collected data cover periods from one second to fifteen seconds and is shown in Figure 6.1(b). From Figure 6.1(b)

we can observe that with a period of less than 6 seconds, the radio request delay is very short, around 300 milliseconds. Above 6 seconds it is around 1100 milliseconds with some fluctuations. The limit of 6 seconds matches the power-off delay for the radio from active to idle, determined later in this section. The reason is that when the radio powers off it can take the radio up to several seconds to be activated again as stated in the technical report published by Nokia [79]. The fluctuations can be explained by variations in the communication path from the phone to the server over both the cellular network and the Internet, e.g., lost packets and delays. Based on these results we have chosen to model the radio request delay as the function req_r given in Equation 6.2. For both the function req_g and req_r we have focused on the average case which (because of the existence of higher delays) trades energy-efficiency over robustness.



(a) GPS request delays. (b) Radio request delays.

Figure 6.1: Request delays for GPS and the radio for different periods between requests on a N95.

Power-Off Delays

The power-off delay which is the time a feature takes to power off after the last usage has also been measured using the same experimental setup as described in Section 6.1. To determine the power-off delays, thirty minutes of data was collected where each minute, a GPS position was requested, then when a position was returned the position data was sent to a server and when the data was sent, the connection was closed. To determine the power-off delays, the power consumption traces were manually inspected and timestamps for the enabling and disabling of different features were entered into a trace file. The enabling and disabling of a feature could be determined from knowledge about the collection procedure and from knowledge about the power consumption of each feature, as determined by the experiments presented in Section 6.1. From the entries in the trace file the values listed in Table 6.2 were calculated. The results indicate that the power-off delay for the GPS and for radio idle is around thirty seconds and a little below six seconds for radio active. The power-off delay for radio idle is relative to when radio active has powered off to

Table 6.2: N95 power-off delays for features.

Feature	Avg. [second]	Std. Dev [second]
GPS	30.0	0.735
Radio idle	31.3	0.337
Radio active	5.45	0.774

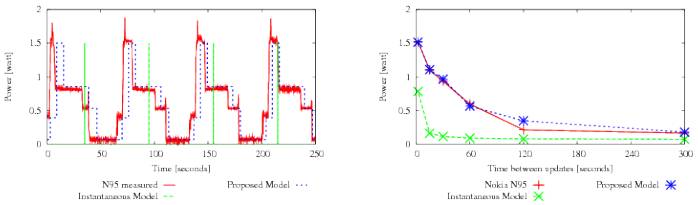
idle mode. The reason no power-off delay is listed for the accelerometer is that the accelerometer did not power off when requested to by our Python code. Only when the interpreter was stopped the power consumption dropped for the accelerometer. Furthermore the accelerometer uses an order of magnitude less power than the radio and the GPS. Therefore the accelerometer is treated in the same manner as the idle consumption, i.e., as a constant power usage.

6.3 Device Model Validation

To validate the proposed device model, we now compare the average power consumption for the periodic sampling calculated with this new model to the power consumption traces collected on a N95 phone. These traces were mentioned earlier in Section 1 and have not been used in the above sections to derive the model. Therefore they can be used to validate that the model can explain the power consumption of a real phone with high precision. Figure 6.2(a) plots data from the collected trace for 60 seconds periodic tracking, overlaid with the predicted power consumption of the two other models. We can see how the proposed model closely matches the real power consumption, whereas the instantaneous model does not. Average values have also been calculated for a 30-minute scenario, which match the length of the traces collected on the N95 phone. The results of the collected traces and the two models have been plotted in Figure 6.2(b). The results show that the new model can describe the power consumption of a real phone with a much higher precision. Therefore this model can be used to inform the design of our tracking techniques towards minimizing the power consumption, whereas the instantaneous model has misinformed earlier research.

7 Discussion

The presented work have been based on a single device, i.e., the Nokia N95 8GB phone. Furthermore, we implemented our system using Python. To explore if our system can be generalized to other devices we measured the parameters for the device model on another Symbian S60 phone, i.e., the newer Nokia N96 phone. We found that the N96 phone's delays and power consumption are comparable to the N95. Our system can thus assumably be generalized to other devices running Symbian S60 and Python. Further work



(a) Over time for 60 seconds periodic tracking.

(b) Average values.

Figure 6.2: Power consumption on a Nokia N95, with the instantaneous model and the proposed model.

is needed to evaluate if our model generalizes to other programming environment, e.g., Symbian C++ or J2ME, other operating systems like Microsoft Windows Mobile and other brands of phones.

One method to address that device models might have to be parameterized for different brands of phones is auto calibration. Some operating systems provide an API for collecting power measurements. Given such an API it would be possible to write an application that could automatically profile a new device for power consumption and delays by starting and stopping features and measuring the consumed power when they power off. The needed parameters for the device model could then be calculated from the collected profiling data.

In our emulations and deployments we only experienced very bad GPS accuracy above 200 meter a few times. One of the methods EnTracked use to fight such situations is to use the GPS-estimated uncertainty to quickly schedule a new measurement if a potential bad measurement is received. For the cell network we generally experienced good communication throughput. However, in our logs we see that some resending of packets have taken place, and thereby increasing the delay for delivering position updates to the server.

In this work we evaluated tradeoffs between robustness and energy efficiency. In the case where we traded energy efficiency for robustness the power consumption was increased from 0.574 watt to 0.864 watt, which is a substantial increase. However, for this case we only changed one out of several possible adjustments points in our system. For instance, when we selected the delays in the device model we selected the average values instead of selecting more conservative values such as the average and two times the standard deviation. Therefore it would be relevant to study the different adjustment points in more detail, in order to understand which ones are the best to use for tuning the system towards either robustness or energy efficiency.

A simple threshold-based algorithm for movement detection was applied but more advanced algorithms do exists. Advanced algorithms could deter-

mine the mode of transportation such as those proposed by Reddy et al. [89]. It would be relevant to extend the systems with such algorithms to allow the tracking to scale to biking and car driving as well. Furthermore such algorithms are also better at handling the many different ways that a person can carry a phone; and ignore movements that are not part of an actual movement, e.g., gestures.

An optimization for the system would be to skip sending position updates to the server if the newly measured position is close to the one that the server already know. Also it could increase power savings for EnTracked if the accelerometer could be powered off when not needed; however, due to the Python library used, this is not possible in the current version. Also more power could be saved if the Python library allowed control of or simply was able to minimize the power-off delays. In this paper we focused on position-based applications that run on a server or a different device than the tracked device. However, for applications running locally on the device the proposed system could be used for optimizing the use of the GPS only.

8 Conclusions

The primary contribution of this paper is the novel EnTracked system that can track mobile devices robustly and energy-efficiently. We profiled how devices consume power for tracking and proposed a device model that can account for the real power consumption of a concrete device family. Furthermore, we propose methods for position tracking that take the changing system conditions into account, specifically radio delays, positioning delays and position accuracy. We also proposed a method that can minimize power consumption and satisfy robustness by calculating the optimal plan for when to power on and off features of the mobile devices such as the GPS module.

The results of our emulation was that the proposed methods can lower the energy consumption considerably and remain robust when faced with changing system conditions. These emulation results was validated by our real-world deployment where a mobile device was successfully energy-efficiently tracked in an urban environment. The results also provide insights into the limitations of our system and led to discussions on how to address these, e.g., by changing the trade-off between robustness and energy efficiency.

In our ongoing work we are trying to address several issues. These are: First, a further exploration of how to tune parameters of our system to realize the best trade-offs between robustness and energy efficiency. Second, propose methods for automatically determine the parameters of our device model for new devices. Third, apply the proposed methods and findings to other positioning technologies such as location fingerprinting [54].

Acknowledgments

We thank Anders Kabell Kristensen for helping collecting measurements. The authors acknowledge the financial support granted by the *Danish National Advanced Technology Foundation* under J.nr. 009-2007-2.

Paper IV

Exposing Position Uncertainty in Middleware

Jakob Langdal Jensen Kari Rye Schougaard
Mikkel Baun Kjærgaard Thomas Toftkjær

Abstract

Traditionally, the goal for positioning middleware is to provide developers with seamless position transparency, i.e., providing a connection between the application domain and the positioning sensors while hiding the complexity of the positioning technologies in use. A key part of the hidden complexity is the uncertainty associated to positions caused by inherent limitations when using sensors to convert physical phenomena to digital representations. We propose to use the notion of seamful design for developers to design a positioning middleware that provides transparent positioning and still allows developers some control of the uncertainty aspects of the positioning process. The design presented in this paper shows how uncertainty of positioning can be conceptualized and internalized into a positioning middleware. Furthermore, we argue that a developer who is interacting with uncertainty concepts is best supported when provided with a programming method with declarative constructs.

Published as: Langdal, J., Schougaard, K. R., Kjærgaard, M. B. and Toftkjær, T., Exposing Position Uncertainty in Middleware. In *Proceedings of the 2nd International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, 2010 (M-MPAC'10)

1 Introduction

Position information is a central form of context information, which is often used in pervasive computing applications. In this paper we present the design of a middleware for working with positioning systems. The middleware supports working with a cross-layer concern for positioning, namely uncertainty.

In pervasive computing the aim is often seamlessness. However, in some cases seamless use of for example sensors is problematic. Sensors are inevitably only accurate to a certain degree and will experience errors. In a seamless design it can be very difficult to know or let alone do anything about inaccuracies or errors. We therefore explore how uncertainty for position sensors can be expressed as a seam and how interaction with this seam can be supported in the middleware.

1.1 Contributions

We present the issues that have to be considered in order to enable seamful interaction (Section 3). The issues are presented as steps in a process for enabling interaction with a seam – a process to design middleware that support seamful interaction for developers.

We carry out the steps in the proposed process for position related uncertainty (Sections 3, 4, and 5)

We explore how seamless and seamful functionality can be combined by considering how an example would be implemented in three different programming styles (Section 5). The example implementations are evaluated according to localization of changes, separation of concerns, preservation of exiting code, separation of configuration code and handling code, and scalability in the number of concerns. We believe that the findings are generalizable to programmatic interaction with other kinds of seams.

2 Seamful Design for Developers

Motivated by the inherent imperfection of sensing technologies used in ubiquitous computing, several authors such as Chalmers and Galani [26], and Benford et al. [11] have argued for contrasting the goal of *seamless* design with one of *seamful* design. They define seams as "[...] the places where [components and technologies] may imperfectly connect to one another or to the physical environment." [11, p.126]. The goal for seamful design is to make the computer only appear in a designed manner so "one can selectively focus on and reveal [seams] when the task is to understand or even change the infrastructure." [26, p. 251].

Following this line of thought, we get that systems designed under the assumption that components within the system will connect and interact perfectly can be said to express a seamless design. A feature of seamless design is

that it aims to adhere to the principle of information hiding [84]. A benefit of seamless design and information hiding is that implementations are encapsulated and inaccessible outside its boundaries. This shields components of the system from being affected by changes inside other components. A seamful design, on the other hand, assumes that some components will connect perfectly for which a seamless design can be applied and other components will not. For the components that do not perfectly connect to other components or the physical environment the aim in addition to information hiding is to, in a designed manner, expose and internalize key aspects which capture the imperfect connection. Although, seamful design might break encapsulation, augmentation of existing components with a seamful design of carefully chosen parts can be useful.

Chalmers, Galani, and Benford et al. directed seamful design towards the end user. However Jensen et al [63] present seamful design for developers. Here, our focus is also on the developers who are subject to the same kind of imperfect connections, only they occur during application development. For the developer, the technologies that may imperfectly connect might both be software components, or as in the case of a position based application, the connection between the positioning technology and the physical environment. For a positioning middleware, this means that instead of only focusing on position transparency, the middleware should, in a designed manner, expose and internalize key aspects of positioning that capture the imperfection. In the field of positioning, imperfections are traditionally denoted as uncertainties. Previous positioning middleware designed for the traditional goal of transparency tends to result in the omission of explicit representation of imperfections in technologies and hides uncertainty for the developer [70, 62]. In some cases, this leaves the application developers at a loss, because the middleware they employ does not provide adequate supports for handling imperfections of the underlying technologies. The concept of seamful design for developers is proposed as a basis for developing middleware for domains influenced by technological imperfections.

3 Application

To apply seamful design to a problem we argue that a middleware developer has to consider the following issues:

1. Identify component relations that constitute a seam.
2. Identify key aspects which capture the seam.
3. Consider how key aspects can be conceptualized.
4. Internalize the aspects into the middleware and expose them to application developers.

In this section we discuss the first issue in a generic setting, and the three other issues are discussed in the context of applying seamful design to the domain of positioning in later sections. Here we present the issues as a sequential process, however, in practice seamful design is an iterative learning process that often requires a developer to re-iterate over components, key aspects and how to conceptualize these. For seamful design for users, Chalmers and Galani [26] have identified the same iterative nature of the design process. As a developer will gradually gain more and more experience with seamful design one might hypothesize that fever iterations might be needed.

To identify components that are affected by seams a first step is to consider technologies that are obvious candidates, e.g., wireless networking due to disconnections, delays or poor throughput, battery powered devices, because battery power is a limited resource, and technologies based on sensing the physical world, e.g., positioning, RFID etc.. In general, it might be hard to spot all seams, thus the discovery process does rely on previous experiences of developers and users.

3.1 Uncertainty as a Seam

In the domain of positioning applications, the positioning process contains a number of inherent properties that directly contribute to the overall quality of the positioning functionality. Following the discussion on seamful design, these properties constitute a seam in the specific domain. We argue that uncertainty is an aspect of positioning that captures a subset of the seam, and can be presented to the developer as a concept for understanding and focusing on the seam. In the further analysis we limit ourselves to the aspect of uncertainty, however, other aspects exist which could be applied to this type of imperfection, e.g., classical software engineering aspects like availability, performance, and security. However, from an application point of view, uncertainty is among the most important aspects to cover for positioning.

3.2 Conceptualization of Uncertainty

Having chosen uncertainty as an aspect capturing a seam in the positioning domain we must identify how this aspect can be conceptualized.

Measurements

One part of uncertainty can be conceptualized from measurement information provided by actual positioning sensors. Some positioning sensors provide estimates of current accuracy and precision. Here, accuracy refers to the closeness of several position fixes to the true, but unknown position of a target. Precision, on the other hand, refers to the closeness of a number of position fixes to their mean values as defined by Küpper [60]. Both of these values can either be provided at a specfic confidence level or as complete distributions

for each of the axes in a given coordinate system. Furthermore, because most positioning sensors sample the position with some discrete sampling frequency or on-demand, as opposed to continuously, there might be a delay between the time when a position is measured to the time when that position is delivered to an application. Therefore, we conceptualize this property as the *staleness* of a position reading, i.e., staleness becomes a measure of how the trustworthiness of a position measurement is degraded over time. The concept of staleness is similar in nature to the freshness of a sensor reading as defined by Ranganathan et al. [88].

Models

Uncertainty can also be conceptualized from one or several models of how the positioning performs. Given an area of interest an accuracy model can provide information about with which accuracy and precision you can expect positioning to work (as presented for 802.11 positioning by Lemelson et al. [65]). Such a model can be constructed based on knowledge about the type of sensors used, the surrounding environment, e.g., buildings and their interior parts like walls and openings. Such models can provide predictions for accuracy and precision, either as a specific confidence level or as a complete distribution. Furthermore, a latency model as presented by Kjærgaard et al. [53] can predict how fast we can get a new position fix which is especially important for GPS based positioning where the time to fix depends on the state of the GPS receiver and can vary from one second to several minutes.

High-level functionality

Furthermore, uncertainty can be conceptualized for high-level positioning functionality like proximity detection and target destination prediction. Assuming that the positioning process includes the calculation of enough parameters, uncertainty concepts for the probability of positions returned by the positioning system can be conceptualized. For instance, for proximity detection with a five meter threshold, the uncertainty concept would represent the likelihood of proximity events actually being correct.

3.3 Inspection and Internalization of Concepts

Given a list of concepts, a component can implement these in two manners. First, they can be implemented for inspection, which means that other components can retrieve data about the concepts (API-like). Second, the concepts can be internalized, which is an extension of the inspection ability, where a component provides an abstraction of the concept and functionality to support the use of that concept. At the same time, the concept might be used within the component for internal reasoning (Framework-like).

A positioning middleware designed for completely seamless use might deal only with positions and disregards any concepts of uncertainty. On a scale of how seamless/seamful a design is, this is an extremely seamless way of dealing with positioning for the developer. A more balanced approach might be a middleware where concepts of uncertainty are made inspectable, and some concepts internalized. At the seamful extreme of the scale would be, a seamfully designed positioning middleware that disregards any concepts of positioning, dealing only with uncertainty. Given probability distributions over space, position information can be inferred, e.g., as a maximum likelihood estimates. These distributions can also be used to calculate high-level information like proximity detection. One drawback of the seamlessly designed middleware is that is does not have any concepts for uncertainty, which at the same time can be an advantage, because of its the simplicity for the developer. Contrastingly, the complexity involved when developing applications using probability distributions is a drawback of seamfully designed positioning middleware. Furthermore, a performance penalty might exist for such middleware, because, the complexity of data structures for distributions are more computationally demanding than data structures for basic coordinate sets. The advantage, however, of a completely seamful design is that it provides the needed concepts for uncertainty all the way to the application developer.

In this paper we argue, that a positioning middleware should be designed for both seamless and seamful use, with concepts for both positioning and uncertainty internalized. The reason for this is that such a middleware should not only support one type of developers, but developers with different skills, short and long schedules, and different type of applications. Given a less skilled developer, who has to develop an application, e.g., for museum guidance, alone and in a short time frame, would most likely develop the application using positioning constructs and might handle only a few issues using the uncertainty concepts. If we instead consider that a skilled developer, who is in a large team and is working with a long time frame, has to develop a firefighter support system based on positioning, he might develop most of the application using only uncertainty concepts and only, in a few places, use the positioning constructs, e.g., due to performance constraints or in non-critical areas of the application.

Therefore, we need a positioning middleware that contain both seamful and seamless functionality and which provides separation and eases the combination of seamlessly and seamfully designed functionality. Our positioning middleware design therefore has to satisfy the following two demands. First, we want to be able to structurally separate the functionality designed for seamless use from the functionality designed for seamful use. Second, we want developers to be able to use functionality that is seamfully designed in the same context as functionality that is seamlessly designed. Finally, the second demand should be satisfied in such a manner that the developer is able to use seamlessly designed functionality without concern for the seamful

functionality, e.g., if the developer chooses to ignore or is unaware of imperfect connections. At a later point the developer should be able start using functionality designed for seamful use in the same context with little additional effort. How these demands can be satisfied in a seamfully designed software component is considered in Sec. 4 for the case of a positioning middleware.

4 Seamful Design for Positioning Middleware

Positioning applications rely on the capability to establish the position of a target. This positioning is based on readings from physical sensors, often of various makes and models. Traditionally, the primary responsibility for middleware is to hide complexity of a certain domain, and to provide a transparent experience working with heterogeneous technologies. In the case of positioning middleware, this involves hiding the imperfections of sensor technologies which lead to uncertainty of the position. The uncertainty often originates from the internal details of the positioning technologies employed, and these details are usually abstracted by the middleware and hidden from the developer in order to present a clean and understandable interface. Depending on the implementation, some positioning middleware handles uncertainty within the internal processing of position readings, however, the uncertainty is still hidden from the developer, or to some extend accessible through simple inspection.

We propose to tackle uncertainty as a seam in the fabric of the architecture, leading to a middleware with uncertainty explicitly designed into its structure, allowing the developer to access and manipulate the middleware in terms of concepts that are related to uncertainty. As mentioned in Sec. 2, one goal is to achieve structural separation of the functionality designed for seamless use from the functionality which is part of the seamful design.

4.1 Architecture

In agreement with common practice, we propose a middleware design based on a layered architecture consisting of three layers: sensor layer, middleware layer, and application layer. The middleware layer has a core consisting of at least two separate components, one responsible for managing positions, and one responsible for managing uncertainty. Both components have access to the underlying sensor layer and both are accessible by the application level developer. In Fig. 4.1 we see how these components are connected. The responsibility of the positioning component is to provide the seamless developer experience for controlling the positioning middleware, while the uncertainty component is responsible for exposing and internalizing the concepts identified for uncertainty in Sec. 3.1, thus constituting the seamful design.

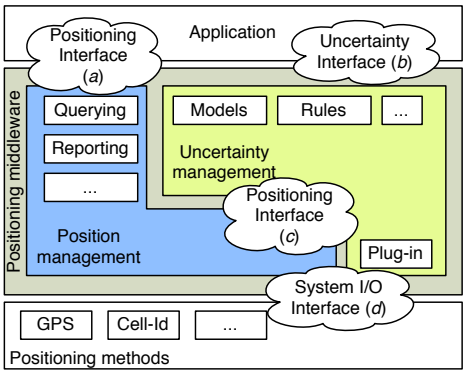


Figure 4.1: An architecture for positioning middleware supporting uncertainty through seamful design.

Positioning

When working seamlessly with positioning, Positioning interface (a) is used as interface between the application layer and the middleware, and the interface is designed for position transparency. In the basic use-case, the positioning middleware is used as a position provider. The provision of positions is controlled through Positioning interface (a), which provides querying and reporting mechanisms. The querying mechanism corresponds to the inspection property described in Sec. 3.1, while reporting is an example of internalization of a concept. Reporting, as it is used in this middleware, is the functionality where the application is notified of changes in positioning when certain criteria is satisfied. One example where reporting represents the internalization of a concept, from the positioning domain, is distance based reporting [67], where the middleware reports positions to the application only when the tracked target has traveled some specified distance. Here, the concept of distance is captured and internalized by the middleware and presented to the developer as an abstraction. In the proposed design, the positioning component is not required to expose any knowledge of uncertainty, and depending on actual implementations, uncertainty can either be ignored or be handled completely hidden from the developer, thus providing a seamless use of the positioning functionality. Several standard frameworks as well as research prototypes for working with positions fit into the positioning component; among the popular ones are the J2ME Location API [70], used in mobile devices, and the research platform TraX [62]. In some cases these positioning systems will provide access to concepts related to uncertainty, e.g., accuracy and precision, as part of the objects representing positions. When such systems are used for the positioning component, the middleware architecture presented in this paper suggests that these properties are exposed through the uncertainty component, even

though this might lead to double representation.

Uncertainty

The second core component encapsulates functionality for conceptualizing and exposing uncertainty. This component constitutes the uncertainty aspect of positioning which is made seamful, and it provides both inspection functionality as well as internalization of uncertainty concepts. When the developer needs to access uncertainty information, Uncertainty interface (b) is used. There are two distinct scenarios for using the uncertainty component.

The first scenario is when the developer is making changes to application state or control flow based on the current level of uncertainty. Here, the inspection ability of the uncertainty component is used, e.g., for branching in a conditional statement or modifying some internal state of the application. To this purpose, the uncertainty component continuously collects and maintains the uncertainty properties of any sensor reading related to the tracked target.

The second scenario is when the developer uses the positioning middleware through the seamless interface (a) and he requires that the middleware deals with an aspect of the uncertainty without it affecting the way he accesses the positioning component. A simple example of this is when the positioning component should change silently to a fallback positioning method if certain uncertainty events occur, e.g., from GPS to Cell-Id. For the uncertainty component to support this, it can control the positioning component directly through Positioning interface (d) or indirectly through the manipulation of the underlying sensor layer via interface (d). Through this mechanism, the uncertainty component is capable of, indirectly, manipulating the positions returned to the application through the interface (a), e.g., for improving the position transparently, providing positions based on estimated behavior patterns, fusion of several sensor values, etc.

Sensing

Both the positioning and the uncertainty component interacts with the underlying sensors through System I/O interface (d). The positioning component uses (d) to get sensor readings. The uncertainty component uses (d) to inject modified positions transparently into the positioning component. Furthermore, the sensing layer plays the role of hardware abstraction layer for both components in the middleware layer.

The proposed middleware has access to a number of positioning methods supported by physical sensors. Both the positioning component and the uncertainty component gives rise to at least three basic requirements which must be fulfilled by the sensors supported and by extension provided by the interface (d). First, the positioning component requires that sensors can provide readings, which can be translated into a position or a coordinate. Second, the

uncertainty component requires that the sensors provide enough information that some level of uncertainty can be estimated. Third, the uncertainty component requires that sensor readings made by the positioning component can be manipulated by the uncertainty component.

5 Combining Seamless and Seamful Functionality

With the middleware providing structural separation of positioning and uncertainty concerns, we propose a solution for how to combine functionality from the two concerns in the same context. The solution is exemplified by the following application. A museum is digitally augmented to provide enhanced experiences where the guests are guided through the museum by a visual representation of their position. Furthermore, special places of interest throughout the museum are presented through an audio track which is triggered by the proximity of the guests. It is assumed that each guest is carrying a positioning device and an instance of the application on a handheld device capable of providing visual and audio presentation. The examples in this section are based on the following hypothetical setting.

The application for museums described above is deployed and running in the final environment. Following this, the administrator of the system discovers that users are experiencing erroneous position indications on their devices when entering specific areas of the museum. To alleviate this problem, the administrator requests the following change in functionality. When horizontal accuracy of the positioning device is greater than x meters, the triggering of audio presentations should be adjusted to be more conservative and the visualization of the guests position should be augmented with a circle indicating the level of accuracy. Furthermore, when no position can be calculated for a period of t seconds, the application should switch to a fallback sensor.

In Fig. 5.1, we see an overview of the three main classes involved for providing the visualization and triggering of audio functionality. There is a class responsible for showing the position on a map, a class responsible for selecting the audio track associated to a specific position, and a class responsible for coordinating the control flow of the application. In the following, some of the changes to the museum application are described using three different programming styles: a simplistic API style, an event based style, and an annotation based declarative style. The styles assume that the changes are implemented using a positioning middleware adhering to the design proposed in Sec 4. For illustration purposes, only three methods are used in the examples, they are: `showOnMap`, `playAudioTrack`, and `locationUpdated`. The method for updating the visual representation is registered with the positioning middleware for periodic updates, and the method for selecting audio is registered as a proximity based callback. A guiding principle for all examples is that implementation of the functionality should attempt to leave the existing code

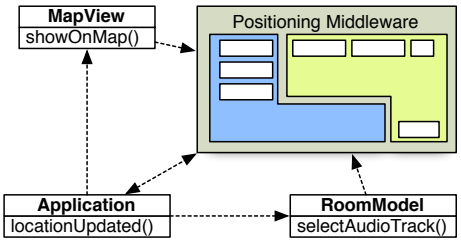


Figure 5.1: Design of the museum application, showing the main classes, and the dependencies to the positioning middleware

unmodified.

5.1 Naive API

The first implementation is an example of how to use the currently available functionality of the J2ME Location API [70]. The following code listing illustrates how the `playAudioTrack` method is modified to react to changes in horizontal accuracy. The method `getHorizontalAccuracy` provides an estimation of the horizontal error in meters, specified as the 1-σ standard deviation, and the method `addProximityListener` instructs the middleware to call a method on the provided listener object when the tracked target is detected to be within range of the coordinate as specified in the parameters. The marked lines and the method signature belongs to the original method.

```
public void playAudioTrack(
  Coordinates registeredCoordinates,
  Location location) {
  float accuracy = location.getQualifiedCoordinates()
                    .getHorizontalAccuracy();
  if (accuracy < x) {
⇒    AudioTrack audioTrack =
    audioTracks.get(registeredCoordinates);
⇒    audioTrack.play();
⇒    locationProvider.addProximityListener(this,
⇒      registeredCoordinates,  proximityRadius);
  } else
    locationProvider.addProximityListener(this,
      registeredCoordinates,  proximityRadius -
        .getQualifiedCoordinates()
        .getHorizontalAccuracy());
    }
}
```

Only the proximity part of the use-case is shown here, however, the update of the visualization functionality, represented by the `showOnMap` method, is analogue to this. The fallback functionality required by the use-case is also omitted in the example, however, it can be implemented reacting to a timer and obtaining a new location provider from the middleware, i.e., the developer has to explicitly define the functionality with little support from the middleware. The implementation, as it is shown in the listing above, handles the decrease of the threshold for proximity detection, however, it does not increase it again when the horizontal accuracy improves. To implement this, the accuracy needs to be monitored. This can be achieved by adding another proximity callback for the original radius or by adding a periodic update method. However, both these solutions will require some book-keeping, e.g., registration and de-registration of the correct proximity listeners, which will result in a significant amount of code. Moreover, this type of book-keeping is required for each part of the code which receives positions from the middleware.

The implementation does not encourage separation of uncertainty code from position code. Registration of proximity event handlers depends on horizontal accuracy levels. Therefore, the accuracy level is inspected with the purpose of setting a new proximity threshold. Instead of querying the accuracy within the method, the registration management could be separated into different methods. While this is indeed possible, it would require massive restructuring of the existing code. Furthermore, as code for managing uncertainty is mixed into the existing methods for handling position updates, a detailed understanding of the existing code is required to make the modifications.

Following this, the configuration of how the positioning middleware should behave is not separated from the actual handling of the uncertainty property. The parameters used for changing the proximity threshold is only available when proximity is detected. Therefore, the reconfiguration of the new proximity threshold is tightly linked to uncertainty handling code. Instead of branching on the accuracy within the `playAudioTrack` method, the implementation might extract the branching structure and wrap the call to the method. However, this solution would still require a change of the existing method, because, the original uses the same middleware configuration as would be used in the extracted branching structure. Finally, the implementation using only the functionality of J2ME Location API scales poorly with the number of concerns involved. The features of the API-style approach for interacting with the middleware is summarized in Table 5.1 along with the features of the event-style and the annotation-style approaches.

5.2 Events

The second implementation assumes, that the middleware raises events related to horizontal accuracy, and that the concept is internalized so that the

middleware can determine if a specified level of accuracy is maintained. In the listing, events are raised when horizontal accuracy exceeds a specified level, and assuming the event handler `AccuracyLimitExceededHandler` is registered with the middleware, the following code listing is sufficient to realize the uncertainty handling:

```
public void onAccuracyLimitExceeded(
  Location location, int accuracy) {
  for (Coordinate c : pointsOfInterest) {
    locationProvider.removeProximityListener(this);
    locationProvider.addProximityListener(this, c,
      radius-accuracy);
  }
}
```

In addition, the event handlers must be configured, which might be implemented by the following code:

```
locationProvicer.addAccuracyLimitHandler(this, x);
```

As with the API-style implementation, the visualization and fallback scenarios are left out of the listing. The visualization can be implemented as a new method in the `MapView` class adding a map overlay and hooked up with an event listener like above. The fallback feature is implemented analogous to the proximity feature shown. A feature of the implementation using events is that the original code is only modified with a single line, i.e., for registration of the event listener. It should even be possible to do this registration in a centralized way, in effect collecting all uncertainty configuration at a single point in the application. Furthermore, the developer is not required to implement functionality for determining whether the accuracy level is exceeded as the middleware has internalized the concept of accuracy However, the application developer is required to have detailed knowledge of the positioning middleware in order to handle the uncertainty. In the example shown for event handling, the way to change the behavior of the positioning middleware involves manipulating the proximity listener registrations. This requires the developer to be aware of the exact behavior of proximity updates and listening mechanisms as well as any possible side-effects the manipulation might have on other parts of the application. Furthermore, the specific handling of the consequences of uncertainty is not generalized in a way that it can be used for situations similar to the one above. Therefore, the solution of event handlers might not scale particular effectively with the number of concerns of uncertainty handled.

5.3 Annotations

The final example of interaction is to use Java 1.5 metadata annotations [16] for decorating the code. The applied positioning middleware provides access

to the internalized concepts of uncertainty through annotations which can be applied to methods and classes.

```
@Proximity
@RequiresHorizontalAccuracy(limit=x)
@AllowFusion
@Fallback(timeout=t,method="auto")
public void playAudioTrack(
  Coordinates registeredCoordinates,
  Location location) {
  AudioTrack audioTrack =
  audioTracks.get(registeredCoordinates);
  audioTrack.play();
}
...
@RequiresHorizontalAccuracy(limit=x)
@UserHandler(handler=MapOverlayer.class)
@UserOptimization(optimizer=MyOptimizer.class)
public void showOnMap(Coordinates c) {
  ...
}
```

The first two annotations of the `playAudioTrack` method tells the positioning middleware that the method requires a horizontal accuracy of at least x to be notified of proximity to a target. The middleware has internalized the concepts of horizontal accuracy and proximity. Therefore, the middleware is capable of using these concepts to ensure that the method is only called if the accuracy limit is satisfied. Moreover, the middleware is free to optimize the position by employing any strategy available, e.g., fusion of more sensors (`@AllowFusion`). A strategy could even be to use an optimization strategy provided by the application (`@UserOptimization`). The `showOnMap` method is annotated with the annotation `@UserHandler` which tells the middleware to use the class specified as a handler of the uncertainty when the limit is exceeded.

The use of annotations to declare where uncertainty needs to be handled leads to the clear separation of concerns, leaving the existing application code which uses positions unaffected. Furthermore, the addition of the annotations can be localized to specific parts of the application where uncertainty is a problem. Because the annotations capture specific concepts of uncertainty, the technique scales well with the number of concerns. In the example above, if the method was called based on distance traveled as well as proximity, an annotation called `@DistanceBased` could be prepended to the method declaration. This tells the positioning middleware that the accuracy limit should be maintained for distance based reporting as well. Finally, the use of annotations

Table 5.1: Properties of the three styles

Property	Naive API	Event	Annotation
Localizes changes to existing code	no	yes	yes
Separates uncertainty and position code	no	no	yes
Preserves existing code	no	yes	yes
Separates configuration from handling	no	yes	yes
Scales well with number of concerns	no	no	yes

leaves the configuration of the positioning middleware completely separated from the implementation of the actual handling of uncertainty. Given the five properties listed in Table 5.1 the annotative style is the only one which can satisfy all of them.

6 Discussion

In this paper we have been primarily focused on using seamful design for developers within the domain of positioning applications. However, we believe that the methodology is useful in other domains as well. To determine to what extend this generalization is possible further work is needed. A first step in determining the generality is to identify how seams in other application domains compares to the properties of uncertainty.

Within the domain of positioning applications, uncertainty is not the only aspect that might be exposed through seamful design. As positioning technologies become more pervasive in everyday life, developers of positioning applications must be increasingly aware of potential privacy concerns. The nature of privacy concerns is in some cases very similar to those of uncertainty. One way to improve privacy is for the end user to be able to adjust the accuracy of reported positions which is a functionality well suited to be captured by a seamful design.

In our application of the proposed design process, we only identified a few concepts relating to uncertainty, however, there definitely exists many other, highly relevant, concepts. Furthermore, the concepts already identified are not definitive and may be extended or modified by further iterations of the design process. In general, the concepts identified in the process are likely to be based on the experiences of the actual designers using the process in combination with the specific application domain in which the process is applied. Further study of how uncertainty can be conceptualized for a broad range of application domains might provide insights into concepts that are common to uncertainty in general.

The middleware presented in this paper is applied to a simple application domain. An evaluation of the middleware used for more demanding applications should contribute to a better understanding of how the middleware

performs.

In this paper, we presented a middleware with an uncertainty component reflecting the features of a positioning component modeled as JSR-179 which is a light-weight middleware. In future work, more complex positioning middleware should be augmented with a seamful uncertainty component to see if the complexity of the uncertainty component is dependent on the complexity of the positioning component.

7 Related Work

In this section we will cover related work with respect to existing positioning middleware and to reflective middleware.

7.1 Positioning Middleware

Several types of middleware have been proposed to provide position transparency, handle sensor heterogeneity, and to provide high-level functionality such as reasoning about spatial relationships.

The TraX [62] middleware focuses on message efficient tracking of targets and the detection of spatial relationships among multiple targets. The middleware do not providing uncertainty information for positions and detections. The J2ME Location API [70] provides positions from different positioning providers and supports the detection of proximity. Furthermore, the API provides access to uncertainty information in terms of estimated horizontal and vertical accuracy for each delivered position.

The Location Stack [42] focus on handling sensor heterogeneity and sensor fusion. This middleware provides information about uncertainty using discrete probability distributions tightly linked to the use of particle filters for sensor fusion. The use of particle filters has influenced their choice of describing uncertainty of positions as conditional probabilities for use in the particle filter.

The positioning middleware MiddleWhere [88], provides and fuses positions from several sensor sources and provides functionality for working with spatial relationships. The middleware associates uncertainty as a confidence level with positions and calculates spatial relationships. Furthermore, the middleware handles the temporal degradation of the quality of positions by reducing the confidence over time. The paper furthermore recognize that developers might have trouble interpreting probabilities in all circumstances and propose to simplify the confidence values into four states (low, medium, high, very high).

To sum up, existing positioning middleware spans from not considering uncertainty over time-degraded confidence levels to detailed representations as particle filters. The different examples of middleware generally provide inspection in an API-like fashion, but some also provide event constructs for

positioning functionality that take uncertainty into account. In comparison, our work apply seamful design for developers to conceptualize key aspects of uncertainty. The proposed concepts for uncertainty capture more aspects of uncertainty than previous positioning middleware, e.g., the use of models to capture uncertainty aspects. Furthermore, we propose to separate uncertainty and positioning functionality which supports developers with different skills and task when using the positioning middleware with different goals. To combine the functionality of positioning and uncertainty functionality we argue for a declarative style of design based on code annotation.

8 Conclusion

We explored the notion of seamful design for developers in designing a positioning middleware that enable developers to deal with uncertainty in a structured manner while minimizing the effect on position transparency. This is done by conceptualizing key aspects of positioning that has an influence on uncertainty. Furthermore, we propose an architecture for a positioning middleware with uncertainty handling along with an analysis of methods used to interact with the middleware. Finally, we argue, that the best suited programming methods is to use a declarative method that: enables a high locality of implementation; has explicit representation; can leave existing domain code untouched; and enables scaling of uncertainty handling with only minor increase in complexity.

This work outlines several paths of future work. First, it would be of interest to evaluate the designed positioning middleware with more demanding applications such as fire fighter support and herd management. Second, augmentation of a more complex positioning middleware with a seamful uncertainty component to see if the complexity of the uncertainty component is dependent on the complexity of the positioning component. Thirdly, study seamful design for developers for different developers with different skills and tasks. Fourthly, further study into how uncertainty can be conceptualized. Finally, apply seamful design for developers to other domains.

Acknowledgements

The authors acknowledge the financial support granted by the Danish National Advanced Technology Foundation under J.nr. 009-2007-2.

Paper V

Implementing Tactics for Positioning Quality Improvements in Lightweight Extensions to Positioning Middleware

Jakob Langdal Jensen Kari Rye Schougaard

Abstract

Location aware application development can be effectively supported by using a positioning middleware. However, for many pervasive computing applications, positioning quality can be improved by exploiting domain or application specific information in an implementation of a positioning quality tactic. To do this, access is required to specific details of the positioning process, that is usually hidden by the positioning middleware. By implementing, evaluating, and analyzing three tactics, we show that positioning quality tactics can be implemented as lightweight extensions to adaptable positioning middleware. We conclude that quality improvements can be implemented as middleware extensions in a compact, learnable and modular way. Thus, extending an adaptable middleware is, indeed, a desirable approach to achieve application or domain specific quality improvements, compared to implementing all of the position processing in an ad hoc way.

Unpublished, in preparation.

1 Introduction

Location-aware application development often involves using a broad range of positioning technologies. Positioning middleware effectively supports the developer in dealing with the sensor heterogeneity [70, 42, 88, 9, 63]. However, the differences in accuracy, availability, latency, power efficiency, and other positioning related qualities cannot be hidden from the application altogether. Sometimes the lack of quality or the quality variations are problematic for the application. Thus, there exists a range of positioning quality tactics[1] for improving specific qualities. Often, application developers can exploit knowledge of the application domain or the application logic to achieve a specific instantiation of a tactic. This, however, requires access to low level details in the positioning process. As middleware is traditionally designed for transparent use, this requirement often means that the application developer cannot use the middleware.

An adaptable positioning middleware is a positioning middleware that supports accessing and modifying the internal middleware functionality. The approach of (re)configuring and extending an adaptable middleware to tailor it for specific circumstances is a major topic in the field of distributed middleware, however most work is focused on reconfigurations.

We claim that it is possible to implement specific instantiations of tactics as extensions to an adaptable positioning middleware and, moreover, that the middleware extensions can be compact, learnable and modular. When middleware adaptations are proven to be an applicable approach for handling quality improvements application developers can choose to use a middleware instead of building ad hoc solutions into their applications.

This paper explains a number of implementations of positioning quality extensions of an adaptable positioning middleware. The extensions are aimed for use by application developers in the relevant domain. The concrete implementations adapts the PerPos middleware [63], however, we state the requirements of the extensions in general terms of access to the positioning process and we also explain how these requirements could be fulfilled by generic reflective positioning middleware.

Contributions. Our main contribution is to show that a number of positioning quality tactics can be integrated in an adaptable positioning middleware in a compact, learnable and modular way. This is shown by implementing and analyzing the following tactics with regards to strengths and limitations:

- Accuracy: Filtering based on sensor input detail.

[1] The use of the term tactic is analogue to its use in software architecture, where a tactic is a fundamental design decision and a proven means to achieve a quality [7]

- Power efficiency: Toggling a power consuming sensor on/off based on expected signal reception.
- Reliability: Averaging over sensor input values based on movement model.

2 Adaptable Middlewares

The heterogeneous environments of pervasive computing have sparked research in adaptable middlewares [63, 23, 92]. Adaptability can be implemented directly in the middleware as in the PerPos middleware [63] or by building the middleware on top of a generic reflective middleware [21, 31]. The adaptation can be achieved through explicit reconfiguration of components or through extensions inserted at specific interception points, e.g., in the bindings between components. We focus on the latter, insertable *extensions*.

The implementations of position quality tactics presented in this paper depend on a number of adaptation features in the middleware. Generally, the approach is limited to middleware where the core functionality of extracting a high level position from heterogenous low level sensor input is implemented by a number of components that process the input from the position sensors in a progressive refinement of position information. In this setting the tactic implementations require:

1. Ability to access the input values for a component method.
2. Ability to access the output value of a component method.
3. Ability to cancel the output of a component method.
4. Ability to replace the output of a component method with another value.
5. Ability to add data to the output of a component method.
6. Ability to access the implementing class of a component and ability to call methods of the implementing class.

The first 4 abilities are, generally, satisfied by `before`, `around` and `after` constructs in languages like Lisp and by the corresponding aspect oriented programming constructs. In Fractal [21] they are supported by redefining the bindings between the components to allow interception or by using component controllers to intercept requests. Adding data to the output of a component method can be implemented by adding a parallel call that logically happens at the same time as the original call. The last ability is satisfiable with generic language reflection.

Concretely, possible extensions depend on the implementation of the middleware. It is simpler to develop an alternative behavior for a component method that carries out a well-defined step in a chain of processing steps than

Table 3.1: Hooks available to features in PerPos

Name	Description
`processOutput`	Process data produced by a component or a feature attached to it.
`processInput`	Process data received by a component.
`cancel`	Allows a feature attached to a component to cancel further processing of data done by the component.
`produce`	Allows a feature to produce data on behalf of a component.
`reflect`	Allows a feature access to the implementing artefacts of a component through language based reflection.

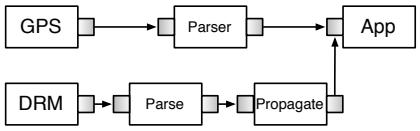
for a complex method that takes raw sensor data as input and outputs a high-level position. Moreover, when accessing the implementing class of a component, the implementation of this class determines the kind of information that can be extracted and to what extend the component can be controlled. The extension developer needs semantic understanding of the implementing class. Middleware designed for being opened will take this into account.

3 Concrete Positioning Middleware Quality Extensions

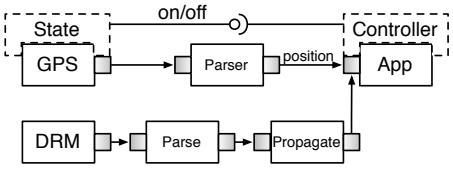
Using an adaptive middleware we have implemented tactics for: improving power efficiency, accuracy, and reliability in an application that uses a Dead Reckoning Module (DRM) and a GPS device for obtaining positions in both indoor and outdoor environments. The application relies on the GPS device to provide absolute positions, and when the satellite reception is blocked, it uses the DRM to keep updating the position based on relative movement events.

In the following we present how the three tactics have been implemented as extensions to PerPos [63], an adaptable positioning middleware, which we will start by introducing. As the instantiations of the tactics are dependent on details relevant only to the concrete application, generic positioning middlewares are unlikely to include implementations of this kind of tactics.

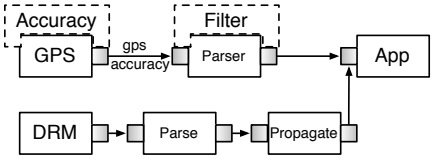
The PerPos middleware [63] exposes a model (a meta representation) of the positioning process of the middleware (Fig. 3.1(a)). The PerPos middleware provides methods for manipulating the structure of this model. Furthermore, the functionality of the middleware can be extended by applying augmentations, called *Features*, to the components. PerPos exposes a number of hooks allowing interception of data between components and manipulation of the individual components. The relevant hooks provided by PerPos are shown in Table 3.1



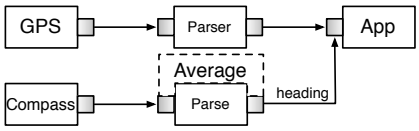
(a) The basic model exposed by the positioning middleware representing the unmodified state of the application.



(b) The model as it is exposed after being extended with features implementing a power conserving tactic.



(c) The model extended with filtering based on GPS accuracy.



(d) The model extended with a feature implementing tactic that averages compass readings when experiencing strong magnetic fluctuations.

Figure 3.1: The model of the positioning process can be explicitly exposed and manipulated through the adaptable positioning middleware.

3.1 Power Conservation

The power conserving tactic uses knowledge of reception in the deployment area to turn off the GPS device when GPS reception is not possible and turning it on again when reception becomes possible (based on historic knowledge of reception). It is implemented by extending the positioning middleware at two points. The application uses the positioning middleware to connect to two sensors, a DRM and a GPS device. The power conserving tactic is implemented as two features, `State` and `Controller`, which are attached to the `GPS` and `App` components as illustrated in Fig. 3.1(b). The `State` feature exploits that the PerPos middleware supports the language reflection requirement (number 6) through the `reflect` hook. The feature uses detailed knowledge of the `GPS` component to expose a method for toggling the power for the underlying GPS device. This method is made accesible through the meta model of the positioning processing. The `Controller` feature uses the `processInput` hook to monitor whether or not the positions received as input to the `App` component lie within areas with good GPS reception. Based on the monitored positions the `Controller` feature uses the `State` feature to toggle the power state of the GPS device.

3.2 Position Accuracy

In the second tactic applied to the application, a filter is inserted before positions are delivered to the application. The filter tests whether the accuracy of the current GPS readings are within an acceptable range and allows only acceptable readings to be passed on to the application. Thus, the tactic improves the accuracy of the received readings. The implementation of this tactic also consists of two features that are applied to the application model (Fig. 3.1(c)). The `Accuracy` feature uses the `processOutput` hook to read the output of the `GPS` component. Based on this output it calculates the current accuracy of the GPS and uses the `produce` hook to augment the data produced by the `GPS` component with accuracy data. The `Filter` feature is attached to the `Parse` component and it uses the `processInput` hook to intercept input. Based on the received accuracy data, the filter decides if the `Parse` component is allowed to process the GPS data. The decision is effectuated using the `cancel` hook to instruct the `Parse` component not to send output produced from unacceptable input.

3.3 Reliability

The third tactic applies to a modified sensor setup. In this application the DRM is replaced with a simple electronic compass. Electronic compasses are highly susceptible to changes in the magnetic fields of the local environment causing the readings to fluctuate. Therefore, one tactic for improving the consistency of compass readings is to provide applications with a rolling average of the heading instead of providing the discrete readings directly. The concrete tactic presented here is built on the assumption that the general motion of the user can be codified in a motion model, e.g., walking, running, driving etc. This motion model is integrated into the parameters of the rolling average calculated by the extension. The tactic is implemented as a single feature, `Average`, attached to the `Parse` component (Fig. 3.1(d)). The feature uses the `processOutput` and the `cancel` hooks to intercept and prevent heading readings being propagated to the application. While intercepting, the feature calculates the rolling average and uses the `produce` hook to provide the application with the result.

3.4 Evaluation of the Concrete Extensions

In this section we evaluate the compactness, learnability and modularity of the concrete extensions. The numbers presented in this section stem from the concrete choice of tactics and underlying middleware. However, we argue that they show a general trend.

Compact Code The "intelligence" involved in instantiating the tactic can be arbitrarily complex, and this intelligence will be necessary both when mak-

ing an ad hoc solution and when adapting a middleware. We therefore evaluate on the code, that must be written to attach the tactic to the middleware process. For the concrete examples this amounts to 25, 27 and 30 lines of code and 8, 10, and 5 lines of configuration information (listed in the following order: power conservation, accuracy and reliability). All line counts are excluding auto generated artefacts, e.g., import statements. This should be compared to the number of lines of code in the middleware components for parsing and extracting the high-level position information from the sensor output. Code that would have to be implemented by the application developer if not using a middleware. For the concrete middleware used those numbers are approximately: $GPS \approx 4000$ lines, DRM ≈ 1300 lines and $Compass \approx 250$ lines.

Learnability By learnability we mean "The capability of the software product to enable the user to learn its application" [46]. In this context the learnability is of the approach, i.e., of the adaptation facilities of the PerPos middleware [63]. The adaptation facilities consist of the 5 hooks specified in Table 3.1. When comparing to other APIs, we think that learning to use the 5 hooks and understanding how the hooks weave the extension developers' code into the middleware is manageable for programmers with understanding of generic interception principles.

Modularization Each of the example tactic implementations is defined in its own class, and the inclusion of the tactic is specified in the model. This makes it easier to change the tactic and to in- or exclude it, than if the code was inserted as an integrated part of several position processing components.

3.5 Analysis of the Middleware Extension Approach

The extensions presented in this paper are just that, extensions of existing behavior. In particular, they exploit the fact that the progressive processing of sensor readings into high-level abstractions can be explicitly modeled by middleware and thereby motivate reuse. In component based systems it is, generally, possible to reconfigure existing components and add new components that fit into the structure. Some positioning quality tactics will be implemented by reconfiguring existing components, e.g., improving latency by using fast but imprecise positioning techniques when starting a position-based application, others by adding a new component, e.g., decreasing uncertainty by combining sensor input from different kinds of sensors.

When the application developer wants to change how a certain functionality in the middleware is carried out, it is easier to extend the existing components. However, extensions are limited to change existing functionality indirectly. Extensions can change applications and high-level middleware functionality by manipulating the processing of sensor data, thereby changing stimuli to existing functionality. Therefore, the scope of extensions are

directly dependent upon the granularity with which the middleware exposes the refinement process.

4 Related Work

The related work falls in two categories. Tactics used within application or as an integrated parts of a middleware; and tactics implemented as middleware adaptations.

The uses and studies of positioning tactics referenced here involve both different qualities and different use settings, there are too many to reference all. First a number of tactics for improving energy efficiency. The tactic of turning down the positioning frequency either using a frequency based scheme or a motion based is used by [6]. Turning off unused sensors for achieving power-efficiency is used by [6] For applications that uses the location of several mobile devices in a central application, the tactic of communicating the location less frequently has been researched in [34, 78] and is in [6] combined with server side predictions. Second, the uncertainty improving tactic of combining sensor input from different kinds of sensors, using an error model, is included in the middlewares presented in [42, 88] and used in [97]. And finally, the tactic described in Sec. 3.5 for improving latency is well known from smartphones. Our approach distinguishes itself from these approaches in that the tactics are implemented as extensions to, and are optional parts of a positioning middleware.

Middleware adaptations to improve certain qualities include the use of policies to define middleware behavior. In [23], policies are used to reconfigure the middleware behavior when the connectivity is changing. Policies can be used for context-based adaptation of the configuration of the positioning system in [9] and context-based reconfiguration of the middleware to change the properties like communication strategy in [92]. The adaptations presented in [23, 92] are solely based on reconfiguring the components in use, where we focus on component adaptation. Moreover, the presentations in the papers are focused on presenting the adaptation capabilities of the middleware rather than focusing on analyzing and understanding the adaptations which is the goal of this paper.

5 Conclusion

We have presented three middleware extensions, that show that positioning quality improving tactics can be implemented as extensions. The example extensions are implemented by:

- accessing the implementing class of a sensor component in order to issue control commands for the sensor

- accessing the implementing class of a component in order to extract extra information
- sending extra data to the next component in the processing line
- use information from input to a component to filter the components output
- use a function on the output(s) of a component (concretely average) before sending the output to the next component

Many other tactics can be implemented by using this approach. As the extensions are compact and learnable, this opens the possibility of using a positioning middleware, even when the application demands certain quality improvements that would be difficult to implement on top of a traditional transparent positioning middleware.

An extension of this work is to implement more tactics, e.g., those mentioned in related work. We plan to investigate how the use of tactics to improve one quality affects other qualities and how the use of several tactics work together when implemented as extensions to a middleware.

Paper VI

Quality trade-offs in position based applications: A survey and taxonomy of tactics for improving positioning related qualities.

Jakob Langdal Jensen Kari Rye Schougaard

Abstract

Location-based applications often need to improve a specific positioning related quality such as accuracy, reliability, freshness, or power efficiency. Nonetheless, there does not exist a survey of tactics to improve positioning related qualities and how they influence various positioning qualities. Location Based applications have become more widespread, especially because of the availability of cheap GPS devices and the popularity of smart phones with several positioning possibilities. This means that the task of writing applications that use positioning is becoming a mainstream task instead of a specialist job. We provide an overview of positioning specific qualities. Furthermore, we introduce the concept of

Unpublished, in preparation.

tactics for improving positioning related qualities. We present a taxonomy of positioning related tactics that includes both best practice tactics and research based tactics. A tactic will be targeted to improve a specific quality, however, other qualities may increase or decrease as a consequence of using this tactic. We argue for the quality improvements and deteriorations that follow from a tactic based on the literature, analyses, and experiments. This taxonomy of positioning tactics will improve the ability of developers to produce high-quality location based applications.

1 Introduction

High-quality positioning is emerging as a problem for application developers as positioning applications become more widespread, for example, through the penetration of positioning enabled smart-phones. Positioning applications, which may use location based services, are affected by a number of qualities that are inherently interdependent. Location-aware application development often involves using a broad range of positioning technologies. However, the differences and dynamic variations in accuracy, reliability, freshness, power efficiency, and other positioning related qualities cannot be hidden from the application altogether. Sometimes the lack of quality or the quality variations are problematic for the application.

There exist, as best practices and research results, a range of positioning quality tactics[1] for improving specific qualities. Often, application developers can exploit knowledge of the application domain or the application logic to achieve a specific instantiation of a tactic.

An overview of these tactics has not been available. This means that, first, the approaches are not well known. Second, many of them have not been described as a generally applicable tactic. Third, often only the quality that is improved is considered in the evaluation of the tactic; interrelationship between the qualities it affects is often not investigated.

When choosing a tactic for improving a quality the consequences for other qualities must be considered. Having knowledge of proven tactics and of how tactics affect relevant qualities both positively and negatively, application developers can improve their choices of tactics and, consequently, the quality of their positioning applications.

1.1 Contributions

The contributions in this paper are mainly that of a survey: giving an overview of existing approaches to positioning quality improvements. We include approaches that lie at a middleware or application level. Therefore, approaches

[1] The use of the term tactic is analogue to its use in software architecture, where a tactic is a fundamental design decision and a proven means to achieve a quality [7]

that are specific for a certain positioning technology or a certain sensor are not included. More specifically we:

- Identify and define the qualities that have a specific relevance for positioning.
- Identify and describe tactics for improving positioning qualities.
- Provide a taxonomy for the tactics.
- Present an overview of how each tactic influences the whole range of positioning qualities.

1.2 Method and Considerations

When the developer has defined the application, a number of positioning related quality requirements – often originating from business goals – will be known, others will become apparent in the course of the development process. Tactics are the (well) known tools to achieve these qualities.

In this positioning quality overview and positioning tactics taxonomy we have collected tactics from state of the art positioning systems, positioning middleware, and research in positioning quality. We argue for the stated quality implications of the tactics with a mix of references to the literature, statements of observable effects, and analyses, and we supplement this with our own experiments where necessary.

Knowledge of quality implications of a tactic based on experiments or user experience will always stem from one or a number of concrete implementations of that tactic. Maybe some clever implementation of a tactic will not have the same drawback on other qualities. And, deficient implementation of a tactic will not achieve the stated quality improvements. Therefore, we do not claim that the use of a certain tactic in all cases will have the stated quality implications. Nonetheless, we argue that there is great value in getting an overview of the current knowledge of quality implications of positioning tactics.

2 Positioning Qualities

A number of software architecture qualities are relevant specifically for the positioning part of the software, we refer to these as *positioning qualities*. We give a definition of positioning qualities in the following. Every presented tactic in the following section is directed at improving one of these qualities.

We state the positioning specific version of the qualities below. Naming, classification and definition of the qualities follow the ISO 9126 standard [46]. However, many of the qualities presented in the standard have no specific version for positioning software; these are not included in the qualities we

investigate and are not included below. We introduce some qualities and subqualities where these are of special interest in positioning software. We make a note of additions to the standard below.

Functionality "The capability of the [positioning subsystem] to provide [positioning] which meet stated and implied needs when used under specified conditions" [46, p. 7][2].

- **Accuracy** The capability of the positioning subsystem to provide positions within the needed maximal distance to the true position.
- **Precision** The capability of the positioning subsystem to provide positions with the needed maximal distance between provided positions based on the same true position.
- **Security** "The capability of the [positioning subsystem] to protect [position data] so that unauthorized persons or systems cannot read or modify them and authorized persons and systems are not denied access to them." [46, p. 8]

Reliability "The capability of the [positioning subsystem] to maintain a specified level of performance when used under specified conditions." [46, p. 8]

- **Integrity** The capability of the positioning subsystem to take relevant measures to avoid incorrect results of functions because of poor quality of positions[3].
- **Availability** The capability of the positioning subsystem to perform required functionality at a given point in time. [4]

Efficiency "The capability of the [positioning subsystem] to provide an appropriate performance, relative to the amount of resources used, under stated conditions." [46, p. 10]

- **Time Behavior** "The capability of the [positioning subsystem] to provide appropriate response and processing times and throughput rates when performing [positioning], under stated conditions." [46, p. 10]. Of special concern for positioning is responsiveness, freshness and time variance[5].

[2]The subdivision of the functionality quality is not part of ISO 9126 [46]. However, for positioning there is a well defined distinction between accuracy and precision.

[3]This quality is not part of the ISO 9126 standard.

[4]In the ISO9126 standard availability is split into maturity that controls the frequency of failure and fault tolerance and recoverability that control the length of degraded operation or downtime following each failure. The availability tactics we present could be categorized as maturity tactics, if the inability of sensors to deliver positions at all times is considered a fault.

[5]The subdivision of time behavior is not part of ISO 9126 [46].

Responsiveness The capability of the positioning subsystem to provide appropriate response time for the first position it delivers, under stated conditions[6].

Freshness The capability of the positioning subsystem to provide appropriate processing time for delivering positions, under stated conditions.

Time Variation The capability of the positioning subsystem to provide appropriate time variation for delivering a stream of positions, under stated conditions.

Resource Utilization "The capability of the [positioning subsystem] to use appropriate amounts and types of resources when the software performs its function under stated conditions." [46, p. 10]. Of special concern is power and bandwidth efficiency[7].

Power Efficiency The capability of the positioning subsystem to use appropriate amounts of power when the subsystem perform positioning under stated conditions.

Bandwidth Efficiency The capability of the positioning subsystem to use appropriate amounts of bandwidth for communicating positions under stated conditions.

3 Positioning Tactics Taxonomy

The qualities of a positioning application depends on the design decisions of the positioning software. These design decision has been termed tactics [7]. According to Bass et al., a tactic is a design decision that influences the activities undertaken after a certain stimulus. It controls activities in order to achieve a specific quality. See Table 3.1 for an overview of all tactics and their primary influences on positioning qualities.

Qualities of the system as a whole also depends on and can be improved in the sensor hardware or low-level software for computing a position from received signals. Tactics for this level are not included in this taxonomy as we focus on what can be achieved in a location based system that receives positions from one or a number of sensors. A few examples of low-level accuracy tactics are using environment information, e.g., a floor map, to provide information on signal propagation [47]. or using an antenna array instead of one antenna in a GPS sensor [77]. Tactics included in the taxonomy are general at least for a category of positioning techniques.

Tactics can refine other tactics. At a fine level of granularity one cannot distinguish between a refinement of a tactic and an implementation of a tactic. Therefore, a list of tactics is inexhaustible. We present proven tactics at a

[6]This quality is for GPS positioning called time-to-first-fix

[7]This subdivision is not part of the ISO 9126 standard.

Tactic name	**Quality characteristics and sub-characteristics**									
	Functionality			Reliability		Efficiency				
						Time Behavior			Resource Utilization	
	Accuracy	Precision	Security	Integrity	Availability	Response time	Freshness	Time variance	Power	Bandwidth
Sensor Fusion	+	+					–		–	
Use Environment Information	+									
Constrain by Motion Model	+									
Statistical Temporal Filtering	±	+					±			
Use Terminal Based Positioning			+							
Anonymize Data										
- Frequently Change Pseudonym			+							
- Report k-anonymity Region	–		+				–	–		
Hide True Position										
- Append Position Dummies	–		+						–	–
- Degrade Data Accuracy	–		+		(+)					(+)
- Use Application Specific Queries			+		(+)	(+)			(+)	(+)
Manage Information Receivers										
- Prompt User			+			–	–	–		
- Use Policies			+							
Fault Detection				+					–	–
Error Estimation				+					–	–
Graceful Degradation	–			+	±					
Expand Coverage					+					
Combine Coverage Areas					+				–	
Use Fast Positioning Technique	±					+				
Constraint Data Age							+	–		
Extrapolate for Missing Values	±	±						+		
Sensor Selection	–							–	+	(+)
Duty Cycling										
–Static	–						–	–	+	(+)
–Dynamic Using Motion Information	–							–	+	(+)
–Dynamic Using Application Logic	–					–		–	+	(+)
–Dynamic Using Motion Prediction	–							–	+	(+)
Decrease Number of Messages										
–Cache Position Data	–							–	(+)	+
–Periodic Querying or Reporting	–							–	(+)	+
–Only Communicate Deviations						(+)	(+)		(+)	+
–Distance Based Reporting	–								(+)	+
–Movement Based Reporting	–								(+)	+
–Aggregate Data						–	–		(+)	+
–Local Caching and Processing				–					(+)	+
Decrease Size of Messages										
–Asymmetrical Compression									(+)	+
Schedule Data Communication							–	–	(+)	+
Use Application Specific Protocols									(+)	+

Table 3.1: Overview of tactics and their quality implications. Legend: "+": increase of the quality. "(+)": derived increase of the quality. "–": decrease of the quality. "±" : may have an effect on the quality.

level of generality, where an overview is still possible. Further refinement or elaboration may be found in the references.

An analysis of implications for the targeted and other qualities is provided for the tactics. This analysis is based on findings published in the literature and our own analyses and experiments. Experiments are carried out in the PerPos positioning middleware [63]. The basic structure of the PerPos middleware is a processing graph, where position processing components process input from sensors in order to arrive at a high-level position. The middleware exposes a meta-model of the positioning process and allows adaptation with custom made code called *Features*. Features can inject code before or after the processing that takes place in a specific position processing component. They can also access the concrete component. PerPos is suitable for tactic experiments because the middleware is open, and tactics can be inserted as Features and easily removed again.

3.1 Functionality Tactics

Accuracy Tactics

The goal of accuracy tactics is to deliver positions that lie within a maximal distance from the true position. Position data arrives from concrete position systems – possibly implemented in hardware. The positioning software processes the position data and delivers a position. Accuracy tactics control the accuracy of delivered position data.

Sensor Fusion Output from several sensors – preferably different kinds of sensors – for the same target is combined into one position. Statistical methods are used to combine the position data from different sensors, while taking into account an error model for the sensors [42, 88].

In [6] WiFi and Bluetooth is combined in a relative peer-based positioning system. For the distance between two devices they report an average error of 3.40 m for Bluetooth alone, 3.91 m for WiFi alone, and improve this to average error of 1.41 m for a combination of Bluetooth and WiFi [6, p. 14].

This tactic will also improve precision, as extreme readings from one sensor is drawn towards the position delivered from the other sensors. Without other measures, the tactic will result in lower power efficiency and lower availability, because more sensors have to be turned on and available at the same time.

Use Environment Information Output from sensors are corrected by using information of the environment encoded in a model, e.g., a road model for vehicular scenarios or a building model for indoor scenarios. Improbable positions are filtered out or annotated with data indicating

its low probability. For outdoor positioning, improbable positions include positions in areas covered by a building, or, for most applications, by water. Application specific knowledge is also usable, e.g., that cars do not drive on foot paths, or that an animal is restricted in its movements to certain areas in a stable. For indoor positioning, improbable positions include positions inside walls or where a route from the previous position would have to cross a wall without any openings. The environment information can be used in a processing algorithm [76], or be fed into a statistical method along with information from other sources, e.g., sensor error models [97]. This tactic is widely used in satellite navigation equipment for vehicles, where several position reading with a large distance from a road must occur before the software layer reports a position outside the road network.

No additional quality information.

Constrain by Motion Model Application specific knowledge of (maximal/minimal) speed, route taken, or other movement patterns are used to improve on position accuracy by filtering out or assigning a low probability to positions that does not fit the model. The information can be used in a processing algorithm [76] or be fed into a statistical method along with, e.g., sensor error models [97]. This tactic is widely used in smartphones where positions are not reported to jump back and forth, but to continue more or less in the same direction.

No additional quality information.

Precision Tactics

The goal of precision tactics is to deliver positions, where positions based on the same true position lie within a maximal distance to each other. Position data arrives from concrete position systems – possibly implemented in hardware. The positioning software processes the position data and delivers a position. Precision tactics control the distance between delivered position data.

Statistical Temporal Filtering Fluctuations in sensor output are leveled out by using information from a number of the most recently collected measurements. Averaging sensor readings are, especially, used for readings from sensors such as digital compasses and accelerometers. These sensors often output data with a high frequency and have a high degree of variation, i.e., low precision. A simple implementation averages over the last number of data entities. A more complex implementation uses weights for example based on the elapsed time period. Information on the movement patterns of the target can also be used. In general, this

tactic is useful for sensor readings where the average accuracy is high, but the precision is low.

No additional quality information.

Security Tactics

Our ISO 9126 based formulation of the security quality, is close to Langheinrich's definition of privacy in ubiquitous computing [64]. Indeed, positioning specific tactics under this heading are concerned with security as a non monotonic function, where both to little security/privacy (exposure) and to much security/privacy (isolation) may be problematic. The goal of security tactics is to control the level of access to location information produced in the system.

The tactics we present here are special for position data. Other tactics that apply generally to data such as encryption, or anonymous routing, are not included.

The tactics are aimed at two levels of the application. The positioning level, where a position is obtained, and a possible service call level, where the user invokes a location based service. Some applications or services will not work without knowing the identity of the user, in these cases privacy cannot be obtained. However, other services may work with pseudonyms (where the true identity of the user is hidden) and others again without any identification of the user [13].

Use Terminal Based Positioning Systems Improve privacy by avoiding central knowledge of the position and avoiding communication of the position. In positioning systems such as GPS, Cricket [85] and Intel's Privacy Observant Location System the device that is positioned computes the position itself based on signals emitted from the infrastructure. These systems are opposed to systems where, e.g., cell towers, or WiFi position systems calculates the position for the device.

The influence on other qualities will depend on which infrastructure based positioning technology is exchanged with which terminal based positioning system.

Anonymize Data The privacy of location data is improved by attaching a pseudonym to the data. However, anonymization is vulnerable to attacks. By analyzing anonymized traces it is possible to reveal important places in the trace, e.g., home, work, desk etc. [4, 40, 43, 48, 73]. Together with other information sources the identity of the owner of the trace data can be revealed, thus, necessitating schemes for preventing such attacks.

Frequently Change Pseudonym In [13] Beresford and Stajano advocate frequently changing the pseudonym, but also warns about

information that may allow an attacker to link all the pseudonyms of a user.

No additional quality information.

Report k-anonymity Region Instead of reporting an exact location and time period, the user reports a region and time period that contains at least $k-1$ other persons [39]. This approach may be extended by taking into consideration the history of the users, providing historical-k-anonymity [14], the size of the area by reporting the minimal area that still contains $k-1$ persons [74], or by ensuring the user is given a new pseudonym in a *mix-zone* where users are not tracked and the tracks of k different users would be mixed up [13].

When a larger region is reported instead of a more fine-grained position, this means that the accuracy of the position degrades. To achieve k-anonymity with a more fine-grained position, the time-period may be extended, this will have a negative effect on the freshness and time variation. The higher k in k-anonymity the lower accuracy or freshness. In [39] a minimal k of 5 and mean k of 10 was experimentally achieved for cars based on traffic counts, with an accuracy of 30 m in an area covering expressways up to 250 m for an area covering city blocks. Furthermore, they report that the "spatial resolution can be significantly improved through a several seconds reduction in temporal resolution" [39].

In the experimental setup of use of mix zones in an office environment reported in [13], the privacy is not considered satisfactory. If the location update frequency is 1 min the median cardinality of the anonymity set is only 1, rising to 10 for a location update frequency of 1h [13, p. 52]. Furthermore, the direction of tracks may be used by and attacker as additional knowledge, which will help the attacker de-anonymize the mixed tracks as a u turn is more improbable than going straight ahead. However, with larger mix zones, that contain many possible routes and more people – for example parts of a city center – the approach might still work.

Append Position Dummies The true position of the user is concealed by appending multiple false positions when reporting a position or using a location based service. In order to fool attackers it is important that the dummy movements are realistic, for example by taking probable movement speed and presence of other users into account [51].

The added generation and communication of dummy positions cause a decrease in Bandwidth Efficiency and therefore in Power Efficiency and Freshness. Furthermore, the accuracy of the reported positions is

also lowered depending on the distribution characteristics of dummy positions.

Degrade data accuracy To improve data privacy by degrading the knowledge an attacker has of the users positions, the accuracy of the positions can be degraded. For positions given in coordinates an error can be added to the position. For symbolic positions the granularity of the position can be coarsened, e.g., specifying a building instead of a room.

When positions are communicated continuously, as for tracking applications, the accuracy has to be degraded substantially to prevent attacks using pattern matching algorithms of the traces. Often the coarser location information is cheap to compute, thereby not affecting the time behavior negatively. Coarse-grained positions may be obtained with better availability and power efficiency.

Use Application Specific Queries Application specific queries may degrade the information in the queries. For example, weather information is only available for regions, therefore, the same service quality can be achieved by providing a region instead of a position when querying about the locale weather.

This tactic does not seem to have negative consequences for other qualities. Often the coarser location information is cheap to compute, thereby possibly improving the time behavior. Coarse-grained positions may be obtained with better availability and power efficiency.

Prompt User When positions or other location information is queried or used by an application the user can be asked for consent. An example of this tactic can be observed on the Apple iPhone, where the user is prompted for consent whenever location information is needed.

The query process will degrade time behavior. Furthermore, in some cases it can be difficult for the user to keep the overview of the consequences for privacy, when giving out location information.

Use Policies A policy that specifies which applications or which users may access location information, whether they are allowed to get location information once or track the user, and finally at which level of detail they may access the information.

No further quality information.

3.2 Reliability Tactics

Integrity Tactics

The goal of integrity tactics is to enable the application to take appropriate measures if the quality of the positions is too low. A position is produced and,

possibly, communicated to some other part of the system. Integrity tactics control the measure of certainty that can be attached to the quality of the position.

Fault Detection A ping/echo or heartbeat mechanism, where the time between pings or heartbeats is set according to the maximum or average speed of the device and the needed accuracy level will ensure that the user of a position of another device is notified of a fault in the mobile device. This tactic is used in combination with communication efficiency tactics, where it is the mobile device that initiates the sending of a position and it only sends it when some requirements are fulfilled.

Using a ping/echo mechanism or a heartbeat means that the Bandwidth Efficiency degrades as more messages are send. This in turn means that the Power Efficiency also degrades.

Use Error Estimates Some positioning systems produce an error estimate, as for example the HDOP of a GPS device. This error estimate can be forwarded to the higher levels of the application and taken into account in the functionality. In [88] each position report is augmented with a confidence measure encoding the error estimates provided by low-level sensors.

Augmenting data with error estimates can potentially have a negative effect on resource utilization as it can lead to more communication and possibly higher power usage.

Graceful Degradation Survey position quality, when some quality decrease below a certain threshold, take measures to improve it, for example by switching to another positioning system or by activating other positioning tactics. When the quality cannot be improved, close down the positioning instead of forwarding low quality positions. Degrading the performance of the positioning system has a potentially negative impact on accuracy, precision and availability.

Availability Tactics

Availability tactics control the ability of the system to deliver positions, when they are needed.

Expand Coverage The coverage of various positioning systems varies: some cover outdoor, some indoor, some worldwide, some inhabitatet areas, some only a specific room. For improved availability use the positioning system that best covers where the application is used.

Combine Coverage Areas The availability of a positioning application can be improved if, for example, a positioning system that covers outdoor

areas is combined with a positioning system that covers indoor areas. When combining positioning systems the power efficiency decreases.

3.3 Efficiency Tactics

Time Behavior Tactics

The goal of time behavior tactics is that positions are delivered within some time constraint. A request for a position arrives – either a one time position request or a timing event for a position stream request – then the system produces a position. Time behavior tactics control the time within which a position is delivered.

Responsiveness Tactics

Use Fast Positioning Technique To achieve better responsiveness, when the position request arrives deliver positions from the fastest positioning sensors available, even if these positions may be inadequate in other qualities.

This tactic can be observed in current smartphones where GPS based applications may answer with a faster but less accurate position based on GSM based positioning or WiFi based positioning, while the GPS gets the first location fix. The first position fix for a GPS receiver may be considerably slower than the following fixes[8]. The implication on other qualities of using this tactic depends on the quality profile of the alternative positioning technique.

Freshness Tactics

Constraint Data Age For tracking applications that record other types of information than position data, the tracked information may be aggregated into a state, and position information that is older than a certain time constraint is excluded from the state.

This tactic may cause the Time Variation quality to decrease because some readings may be excluded from the dataset.

The distributed sensor network middleware EnviroTrack [1], allows the application developer to define a freshness constraint, which is used to delimit which sensor readings is attached to a *state* of a tracked object. They do not evaluate how the freshness constraint affects positioning qualities of applications implemented on top of the middleware as this is not the focus of the project.

[8]For a cold start: the receiver has no knowledge of the satellite constellations, and this information (the almanac) must first be received; for a hot-start the receiver must *acquire* the signal from each satellite by searching for the right alignment of the transmitted and the local copy of the pseudo random noise (PRN) code.

Time Variation Tactics

Extrapolate For Missing Values For missing or late values a position is calculated based on historic data. Static information of, e.g., maximum or average speed or typical movement patterns, or dynamic information of, e.g., current speed and direction can be used as a part of the extrapolation.

This tactic can be observed in satellite navigation systems for cars, where the position of the car is propagated along the current road when traveling in urban canyons, tunnels etc. In [76] the tactic is used in order to produce position values to pair with data from other sensors when the position data was not available.

The accuracy of the position is supposedly degraded (since extrapolation is not preferred over sensor-based data). The precision is improved because consecutive positions are affected by the same imprecision factors.

Resource Utilization

The goal of resource utilization tactics is that positions are delivered with an appropriate use of ressources. Resources of special concern for positioning applications are power and bandwidth. A position request arrives, possibly from a remote part of the system, the device is positioned and possibly communicates its position remotely. Resource utilization tactics control the resources that are consumed for positioning the device and communicating the positions.

Power Efficiency Tactics

Sensor Selection The strategy of sensor selection saves power by switching between sensors with the goal to use the sensors which uses the least amount of power to provide positions that satisfy the service requirements. This strategy is relevant in cases where services have changing requirements to positioning accuracy and several sensors are available, e.g., WiFi, GSM or GPS, all of which have different properties for power consumption and positioning accuracy.

Typically, the most power efficient positioning feature is used to obtain a position, however, this is likely a positioning feature that lacks in other qualities. When the position fulfills certain criteria, other positioning features may be turned on in order to obtain a position of higher quality.

Deblauwe and Ruppel combines this tactic with *Duty Cycling* [32]. They use **GSM!** (**GSM!**) positioning to determine if the user is inside a geographical area. The device is required to send a position update to the server when it is no longer in the area. Only when the current **GSM!** cell is not entirely within the area, GPS is turned on to obtain a position

with higher accuracy. Evaluation results for this method indicate power savings of up to 80%.

Duty Cycling Use of sensors scanning for radio signals for example for GPS or WiFi positioning is expensive in terms of power. If sensors are turned off when not needed, this will lead to increased power efficiency. Sensors, such as GPS and WiFi, that are expensive to keep in an idle state, need to be turned off, while, e.g., Bluetooth sensors can be left in an idle state, as long as no scans are carried out [6]. Sensors may not be able to deliver a position instantaneously, but need to be turned on for some time before delivering a position. Moreover, they may continue to use energy for a while after having been turned off [53, for GPS]. The gain in power efficiency depends on the length of the time-interval the sensor is turned off, and, on certain sensor specific delays as well. As no positions are produced while sensors are turned off, fewer positions may be communicated, thus increasing bandwidth efficiency. However, in order to retain a high time variation quality the old position may be communicated. Therefore, the possible increase in bandwidth efficiency does not follow automatically. Evaluations of power efficiency tactics often compare power efficiency to accuracy, assuming that the old position express the position of subsequent movements. Instead, the tactics could be seen to decrease time variation quality. The tactics are often combined with *Extrapolate for Missing Values* or *Sensor Selection* to produce new positions that can be used in the accuracy measurements instead of the simplest solution: continuing to use the last position produced by the sensor that has been turned off. The decrease in accuracy depends either on motion patterns and the ability to predict them or the accuracy of the "low power positioning technique". In the following, four duty cycling tactics are presented.

Static The sensor is turned on and off according to a fixed schedule.

The power savings are easy to predict with this tactic, given that the cost of turning on the sensor, the period it needs to be turned on in order to get a (good) position, and the cost of powering down is known. The tactic will either result in a decrease in freshness, or an accuracy that depends on motion patterns.

Kjaergaard et al. [53] use this tactic to compare more refined tactics against. For applications for pedestrians they assume a maximum speed of $10m/sec$. For an accuracy tolerance of 25, 100 and 200 meters, this gives a time interval for needed positions of 2.5 sec, 10 sec, and 20 sec respectively. This gives power savings of 0.0% for a needed accuracy of 25 meters, 9.19% for $100m$, and 13.58% for $200m$.

Youssef et al evaluate the power savings of this tactic implemented on an HTC Dream, Android phone [99]. They combine this tactic with *Sensor Selection*: While the high power GPS is turned off, they use dead-reckogning based on the low power accelerometer and compass for positioning. In experiments driving respectively on a high-way and in a city, they measure the decrease in accuracy for varying lengths of the period the GPS is turned off. They find that the accuracy error increases linearly, while the decrease in power usage is exponential.

Dynamic Using Motion Information The time period the sensor is turned off is based on movement detected for the current sensor input or motion detected by additional sensors (sensor replacement). High-level motion means the sensor is not turned off or only turned off for a short while, while no or a low level of motion means the sensor is turned off for a longer time period.

The improvements in energy efficiency will depend on how a high or low level of motion is defined and the decrease in accuracy on how accurately the prediction or detection of motion is.

A metric that describes the amount of change in positions of devices participating in a multi-target application is described in [6]. The time period between scans used for positioning is adapted based on this metric.

EnTracked [53] uses an accelerometer to determine if the device is moving or not. When the device is stationary, the GPS, which is used for positioning, is turned off. When movement is detected by using the accelerometer, the speed of the device is obtained from the GPS. The current speed is used in a dynamic programming algorithm to schedule when to turn on the sensor while taking into account the needed accuracy and an error model of GPS positioning. In emulation experiments the approach is measured to give power savings of 43.36% with an accuracy limit of 25 meters, 48.73% with 100 meters, and 56.2% with 200 meters. The device is stationary for 67.1% of the time in the experiments. A real-time experimental evaluation of the system, where pedestrians were tracked walking in a residential area, gave power savings of 62.3% with an accuracy limit of 100 meters and 69.7% with 200 meters.

Dynamic Using Application Logic Application specific context information can be used to determine when the sensors are to be turned off.

Banerjee et al. [6] upload to the cloud information of whether any devices in the vicinity has its back-light turned on. As the application is a social application showing information of friends in the vicinity, they can hold off from positioning when nobody is inter-

acting with the application. Based on a study of use patterns they argue that 30% power may be conserved by using this scheme. In this version of the tactic responsiveness will decrease as positioning will have to be carried out when a user starts interacting with the application, instead of this information being collected at all times and ready to use.

Dynamic Using Motion Prediction Future motions of the device may be predicted by statistical methods. Environment information can be taken into account.

In CAPS [83] the tactic is combined with *Sensor Selection*. The current position is estimated by combining the cell-ID information freely available on smartphones with a position prediction based on the history of past GPS coordinates. When a switch to a new cell is observed, the history is used to choose a position on the line that constitutes a border between the relevant cells. The same technique can be used for the number of times a cell-ID has been observed along with an interpolation of the movement. The GPS is turned on if necessary to learn and build the history of the user's routes. Experiments on 4 routes showing different characteristics show that CAPS keeps the median positioning error below 75 meters and GPS usage ratio below 4%.

Bandwidth Efficiency Tactics

Decrease Number of Messages Use an efficient protocol for communicating positions, that requires as few messages as possible. Positioning applications are most commonly deployed, at least partly on mobile devices. Consequently, when position information is transferred between a mobile device and a server or another mobile device, a wireless channel has to be used. Here bandwidth is low and expensive. For an overview of communication protocols, with simulations varying parameters as number of position requests per second, ratio between average and maximum speed, in order to compare communication efficiency and positioning accuracy of various protocols, see [67].

Cache Position Data For querying protocols where the position is requested by another entity, the current position can be cached, thereby, possibly, avoiding to forward some position queries to the mobile device. The server or the position requesting device estimates the accuracy of the cached position and decides whether to query the device for a new position. A pessimistic protocol uses the age of the position and the maximum speed of the device to determine the accuracy of the cached position. An optimistic protocol instead uses the average speed.

The analysis by Leonhardi and Rothermel [67] shows that the bandwidth efficiency of using this protocol, depends on the demanded level of accuracy of the position and on the maximum or average speed of the device. If the level of accuracy demanded is low and/or the speed is low, many position updates can be avoided.

Periodic Querying or Reporting For querying protocols that stores positions of devices on a server, the server can query the mobile device periodically, and return the stored position between queries. Similarly, for a reporting protocol, where the position is reported by the mobile device, the position information can be send periodically, instead of every time the sensor system has determined a new position.

The analysis by Leonhardi and Rothermel [67] shows that the bandwidth efficiency depends on the time threshold. In order to achieve a certain lower bound on accuracy of the position, the time threshold has to be set to the time the device takes to move the needed accuracy minus the accuracy of the stored position. If the application only needs an average accuracy, the average speed can be used instead of the maximum.

Only Communicate Deviations For reporting protocols the mobile device and the user of the position can share a scheme for predicting movements. A new position is only send, when it deviates (with a certain level of accuracy) from the predicted position. Using this tactic may improve on both response time and freshness as the predicted positions can be calculated on the client.

Distance Based Reporting For a reporting protocol the mobile device sends a position update only when it has moved a certain distance since the last update.

The analysis by Leonhardi and Rothermel [67] shows that the bandwidth efficiency of this tactic depends on the movement patterns of the mobile device, the slower the device moves the more bandwidth efficiency. To achieve a specific minimum accuracy at the server or device using the position, the distance threshold is set to the minimum accuracy expressed as a distance minus the accuracy of the last sent position.

Asymmetrical Compression Compress Data asymmetrically using a cheap mechanism for compression on the mobile device, and a possibly more expensive on the server side.

No additional quality information.

4 Related Work

Krumm presents a survey of computational location privacy [59]. The security tactics we present are influenced by the categorization in this paper.

Kjærgaard presents an overview of approaches to power efficiency [55]. The approaches are subdivided into categories. The categories for power efficiency in our taxonomy has been inspired by this work. However, we have collected two of the categories in a more coarse-grained category to make visible common factors, and we include more tactics.

Leonhardi and Rothermel present an overview of position communication protocols most of our tactics for communication are inspired by these protocols. In addition to the information they provide we also report quality improvements from other projects.

As the overviews presented above focus on respectively privacy, power efficiency and bandwidth efficiency, a more thorough description of the approaches is provided. Our work on the other hand, is more comprehensive in terms of the number of qualities that are included. Furthermore, we make the link to the tactics of software architecture and consider the consequences for other qualities when applying the solutions.

5 Conclusion

This survey and taxonomy of positioning quality tactics concentrates on tactics that can be employed on the middleware or application level of a positioning application. Research studies show that many of these tactics do, indeed, increase the quality they are targeted to improve. The use of tactics in consumer products, prove the acceptance of the need for quality improvements in a broader audience.

The research and applied knowledge of positioning qualities is still young. The list of generally applicable tactics for improving positioning qualities will grow in the years to come. Moreover, as the field matures some of the tactics that are mostly used in research will become proven approaches in applied positioning. In order to inform and advance this application of research results we need more experiments to investigate how tactics increase some qualities and decrease others. Furthermore, we need to investigate how the application of several tactics influence each other. Nonetheless, we expect this survey and taxonomy of positioning quality tactics to support positioning application developers in improving the quality of their applications.

Bibliography

[1] Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on* (2004), IEEE, pp. 582–589.

[2] Anand, M., Nightingale, E. B., and Flinn, J. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services* (2004).

[3] Apple. Core Location Framework Reference, 2011.

[4] Ashbrook, D., and Starner, T. Using gps to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing 7*, 5 (2003), 275–286.

[5] Balz, M., Striewe, M., and Goedicke, M. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@ run. time at MODELS* (2008), pp. 6–15.

[6] Banerjee, N., Agarwal, S., Bahl, P., Chandra, R., Wolman, A., and Corner, M. Virtual compass: Relative positioning to sense mobile social interactions. In *Pervasive Computing*, vol. 6030 of *Lecture Notes in Computer Science*. Springer, 2010.

[7] Bass, L., Clements, P., and Kazman, R. *Software architecture in practice, 2nd Edition*. Addison-Wesley Professional, 2003.

[8] Becker, C., Handte, M., Schiele, G., and Rothermel, K. PCOM - A Component System for Pervasive Computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)* (2004), pp. 67–76.

[9] Bellavista, P., Corradi, A., and Giannelli, C. The PoSIM middleware for translucent and context-aware integrated management of heterogeneous positioning systems. *Computer Communications 31*, 6 (2008).

[10] Benford, S., Anastasi, R., Flintham, M., Drozd, A., Crabtree, A., Greenhalgh, C., Tandavanitj, N., and Adams, M. Coping with uncertainty in a location-based game. *IEEE Pervasive Computing 2*, 3 (July 2003), 34–41.

[11] Benford, S., Crabtree, A., Flintham, M., Drozd, A., Anastasi, R., Paxton, M., Tandavanitj, N., Adams, M., and Row-Farr, J. Can you see me now? *ACM Trans. Comput.-Hum. Interact 13*, 1 (2006), 100–133.

[12] Benford, S., Seager, W., Flintham, M., Anastasi, R., Rowland, D., Humble, J., Stanton, D., Bowers, J., Tandavanitj, N., Adams, M., Row-Farr, J., Oldroyd, A., and Sutton, J. The error of our ways: The experience of self-reported position in a location-based game. In *Proceedings of the 6th International Conference on Ubiquitous Computing* (2004), pp. 70–87.

[13] Beresford, A., and Stajano, F. Location privacy in pervasive computing. *Pervasive Computing, IEEE 2*, 1 (2003), 46–55.

[14] Bettini, C., Wang, X., and Jajodia, S. Protecting privacy against location-based personal identification. *Secure Data Management* (2005), 185–199.

[15] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., et al. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online 2*, 6 (2001), 1–40.

[16] Bloch, J. *JSR 175: A Metadata Facility for the Java Programming Language*. Sun Microsystems, Inc., 2002.

[17] Blunck, H., Godsk, T., Grønbæk, K., Kjærgaard, M. B., Jensen, J. L., Scharling, T., Schougaard, K. R., and Toftkjær, T. PerPos: a platform providing cloud services for pervasive positioning. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application - COM.Geo '10* (New York, New York, USA, June 2010), ACM Press, p. 1.

[18] Blunck, H., Kjærgaard, M. B., and Toftegaard, T. Sensing and Classifying Impairments of GPS Reception on Mobile Devices. In *To appear in Proceedings of the Ninth International Conference on Pervasive Computing (Pervasive 2011)* (2011), Springer.

[19] Bracha, G., and Ungar, D. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM SIGPLAN Notices* (2004), vol. 39, ACM, pp. 331–344.

[20] Brown, L., Karasyov, K. A., Levedev, V. P., Starikovskiy, A. Y., and Stanley, R. P. Linux Laptop Battery Life. In *Proc. of the Linux Symposium* (2006).

[21] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J. The fractal component model and its support in java. *Software: Practice and Experience 36*, 11-12 (2006).

[22] Cai, Y., and Xu, T. Design, analysis, and implementation of a large-scale real-time location-based information sharing system. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008).

[23] Capra, L., Emmerich, W., and Mascolo, C. Reflective middleware solutions for context-aware applications. In *REFLECTION 2001*, vol. 2192 of *Lecture Notes in Computer Science*. Springer, 2001.

[24] Capra, L., Emmerich, W., and Mascolo, C. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering 29*, 10 (2003), 929–945.

[25] Chalmers, M. *Seamful design: showing the seams in wearable computing.* IEE, 2003.

[26] Chalmers, M., and Galani, A. Seamful interweaving: heterogeneity in the theory and design of interactive systems. In *Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques* (2004), ACM, pp. 243–252.

[27] Chen, G., and Kotz, D. Solar: An open platform for context-aware mobile applications. In *Proceedings of the First International Conference on Pervasive Computing* (2002), pp. 41–47.

[28] Chen, Y., Chen, X. Y., Rao, F. Y., Yu, X. L., Li, Y., and Liu, D. Lore: an infrastructure to support location-aware services. *IBM J. Res. Dev. 48*, 5/6 (2004), 601–615.

[29] Cheng, S., Garlan, D., Schmerl, B., Sousa, J., Spitnagel, B., and Steenkiste, P. Using architectural style as a basis for system self-repair. In *Proceedings of the IFIP 17th World Computer Congress-TC2 Stream/3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance* (2002), pp. 45–59.

[30] Civilis, A., Jensen, C. S., and Pakalnis, S. Techniques for efficient road-network-based tracking of moving objects. *IEEE Trans. Knowl. Data Eng. 17*, 5 (2005), 698–712.

[31] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. A generic component model for building systems software. *ACM Trans. Comput. Syst. 26*, 1 (2008).

[32] Deblauwe, N., and Ruppel, P. Combining gps and gsm cell-id positioning for proactive location-based services. *Mobile and Ubiquitous Systems, Annual International Conference on 0* (2007), 1–7.

[33] Farrell, T., Cheng, R., and Rothermel, K. Energy-efficient monitoring of mobile objects with uncertainty-aware tolerances. *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International* (Sept. 2007), 129–140.

[34] Farrell, T., Lange, R., and Rothermel, K. Energy-efficient tracking of mobile objects with early distance-based reporting. In *MobiQuitous* (2007).

[35] Garlan, D., and Schmerl, B. *Using Architectural Models at Runtime: Research Challenges*. Springer, 2004, pp. 200–205.

[36] Gjerlufsen, T., Ingstrup, M., and Olsen, J. Mirrors of meaning: Supporting inspectable runtime models. *Computer 42*, 10 (2009), 61–68.

[37] Glasgow, I. M., Glasgow, M. C., Sussex, Y. R., and Sussex, H. S. Seamful ubiquity : Beyond seamless integration. *Ieee Pervasive Computing* (2002).

[38] Graumann, D., Hightower, J., Lara, W., and Borriello, G. Real-world implementation of the location stack: The universal location framework. In *Fifth IEEE Workshop on Mobile Computing Systems and Applications, 2003. Proceedings* (2003), IEEE Computer Society, pp. 122–128.

[39] Gruteser, M., and Grunwald, D. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (2003), ACM, pp. 31–42.

[40] Hariharan, R., and Toyama, K. Project lachesis: parsing and modeling location histories. *Geographic Information Science* (2004), 106–124.

[41] Hightower, J., and Borriello, G. Particle filters for location estimation in ubiquitous computing: A case study. In *Proceedings of the 6th International Conference of Ubiquitous Computing* (2004), pp. 88–106.

[42] Hightower, J., Brumitt, B., and Borriello, G. The location stack: A layered model for location in ubiquitous computing. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop*

on (Los Alamitos, CA, USA, 2002), vol. 0, IEEE Computer Society, pp. 22–28.

[43] Hightower, J., Consolvo, S., LaMarca, A., Smith, I., and Hughes, J. Learning and recognizing the places we go. *UbiComp 2005: Ubiquitous Computing* (2005), 159–176.

[44] Højteknologifonden. Annual Report 2008. Tech. rep., Højteknologifonden, 2008.

[45] Ingstrup, M., and Hansen, K. A declarative approach to architectural reflection. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on* (2005), pp. 149 –158.

[46] ISO/IEC. *Software engineering – Product quality – Part 1: Quality model*, 2001. ISO/IEC 9126-1:2001.

[47] Ji, Y., Biaz, S., Pandey, S., and Agrawal, P. ARIADNE: a dynamic indoor signal map construction and localization system. In *Proceedings of the 4th international conference on Mobile systems, applications and services, June* (2006), pp. 19–22.

[48] Kang, J., Welbourne, W., Stewart, B., and Borriello, G. Extracting places from traces of locations. *ACM SIGMOBILE Mobile Computing and Communications Review 9*, 3 (2005), 58–68.

[49] Kaplan, E., and Hegarty, C. *Understanding GPS: Principles and Applications*, second edition ed. Artech House Incorporated, 2005.

[50] Kiczales, G. Aspect-oriented programming. *ACM Computing Surveys (CSUR) 28*, 4es (1996), 154.

[51] Kido, H., Yanagisawa, Y., and Satoh, T. An anonymous communication technique using dummies for location-based services. In *Pervasive Services, 2005. ICPS'05. Proceedings. International Conference on* (2005), IEEE, pp. 88–97.

[52] Kjærgaard, M., Blunck, H., Godsk, T., Toftkjær, T., Christensen, D., and Grønbæk, K. Indoor Positioning Using GPS Revisited. In *Proceedings of the Eight International Conference on Pervasive Computing (Pervasive 2010)* (Berlin, Heidelberg, 2010), P. Floréen, A. Krüger, and M. Spasojevic, Eds., vol. 6030 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 38–56.

[53] Kjærgaard, M., Langdal, J., Godsk, T., and Toftkjær, T. Entracked: energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th international conference on Mobile systems, applications, and services* (2009), ACM, pp. 221–234.

[54] Kjærgaard, M. B. A Taxonomy for Radio Location Fingerprinting. In *Proceedings of the Third International Symposium on Location and Context Awareness* (2007).

[55] Kjærgaard, M. B. Minimizing the power consumption of location-based services on mobile phones. *Pervasive Computing, IEEE PP*, 99 (2010), 1.

[56] Kjærgaard, M. B., and Weckemann, K. PosQ: Unsupervised Fingerprinting and Visualization of GPS Positioning Quality. In *Proceedings of the Second International Conference on Mobile Computing, Applications, and Services (MobiCASE 2010)* (2010), Springer.

[57] Klues, K., Hackmann, G., Chipara, O., and Lu, C. A component-based architecture for power-efficient media access control in wireless sensor networks. *Proceedings of the 5th international conference on Embedded networked sensor systems - SenSys '07* (2007), 59.

[58] Kon, F., Costa, F., Blair, G., and Campbell, R. H. The case for reflective middleware. *Communications of the ACM 45*, 6 (2002), 33–38.

[59] Krumm, J. A survey of computational location privacy. *Personal and Ubiquitous Computing 13*, 6 (2009), 391–399.

[60] Küpper, A. *Location-Based Services: Fundamentals and Operation*. Wiley, October 2005.

[61] Küpper, A., and Treu, G. Efficient proximity and separation detection among mobile targets for supporting location-based community services. *Mobile Computing and Communications Review 10*, 3 (2006), 1–12.

[62] Kupper, A., Treu, G., and Linnhoff-Popien, C. Trax: a device-centric middleware framework for location-based services. *Communications Magazine, IEEE 44*, 9 (2006), 114–120.

[63] Langdal, J., Schougaard, K. R., Kjærgaard, M. B., and Toftkjær, T. Perpos: a translucent positioning middleware supporting adaptation of internal positioning processes. In *Middleware '10: Proceedings of the 11th ACM/IFIP/USENIX International Conference on Middleware* (2010).

[64] Langheinrich, M. Privacy in ubiquitous computing. In *Ubiquitous Computing Fundamentals*, J. Krumm, Ed. CRC Press, 2010.

[65] Lemelson, H., Kjærgaard, M. B., Hansen, R., and King, T. Error estimation for indoor 802.11 location fingerprinting. In *Proceedings of the 4th International Symposium on Location and Context Awareness* (2009), pp. 138–155.

[66] Leonhardi, A., Nicu, C., and Rothermel, K. A map-based dead-reckoning protocol for updating location information. In *Proceedings of 16th Int. Parallel and Distributed Processing Symposium* (2002).

[67] Leonhardi, A., and Rothermel, K. A comparison of protocols for updating location information. *Cluster Computing 4*, 4 (October 2001), 355–367.

[68] Liu, J., and Zhong, L. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008).

[69] Location Acquisition API: Using Location Acquisition API. http://www.forum.nokia.com, 2007.

[70] Loytana, K. *JSR 179: Location API for J2ME*. Nokia Corporation, 2006.

[71] Maes, P. Computational reflection. *The Knowledge Engineering Review 3*, 01 (July 2009), 1.

[72] March, S. T., and Smith, G. F. Design and natural science research on information technology. *Decision Support Systems 15*, 4 (Dec. 1995), 251–266.

[73] Marmasse, N., and Schmandt, C. Location-aware information delivery with commotion. In *Handheld and Ubiquitous Computing* (2000), Springer, pp. 361–370.

[74] Mascetti, S., and Bettini, C. A comparison of spatial generalization algorithms for lbs privacy preservation. In *Mobile Data Management, 2007 International Conference on* (2007), IEEE, pp. 258–262.

[75] Mills, D. *Computer Network Time Synchronization - the Network Time Protocol*. CRC Press, 2006.

[76] Milton, R., and Steed, A. Correcting gps readings from a tracked mobile sensor. In *Proceedings of the First International Workshop on Location- and Context-Awareness* (2005), pp. 83–94.

[77] Misra, P., and Enge, P. *Global Positioning System: Signals, Measurements, and Performance*. Ganga-Jamuna Press, 2006.

[78] Musolesi, M., Piraccini, M., Fodor, K., Corradi, A., and Campbell, A. Supporting energy-efficient uploading strategies for continuous sensing applications on mobile phones. In *Pervasive Computing*, vol. 6030 of *Lecture Notes in Computer Science*. Springer, 2010.

[79] Nokia. *S60 Platform: Effective Power and Resource Management, Version 3.1.* Nokia, 2007.

[80] Nokia - Energy Profiler. http://www.nokia.com, 2008.

[81] Oppermann, L., Broll, G., Capra, M., and Benford, S. Extending authoring tools for location-aware applications with an infrastructure visualization layer. In *In Proceedings of the 8th International Conference on Ubiquitous Computing* (2006), pp. 52–68.

[82] OSGi Alliance. Open Services Gateway Initiative. *Specification download* `http://www.osgi.org/Download/Release4V42` *(Online, cited feb. 18, 2010)* (2009).

[83] Paek, J., Kim, K.-H., Singh, J. P., and Govindan, R. Energy-efficient positioning for smartphones using cell-id sequence matching. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 293–306.

[84] Parnas, D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (1972), 1053–1058.

[85] Priyantha, N., Chakraborty, A., and Balakrishnan, H. The cricket location-support system. In *Proceedings of the 6th annual international conference on Mobile computing and networking* (2000), ACM, pp. 32–43.

[86] Python for S60. http://sourceforge.net/projects/pys60, 2008.

[87] Rajamani, V., Julien, C., Payton, J., and Roman, G.-C. PAQ: Persistent Adaptive Query Middleware for Dynamic Environments. In *Proceedings of the 10th International Middleware Conference* (2009), pp. 226–246.

[88] Ranganathan, A., Al-Muhtadi, J., Chetan, S., Campbell, R. H., and Mickunas, M. D. Middlewhere: A middleware for location awareness in ubiquitous computing applications. In *Proceedings of the ACM/IFIP/USENIX 5th International Middleware Conference* (2004).

[89] Reddy, S., Burke, J., Estrin, D., Hansen, M., and Srivastava, M. Determining transportation mode on mobile phones. In *Proceedings of The 12th IEEE Int. Symposium on Wearable Computers* (2008).

[90] Riva, O. Contory: A middleware for the provisioning of context information on smart phones. In *Proceedings of the 7th International Middleware Conference* (2006), pp. 219–239.

[91] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. Discovering architectures from running systems. *IEEE Transactions on Software Engineering 32*, 7 (2006), 454–466.

[92] Sivaharan, T., Blair, G., and Coulson, G. Green: A configurable and reconfigurable publish-subscribe middleware for pervasive computing. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, R. Meersman and Z. Tari, Eds., vol. 3760 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 732–749.

[93] Tilak, S., Kolar, V., Abu-ghazaleh, N. B., and don Kang, K. Dynamic localization protocols for mobile sensor networks. In *Proceedings of IWSEEASN* (2005).

[94] Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 94–104.

[95] Welbourne, E., Khoussainova, N., Letchner, J., Li, Y., Balazinska, M., Borriello, G., and Suciu, D. Cascadia: a system for specifying, detecting, and managing RFID events. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008), pp. 281–294.

[96] wen You, C., Huang, P., Chu, H.-H., Chen, Y.-C., Chiang, J.-R., and Lau, S.-Y. Impact of sensor-enhanced mobility prediction on the design of energy-efficient localization. *Ad Hoc Networks 6*, 8 (2008), 1221–1237.

[97] Woodman, O., and Harle, R. Rf-based initialisation for inertial pedestrian tracking. In *Pervasive Computing*, vol. 5538 of *Lecture Notes in Computer Science*. Springer, 2009.

[98] Yau, S. S., Karim, F., Wang, Y., Wang, B., and Gupta, S. K. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing 1* (2002), 33–40.

[99] Youssef, M., Yosef, M. A., and El-Derini, M. N. Gac: Energy-efficient hybrid gps-accelerometer-compass gsm localization. In *GLOBECOM* (2010), pp. 1–5.

[100] Zheng, Y., Liu, L., Wang, L., and Xie, X. Learning transportation mode from raw gps data for geographic applications on the web. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008* (2008), pp. 247–256.