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

Jakob Langdal[1], Kari R. Schougaard[2],
Mikkel B. Kjærgaard[2], and Thomas Toftkjær[3]

[1] Alexandra Institute Ltd., Denmark
jakob.langdal@alexandra.dk
[2] Department of Computer Science, Aarhus University, Denmark
{kari,mikkelbk}@cs.au.dk
[3] Systematic A/S, Denmark
ttr@systematic.com

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 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 [1], visualizing the positioning infrastructure [2], minimizing energy consumption of location-aware applications [3] or adding high-level reasoning based on machine learning [4]. Therefore, a positioning middleware that gives a structured cross-level access to the positioning mechanisms is needed.

I. Gupta and C. Mascolo (Eds.): Middleware 2010, LNCS 6452, pp. 232–251, 2010.

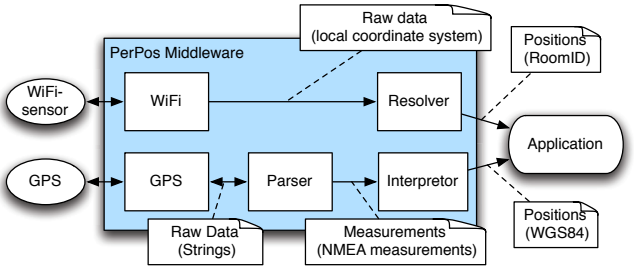


Fig. 1. Concrete positioning processes for the example Room Number Application

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 Global Positioning System (GPS) receiver and WiFi-signal strength measurements, and interfacing with a server containing an indoor WiFi positioning system [3] 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. 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 [1] 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 high-level functionality and Application Programming Interface (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 [2], manage sensors to minimizing energy consumption [3] 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 [4]. 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 [5], the Location Stack [6], and PoSIM [7]. 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. [8].

In connection with the second requirement positioning middleware, such as MiddleWhere [5] and the Location Stack [6], 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 [7] 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 [5] and PoSIM [7] 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 [9] 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 [10] for

generic context fusion, PAQ [11] which supports generic queries over temporal-spatial data and PCOM [12] 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 three 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 [5, 6]. 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 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 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. 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 Java Location API for J2ME (JSR-179) [13] 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.

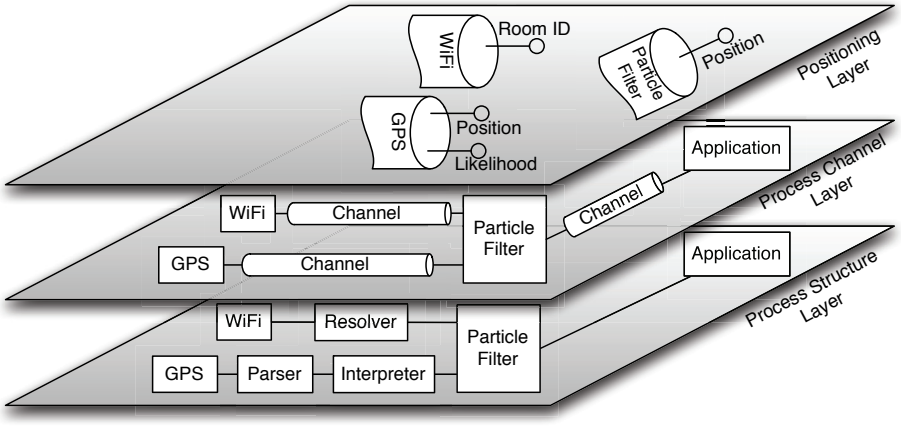


Fig. 2. The three levels of abstraction on the positioning process provided by the PerPos middleware

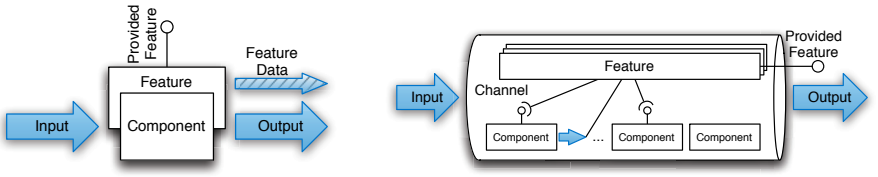
At the second layer the API provides access to an abstract structure of the underlying positioning process presented as a tree consisting of tree basic node types and the processing channels connecting them. The nodes are either: data sources, components that merges data sources, or the root node representing 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.

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, 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.



(a) Component feature (b) Channel feature

Fig. 3. The two kinds of extension features in the PerPos middleware

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 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 3(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. 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 3(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 4. 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

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

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

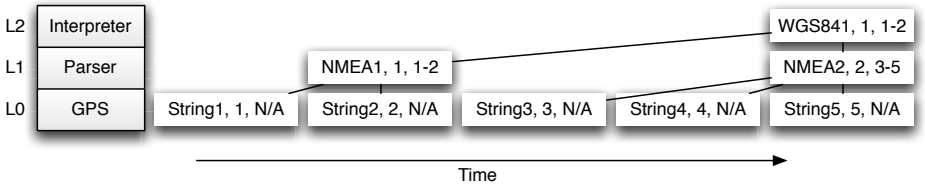


Fig. 4. An example data tree for the GPS Channel

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 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 Horizontal Dilution of Precision (HDOP) information. Because, the Interpretor is implemented so that it only returns a value when a valid position is produced several National Marine Electronics Association (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 [14]. To summarize, the combined effect of providing 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 [15]. 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 [8], 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 Universal Location Framework (ULF), an implementation of The Location Stack [8], 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 [7] 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.

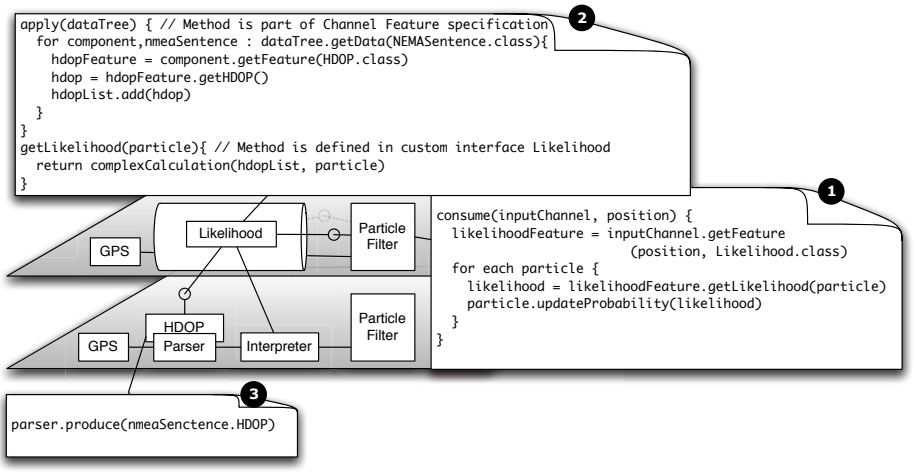


Fig. 5. Code snippets used to provide the particle filter with a likelihood estimate based on HDOP values. The code is shown as pseudo code for clarity, the actual code is Java.

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 5 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,

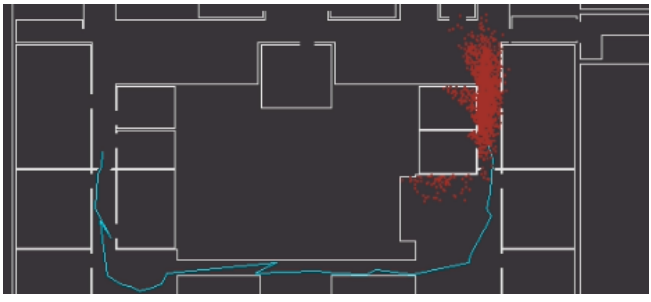


Fig. 6. Example run of a particle filter implemented using the PerPos middleware. Red dots indicate particle positions, the blue line indicates the evaluated trace, and white lines indicate walls.

taking the place of the sensors. Using this approach we were able to produce a refined trace as shown in Figure 6.

In the Location Stack [6] or MiddleWhere [16] 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 [7] 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 written. 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 [3]. 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 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 7. 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

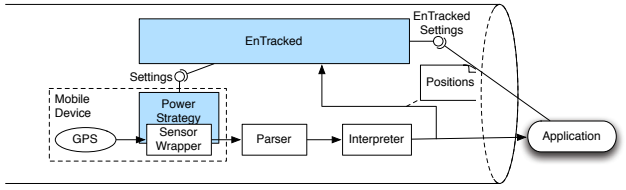


Fig. 7. Processing graph for the implementation of EnTracked using the extensible PerPos API

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 [15]. 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 [16] 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 [6] places obtained Measurements in a database and does not discuss sensor configuration. How to implement a power consumption scheme using PoSIM is discussed in [7]. 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 [6] and MiddleWhere [16] 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 [7] 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" [9, p. 37]. Positioning middleware designed for the traditional goal of transparency aims for the widely recognized principle of information hiding [17] 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 [13, 18]. 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 [19], 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 [20], and Benford et al. [21] 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." [21, 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." [20, 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. [8]. 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. [5] 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) [22] focuses on providing high-level location data together with sensor fusion and intelligent notification. Cascadia by Welbourne et al. [23] 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. [6] 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 [8]. 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. [7] 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 [24] 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 network infrastructure. To address this problem, reflective middleware has been proposed, as described by Kon et al. [9], 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. [25] 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. [12] provides adaptation for pervasive component-based systems by contracts that specify dependencies between components and resources. PAQ [11] 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.

References

1. Hightower, J., Borriello, G.: Particle filters for location estimation in ubiquitous computing: A case study. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 88–106. Springer, Heidelberg (2004)
2. Oppermann, L., Broll, G., Capra, M., Benford, S.: Extending authoring tools for location-aware applications with an infrastructure visualization layer. In: Dourish, P., Friday, A. (eds.) UbiComp 2006. LNCS, vol. 4206, pp. 52–68. Springer, Heidelberg (2006)
3. Kjærgaard, M.B., Langdal, J., Godsk, T., 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)
4. Zheng, Y., Liu, L., Wang, L., 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, pp. 247–256 (2008)
5. Ranganathan, A., Al-Muhtadi, J., Chetan, S., Campbell, R., Mickunas, M.D.: MiddleWhere: a Middleware for Location Awareness in Pervasive Computing Applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 397–416. Springer, Heidelberg (2004)
6. Hightower, J., Brumitt, B., Borriello, G.: The location stack: a layered model for location in ubiquitous computing. In: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (2002)
7. Bellavista, P., Corradi, A., Giannelli, C.: The PoSIM middleware for translucent and context-aware integrated management of heterogeneous positioning systems. Computer Communications 31(6), 1078–1090 (2008)
8. Graumann, D., Hightower, J., Lara, W., Borriello, G.: Real-world implementation of the location stack: The universal location framework. In: Proceedings of Fifth IEEE Workshop on Mobile Computing Systems and Applications, pp. 122–128. IEEE Computer Society, Los Alamitos (2003)

9. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Communications of the ACM 45(6), 33–38 (2002)
10. Chen, G., Kotz, D.: Solar: An open platform for context-aware mobile applications. In: Mattern, F., Naghshineh, M. (eds.) PERVASIVE 2002. LNCS, vol. 2414, pp. 41–47. Springer, Heidelberg (2002)
11. Rajamani, V., Julien, C., Payton, J., Roman, G.C.: PAQ: Persistent Adaptive Query Middleware for Dynamic Environments. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 226–246. Springer, Heidelberg (2009)
12. Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A Component System for Pervasive Computing. In: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), pp. 67–76 (2004)
13. Loytana, K.: JSR 179: Location API for J2ME. Nokia Corporation (2006)
14. Blunck, H., Godsk, T., Grønbæk, K., Kjærgaard, M.B., Jensen, J.L., Scharling, T., Schougaard, K.R., Toftkjær, T.: Perpos: A platform providing cloud services for pervasive positioning. In: COM.Geo 2010, 1st International Conference on Computing for Geospatial Research & Application (2010)
15. Alliance, O.: Open Services Gateway Initiative. Specification download (2009), `http://www.osgi.org/Download/Release4V42` (Online, cited February 18, 2010)
16. Ranganathan, A., Al-Muhtadi, J., Chetan, S., Campbell, R.H., Mickunas, M.D.: Middlewhere: A middleware for location awareness in ubiquitous computing applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 397–416. Springer, Heidelberg (2004)
17. Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
18. Kupper, A., Treu, G., Linnhoff-Popien, C.: Trax: a device-centric middleware framework for location-based services. IEEE Communications Magazine 44(9), 114–120 (2006)
19. Weiser, M.: The computer for the 21st century. Scientific American 265(3), 94–104 (1991)
20. Chalmers, M., 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, pp. 243–252. ACM, New York (2004)
21. Benford, S., Crabtree, A., Flintham, M., Drozd, A., Anastasi, R., Paxton, M., Tandavanitj, N., Adams, M., Row-Farr, J.: Can you see me now? ACM Trans. Comput. 13(1), 100–133 (2006)
22. Chen, Y., Chen, X.Y., Rao, F.Y., Yu, X.L., Li, Y., Liu, D.: Lore: an infrastructure to support location-aware services. IBM J. Res. Dev. 48(5/6), 601–615 (2004)
23. Welbourne, E., Khoussainova, N., Letchner, J., Li, Y., Balazinska, M., Borriello, G., 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, pp. 281–294 (2008)
24. Riva, O.: Contory: A middleware for the provisioning of context information on smart phones. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 219–239. Springer, Heidelberg (2006)
25. Capra, L., Emmerich, W., Mascolo, C.: Carisma: Context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering 29(10), 929–945 (2003)