Towards a Services Platform for Mobile Context-Aware Applications

Patrícia Dockhorn Costa[1], Luís Ferreira Pires[1], Marten van Sinderen[1] and José Gonçalves Pereira Filho[2]

[1]Centre for Telematics and Information Technology, University of Twente, PO Box 217, 7500 AE Enschede, the Netherlands

[2]Universidade Federal do Espírito Santo, Dept. de Informática, Av. Fernando Ferrari s/n, CEP 29060-900 Vitória (ES), Brazil

Abstract. Context-aware services platforms aim at supporting the handling of contextual information in order to provide better user-tailored services. This paper proposes a novel services platform architecture to support mobile context-aware applications, giving emphasis to the configurability of the platform's generic functionality. The paper introduces concepts and a language to cope with configurability aspects. The paper also reports on the implementation of this architecture in the WASP[a] platform, which is a Web services-based context-aware services platform that runs on top of 3G networks.

1 Introduction

Context-awareness has emerged as an important and desirable feature in distributed mobile applications. This feature deals with the ability of applications to utilize information about the user's environment (context) in order to dynamically select and execute relevant services that better match the user needs [2]. In a ubiquitous environment, with many services available at any time, context information is especially important to help determining which services are relevant for the user [4].

Building context-aware systems involves the consideration of several new challenges mainly related to the gathering/sensing, modelling, storing, distributing and monitoring of contextual information. These challenges motivate the need for proper architectural support.

There have been many initiatives towards architectural support for context-aware applications. In particular, considerable effort has been spent on the development of *infrastructure software* to support the development and/or operation of context-aware applications. *Infrastructure software* comprises code libraries or runtime environments that provide high-level abstractions to shield application developers from the interactions with (lower-level) data repositories, hardware devices and

[a] The WASP (Web Architectures for Service platforms) project is funded by the *Freeband Knowledge Impulse* joint initiative of the Dutch government, knowledge institutes and industry.

software constructs [4]. Among these infrastructures, we have seen the emergence of *context-aware services platforms*, which aim at providing support for application designers to conceive their applications by using services, mechanisms and interfaces that shield them from the complexity of handling contextual information [3].

Currently available platforms, however, offer a limited level of configurability. Ideally, a platform for context-aware applications should facilitate the creation and the dynamic deployment of a large range of context-aware applications that are unanticipated during the design of the platform. In this paper, we define a services platform architecture for context-aware applications, giving emphasis to the configurability and extensibility of the platform's generic functionality. We present a descriptive language, coined WSL (WASP Subscription Language), which enables applications to dynamically configure the platform to fulfil their needs.

The remainder of this paper is structured as follows: Section 2 gives an overview of our proposed context-aware services platform, Section 3 defines the concepts applied in the platform development, Section 4 describes WSL, Section 5 introduces the platform components, Section 6 presents the prototype and an application example, and Section 7 gives final remarks and identifies topics for further study.

2 Overview of the Services Platform

The services platform forms the system environment for context-aware mobile applications. It supports the scenario in which context information is gathered from *Context Providers* (sensors or third-party information providers) and services are implemented by third-party service providers.

The services platform aims at delivering the most adequate services based on both application requirements and contextual facts (see Fig. 1). Applications describe their requirements by defining the desired services and the contextual conditions in which the services should be provided. The platform should autonomously react to *reaction rules*, which are defined in terms of conditions to be checked against contextual facts.

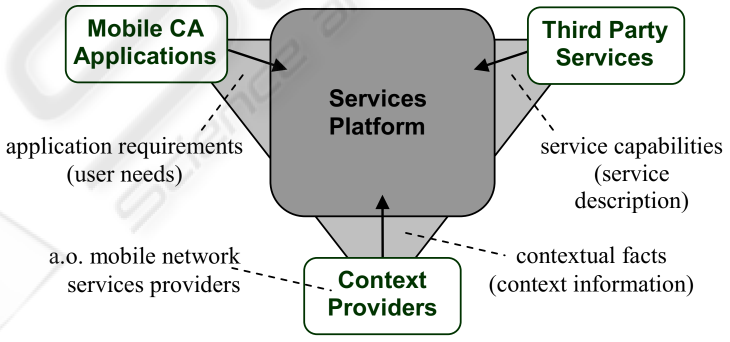

Mobile CA Applications
Third Party Services
Services Platform
application requirements (user needs)
service capabilities (service description)
a.o. mobile network services providers
contextual facts (context information)
Context Providers

Fig. 1. Overview of the services platform

In [3] we have identified the essential requirements to be satisfied by the services platform, which include:

- *Context handling*: the platform should provide efficient mechanisms to gather, store, distribute and monitor contextual information;
- *Reactive behaviour*: the platform should allow the specification of reaction rules. Moreover, it should be able to react according to the specified rules;
- *Configurability*: the platform should be able to support context-aware applications that are unanticipated during platform design. For that, mechanisms of configurability and the use of generic components need to be considered.

We have addressed all these requirements when designing the platform architecture, but most of our efforts have been spent on developing an architecture with a high level of configurability. The proposed solution includes the definition of a subscription language, which allows applications to dynamically expose their needs to the platform.

Fig. 2 depicts the proposed services platform architecture, which contains three main components: *Monitor*, *Registry* and *Context Interpreter*.

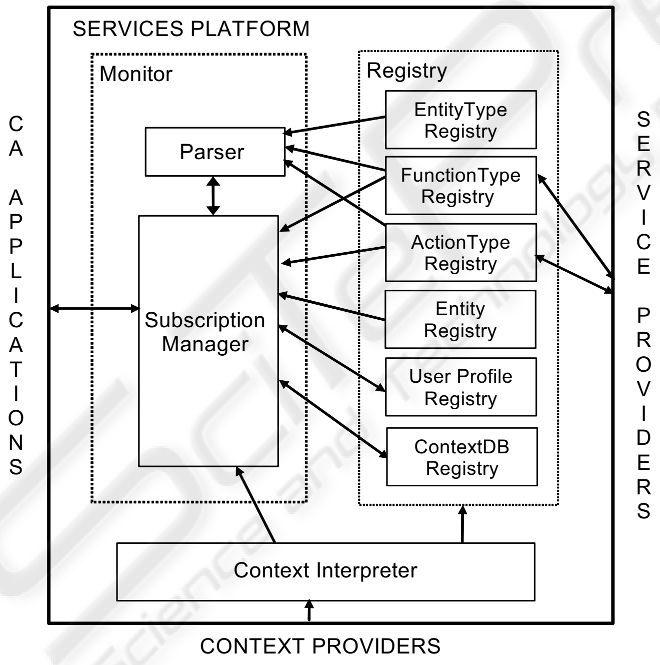


Fig. 2. Services platform architecture

The *Context Interpreter* gathers contextual information from *Context Providers* (sensors or third-party providers), manipulates contextual information and makes it uniformly available to the rest of the platform. The *Registry* maintains information necessary to support the interpretation of application requirements and the execution of services. The *Monitor* is the core of the platform, since it is responsible for receiving and interpreting application requests and making them active within the platform.

In our approach, application↔platform interactions are dynamically configured through the definition of *application subscriptions*. In a subscription, an application is capable of dynamically exposing its requirements to the platform, which composes at runtime new tailored services from the set of available services.

The services platform architecture proposed in this paper has been designed and implemented in the WASP (Web Architectures for Services platforms) project. The prototype implementation is referred to as the *WASP Platform*. The WASP project is concerned with the definition and validation of a services platform to facilitate the development and deployment of mobile context-aware applications, called *WASP applications*, on top of 3G networks [7], using Web Services technologies [12]. The 3G networks provide the telecommunication infrastructure for mobile terminals. In addition, a 3G network can play the role of a Context Provider, since it is able to provide the current location of its users. 3G network functions are accessed using the Parlay X [8], a Web services interface. Web Services technologies are used to support application↔platform and platform↔service provider interactions.

3 Concepts

In order to effectively and consistently manipulate the contextual knowledge of the platform, we need to organize, represent, and describe it in a model. For this purpose we have introduced a context model. Whenever such a model is available, it can be used as basis for common understanding between platform and applications developers, and service providers.

3.1 The Platform Context Model

The services platform manipulates *data entities*, which represent objects of the real world (users, restaurants, museums, roads, vehicles, etc.). *Attributes* (age, area, address, etc.) and *Context* (time, location, activity, etc.) are associated with data entities.

The UML class diagram depicted in Fig. 3 shows a possible configuration of the context model proposed for the platform. This diagram shows only an example configuration, with entities *restaurant* and *user*, and their relationship with the *location* context. The model can be extended if required by the applications, by (dynamically) adding new entity types, such as, e.g., *museum* and *supermarket* and context types, such as, e.g., *time* and *activity*.

The model presents three instantiation levels, namely a *metamodel*, a *model* and an *object* level. The *metamodel* level is embedded in the platform and is defined during the platform design-time, but remains unchanged during runtime. The *model* and the *object* levels can be dynamically changed during runtime. They represent instances of the *metamodel* and the *model* levels, respectively. The context model is discussed in more detail in [2].

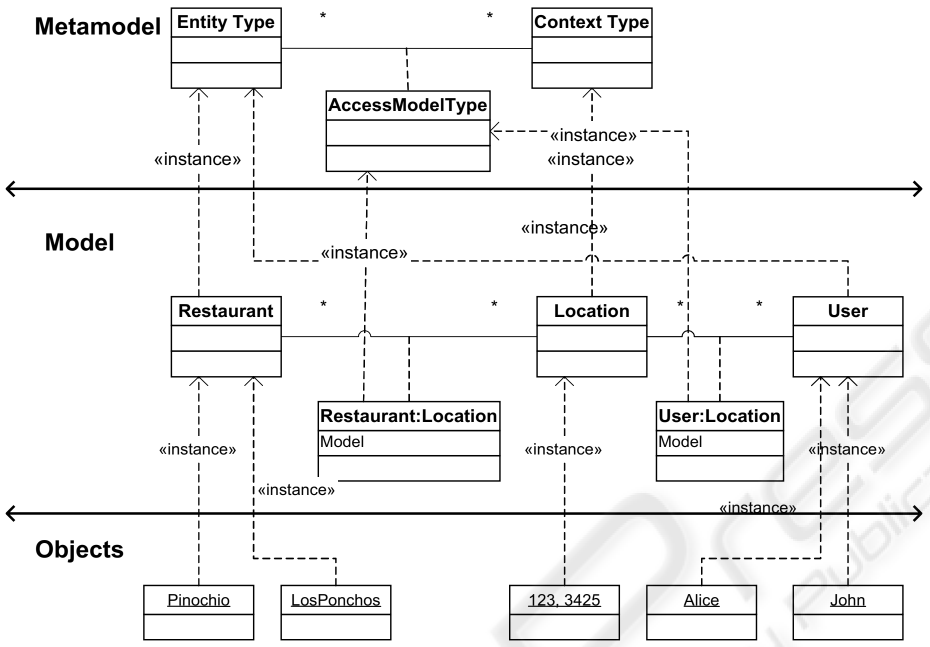


Fig. 3. Context Model

Fig. 3 shows a model configuration in which entities types *Restaurant* and *User* (*model*) are instances of *Entity Type* (*metamodel*). Moreover, the context *Location* (*model*) is instance of *Context Type* (*metamodel*). Similarly, *Pinochio* and *LosPonchos* (*object*) are instances of entity *Restaurant* (*model*) and *Alice a*nd *John* (*object*) are instances of entity *User* (*model*).

(Hierarchical) relations between *entity types* can also be defined, varying from simple categorizations of entity types to complex *ontologies* [9]. A common representation of this knowledge is essential for the interoperability of the platform and its environment.

3.2 Services

We have defined two types of service units to be used as building blocks for services in the platform: (i) service units that reason about contextual circumstances, which we call *functions* and (ii) service units that are triggered and react to some determined contextual fact, which we call *actions*.

A function is a service unit that performs a computation with no side-effects, i.e., the computation does not change the current status of the platform. An example of function is the `isInside` operation, with parameters of type *container* and *person*. This functions returns *true* if the given *person* instance is inside the given *container* instance, and *false* otherwise.

An action is a service unit that performs a computation with side-effects for one or more parties involved in the system. An example of *action* is the `sendAmbulance` operation, which sends an ambulance to a physical place. The invocation of functions and actions follows the request-response pattern.

Functions and actions are composed together by the platform to form new services. The service compositions are described by the applications, by means of applications subscriptions written in WSL.

4 The WASP Subscription Language (WSL)

WSL is a descriptive language that we have developed in order to be able to specify application subscriptions. Initially, we identified two essential requirements with respect to the elements of this language: (i) a way to specify the reactions of the platform to stimuli from the environment and (ii) a way to correlate events that eventually trigger the specified reactions.

Subscriptions can be either parameterized or not. Parameterization is necessary when the rule (subscription) applies to a collection of entities, since it would be cumbersome to write a subscription for each target entity. Sometimes it is also necessary to select entities of a collection for which a certain condition holds.

We illustrate the WSL clauses in the sequel by means of examples that use the functions `count` and `IsInside`, and the action `sendSms.` `Count` returns the number of elements in a collection; `IsInside` returns *true* if an entity is inside a given *container*, and *false* otherwise; and `sendSms` sends a message to a set of users. The WSL (abstract) syntax has been completely specified in [2], using both EBNF and UML class diagrams.

4.1 The SELECT clause

The `SELECT` clause returns a collection of entities for which a given filtering expression holds true. This clause allows the selection of a subset of a collection that respects a filtering condition defined as a logical combination of contextual facts and/or attributes. Its abstract syntax is as follows:

```
SELECT (<collection-of-entities>; <var>;
        <filtering-expression-involving-var>)
```

A select clause that returns a collection of users currently located in the city of Enschede can be defined as:

```
SELECT (entity.user.*; u; u.location.city ==
"Enschede")
```

where `entity.user.*` represents the collection of all users in the system, `u` is the variable to designate the elements of the collection and `u.location.city == "Enschede"` is the logical expression that filters the designated collection of users by selecting the ones that are in Enschede.

4.2 The ACTION-GUARD clause

The ACTION-GUARD clause defines that one or more actions should be triggered as a consequence of a correlation of events. This clause allows one to represent actions that are performed by the platform as a reaction to stimuli defined in a logical expression. Its abstract syntax is as follows:

```
ACTION <action> [GUARD <correlation-of-events>]
```

An action to send an SMS message to a user if he is inside a movie theatre can be defined as:

```
ACTION SendSms (entity.user.John,
 "Hey John, Cola and movie, a perfect combination!");
GUARD
(count (SELECT (entity.cinema.*; c;
    ( isInside(entity.user.John, c) AND
      (c.location.city == "Enschede") ) )
        ) > 0 )
```

The statement above defines that a message should be sent to John if John is inside a movie theatre and this movie theatre is located in Enschede. The SELECT clause is used to select a collection of movie theatres in Enschede where user John currently is. The selected collection has 0 or 1 element (either the user is in one or in zero movie theatres). If the user is inside a movie theatre, an advertisement is sent; otherwise the action is not triggered.

An ACTION-GUARD clause can be used to define an application subscription, since it tells the platform which correlation of events has to be monitored, and which actions should be triggered.

4.3 The SCOPE clause

The SCOPE clause defines a collection of target entities for which the subscription defined in a nested ACTION-GUARD clause should be applied. The SCOPE clause has been introduced to support subscription parameterization. Its abstract syntax is as follows:

```
SCOPE (<collection-of-entities>; var){
ACTION <action-involving-var>
   [GUARD <correlation-of-events>]}
```

The SCOPE clause can be used to define the scenario "Send an advertisement to every user inside the movie theatres in Enschede" in the following way:

```
SCOPE ((SELECT (entity.user.*; u2; u2.location.city ==
"Enschede")); u)
{
  ACTION SendSms (u,
         "Cola and movie, a perfect combination!");
    GUARD
    ( count (SELECT (entity.cinema.*; c;
        (isInside(u,c) AND
        (c.location.city == "Enschede") ) )
             ) >0 )
}
```

The SELECT clause returns a collection of users located in Enschede in that given moment. The nested ACTION-GUARD clause applies to each of the selected users, which are named u in the SCOPE clause.

5 Platform Components

The three main components of the services platform architecture are the *Context Interpreter*, the *Registry* and the *Monitor*.

5.1 Context Interpreter

The *Context Interpreter* gathers contextual information from *Context Providers* (sensors or third-party providers) and makes it uniformly available to the rest of the platform. The interpreter may also perform:

– *Context aggregation*: the context interpreter provides contextual information about a certain entity by gathering and aggregating context from a set of context providers, if necessary;
– *Context inference*: the context interpreter infers context from other contexts. Inference rules may be used to perform this activity.

5.2 Registry

The *Registry* component consists of a collection of registries that contain and maintain the information represented in the *data entity model* (see Fig. 3). These registries provide essential information to support the deployment of applications in the platform. We have defined six registries:

– *Entity Type Registry*: stores entity types, and their correspondent attributes and context types. Examples of entity types are person, cinema, restaurant and museum; examples of attributes are age and address; examples of context types are location, velocity and activity. Some types of contexts apply only to specific entity types. For example, velocity may be applied to a person but not to a movie theatre. The Entity Type registry manages all possible combinations of context types and entity types, being the actual representation of the *Model* level of the context model;
– *Entity Registry*: stores instances of entity types. For example, it might store the instances *Alice* and *John* of the entity type *person,* and *Pizza Hut* of entity type *restaurant*;
– *Function* and *Action Type Registries*: store, match and retrieve functions and actions profiles, respectively. Actions and functions are published and implemented by third-parties service providers;

- *User Profile Registry*: manages user profiles. Significant facts can be collected directly from the user profiles, like, e.g., as gender, date of birth, name, preferences, etc. These facts can be considered as contextual information, in the sense that they describe the environment in which a user operates;
- *ContextDB Registry*: preserves contextual information over time (history). Keeping context history is essential to allow context inference based on past occurrences.

5.3 Monitor

The core of the platform architecture is the *Monitor* module, which is responsible for interpreting and managing the application subscriptions. In order to perform its operations, the *Monitor* makes use of the data available in the Registry and the contextual information provided by the Context Interpreter.

Application↔platform interactions are dynamically configured through the definition of application subscriptions written in WSL, as discussed in Section 4. Using WSL one can refer to entities (their context and attributes) and the combination of actions and functions in order to express the desired service in terms of a subscription. Furthermore, new entities, context, attributes, actions and functions can be added to the platform on demand. This implies that different forms of possibly complex services can be composed and deployed at runtime by the platform, making it highly configurable.

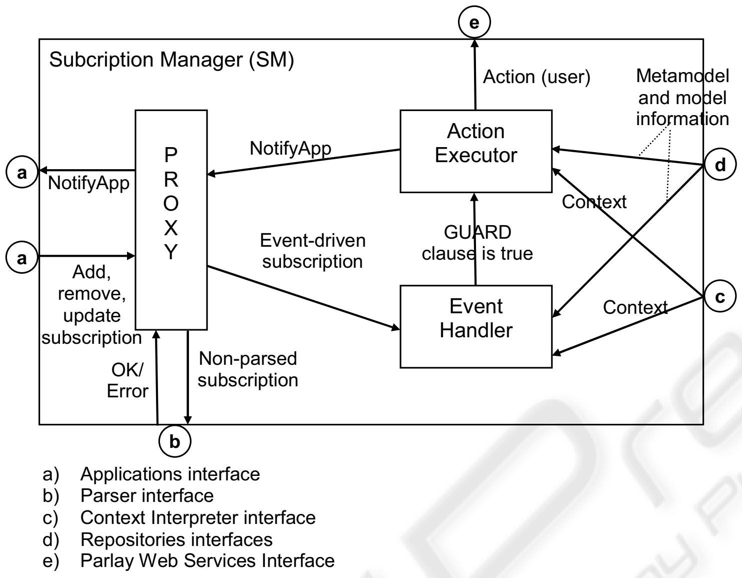
The Monitor contains a Parser and a Subscription Manager component.

Parser. The Parser component is responsible for verifying whether the subscriptions are syntactically and semantically correct, having as reference the syntax of WSL [2]. The parsing result is a tree containing WSL primitive elements. There are two levels of semantic checking:

1. A model checking level using the platform entity metamodel (Fig. 3). At this level, the Parser verifies the existence of the entity types and whether the combinations of context type and entity type are meaningful. Furthermore, the Parser needs to verify the semantics of functions and actions;
2. An instance checking level using the instance repositories to check the existence of the entities (final instances).

Subscription Manager (SM). The SM provides an interface for applications to add, remove or update subscriptions. Furthermore, applications should provide a notification interface for possible callbacks from the platform. Fig. 4 depicts the internal structure of the Subscription Manager.

Applications subscribe to the platform by dynamically adding application subscriptions. Moreover, existing application subscriptions can be updated or removed. An added or updated subscription is parsed and verified for syntax and semantic integrity. If no errors are found, the subscription is forwarded to the *event handler*, which constantly checks the condition in the `GUARD` clause. The frequency in which checks are performed depends on how often contextual information is provided by the *Context Interpreter*. In general, the GUARD clause is checked whenever notifications of context changes are received from the *Context Interpreter*.



Subcription Manager (SM)
P R O X Y
Action Executor
Event Handler
NotifyApp
NotifyApp
Add, remove, update subscription
OK/ Error
Non-parsed subscription
Event-driven subscription
GUARD clause is true
Action (user)
Metamodel and model information
Context
Context
a
a
b
c
d
e
a) Applications interface
b) Parser interface
c) Context Interpreter interface
d) Repositories interfaces
e) Parlay Web Services Interface

Fig. 4. Subscription manager

When a condition in the GUARD clause becomes true, the corresponding action is triggered, which could be a simple callback to the application or a more complex task on the users' device. Fig. 5 shows the interactions that take place in the *Subscription Manager* in order to handle a subscription.

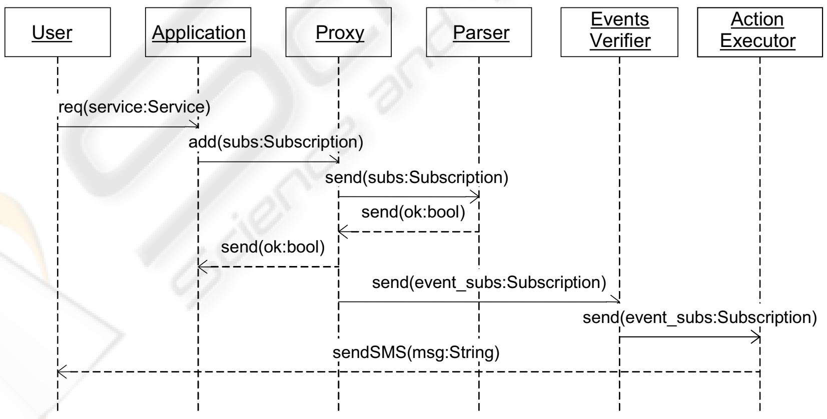


Fig. 5. Interactions necessary for handling a subscription

In Fig. 5 the user requests a service to the application, which consists of sending him a message in case one of his colleagues is close to him. The application describes the service composition by means of a subscription and adds it to the platform. The *Proxy* gets the subscription and asks the *Parser* to check it. Once the subscription is checked, it is forwarded to the *EventVerifier*. The *EventVerifier* checks whether there is a colleague of the user close by him. In case this condition becomes true, the *ActionExecutor* triggers the `sendSMS` action.

6 Validation Scenario

The main goal of the services platform prototype built in the WASP project has been to demonstrate and validate the concepts and the main architectural elements. In this prototype we have focused on the application↔platform interactions.

6.1 Prototype

We have prototyped the Monitor (*Subscription Manager* and the *Parser*), some of the *Registries* and a simplified *Context Interpreter*. The application↔platform and platform↔context provider interactions have been implemented. The prototype does not include the implementation of the platform↔service providers interactions. Essential actions and functions have been hard-coded in the platform with which our scenarios have been performed (mainly location-aware scenarios). Future implementations of the prototype will consider these interactions, by implementing the action type registry and function type registry. Service discovery can also be implemented, e.g., by using UDDI [13].

We have defined an XML Schema that represents the WSL syntax [2], so that application subscriptions can be written as XML documents and validated using this WSL XML Schema. The WSL parser reads application subscriptions in XML format and maps them into Java classes, which are automatically compiled and executed during runtime.

We have used Web Services technologies and Java language for implementing the prototype. The WASP platform interface is offered as a web service end-point, allowing the operations to be remotely called by the platform applications. Furthermore, we also have implemented the users' terminals as a web service end-point to allow callbacks from the platform. We have used JAX-RPC [10] to automatically generate the WSDL file from Java interfaces and the W3C's Document Object Model (DOM) [11] to develop our XML-based WSL parser.

6.2 Example: Taxicab Application

We illustrate the use of the platform with a context-aware taxicab application. Users request a taxicab directly at the taxicab company's web site with no need to inform their current location. Furthermore, users get a message when the requested taxicab approaches their location. Fig. 6 depicts the sequence diagram of this scenario.

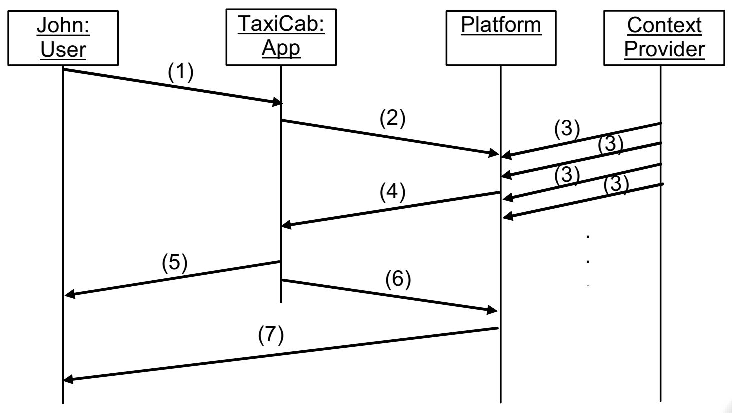


Fig. 6. Sequence diagram for the taxicab scenario

Table 1 gives the messages that correspond to the numbering in Fig. 6.

Table 1. Exchanged messages in the taxicab scenario

Message Number	Message Contents
(1)	"I need a cab"
(2)	`ACTION NotifyApp( bookTaxicab (SELECT (entity.taxicab.*; tc; (CloseBy (tc, entity.user.John, 3000)) AND (tc.company = "ABC"))));`
(3)	John's location and taxicabs' locations
(4)	The booked taxicab identification and the approximate taxicab arrival time
(5)	"The taxicab will arrive in approximately 5 minutes"
(6)	`ACTION SendSMS(entity.user.John, "Your taxicab has arrived."); GUARD CloseBy(entity.user.John, entity.taxicab.cab1234, 50)`
(7)	SMS with the text "your taxi has arrived"

In Table 1, message (1) represents the user request for a taxicab. Message (2) is the application subscription in WSL for a service composed using actions `NotifyApp` and `bookTaxicab`, and function `closeby` as building blocks. Action `NotifyApp` causes a callback from the platform to the application, in which the results of action `bookTaxicab` are informed. These results are the identification of the taxicab and an approximate arrival time. Besides returning this information, `bookTaxicab` is a third-party service that books a taxicab from a given collection. We assume in this scenario that the context provider is able of providing the current locations of taxicabs and users (message (3)).

An extra service offered by this context-aware taxicab application is that the user gets a message when the taxicab approaches the current location of the user. Message (6) is the application subscription used to define this extra service.

7 Final Remarks

Most of the current approaches for building context-aware services platforms ignore the dynamic (runtime) deployment of mobile context-aware applications on top of these platforms. The architecture presented in [6] focuses on the handling of different types of adaptation mechanisms for device-oriented services, and addresses resource availability and adaptability. A language for coordination of events intended for the enterprise domain has been reported in [5]. An infrastructure based on a distributed database and a dynamic decentralized resource discovery service for wearable computing has been described in [1].

Our approach supports the configuration of applications↔platform interactions at runtime. Furthermore, our platform can be extended with additional functions, actions and data entities, making it appropriate for a large range of (unanticipated) context-aware applications.

In order to allow dynamic configuration of interactions we have introduced WSL, which is a descriptive language developed especially for this purpose. We have decided to develop our own relatively simple language in order to be able to start experimenting with our configuration approach right away. By using XML-based techniques, the development of language manipulation support (parsers and interpreters) has been facilitated. In future we will investigate how WSL can be replaced by standard languages.

We have used Web Services technology to support the interactions of the platform with its environment. As a consequence, third-party applications can access the services offered by the platform through widely-used Internet protocols. In addition, Web Services technology facilitates the extension of the platform by third-party service providers, which can provide additional functions and actions as web services.

Defining a comprehensive architecture for a context-aware services platform is a non-trivial task. It involves several issues and domains, such as ubiquitous computing, artificial intelligence, human-computer interaction, and other crosscutting issues such as security and privacy, scalability and performance. Another effort within the WASP project has aimed at designing a privacy architecture for the WASP platform [14], providing the users of the platform with privacy control while being unobtrusive.

In the WASP project we have also investigated the applicability and usefulness of Semantic Web technologies for the representation of contextual information, leading to the development of context ontologies and the use of reasoners to detect context conditions [9]. Further work in the WASP project will consider the integration of the different results, such as the overall architecture and the privacy support, giving special attention to the use of Semantic Web technologies. We believe that the use of ontologies and reasoners are promising techniques to enhance the reusability, flexibility and intelligence of context-aware services platforms and applications.

References

1. DeVaul, R. et al.: The Ektara Architecture. MIT Technical Report (2000)
2. Dockhorn Costa, P.: Towards a Services Platform for Context-Aware Applications. Master Thesis. University of Twente, the Netherlands (2003)
3. Dockhorn Costa, P., Pereira Filho, J.G., van Sinderen, M.: Architectural Requirements for Building Context-Aware Services Platforms. In: E. Halasz et al (eds.): Proceedings of 9th Open European Summer School and IFIP Workshop on Next Generation Networks (EUNICE 2003). Hungary (2003)
4. Edwards, W.K., Bellotti, V., Dey, A.K., Newman, M.W.: Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI 2003). Fort Lauderdale, Florida, USA (2003)
5. Efstratiou, C. Cheverst, K., Davies, N., Friday, A.: An Architecture for the Effective Support of Adaptive Context-Aware Applications. In: Tan, K-L., Franklin, M.J., Lui, J.C.S. (eds.): Mobile Data Management: Second International Conference (MDM 2001). Lecture Notes in Computer Science, Vol. 1987. Springer-Verlag (2001) 15-26
6. Indulska, J., Loke, S.W., Rakotonirainy, A., Witana, V., Zaslavsky, A.B.: An Open Architecture for Pervasive System. In: Zielinski, K., Geihs, K., Laurentowski, A. (eds.): Proceedings of the 3rd International Working Conference on Distributed Applications and Interoperable Systems (DAIS 2001). Kluwer (2001) 175-188
7. Laar, V.: Requirements for the 3G Platform. WASP Deliverable D1.1 (2003)
8. Parlay Group: Parlay X Web Services White Paper (2002) [http://www.parlay.org/ about/parlay_x/ParlayX-WhitePaper-1.0.pdf]
9. Rios D., Dockhorn Costa, P., Guizzardi, G., Ferreira Pires, L., Pereira Filho, J.G., van Sinderen M.: Using Ontologies for Modeling Context-Aware Services Platforms. In: Workshop on Ontologies to Complement Software Architectures (OOPSLA 2003). Anaheim, CA, USA (2003)
10. Sun Microsystems: Java API for XML-Based RPC (JAX-RPC) Specification 1.0, JSR-101. [http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec1]
11. World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification (1998) [http://www.w3.org/TR/REC-DOM-Level-1/]
12. World Wide Web Consortium: Web Services Architecture (2003) [http://www.w3.org/TR/ws-arch/]
13. Universal Description, Discovery and Integration (UDDI) project: UDDI: Specifications. [http://www.uddi.org/specification.html]
14. Zuidweg, M., Pereira Filho, J.G., van Sinderen M.: Using P3P in a Web Services-Based Context-Aware Application Platform. In: E. Halasz et al (eds.): Proceedings of 9th Open European Summer School and IFIP Workshop on Next Generation Networks (EUNICE 2003). Hungary (2003)