Available online at www.sciencedirect.com
ScienceDirect
Pervasive and Mobile Computing 4 (2008) 360–389
pervasive and mobile computing
www.elsevier.com/locate/pmc

CoWSAMI: Interface-aware context gathering in ambient intelligence environments

Dionysis Athanasopoulos[a], Apostolos V. Zarras[a,*], Valerie Issarny[b], Evaggelia Pitoura[a], Panos Vassiliadis[a]

[a] *Department of Computer Science, University of Ioannina, GR 45110 Ioannina, Greece*
[b] *INRIA - UR Rocquencourt - France*

Received 18 April 2007; received in revised form 12 November 2007; accepted 11 December 2007
Available online 4 January 2008

Abstract

In this paper we present CoWSAMI, a middleware infrastructure that enables context awareness in open ambient intelligence environments, consisting of mobile users and context sources that become dynamically available as the users move from one location to another. A central requirement in such dynamic scenarios is to be able to integrate new context sources and users at run-time. CoWSAMI exploits a novel approach towards this goal. The proposed approach is based on utilizing Web services as interfaces to context sources and dynamically updatable relational views for storing, aggregating and interpreting context. Context rules are employed to provide mappings that specify how to populate context relations, with respect to the different context sources that become dynamically available. An underlying context sources discovery mechanism is utilized to maintain context information up to date as context sources, and users get dynamically involved.

Keywords: Ambient intelligence environments; Context awareness; Middleware; Web services

* Corresponding author.
E-mail addresses: dathanas@cs.uoi.gr (D. Athanasopoulos), zarras@cs.uoi.gr (A.V. Zarras), Valerie.Issarny@inria.fr (V. Issarny), pitoura@cs.uoi.gr (E. Pitoura), pvassil@cs.uoi.gr (P. Vassiliadis).

1574-1192/$ - see front matter
doi:10.1016/j.pmcj.2007.12.004

1. Introduction

Ambient Intelligence (AmI) refers to the development of environments that are aware and responsive to the presence of people [1–3]. AmI relies on Weiser's pioneer vision on ubiquitous computing [4]. Ubiquitous or pervasive computing [5] foresees a digital world consisting of interconnected electronic devices that support the quotidian activities of people. AmI goes one step further from the aforementioned initiatives by putting a specific focus on the users and their experience with the electronic facilities that surround them. This focus augments ubiquitous computing and networking with additional requirements for *natural user-friendly interaction* and *context awareness*.

Context in computing systems is defined as "*a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*" [6]. Typical middleware infrastructures that focus on context awareness provide basic programming means that allow defining context models, consisting of basic notions (e.g. location, battery, temperature) that affect the interaction between the user and the application. Based on these definitions, they facilitate the gathering and interpretation of context information. To achieve this goal, a layered architectural style [7,6,8] is typically employed. The lowest layer comprises *context sources*, i.e. elements that hide details related to the acquisition of raw context data (e.g. the use of sensors, GPS). The middle layer consists of *context aggregators*, which collect raw context data provided by context sources. Finally, the upper layer comprises *context interpreters*, which use context information to derive certain conclusions and determine corresponding adaptation actions concerning the user and the application.

The particular *problem* we are dealing with in this paper is *the dynamic integration of context sources in open AmI environments*, i.e., environments consisting of mobile context users and context sources which become available on-the-fly as the users move from one location to another. In open AmI environments we must facilitate the fusion of information offered through varying interfaces, provided by the context sources that become dynamically available. The ISTAG AmI Car and Emergency Rescue scenarios give typical examples of what we call open AmI environments [9]. Specifically, these scenarios involve mobile users that move across different locations, possibly using intelligent vehicles. The context that interests them may concern information related to their travel, the weather, the traffic, medical care, etc. Such information may be offered by varying sources in the form of services that are available in each location.

The dynamic nature of open AmI environments renders the use of existing context-aware middleware infrastructures problematic towards the development of such environments. The current practice on the integration of context users and context sources in these infrastructures is focused around two different approaches: context information is provided to the aggregators either on-demand (i.e. the aggregators contact the context sources) [7,10], or on-update (i.e. the context sources contact the aggregators) [7,11–13]. Both of the previous approaches are realized by imposing specific functional dependencies on the elements involved, i.e., either the aggregators must have prior knowledge of the interfaces of context sources, or the inverse. In other words, the current state-of-the-art assumes a tight coupling between context sources and context aggregators. Thus, there is no sufficient answer to the issue of *integrating loosely-coupled context sources and aggregators at run-time*.

Our vision towards a middleware infrastructure that enables context awareness in open AmI environments can be summarized in the following requirements:

- *Loose coupling & customization*. To enable the dynamic integration of context sources in an open AmI environment, the middleware infrastructure must impose the less possible functional constraints to these sources. In particular, the infrastructure must provide an easily customizable aggregator mechanism that adapts its behavior to the users' perception of context and to the interfaces of available context sources to facilitate the gathering of context information.
- *Dynamic discovery of context sources*. Given that context sources become available dynamically, the infrastructure must provide means for their discovery. The context sources discovery should be completely distributed, to effectively support environments that are formulated in a purely ad hoc manner and environments that span across wide areas.
- *User friendliness*. The middleware infrastructure must provide friendly means that would allow defining the users' perception of context and map this perception to the context sources that become dynamically available. Moreover, the infrastructure must provide means that would allow one to easily gather and interpret context information.

In this paper, we propose CoWSAMI as a possible solution to the above issues. CoWSAMI extends our previous work on WSAMI, a light-weight middleware platform for the development of mobile Web services [14]. *CoWSAMI, goes one step further by facilitating the dynamic integration of context users and sources in open AmI environments* based on the following concepts:

- *Interface-aware mapping of externally gathered context information to the internal, user perception of context.* The only requirement imposed by CoWSAMI on the context sources of an open AmI environment is that they should export interfaces that comply with the standard Web service architecture [15]. CoWSAMI further responds to the requirement for loose coupling and customization by offering means for defining dynamically updatable relational views that represent the users' perception of context. Moreover, CoWSAMI allows defining context rules that reflect the mapping between the users' perception of context and the interfaces of available context sources. A corresponding context aggregator is provided for gathering context information. The aggregator is *interface-aware* in the sense that it gathers context by tailoring its behavior with respect to the aforementioned mapping.
- *Multi-policy dynamic discovery of context sources*. To meet the requirement for the dynamic discovery of context sources, a completely distributed mechanism is provided to maintain context information up to date as context sources and users get dynamically involved. Given that the availability of context sources may evolve differently, the discovery mechanism offers the flexibility of associating different discovery policies with different categories of context sources.
- *User-friendly, SQL-based querying*. Typical database query languages provide a simple declarative way for exploring information stored in a database. To satisfy the user-friendliness requirement in CoWSAMI we follow the same direction. In particular,

CoWSAMI offers an easy-to-use SQL-based context interpreter, inspired by our previous work on querying ad hoc communities of peers [16].

The remainder of this paper is structured as follows. Section 2 presents a reference example, used for exemplifying the CoWSAMI approach. Section 3 details the architecture of CoWSAMI. Section 4 details the interface-aware context gathering process, while Section 5 discusses the dynamic context sources discovery process. Section 6 provides an evaluation of the CoWSAMI approach. Section 7 discusses related work. Finally, Section 8 concludes this paper with a summary of our contribution and the future directions of this work.

2. Reference example

The main goal of CoWSAMI is to enable the dynamic integration of mobile users with context sources that become dynamically available in an open AmI environment. To illustrate our approach, we specifically focus on the ISTAG Car scenario [9] that concerns the development of intelligent transport environments. The vision of developing such environments recently gained the attention of IEEE, which released the WAVE (Wireless Access in Vehicular Environments) communications standards [17]. WAVE standards extend the typical IEEE 802.11 to enable networking services amongst vehicles. According to IEEE's vision, drivers shall receive context information about road conditions, red lights, and hazards from other vehicles up to 0.5 Km ahead in highways and up to 0.1 Km ahead in cities. Emergency vehicles that may be useful in the ISTAG Emergency Rescue scenario shall be equipped with longer range WAVE systems. What is important to note at this point is that *the WAVE standard leaves each vehicle manufacturer to decide upon the issues of what sort of data will be provided and how*. Therefore, different vehicle manufacturers are free to come up with various different services that offer context information in vehicular environments.

Into this context, our reference example extends the one used in [14], which has been developed at INRIA for the OZONE IST project [18]. Our AmI environment is aimed at supporting the city of Rocquencourt, near Versailles. The environment offers CyberCars[1] to Rocquencourt citizens and tourists (Fig. 1). CyberCars are unmanned vehicles that can be booked through the Web, towards moving across different sites of the city.

Each CyberCar is equipped with its own computing facilities. Moreover, it may communicate with other entities through a wireless network. The embedded computers of CyberCars offer Web services that provide context information regarding the CyberCars' features (such as velocity, position, brand). As discussed earlier, the interfaces of these services may depend on each different CyberCar manufacturer. The implementation of the services also depends on the manufacturer [17] as it may involve gathering information from manufacturer-specific embedded electronic devices through special purpose command sets, offered by these devices (e.g. in [19] we discuss an approach for developing Web interfaces for GSM/GPRS enabled mobile sensors). Realizing, for

[1] http://www.cybercars.org.

Fig. 1. CyberCars used at INRIA Rocquencourt.

instance, a Web service that reports the velocity of a CyberCar may involve gathering the CyberCar's previous and current position for a particular time interval, using an embedded GPS tracker.[2] The city of Rocquencourt further offers Web services that provide information regarding the city's hotels and restaurants. These services may also vary depending on the different hotel and restaurant companies that offer them. Finally, in our example we assume that the Rocquencourt city hall provides a Web service that reports the location of various sites (such as monuments, organizations, hospitals, shopping centers).

Hence, in our environment the CyberCar, the hotel, the restaurant and the city-hall services are *context sources*, while Rocquencourt citizens and tourists play the role of *mobile context users*. In the open environment that we examine, the context information that interests the CyberCars' passengers may vary. Therefore, each passenger must be provided with an *aggregator* mechanism that enables the definition of the context model that interests him/her. Take, for instance, the case of an English speaking tourist who is interested in the current status of the traffic and the available city hotels. Using his/her PDA, he/she should be able to define a corresponding context model. Similarly, a French speaking Rocquencourt citizen who is only interested in the current status of the city traffic should be able to define a different context model.

One way to discover the current traffic situation for the English tourist and the French citizen is by dynamically discovering the varying Web services of the CyberCars that

[2] http://www.brickhousesecurity.com/gps-car-tracking-vehicle-logging.html.

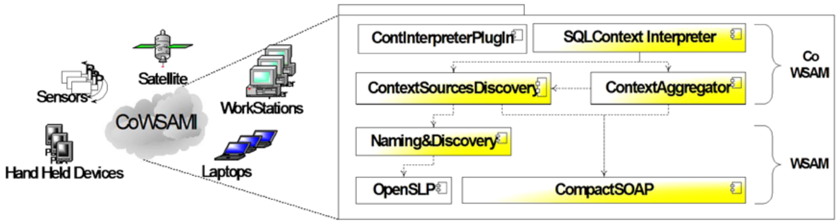


Fig. 2. CoWSAMI architecture.

circulate towards the direction where these users are heading. Information regarding the velocity of these CyberCars must be gathered, using the aggregator. To achieve this loosely-coupled integration, the aggregator must dynamically customize its behavior to the users' specific context models and the manufacturer-specific Web service interfaces of the context sources; in other words the aggregator must be interface-aware.

3. The CoWSAMI architecture

In our view, an AmI environment consists of networked entities, each one of which includes an instance of the CoWSAMI infrastructure (Fig. 2). The CoWSAMI entities may be connected through a LAN or WLAN. CoWSAMI also allows the formulation of pure ad hoc environments where the entities communicate through technologies such as Bluetooth. Finally, the environment may comprise entities located on the Internet. The entities vary from typical stationary workstations and PCs, to handheld and embedded devices such as PDAs, smart-phones and sensors.

In the open environments that we examine, every CoWSAMI entity may play the role of a context user, the role of a context source, or both. A context source provides one or more Web services that offer context information. The Web services are specified using the WSAMI language, which extends standard WSDL [20] with features enabling a more detailed behavioral description. A service specification consists of an *abstract* and a *concrete* part. The abstract part describes the *interface of the service* in terms of a WSDL document, whose URI (Uniform Resource Identifier) is included in the abstract part. Multiple service specifications may have *the same abstract part*, as long as they describe services providing the same interface. The concrete part of the service specification contains binding information (e.g., the endpoint address of the service, the communication protocol used, etc.).

Fig. 3(a) and (b) give examples of service specifications, for two different CyberCars. The upper parts of the figures give the abstract specifications of these entities, while the lower parts provide the WSDL interfaces referenced in these abstract specifications. CyberFrogs (Fig. 3(a)) offer an homonymous Web service whose interface comprises 6 operations that can be invoked to obtain a unique identifier that characterizes each CyberFrog, the CyberFrog's brand, velocity, fuel, and coordinates. On the other hand, CyberCabs (Fig. 3(b)) provide a Web service that comprises a single operation, which can be invoked to obtain all the characteristics of a CyberCab (i.e., its identifier, brand, velocity and coordinates).

Abstract part of WSAMI specification for CyberFrog

```
<Abstract name = "CyberFrog">
<Interface
hrefSchema="http://localhost:8080/CyberFrog.wsdl">
</Abstract>
```

CyberFrog Interface specification in WSDL

```
<wsdl:definitions>
<wsdl:definitions>
<wsdl:message name="getVelocityResponse">
 <wsdl:part name="getVelocityReturn" type="xsd:float" />
</wsdl:message>
------------------------------------------------------------
<wsdl:portType name="CyberFrog">
<wsdl:operation name="getID">
  <wsdl:output message="impl:getIDResponse"
                         name="getIDResponse" />
</wsdl:operation>
<wsdl:operation name="getVelocity">
  <wsdl:output message="impl:getVelocityResponse"
                         name="getVelocityResponse" />
</wsdl:operation>
</wsdl:portType>
------------------------------------------------------------
</wsdl:definitions>
```

(a) The CyberFrog interface.

Fig. 3. Different types of CyberCar interfaces.

In CoWSAMI, user context is specified in terms of *context attributes* (Fig. 4(a)). The information that corresponds to a context attribute is a value provided by a Web service. In general, the AmI environment may comprise multiple context sources that report semantically equivalent information (e.g., two different CyberCars reporting their current velocity). Moreover, a context source may provide values for more than one context attribute (e.g., the velocity and the brand). Therefore, related context attributes are organized into relations (Fig. 4(a)). A *context relation* is characterized by a *name* and consists of *a finite set of context attributes*. The context information modeled by a relation is a finite set of tuples, i.e., a finite subset of the cartesian product of the domains of the attributes that constitute the relation. The relational-based approach we employ for modeling context is also followed by other context-aware middleware infrastructures [8]. However, in most of these cases, context modeling is *controlled* in that there is a unified model defined for the environment. In CoWSAMI we aim at supporting open AmI scenarios in which different users may utilize different definitions of context relations. The relational-based modeling of context information allows formulating queries against relation instances, which are populated at run-time through the use of the CoWSAMI infrastructure.

Abstract part of WSAMI specification for CyberCab

```
<Abstract name = "CyberCab">
<Interface
hrefSchema="http://localhost:8080/CyberCab.wsdl">
</Abstract>
```

CyberCab Interface specification in WSDL

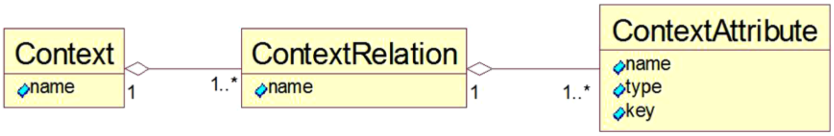
```
<wsdl:definitions>
---------------------------------------------------------------------
<wsdl:message name="getStateResponse">
  <wsdl:part name="ID" type="xsd:int"/>
  <wsdl:part name="Brand" type="xsd:string"/>
  <wsdl:part name="Velocity" type="xsd:float"/>
  <wsdl:part name="Fuel" type="xsd:float"/>
  <wsdl:part name="XPos" type="xsd:float"/>
  <wsdl:part name="YPos" type="xsd:float"/>
</wsdl:message>
---------------------------------------------------------------------
<wsdl:portType name="CyberCab">
<wsdl:operation name="getState">
  <wsdl:output message="impl:getStateResponse"
      name="getStateResponse" />
</wsdl:operation>
</wsdl:portType>
---------------------------------------------------------------------
</wsdl:definitions>
```

(b) The CyberCab interface.

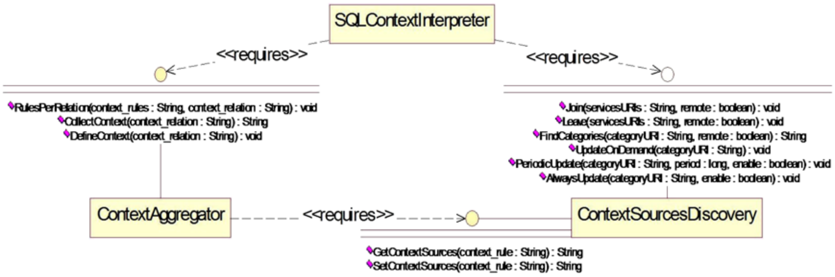
Fig. 3. (*continued*)

In detail, the CoWSAMI infrastructure is structured as depicted in Fig. 2. Its lowest layer comprises CSOAP (Compact SOAP), a light-weight communication mechanism, which allows deploying, invoking and executing Web services on resource-constrained devices. The same layer comprises a standard SLP (Service Location Protocol) server [21] which serves for locating networked CoWSAMI entities in the environment. On top of the SLP server, the Naming&Discovery service realizes a primitive Web service discovery protocol. On top of the primitive Naming&Discovery service, a *context sources discovery mechanism* is incarnated by the ContextSourcesDiscovery service. The same layer further comprises a *context aggregator* mechanism, realized by the ContextAggregator service. Finally, the upper layer of CoWSAMI comprises at least a simple *context interpreter*, realized by the SQLContextInterpreter service. This service provides user-friendly means for gathering context information through the use of its underlying aggregator and discovery services.

Fig. 4(b) gives the interface of the ContextAggregator service which allows performing the following tasks: (1) Define the particular context relations that interest a user; (2) define the context rules that customize the gathering of context information with respect to the interfaces of the context sources that become dynamically available and the context relations of interest; (3) compile the tuples that constitute the context relations of interest.



(a) Context structure.



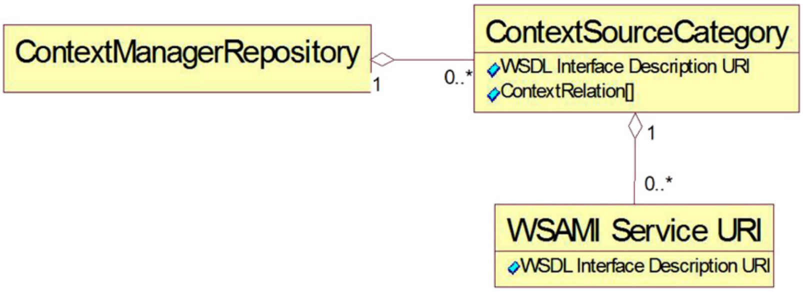
(b) CoWSAMI services.

Fig. 4. Context structure and CoWSAMI service interfaces.

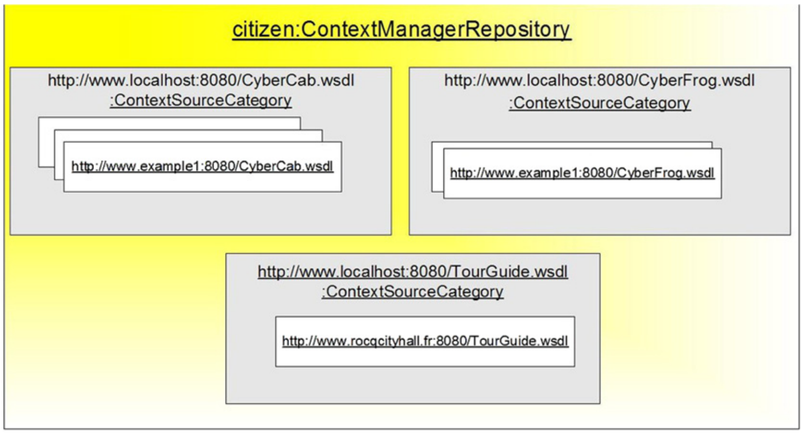
Fig. 4(b) further details the interface of the ContextSourcesDiscovery service. Through this service a CoWSAMI entity that plays the role of a context source can perform the following tasks: (1) Join the AmI environment, i.e., make itself available as a context source to other CoWSAMI entities; (2) leave the AmI environment, i.e., resign from playing the role of a context source. Through the same service, an entity that gathers context information can perform the following tasks: (1) Discover the different Web service interfaces offered by context sources that become dynamically available; (2) populate a repository managed by the ContextSourcesDiscovery service with addressing information concerning context sources that become dynamically available. Finally, the ContextSourcesDiscovery service provides the ContextAggregator service with means for acquiring addressing information concerning the available context sources that contribute to a particular context relation of interest.

The repository of the ContextSourcesDiscovery service consists of a number of categories (Fig. 5(a)). A *category is characterized by the URI of the abstract service description that specifies the particular interface provided by the context sources, belonging to this category*. The category contains URIs that identify service specifications, whose abstract parts reference the URI that characterizes the category. The concrete parts of these service specifications comprise specific addressing information for the available context sources that offer the specified services. The *category is further characterized by the names of the context relations to which it contributes*.

Returning to our reference example, Fig. 5(b) gives a possible snapshot of the repository, managed by the ContextSourcesDiscovery service that is deployed on the French citizen's PDA. The repository comprises 2 categories for CyberCar URIs and a single category for the Rocquencourt city-hall service. The first of the CyberCar categories is characterized by the URI of the CyberFrog interface, given in Fig. 3(a). Similarly, the second category



(a) Repository structure.



(b) Repository instance.

Fig. 5. ContextSourcesDiscovery repository.

is characterized by the URI of the CyberCab interface, given in Fig. 3(b). The CyberFrog category contains 2 URIs of available services, providing this interface. On the other hand, the CyberCab category contains 3 URIs of available CyberCab services.

4. Interface-aware context aggregation

The distinctive feature of the context aggregation process as realized by the ContextAggregator service is that it adapts itself to the varying Web service interfaces offered by context sources that become dynamically available. Achieving this feature relies on two XML schemas that serve for defining context relations and rules. These tasks are further detailed in this section along with the overall context aggregation process.

4.1. *Defining context*

The ContextAggregator service accepts as input (through the SQLContextInterpreter front-end), context definitions that follow the XML schema of Fig. 6(a). Context relations

Context Relations

```
<xsd:element name="ContextRelation">
<xsd:complexType >
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:sequence>
     <xsd:element ref="ContextAttribute" minOccurs="1"
                                  maxOccurs = "unbounded"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Context Attributes

```
<xsd:element name="ContextAttribute">
<xsd:complexType >
  <xsd:attribute name="name"
       type="xsd:string" use="required"/>
  <xsd:attribute name="type"
       type="xsd:anySimpleType" use="required"/>
  <xsd:attribute name="key"
       type="xsd:boolean" default="false"/>
</xsd:complexType>
</xsd:element>
```

(a) Context schema.

```
<Context name="EnglishTouristContext">
<ContextRelation name="TRAFFIC">
  <ContextAttribute name="ID" type="int" key="true" />
  <ContextAttribute name="BRAND" type="string"/>
  <ContextAttribute name="VELOCITY" type="float"/>
  <ContextAttribute name="FUEL" type="float"/>
  <ContextAttribute name="XCOORD" type="float"/>
  <ContextAttribute name="YCOORD" type="float"/>
</ContextRelation>
<ContextRelation name="MAP">
  <ContextAttribute name="SITE" type="string" key="true"/>
  <ContextAttribute name="XCOORD" type="float"/>
  <ContextAttribute name="YCOORD" type="float"/>
</ContextRelation>
<ContextRelation name="HOTELS">
  <ContextAttribute name="ID" type="int" key="true"/>
  <ContextAttribute name="SINGLE_ROOM_PRICE" type="int"/>
  <ContextAttribute name="DOUBLE_ROOM_PRICE" type="int"/>
</ContextRelation>
</Context>
```

```
<Context name="FrenchCitizenContext">
<ContextRelation name="CIRCULATION">
  <ContextAttribute name="ID" type="int" key="true" />
  <ContextAttribute name="MARQUE" type="string"/>
  <ContextAttribute name="VELOCITY" type="float"/>
  <ContextAttribute name="XPOS" type="float"/>
  <ContextAttribute name="YPOS" type="float"/>
</ContextRelation>
<ContextRelation name="PLAN">
  <ContextAttribute name="ENDROIT" type="string" key="true"/>
  <ContextAttribute name="XPOS" type="float"/>
  <ContextAttribute name="YPOS" type="float"/>
</ContextRelation>
</Context>
```

(b) Examples of context definitions.

Fig. 6. Context definition.

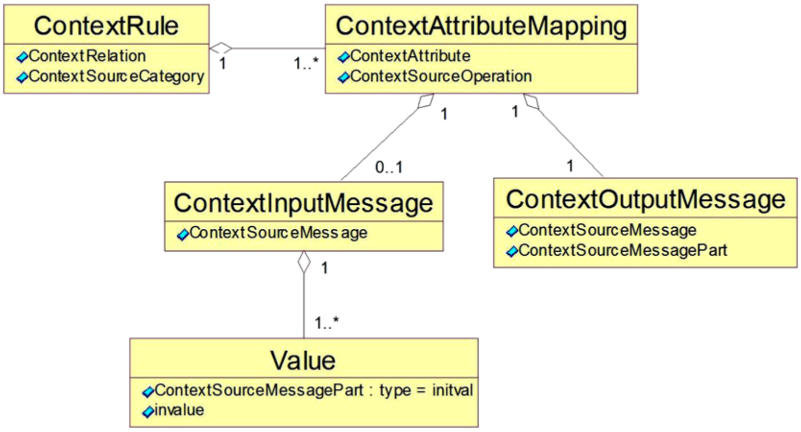
are specified using the ContextRelation tag. Within this tag, it is possible to define one or more context attributes, using the ContextAttribute tag.

In our reference example, the English speaking tourist who is interested in the current traffic situation and the available city hotels may use the three context relations showed on the left part of Fig. 6(b). The first relation is named TRAFFIC and consists of 6 attributes (ID, BRAND, VELOCITY, FUEL, XCOORD and YCOORD), characterizing CyberCars that circulate in the environment. The second relation is named MAP and consists of three attributes that correspond to the name and the coordinates of a site. The third relation is called HOTELS and comprises attributes that characterize city hotels. Similarly, the French speaking Rocquencourt citizen who is only interested in the city traffic may use a simpler context definition consisting of the two relations given on the right side of Fig. 6(b). The first relation is analogous to the TRAFFIC relation defined by the tourist; it is named CIRCULATION and consists of 5 attributes (ID, MARQUE, VITESSE, XPOS and YPOS). The second relation is named PLAN and it is similar to the MAP relation, defined by the tourist.

Technically, a given context definition is parsed by the ContextAggregator service towards finding the constituent context relations, which are persistently stored.

4.2. *Defining context rules*

A context rule must be provided to the ContextAggregator service, upon the discovery (through the ContextSourcesDiscovery service) of a category of context sources that may contribute in populating a context relation that interests the user. In principle, the context



(a) Context rule structure.

Fig. 7. Context rule definition.

rule associates *the attributes of the context relation* with the *operations* offered by the Web services of the discovered category of context sources.

In general, the interface that characterizes a discovered category of context sources consists of a number of PortType definitions (Fig. 3(a), (b)) [20]. A PortType, in turn, consists of the definitions of the operations that can be called on the context sources of this category. Each operation may specify an input and an output message. The former element refers to the SOAP request message sent upon the operation invocation, while the latter element refers to the SOAP message returned in response to the invocation. The specification of a message consists of one or more parts. Each part is characterized by a name and the particular type of datum exchanged between a service and an entity that invokes the service.

Therefore, a context rule that describes how to populate a context relation, using a discovered category of context sources is specified according to the schema given in Fig. 7(b). In particular, the context rule is characterized by the name of the context relation and the URI of the interface specification that characterizes the discovered category. The rule comprises one or more ContextAttributeMapping elements. Each element describes how to obtain a value for an attribute of the context relation, by invoking a corresponding operation of the specified interface. The ContextAttributeMapping element comprises two constituent parts. The first part (specified using the ContextInputMessage tag) is characterized by the type of the request message that should be sent upon the operation invocation and a sequence of values that should be included in the message. The second part (specified using the ContextOutputMessage tag) is characterized by the type of the response message, received after the operation invocation. Moreover, the second part of the mapping specifies the name of the *actual part* of the response message that contains the value of the context attribute.

Every rule given to the ContextAggregator service is parsed and its encapsulated ContextAttributeMapping elements are grouped with respect to the operations that have to

Context Rules

```
<xsd:element name="ContextRule">
<xsd:complexType >
    <xsd:attribute name="ContextRelation" type="xsd:string" />
    <xsd:attribute name="ContextSourceCategory" type="xsd:anyURI" />
    <xsd:sequence>
      <xsd:element ref="ContextAttributeMapping" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Context Attribute Mapping

```
<xsd:element name="ContextAttributeMapping">
<xsd:complexType >
    <xsd:attribute name="ContextAttribute" type="xsd:string" />
    <xsd:attribute name="ContextSourcePortType" type="xsd:string" />
    <xsd:attribute name="ContextSourceOperation" type="xsd:string" />
    <xsd:sequence>
     <xsd:element ref="ContextInputMessage" minOccurs="0"  maxOccurs = "1"/>
     <xsd:element ref="ContextOutputMessage" minOccurs="1"  maxOccurs = "1"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Context Input Message & Values

```
<xsd:element name="ContextInputMessage">
<xsd:complexType >
    <xsd:attribute name="ContextSourceMessage" type="xsd:string" />
    <xsd:sequence>
     <xsd:element ref="Value" minOccurs="1"  maxOccurs = "unbound"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="Value">
<xsd:complexType >
    <xsd:attribute name="ContextSourceMessagePart" type="xsd:string"/>
    <xsd:attribute name="invalue" type="xsd:anySimpleType"/>
</xsd:complexType>
</xsd:element>
```

Context Output Message

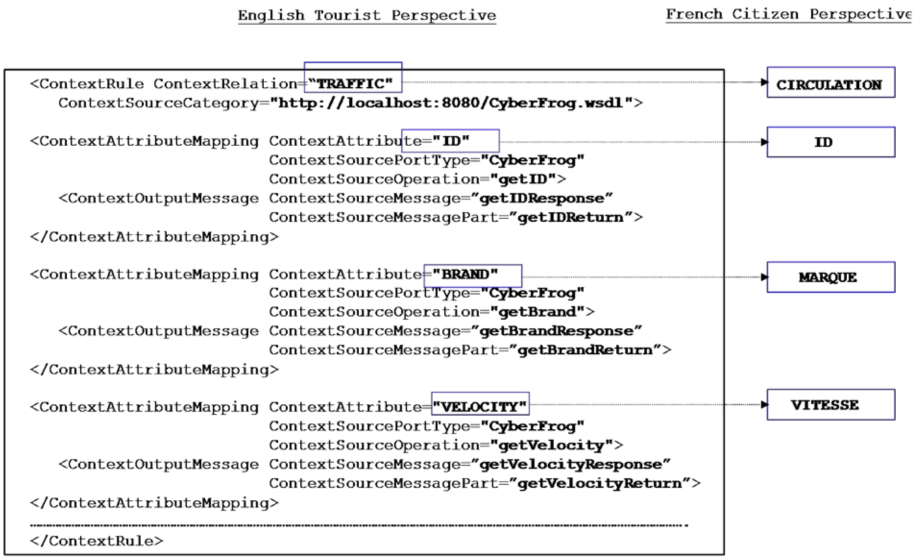
```
<xsd:element name="ContextOutputMessage">
<xsd:complexType >
    <xsd:attribute name="ContextSourceMessage" type="xsd:string" />
    <xsd:attribute name="ContextSourceMessagePart" type="xsd:string"/>
</xsd:complexType>
</xsd:element>
```

(b) Context rule schema.

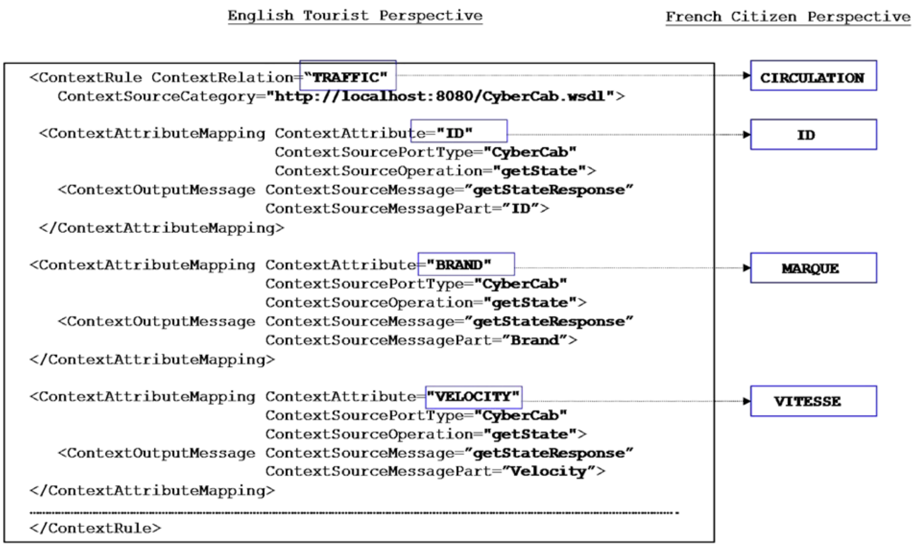
Fig. 7. (*continued*)

be called to fill up the values of context attributes. This grouping is necessary for optimizing the collection of context information in cases where the values of multiple attributes are obtained by calling a single operation (e.g., Fig. 8(b)). Finally, every context rule is stored persistently in the device where the ContextAggregator service is deployed.

Moving on to our reference example, Fig. 8(a) gives a part of the rule that describes how to populate the TRAFFIC relation, employed by the English tourist (Fig. 6(b)), using context sources that offer the CyberFrog interface. The rule specifies a context attribute mapping for each one of the ID, BRAND, VELOCITY, FUEL, XCOORD and YCOORD attributes. To obtain a value for the VELOCITY attribute, for instance, the getVelocity() operation must be invoked. No input message is needed for this invocation since the operation accepts no input parameters. The actual value of VELOCITY is stored



(a) Context rules for the CyberFrog interface.



(b) Context rules for the CyberCab interface.

Fig. 8. Examples of context rules.

in the getVelocityReturn part of the getVelocityResponse message (see Fig. 3(a) for the definitions of these WSDL elements), returned upon the invocation on the aforementioned operation. Fig. 8(b) gives a part of the rule that describes how to populate the same relation, using context sources that belong to the CyberCab category (Fig. 3(b) gives the interface

Fig. 9. Examples of queries issued against context relations.

specification that characterizes this category). To obtain a value for the VELOCITY attribute, the getState() operation must be invoked. The value of VELOCITY is now stored in the Velocity part of the getStateResponse message. Similarly, to obtain a value for the ID attribute, the same operation must be called. The value of the attribute is stored in the ID part of the aforementioned message. Fig. 8(a) and (b) further highlight the main differences between the rules defined for the English tourist's TRAFFIC relation and the rules that should be given for the French citizen's CIRCULATION relation (Fig. 6(b)).

4.3. Collecting context information

Collecting context information amounts to providing an SQL query to the SQLContextInterpreter service (Fig. 9). For every context relation specified in the FROM clause of the query, the ContextAggregator service is invoked to gather related tuples (Fig. 10 step 1). The ContextAggregator retrieves from the repository maintained by the ContextSourcesDiscovery service the different categories of context sources that contribute to the compilation of these tuples, along with the context rules that correspond to these categories (Fig. 10 steps 2–3). The operations prescribed in the rules are invoked on the context sources, using dynamic JAXRPC Web service invocations[3] and the response messages are parsed to obtain the raw context data that serve as values of the context attributes that constitute the context relation (Fig. 10 steps 4–7). Certain transformations (e.g. miles to Km, s to ms, etc) may be further applied on the data by intercepting the JAXRPC Web service invocations with WSAMI customizers [14].

Once the tuples of the context relations involved in the query are assembled, the results are returned to the SQLContextInterpreter service, which processes them and results are returned back to the user. The processing of the tuples is performed differently, depending on whether CoWSAMI is deployed on a resource-constrained device or not. Specifically, for the case of resource-constrained devices, the SQLContextInterpreter comprises a simple set of main memory relational operators that directly process the

[3] http://java.sun.com/webservices/jaxrpc/index.jsp.

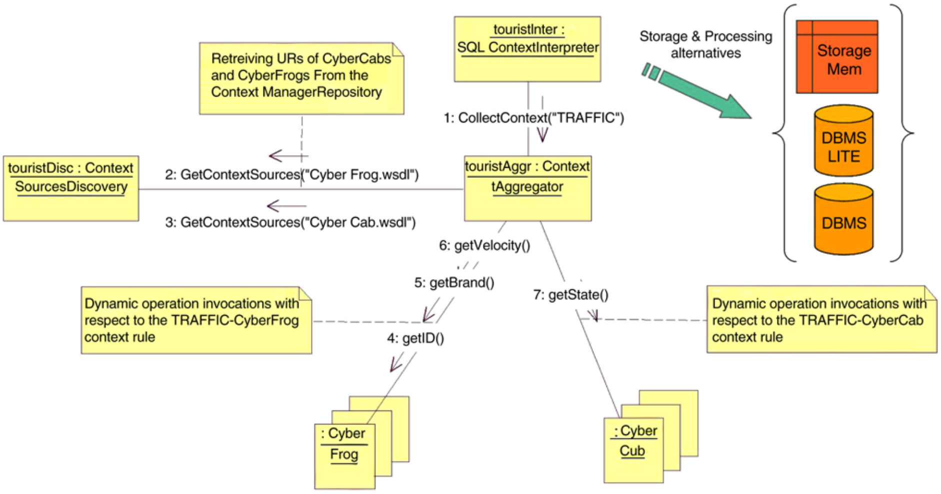


Fig. 10. Collaboration among the CoWSAMI middleware services towards gathering context.

incoming tuples. In this case, the tuples that constitute a context relation are stored in the device storage memory, in files characterized by the name of the relation and the user that defined it. A possible alternative for storing and processing the compiled tuples is through the integration of the SQLContextInterpreter with existing DBMS support for resource-constrained devices (e.g., Oracle Lite [22], IBM DB2 Everyplace [23]). For non-resource-constrained entities, the SQLContextInterpreter can be integrated with a full-blown relational engine. In any of these cases, the contents of the storage media (device storage memory, or DBMS) are updated with respect to a given freshness threshold.

In our reference example, the SQLContextInterpreter accepts as input, queries such as the ones given in Fig. 9. The left part of Fig. 9 shows a query issued by the English tourist against the TRAFFIC relation (Fig. 6(b)). The query returns the average velocity of the CyberCars that circulate in an area defined by the current position of the tourist's CyberCar and the position of the tourist's destination. Similarly, the right part of Fig. 9 shows the query that is issued by the French citizen to check for the current status of the traffic towards the citizen's destination. This query is issued against the CIRCULATION relation (Fig. 6(b)) Answering any of the two queries amounts to collecting the velocity of all the different CyberCars that can be accessed. The job of collecting this information is performed by the ContextAggregator service of each user, with respect to the context rules (Fig. 8) specified for the two different Web service interfaces of Fig. 3, which characterize the CyberCars that are currently available.

5. Multi-policy context sources discovery

Dynamic context sources discovery is the second essential feature provided by CoWSAMI to enable context awareness in open AmI environments consisting of mobile users and context sources that become dynamically available as the users move from one location to another. Given that certain environments may be formulated in a completely *ad hoc* manner we employ a distributed discovery protocol that extends the one realized by the basic WSAMI Naming&Discovery service. The *modus operandi* of this service along with the required extensions that enable the dynamic discovery of context sources is further detailed in the remainder of this section.

5.1. The Naming&Discovery service

In CoWSAMI, a Naming&Discovery service is deployed on every entity along with a standard SLP (Service Location Protocol) server [21], which periodically broadcasts the URI of the Naming&Discovery service. Based on the previous mechanism, the Naming&Discovery service is aware of other reachable Naming&Discovery services. The Naming&Discovery service manages a catalog, named *local*, which contains the URIs of the Web services offered by the CoWSAMI entity. The service further manages a catalog, named *remote*, which acts as a local cache that contains URIs of Web services, deployed on other reachable CoWSAMI entities; these Web services were discovered at some point in the past by the CoWSAMI entity.

The overall service discovery process starts with a request for a particular Web service interface, made by the CoWSAMI entity on its locally deployed Naming&Discovery service, which takes charge of forwarding this request to all other reachable Naming&Discovery services. Each one of them, checks its local and remote repositories for Web services that offer the required interface and replies back to the requesting entity with the URIs of these services.

5.2. The ContextSourcesDiscovery service

The context sources discovery builds upon the generic service discovery protocol discussed in the previous subsection and involves performing the following tasks. The ContextSourcesDiscovery service, on the side of a context user, discovers other reachable ContextSourcesDiscovery services and contacts them to find out about the different categories of context sources that are available in the environment. Alternatively, the user may find out about the different categories of context sources by contacting a universal repository (if there is one available) [14] that maintains these categories with respect to a standardized ontology, employed for the particular environment. Following this, the user selects the categories that can contribute to the context relations that interest him/her. Finally, the user utilizes the ContextSourcesDiscovery service to populate the ContextSourcesDiscovery repository with URIs of Web services that belong to the selected categories.

The ContextSourcesDiscovery repository can be managed with respect to three alternative policies, configured by the user *per different category*. Specifically, the alternative policies are: (1) Update-On-Demand, i.e., the repository contents are updated upon a request issued by the user towards the ContextSourcesDiscovery service; (2) Periodic-Update, i.e., the repository contents are updated periodically, with respect to a period specified by the user to the ContextSourcesDiscovery service; (3) Always-Update, i.e., the repository contents are updated whenever a context source of interest joins or leaves the environment. Associating different categories of context sources with different

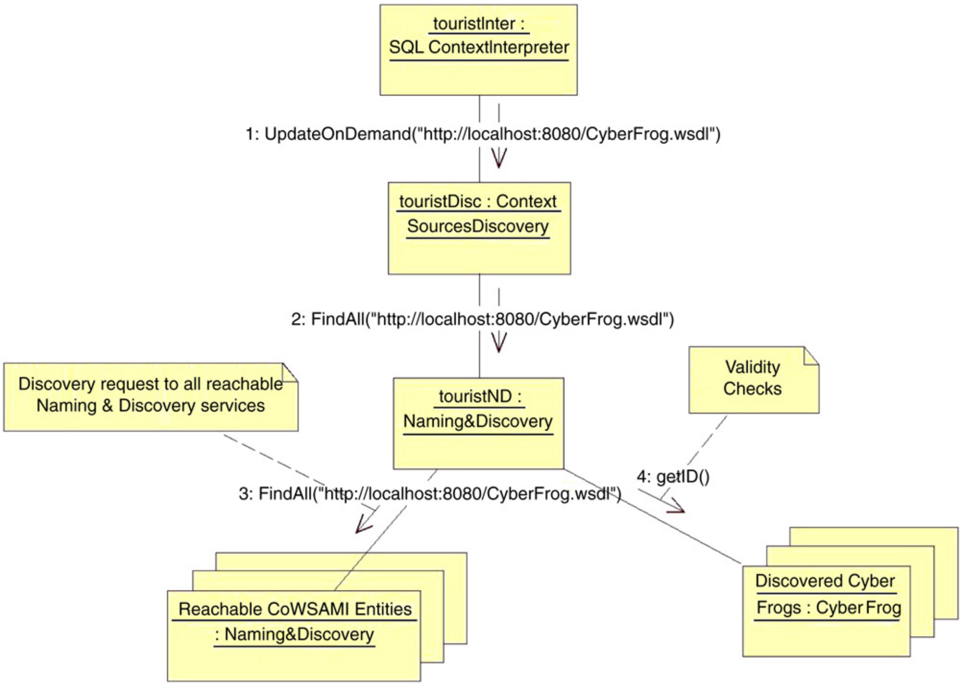


Fig. 11. Collaboration among the CoWSAMI middleware services towards discovering context sources.

update policies is useful considering that the availability of certain categories of services may change faster than the availability of certain others. In our reference example, for instance, it is reasonable to assume that tourists select the Update-On-Demand policy for hotel and restaurant services, since their availability is expected to change rather slowly. On the other hand, the Periodic-Update or the Always-Update policies seem to be more suitable for discovering CyberCars.

The Update-On-Demand policy is activated (through the SQLContextInterpreter front-end), by providing as input to the ContextSourcesDiscovery service a category of context sources that interest the user. In Fig. 11, for instance, the discovery process is activated towards the discovery of CyberFrogs. Following this, the service uses the interface that characterizes the given category as input to the Naming&Discovery service, which takes charge of locating URIs of Web services, providing this interface, as detailed in the previous subsection (Fig. 11 steps 2–3). At this point, it should be noted that some of the URIs that result from the Naming&Discovery service discovery protocol may not be valid. As discussed in the previous subsection, some of the URIs were possibly found in the remote catalogs of other reachable Naming&Discovery services. These catalogs act as local caches. Hence, it is probable that some of the cached URIs correspond to Web services offered by context sources that are no longer available. Using such services for gathering context implies an unnecessary waste of time and resources. To deal with this issue, the ContextSourcesDiscovery service checks the validity of the retrieved URIs by randomly selecting and invoking an operation on the corresponding Web services (Fig. 11 step 4).

The Periodic-Update policy works similarly. In particular, the user activates this policy, by providing a category of context sources and a time period T in ms. Enabling the policy results in the creation of a thread, which resumes its execution every T ms. At this point, the thread performs the steps of the Update-On-Demand policy (Fig. 11) to update the repository contents of the given category.

Finally, the Always-Update policy is also activated by providing a category of context sources as input to the ContextSourcesDiscovery service. After the policy activation, the service performs the basic steps that realize the Update-On-Demand policy (Fig. 11). Following this, the service listens for notifications coming from joining and leaving entities, until the user deactivates the policy. More specifically, whenever a CoWSAMI entity joins the environment as a context source, it uses its locally deployed ContextSourcesDiscovery service for registering the URIs of its Web services to the locally deployed Naming&Discovery service. Then, the ContextSourcesDiscovery service notifies all other reachable ContextSourcesDiscovery services about the entity's intention to join as a context source. On the side of the notified services, it is checked whether some of the Web services offered by the joining entity belong to categories for which the Always-Update policy is enabled. The URIs of such services are stored in the corresponding categories. The ContextSourcesDiscovery service of a leaving entity is also obliged to notify all other reachable ContextSourcesDiscovery services about the entity's intention. Following this, the notified services modify the contents of their repositories accordingly.

6. Evaluation

The assessment of CoWSAMI concerns 3 different aspects. First, we discuss the issue of user-friendliness. Following this, we investigate the memory requirements of the main CoWSAMI services. Finally, we examine the performance of these services.

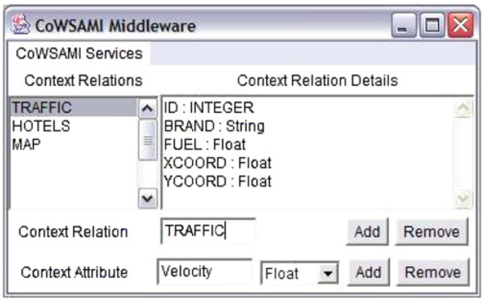
6.1. CoWSAMI and context users

CoWSAMI has been developed while keeping in mind two main target groups: (1) developers that customize the basic CoWSAMI mechanisms towards the realization of context-aware applications for a particular AmI environment and (2) end-users that utilize CoWSAMI applications.

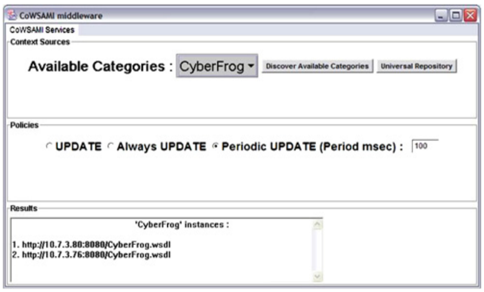
To simplify the work of the developers' target group, we employed the service-oriented paradigm towards the dynamic integration of context sources in open AmI environments. In typical service-oriented environments, information is gathered by orchestrating primitive Web services into composite ones (e.g. through BPEL,[4] or WSFL[5]). These technologies, along with the flexibility of SQL querying and the programmable service-oriented

[4] http://www.ibm.com/developerworks/webservices/library/ws-bpel/.

[5] http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.



(a) Context definition.



(b) Context sources discovery.

Fig. 12. CoWSAMI user interfaces.

interfaces of the basic CoWSAMI mechanisms facilitate the development of context-aware applications over CoWSAMI. As a proof of concept, we developed simple user interfaces on top of the basic CoWSAMI services that facilitate the definition of context relations and rules, the formulation of SQL queries and the customization of the context sources discovery process (e.g. Fig. 12(a), (b)). Using the interface of Fig. 12(a), a user can add/remove/modify context relations and attributes in a quite straightforward way. The interface of Fig. 12(b) serves for the discovery of available categories of context sources. The discovery can be done either in the user's vicinity, or through the universal repository, using the corresponding interface buttons shown in the figure. Following this, the user may specify a particular category of context sources in the drop-down list of the interface and associate this category with a particular update policy, using the check-box that lies below the drop-down list.

Nevertheless, results from the OZONE IST project [18] with end-users of varying ages (20–60 years), education and socio-economic backgrounds, suggested that the unfamiliarity with the Web service technology imposes certain difficulties on using infrastructures that rely on this technology. In an open AmI environment, the end-users will not be sitting comfortably in front of their workstation, having the ability to write down BPEL or WSFL code. Moreover, the typical context users will not be experienced developers. Dealing with the previous issues amounts to developing environment-specific CoWSAMI applications and user interfaces such as the ones given in Fig. 12. Moreover,

Table 1
Summary of memory requirements for CoWSAMI

Middleware	Required memory (MB)			
WSAMI	<=3.9			
CoWSAMI	ContextSourcesDiscovery (Update Policies)			ContextAggregator
	On-Demand	Periodic	Always	
	<=0.75	<=0.85	<=0.75	<=0.6

the work of the users can be simplified by offering them pre-defined context relations, rules and SQL queries, which can be easily developed by the CoWSAMI developers' group of the target AmI environment. Finally, the development of CoWSAMI interfaces and applications involves further interesting issues such as trans-rendering for different types of user devices and multilingual support.

6.2. CoWSAMI memory requirements

Currently, resource-constrained devices come with various technical characteristics (CPU, OS, memory, autonomy, etc.).[6] The available memory in the most recent models of PDAs ranges from 8 MB to 256 MB. However, there still exist few PDA models with less than 4 MB of RAM. Similarly, the memory provided by the most recent smart-phones ranges from 8 MB to 32 MB, while there still exist few models with less than 4 MB of RAM. The overall memory footprint of the WSAMI platform is 3.9 MB [14]. Therefore, it is suitable for CoWSAMI entities that execute on top of devices that have at least 8 MB of memory.

The additional memory requirements of the ContextAggregator and the ContextSourcesDiscovery services vary, depending on the operations that were performed by these services (Table 1). Regarding the ContextAggregator service, the context relations and the context rule definitions are quite cheap since they do not involve any additional invocations on Web services provided by available context sources. On the other hand, the context gathering operation involves contacting context sources towards collecting the information provided by these sources. Hence, its memory requirements are higher. However, even the execution of this operation requires less than 600 kB of memory. Joining or leaving an environment using the ContextSourcesDiscovery service requires less than 750 kB of memory. The realization of the Update-On-Demand and the Always-Update policies also requires less than 750 kB. The realization of the Periodic-Update policy is slightly more expensive, requiring less than 850 kB. The extra overhead of this policy is due to the use of the additional thread that periodically executes the basic steps of the Update-On-Demand policy (Section 5.2).

At this point, it should be mentioned that the aforementioned values were obtained after the optimizations which were performed in the early versions of the ContextAggregator and the ContextSourcesDiscovery services. In the early version of the ContextAggregator

6 http://www.palmzone.net.

service, we observed that the amount of memory required for collecting context information depends on the number of available context sources and the interface of these sources. To keep the memory used by the ContextAggregator constant and independent from the interface of the context sources we explicitly invoke the Java garbage collector right after each Web service invocation which was performed on the context sources. In the early version of the ContextSourcesDiscovery we observed a similar problem; the amount of memory required for updating the repository contents of the service increased with the number of context sources that were available in the environment. The reason behind this was that the ContextSourcesDiscovery service invokes operations on each context source to validate its presence. To avoid this problem and stabilize the amount of memory required to a value that does not depend on the number of available context sources we also explicitly call the Java garbage collector after every Web service invocation which was performed on the context sources.

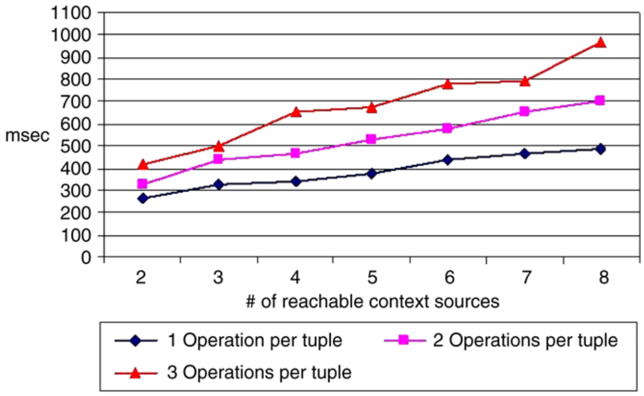
6.3. CoWSAMI performance

To assess the performance of CoWSAMI we performed two different sets of experiments (inspired by our reference example) towards measuring the execution time of basic operations performed by the CoWSAMI services. In the first set we configured an environment consisting of a maximum number of 9 CoWSAMI entities (3 P-IV 256 MB, 5 P-IV 512 MB, 1 P-IV 1 GB), communicating through a typical IEEE 802.11g WLAN. One of the entities played the role of a context user, while the others played the role of context sources. In the second set of experiments we scaled up the number of available users and context sources in a simulated environment whose parameters (i.e. average execution times for: (1) discovering a context source, (2) querying a context source, (3) joining the environment and (4) leaving the environment) resulted from the first set of experiments.

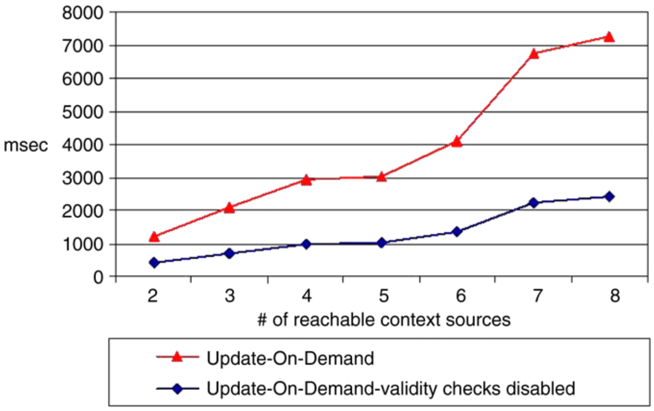
6.3.1. 1st set of experiments

In this set of experiments first we measured the time required for gathering context information for a context relation that consisted of 6 attributes (we assumed the TRAFFIC relation of our reference example). Specifically, we examined the impact of the increasing number of context sources that contribute to the relation. We used different configurations where the number of context sources ranged from 2 to 8. The context user entity performed the SELECT * FROM TRAFFIC query. For every different configuration, we produced 3 variations, corresponding to 3 different categories of context sources. In the first variation a single operation is called on each source to compile the relation tuples. Similarly, in the remaining two variations 2 and 3 operations were called on each source to compile the relation tuples. To realize the 3 variations we used corresponding sets of context rules. As expected, the overhead of the context gathering operation linearly increases, according to the number of reachable context sources and the nature of their interfaces (Fig. 13(a)). Specifically, the measured times in the most complex case where 3 operations were called, arise up to 1000 ms, which is considered to be reasonable.

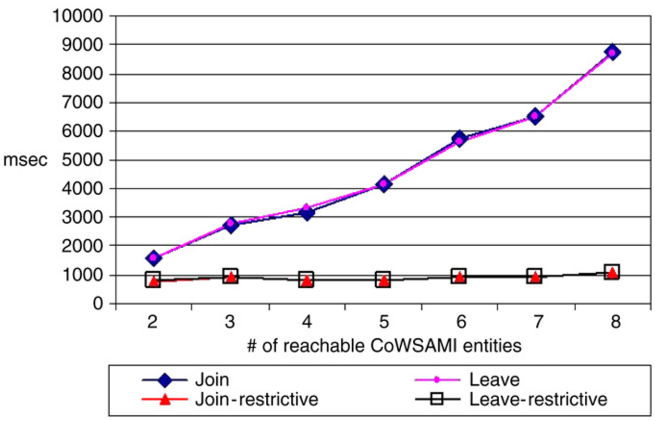
Regarding the discovery of context sources, the Periodic-Update and the Always-Update policies rely on the realization of the Update-On-Demand policy (Section 5.2). Based on this remark, we examined the impact of the increasing number of reachable



(a) Gathering context.



(b) Updating the repository.



(c) Joining and leaving the environment.

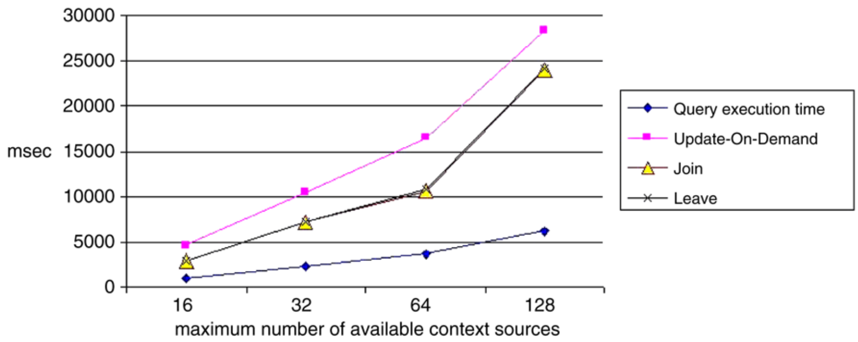
Fig. 13. Performance results — 1st set of experiments.

context sources in the realization of these policies. The number of reachable context sources in this experiment ranged from 2 to 8. All the context sources belonged to the same category, which was given as input to the ContextSourcesDiscovery service deployed on the user entity. In this experiment we examined two different scenarios. In the first one, the validation checks performed by the ContextSourcesDiscovery service on the side of the user were enabled, while in the second one they were disabled. Fig. 13(b) summarizes the results we obtained. We can observe that the overhead of the Update-On-Demand policy increases with the number of context sources. The overhead is partially due to the use of the Naming&Discovery service. As discussed in Section 5.1, the Naming&Discovery service that is deployed on an entity forwards requests for service discovery to all other reachable Naming&Discovery services. The overhead is further due to the validation calls, performed after the discovery of reachable context sources. The comparison between the two scenarios that we examined shows that the impact of the validation checks in the performance of the context sources discovery process is greater than the impact of the Naming&Discovery service.

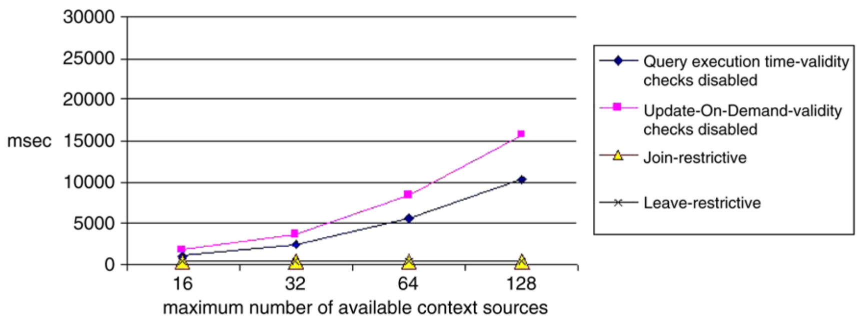
Fig. 13(c) gives the time spent by a context source for joining and leaving the environment. In this particular experiment, we assumed a scenario where the entities may use any of the three update policies discussed in Section 5.2 and a restricted scenario where the entities may use only the Periodic-Update or the Update-On-Demand policies. The overhead of the joining/leaving entity in the first scenario linearly increases with the number of entities that constitute the environment. This increment is expectable given that all of the aforementioned entities are notified whenever a context source joins or leaves the environment. As discussed in Section 5.2, the notifications are necessary for the realization of the Always-Update policy that may be activated by the entities. The Always-Update policy has a significant impact on the performance of our environment even if the policy is disabled, since the notifications are sent by the joining/leaving entity blindly to all the other entities in the environment, independently from the update policy used by these entities. The reason behind this is that in a completely open AmI environment we consider impractical to assume that every entity knows about the update policies used by the other entities. In our restrictive scenario the performance significantly improves. Actually, we obtain a constant overhead for the joining/leaving entity.

6.3.2. 2nd set of experiments

In the second set of experiments we used various configurations where the maximum numbers of context users and sources ranged from 16 to 128. During each one of the experiments, context sources arbitrarily joined and leaved the environment. The numbers of the joining and leaving sources were randomly generated with the constraint that they never exceeded the 20% of the maximum number of context sources assumed. The context users arbitrarily queried the context sources that were available. For each configuration of context users and sources we examined two different scenarios. In the first scenario, the discovery of context sources could be performed with any of the three policies described in Section 5.2. In the second scenario, we assumed that only the Update-On-Demand and the Periodic-Update policies were used. Moreover, the validity checks were disabled for these policies.



(a) 1st scenario.



(b) 2nd scenario.

Fig. 14. Performance results — 2nd set of experiments.

Fig. 14(a) and (b) summarize the results that we obtained for the aforementioned scenarios. Specifically, Fig. 14(a) and (b) give the average execution times spent by the users for discovering and querying context sources and the average execution times spent by the sources for joining and leaving the environment. As expected, the observations made in the first set of experiments are still valid. The average execution times increase linearly with the size of each configuration. The average execution times that concern the discovery of context sources in the second scenario are much better than the ones obtained in the case of the first scenario due to the fact that the validity checks were disabled. Interestingly, the average execution times of the queries in the second scenario are slightly worse compared to ones obtained in the first scenario. The previous observation is the price to pay for not using the Always-Update policy and for disabling the validity checks in the Update-On-Demand and the Periodic-Update policies; certain context users performed useless Web service invocations (which resulted in corresponding exceptions) on context sources that were no longer available in the environment.

Summarizing the results of the 1st and the 2nd sets of experiments, we can reach the following conclusions. The performance of CoWSAMI scales gracefully with respect to the number of mobile users and context sources that get dynamically involved. Moreover, the choices of (1) not using the Always-Update policy and (2) disabling the validity checks performed by the Update-On-Demand and the Periodic-Update policies, significantly improve the performance of the context sources discovery process, while slightly slowing down the performance of the context gathering process. On the other hand, (1) using the Always-Update policy and (2) enabling the validity checks, optimize the context gathering process with a higher expense for the performance of the discovery process. Balancing the trade-off that relates to the configuration of the previous choices is a solution to the issue of meeting the efficiency requirements of the users of a particular AmI environment. Tuning the aforementioned technical choices towards coping with the performance issues of CoWSAMI is a responsibility of the CoWSAMI developers' target group. Nevertheless, we further consider solutions that would allow the users themselves to tune the performance of CoWSAMI according to their efficiency requirements. Specifically, in [24] we proposed SQL^p, an extension to SQL that allows users to query ad hoc communities of services, while setting up various attributes, such as the maximum tolerable query execution time and the maximum number of services to be contacted. Integrating an SQL^p interpreter in CoWSAMI is under consideration in the CoWSAMI future work agenda.

7. Related work

Various infrastructure-based approaches to context-aware computing have been proposed in the past. In this section, we discuss the main features of the prominent ones.

In [25] the authors presented CASS, a middleware infrastructure for context-aware mobile applications. In CASS context aggregation was realized by a context server deployed on a stationary workstation. Context information is stored in a database and SQL queries can be used for manipulating this information. This aspect bears some similarity with our approach where we model context in terms of relations. CASS further provides a rule-based interpreter, which is also deployed on the context server. In CoBrA [26] the context aggregation and interpretation elements are also deployed on a shared server, called context broker. In Hydrogen [10], context aggregation and interpretation take place on a context management element, which can be shared by more than one application that execute on a mobile device.

With regard to open AmI environments, the aforementioned infrastructures do not satisfy the requirement for dynamic context sources discovery as they do not provide such a support. In comparison with the aforementioned infrastructures, CoWSAMI further aims at autonomous context aggregation and interpretation. For that reason, any CoWSAMI entity may comprise and customize its own services for context aggregation and interpretation. We consider this approach to be more suitable for open AmI environments. However, CoWSAMI may also be used for shared context aggregation and interpretation. The fact that the CoWSAMI aggregation and interpretation mechanisms were build as Web services makes it possible to deploy them in a central entity that plays the role of a shared context server.

MobiPADS [27] is another interesting approach to context awareness. Context aggregation relies on event composition, while context interpretation is based on event-condition-action rules. In CORTEX [12,13], context aggregation relies on an hierarchical model of attributes and context interpretation is based on event-condition-action rules.

In CARISMA [28] the main focus is on context interpretation in the presence of conflicts. The authors proposed an interesting approach that deals with this issue based on a microeconomic conflict resolution mechanism. In Context Toolkit [7] the authors discussed the need for a discovery service that allows locating available context sources. In line with the previous one, SOCAM [29] provides dynamic context sources discovery, based on an ontological approach which has been further employed for context aggregation and interpretation. Specifically, a tree-structured configuration of SLS (Service Location Service) servers is employed. In Gaia [11] context aggregation and interpretation rely on first order logic and boolean algebra [30]. Gaia further provides a discovery mechanism that senses the presence of both digital and physical entities.

In comparison with the previous infrastructures that account for the requirement for dynamic context sources discovery, CoWSAMI follows a completely distributed approach that is suitable even for open AmI environments that are formulated in a purely ad hoc manner; in such cases it is not possible to assume the existence of a central discovery server (or a fixed configuration of discovery servers such as the one used in SOCAM). Moreover, the requirement for loose coupling and customization is not taken into consideration by any of the infrastructures discussed in this section. In all cases, infrastructure-specific abstractions must be developed to enable the integration of context sources with the rest of the middleware elements. As opposed to the previous one, CoWSAMI only requires that the context sources offer interfaces that conform to a widely accepted standard (i.e. the Web service architecture).

In a sense, the CoWSAMI perception of open AmI environments is closer to the one employed in the LAICA [31] and the PICO [32] projects. The role of the middleware in LAICA comprises knowing the location of required data and services and translating data provided by heterogeneous agents. PICO aims at the creation of mission-oriented communities of autonomous entities that perform tasks on behalf of the users. However, even in LAICA and PICO, the issues of loose coupling and customization are not taken into consideration.

8. Conclusion

In this paper we proposed CoWSAMI, a middleware infrastructure that allows context awareness in open AmI environments consisting of mobile users and context sources that become dynamically available. CoWSAMI allows the mapping of externally gathered context information to the internal, user perception of context. CoWSAMI is interface-aware in the sense that it gathers context by tailoring its behavior with respect to the aforementioned mapping. Moreover, CoWSAMI facilitates the discovery of context sources based on a completely distributed mechanism that allows the maintenance of context information based on multiple policies. Context information can be subsequently exploited through easy-to-use SQL-based facilities. We have experimented with CoWSAMI and our findings indicate that (1) it requires reasonable amounts of memory resources for its operation and (2) its scale-up with respect to the number of involved context users and sources is graceful. Finally, CoWSAMI provides the developers of AmI environments with a means for balancing the performance trade-off that involves the context sources discovery and the context gathering processes.

CoWSAMI takes us one step closer to the realization of open AmI scenarios. However, several other challenging issues are also involved to this end. Specifically, a commonly agreed ontology that models the different categories of context sources involved in a particular AmI environment can be used as an intermediate representation for simplifying the definitions of context relations and context rules. Up until now, such an ontology has only been considered for the discovery of different categories of context sources. The users' interaction with CoWSAMI can be further improved through pre-defined context relations, rules and queries. Moreover, we investigate solutions that would allow the users to tune the performance of CoWSAMI according to their particular QoS requirements [24]. We particularly focus on the quality dimensions of reliability, performance and trust and we plan to introduce in CoWSAMI, previous work that we had already performed in these fields [33–36].

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. We would also like to thank K. Georgoulas for his contribution to CoWSAMI.

References

[1] E. Aarts, R. Harwig, M. Schuurmans, Ambient intelligence, in: The Invisible Future: The Seamless Integration of Technology into Everyday Life, McGraw-Hill, 2001, pp. 235–250 (Chapter).

[2] E. Aarts, R. Roovers, IC design for ambient intelligence, in: Proceedings of the 6th IEEE Design Automation and Test in Europe Conference and Exhibition, DATE'03, 2003, pp. 2–7.

[3] W. Weber, C. Braun, R. Glaser, Y. Gsottberger, M. Halik, S. Jung, H. Klauk, C. Lauterbach, G. Schmid, X. Shi, T.F. Sturm, G. Stromberg, U. Zschieschang, Ambient intelligence — key technologies in the information age, in: Proceedings of the IEEE International Electron Devices Meeting, IEDM'03, 2003, pp. 1.1.1–1.1.8.

[4] M. Weiser, The computer of the twenty-first century, Scientific American 135 (3) (1991) 94–104.

[5] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Personal Communications 8 (4) (2001) 10–17.

[6] A.K. Dey, Understanding and using context, Personal and Ubiquitous Computing 5 (1) (2001) 4–7.

[7] A.K. Dey, G.D. Abowd, A Context-based Infrastructure for Smart Environments, in: Proceedings of the International Workshop on Managing Interactions in Smart Environments, MANSE '99, 1999, pp. 114–128.

[8] M. Baldauf, S. Dustdar, F. Rosenberg, A survey on context aware systems, Journal of Ad-Hoc and Ubiquitous Computing 2 (4) (2007) 263–277.

[9] IST Advisory Group (ISTAG). Software technologies, embedded systems and distributed systems — A European strategy towards ambient intelligent environment. Technical Report, IST, 2002. http://www.cordis.lu/ist/istag.html.

[10] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, Context-awareness on mobile devices — the hydrogen approach, in: Proceedings of 36th IEEE Hawaii International Conference on System Sciences, HICSS'02, 2002, pp. 292–302.

[11] M. Roman, C.K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, K. Nahrstedt, Gaia: A middleware infrastructure to enable active spaces, IEEE Pervasive Computing 1 (4) (2002) 74–83.

[12] G. Biegel, V. Cahill, A framework for developing mobile, context-aware applications, in: Proceedings of 2nd IEEE Conference on Pervasive Computing and Communications, Percom'04, 2004.

[13] T. Sivaharan, G. Blair, A. Friday, M. Wu, H. Duran-Limon, P. Okanda, C-F. Sorensen, Cooperating sentient vehicles for next generation automobiles, in: Proceedings of ACM MobiSys International Workshop on Applications of Mobile Embedded Systems, 2004.

[14] V. Issarny, D. Sacchetti, F. Tartanoglou, F. Sailhan, R. Chibout, N. Levy, A. Talamona, Developing ambient intelligence systems: A solution based on web services, Journal of Automated Software Engineering 12 (1) (2005) 101–137.

[15] W3C. Web Services Architecture, Technical Report, W3C, 2004. http://www.w3.org/TR/ws-arch/.

[16] A. Zarras, P. Vassiliadis, E. Pitoura, Query management over ad-hoc communities of web services, in: Proceedings of the 2nd IEEE International Conference on Pervasive Services, ICPS'05, 2005, pp. 261–270.

[17] I. Berger, Standards for car talk, The Institute 31 (1) (2007) 1, 6.

[18] OZONE Consortium. New Technologies and Services for Emerging Nomadic Societies. Technical Report, IST, 2002. http://www.extra.research.philips.com/euprojects/ozone/.

[19] Z. Plitsis, I. Fudos, E. Pitoura, A. Zarras, On accessing GSM-enabled mobile sensors, in: Proceedings of the 2nd IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing, ISSNIP'05, 2005, pp. 223–228 (in press).

[20] W3C. Web Services Description Language (WSDL) v2.0, Technical Report, W3C, 2005. http://www.w3.org/TR/wsdl20/.

[21] E. Guttman, C. Perkins, J. Veizades, Service location protocol, Version 2. Technical Report, Network Working Group, 1999. http://www.ietf.org/rfc/rfc2608.txt.

[22] ORACLE. Oracle Database Lite In Depth, Technical Report, ORACLE, 2004. http://www.oracle.com/technology/products/lite/lite_technical_InDepth.pdf.

[23] J.S. Karlsson, A. Lal, C. Leung, T. Pham, IBM DB2 everyplace: A small footprint relational database system, in: Proceedings of the 17th IEEE International Conference on Data Engineering, ICDE'01, 2001, pp. 230–232.

[24] N. Folinas, P. Vassiliadis, E. Pitoura, E. Papapetrou, A. Zarras, Context-aware query processing in ad-hoc environments of peers, Journal of Electronic Commerce in Organizations (JECO) 6 (1) (2007) 62–88.

[25] P. Fahy, S. Clarke, CASS — Middleware for mobile context-aware applications, in: Proceedings of the 2nd ACM SIGMOBILE International Conference on Mobile Systems, Applications and Services, MobiSys'04, 2004.

[26] H. Chen, T. Finin, A. Joshi, L. Kagal, F. Perich, D. Chakraborty, Intelligent agents meet the semantic web in smart spaces, IEEE Internet Computing (11) (2004) 2–12.

[27] A.T. Chan, S.-N. Chuang, MobiPADS: A reflective middleware for context-aware mobile computing, IEEE Transactions on Software Engineering 29 (10) (2003) 1072–1085.

[28] L. Capra, W. Emmerich, C. Mascolo, CARISMA: Context — Aware reflective middleware system for mobile applications, IEEE Transactions on Software Engineering 29 (10) (2003) 929–945.

[29] T. Gu, H.-K. Pung, D.-Q. Zhang, A service-oriented middleware for building context-aware services, Journal of Network and Computer Applications 28 (2005) 1–18.

[30] A. Ranganathan, R.H. Campbell, An infrastructure for context-awareness based on first order logic, Pervasive and Ubiquitous Computing Journal 7 (6) (2003) 353–364.

[31] G. Cabri, L. Ferrari, F. Zambonelli, The LAICA project: agent-based middleware for urban ambient intelligence, in: Proceedings of the 14th IEEE Wetice Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, 2005.

[32] M. Kumar, B.A. Shirazi, S.K. Das, B.Y. Sung, D. Levine, M. Singhal, PICO: A middleware framework for pervasive computing, IEEE Pervasive Computing 2 (3) (2003) 72–79.

[33] F. Sailhan, V. Issarny, Cooperative cashing in ad-hoc networks, in: Proceedings of the 4th IEEE International Conference on Mobile Data Management, MDM'03, 2003, pp. 13–28.

[34] J. Liu, V. Issarny, QoS-aware service location in mobile ad-hoc networks, in: Proceedings of the 5th IEEE International Conference on Mobile Data Management, MDM'04, 2004, pp. 224–235.

[35] F. Papadopoulos, A. Zarras, E. Pitoura, P. Vassiliadis, Timely provisioning of mobile services in critical pervasive environments, in: R. Meersman, Z. Tari, et al. (Eds.), Proceedings of the 7th International Symposium on Distributed Objects and Applications, DOA'2005, in: LNCS, vol. 3760, 2005, pp. 864–881.

[36] J. Liu, V. Issarny, An incentive compatible reputation mechanism for ubiquitous computing environments, International Journal of Information Security 6 (5) (2007) 297–311.

Dionysis Athanasopoulos received his B.Sc. in Computer Science in 2005 from the Computer Science Department, University of Ioannina. From November 2005 till now, he is a M.Sc. degree student at the same Department. His interests include Middleware, Distributed Systems and Pervasive Computing.

Dr. Apostolos V. Zarras received his B.Sc. in Computer Science in 1994 from the Computer Science Department, University of Crete. From the same department he received his M.Sc. in Distributed Systems and Computer Architecture. In 1999 he received his Ph.D. in Distributed Systems and Software Architecture from the University of Rennes I. From 2004 till now he holds a lecturer position at the Department of Computer Science of the University of Ioannina. Apostolos Zarras has to his credit over 20 papers in international conferences, journals and magazines. He is currently a member of the IEEE computer society. His research interests include middleware, model-driven architecture development, quality analysis of software systems and pervasive computing.

Dr. Valerie Issarny received her Ph.D. in computer science from the Université de Rennes 1, in November 1991, where she proposed an exception handling model for concurrent programming. During her Ph.D., she was a member of the LSP (Languages space operating system). From 1993 to 2001, she was an INRIA researcher in the Solidor group, examining solutions to the construction of robust and efficient distributed systems. Since 2002, she has been a Research Director, leading the ARLES group at INRIA Rocquencourt., which investigates solutions to the architecture-based development of ambient intelligence environments.

Dr. Evaggelia Pitoura received her B.Sc. from the University of Patras, Greece in 1990 and her M.Sc. and Ph.D. in computer science from Purdue University in 1993 and 1995, respectively. Since June 2005, she has been an associate professor at the Department of Computer Science of the University of Ioannina, Greece where she leads the distributed data management group. Her publications include more than 70 articles in international journals and conferences and a book on mobile computing. She has also co-authored two tutorials on mobile computing for IEEE ICDE 2000 and 2003. She is the recipient of the best paper award of IEEE ICDE 1999 and two “Recognition of Service Awards” of ACM.

Dr. Panos Vassiliadis obtained his Diploma in Electrical Engineering and his Ph.D. from the Department of Electrical and Computer Engineering, of the National Technical University of Athens (NTUA) in 1995 and 2000, respectively. He joined the Department of Computer Science of the University of Ioannina in 2002. So far, Dr. Vassiliadis’ research has focused on Data Warehouse technology, with particular interest on issues like data warehouse metadata repositories and metadata modeling, data warehouse quality, On-Line Analytical Processing (OLAP), and Extraction-Transformation-Loading (ETL). Apart from the data warehousing field, his on-going research includes topics like (a) the modeling of the architecture & evolution of databases & information systems, (b) web services and (c) pattern management (mainly referring to patterns that are the results of data mining algorithms).