University of Peloponnese

Faculty of Science and technology

Department of Computer Science and Technology

Software and Database Systems Laboratory

The Architecture for Context Management of m-Commerce Applications

Technical Report TR-SSDBL-11-001

Poulcheria Benou, Costas Vassilakis
pbenou@ethnodata.gr, costas@uop.gr

December 2011

Tripoli, Greece

Abstract

This document describes the architectural design and the detailed module design of a context management system for m-commerce applications.

1 Introduction – Requirements and Overall Architecture

In order to determine the context information needs of an m-commerce application a relevant methodology for extracting these needs should be employed. Benou and Vassilakis [1] have already proposed such a methodology. Since the context information of an m-commerce application has been identified (through Benou and Vassilakis' methodology or any other suitable methodology) and properly modeled (through extended UML class diagrams), the next step for the realization of a context-aware application is the designing of the subsystem that will manage the context.
The process of designing the system that will manage context information is common to all context-aware mobile commerce applications (CAMCA). Despite the fact that the context that different CAMCAs manage can be quite diverse, a well-defined context management architecture with standardized interfaces between its components and towards its clients, additionally being extensible in terms of (a) the context factors it is able to manage and (b) methods for context acquisition, may practically be used to support the context management requirements of any CAMCA. Such a standardized architecture will constitute a useful tool for speeding up the development of context-aware applications [2] and minimizing the probability for errors or omissions; furthermore, it will increase the potential for reusability, since context components developed for some application will be able to be incorporated in other applications with few or no changes.

Both the international practice and the state-of-the-art [3][4][5] in the areas of pervasive and ubiquitous computing indicate that a context information management subsystem should be able to:

- *capture* context information from its sources, which are physical and logical sensors, as well as the users. This includes the discovery of the context information sources within its vicinity.
- *store* context information or parts of it, so that it can be exploited in subsequent situations.
- *interpret* the context to a higher level of abstraction, which will be more meaningful (and useful) to the application that will use it. As an example, we can consider the interpretation of a *(longitude, latitude)* pair to a representation of the form *"home," "office"* or *"shopping mall"*.
- *transit* the context information to the application that will use it. Transition should be supported in two modes, i.e. with the initiative being either on the application (request/response or pull paradigm) or on the context management system (pub/sub or push paradigm), since both these modes are considered useful in CAMCAs [6].

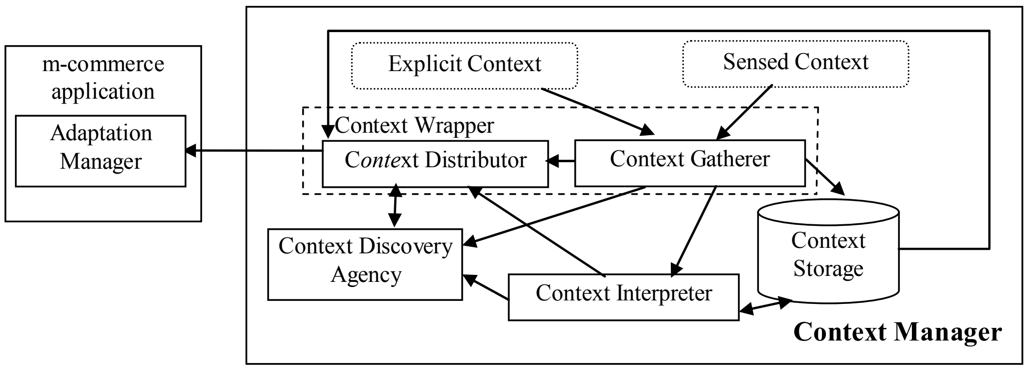

Fig 1 The Context Manager

In accordance with the above requirements, we will present below the design of the *Context Manager* module (Fig 1), which is further decomposed into the following components: i) the *Context Gatherer, ii)* the *Context Interpreter, iii)* the *Context Storage, iv)* the *Context Distributor and iv)* the *Context Discovery Agency.* The *Context Gatherer* is responsible for gathering the context information from the various sources of the application environment. The *Context Interpreter* is responsible for interpreting the context information to a higher level of abstraction. The *Context Storage* is responsible for storing the context information for subsequent use. The *Context Distributor* is responsible for distributing the context information to the applications that need it. The *Context Discovery Agency* is responsible for discovering the context information that can be made available to the interested parties. The interested parties are essentially the components responsible for performing adaptation within various information systems (frequently termed as *adaptation managers*); these components will use the information provided by the Context Manager to perform the adaptation of the application they provide.

In the following subsections we will describe in detail each of the functional components that comprise the Context Manager, as well as the interactions between these components and between the context manager and the adaptation manager component of the m-commerce applications requesting its services.

1.1 Context Wrappers: Gathering and Distributing Context

The Context Gatherer is the subsystem which is responsible for collecting the context information from its sources. Context information may be gathered from *physical sensors* (e.g. location sensors such as GPS, identification sensors such as smartcard or fingerprint readers, motion sensors, etc) [7] or from *logical sensors* (e.g. APIs provided by the operating systems which allow the retrieval of information regarding the processing power, the available software and hardware components, the current time and so forth). Logical sensors include the software modules that retrieve information from the main application database, (e.g. which user is currently logged in, which has been his/her observed behavior up to now, etc). An additional source of context information is the *user*, who is the source of explicitly provided context information, (i.e. information directly entered by the user, such as gender, date of birth and so on; some of this information may, of course, be stored into the main application database and from then on extracted from there). Depending on the source of the context information (physical sensors, logical sensors or users), the mechanisms that will capture it will be designed.

Physical sensors typically react to some environmental stimulus and generate numerical outputs which can be retrieved using low-level, device-specific protocols. *Logical sensors* are realized through software APIs, which the interested party may invoke to obtain the desired context information; logical sensors may read the context information values from a single physical sensor or combine values from multiple physical sensors [8]. Context information is made available from logical sensors either through a periodic monitoring process (*polling*) or through an available notification mechanism (e.g. an operating system API which provides notifications when additional storage space is made available). Finally, *user information sensors* - i.e. sensors delivering context information provided by the user (explicitly provided context information, e.g. information about the age or the likings of the user) - is not retrieved through sensing mechanisms, but is made available through graphical interfaces or through *information integration procedures* (e.g. parsing and processing of XML files, retrieval of information from smart cards and so forth).

Direct incorporation of sensor-dependent code data into applications, usually necessitates low-level coding and leads to tightly-coupled applications with low portability and components with limited reusability [9]. Therefore, in order to decouple the applications from the details of the sensing process, we adopt the *context wrapper* approach, i.e. we introduce a software module that undertakes the responsibility of reading context information from its source, encapsulating the peculiarities and idiosyncrasies of the particular context source and making the context information available for exploitation through a standardized interface, common for all kinds of context information. Fig 2 illustrates the concept of the context wrapper through a UML diagram. Naturally, context wrappers will include source-dependent software, therefore a distinct context wrapper is required for each different context source; the presence of the context wrapper, however, enables us to handle introductions of new context sources or modifications of existing ones by correspondingly creating a new context wrapper or modifying the existing one, leaving the rest of the CAMCA and the context manager system intact.

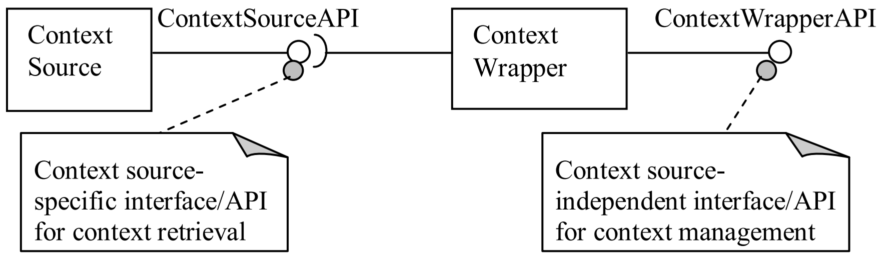
Context Source
ContextSourceAPI
Context Wrapper
ContextWrapperAPI
Context source-specific interface/API for context retrieval
Context source-independent interface/API for context management

Fig 2 A Context Wrapper

As part of their internal operation, context wrappers may *cache* the last value obtained from the managed sensor in local memory to speed up the processing of the requests posed to them.

Regarding their cooperation with other components, context wrappers provide the following functionalities:

1. they allow external entities, (e.g. adaptation managers of CAMCAs), to retrieve the values produced by the context source they manage, thus implementing the *pull* paradigm. As a response to such queries, the wrapper may probe the context source for a new value, use the last one retrieved from the context source and cached, if it is deemed valid or even retrieve a value previously stored in the context store.
2. they allow external entities to *subscribe* to notifications provided by the wrapper. These notifications allow interested applications to be informed about changes on the values of the context information sensed by some particular wrapper. They are sent whenever a subscriber-specified condition is met – e.g. for a wrapper managing a GPS device, a relevant condition could be "the location has changed by 200m or more". The subscription mechanism effectively implements the pub/sub paradigm.
3. they store the values obtained in the context store for later usage.
4. they offer *reflection* capabilities, through which a context wrapper may be queried regarding the *context properties* it "measures" (e.g. user identity or user location; in the following, this information will be termed), which metadata are pertinent to each specific property (e.g. if a wrapper "measures"

temperature, an indication whether temperature is measured in Centigrade or Fahrenheit degrees) and the list of the notifications it provides (e.g. for a wrapper measuring temperature, “temperature increased,” “temperature dropped,” “temperature changed,” “temperature above threshold” and so forth).

5. they register themselves with the Context Information Discovery Agency. This registration allows the wrapper to be discovered by other software components (context information aggregation wrappers, adaptation managers, etc), as described in section 1.4, below. They also unregister themselves from the Context Information Discovery Agency when they cease their operation.
6. they enable their detection from the Discovery Agency, thus allowing the Context Information Discovery Agency to populate its context provider repository. This may be practically implemented by having the Discovery Agency periodically broadcast requests for the specific service and automatically register to its repository those context wrappers that will respond to the broadcast. These broadcasts also allow the Context Information Discovery Agency to determine which wrappers remain operational and which have ceased functioning.

According to the above list of offered functionalities, the context wrapper interface depicted in Fig 2 can be refined as shown in Fig 3.

Essentially, context wrappers implement the *context gatherer* and the *context distributor* of the architecture depicted in Fig 1, with the code liaising with the context source interface (cf. Fig 2, Fig 3) implementing the *context gatherer* and the code realizing the context source-independent interface/API (and more specifically the ContextQuery and ContextNotification interfaces of Fig 3) being the *context distributor*. More specifically, the ContextQuery and ContextNotification interfaces of Fig 3 implement the distribution of context information to interested parties, while interfaces ContextReflection, ContextDiscoverable and ContextDataStoreCom facilitate aspects of the context distributor’s operation in the overall architecture.

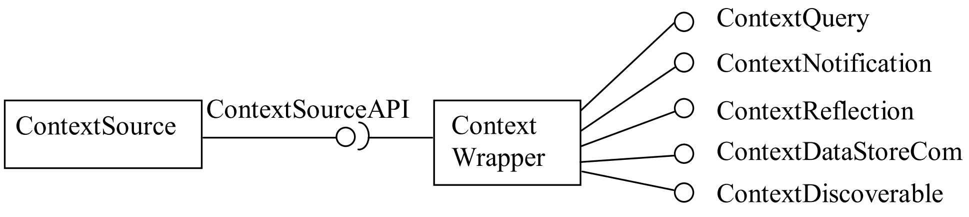

Fig 3 Refined Context Wrapper Interface

According to the design specification presented above, in order to define a context wrapper, its designer should specify the following:

i) its properties
ii) the metadata of its properties
iii) the notifications it provides
iv) the mechanisms through which the wrapper will obtain the values from the context source
v) the conditions under which the obtained values will be stored in the context store. This is needed to avoid over-populating the context store with unneeded values. For example, a GPS sensor may store the value in the context store, if it has changed more than 0.5 km from the last value stored in the context store or if no other value has been stored in the last hour.
vi) the algorithm through which the wrapper will decide if an incoming query will be honored by using a value from the local cache, from the context store or a fresh value obtained from the context source.

The details of the interfaces through which the wrappers communicate with external software entities (i.e. details on the request response dialogues and notification messages) are described in section 1.2 below. We must note here that the design presented above directly supports configurations where the context wrapper is not located on the same machine as the context source it manages. This is important for cases where some sensor is an embedded device with limited CPU power, communication capabilities or increased needs for energy preservation. In such cases, the sensor only needs to make available the data using a prominent mode (e.g. through an RS-232 connection or via Bluetooth), while the context wrapper will run on suitable hardware and undertake the tasks of context information gathering and distribution.

A context wrapper provides information originating from a particular context source, i.e. physical or logical sensor, or the user. In many cases, however, the information required for an entity (person, location or object) is essentially an aggregation of the data elements provided by multiple context information wrappers, which may also need to be combined with additional information from the context information store. Therefore, it is necessary to introduce software components that implement this form of aggregation and which are called *context information aggregators*. Their functionality is similar to that of context wrappers, in the sense that they can respond to queries, produce notifications and store the context information they acquire. These software components can in turn *query* or *subscribe to* other context information wrappers so as to obtain the elements of context information they are interested in. Furthermore, they can retrieve information from the context information store, which may be used together with the data obtained through queries or incoming notifications to produce aggregated context information; the latter will be made available to interested parties for further perusal. Context aggregators are similar to logical sensors, differing only in the aspect that context information is retrieved from context wrappers instead of context source-dependent APIs.

1.2 The Context Information Distributor: Interface Details and Messages Exchanged

The context distributor (i.e. the ContextQuery and ContextNotification interfaces of the context wrapper) undertakes the task of making the context information available to the interested parties (notably the adaptation managers of CACMs) in a standardized and uniform manner. More specifically, it allows for the distribution of the context information according to both the *request-response* and the *event-triggered* paradigm [10], corresponding to the "pull" and "push" context information distribution [6]. According to the *request-response* (pull) paradigm, context information is given as a response to explicit requests, while according to the *event-triggered* (push) paradigm, the context distributor arranges for sending context information to *subscribers* when certain events occur. The context information distributor is implemented through the query and notification mechanisms built in the context information wrappers and realized by the ContextQuery and ContextNotification interfaces, respectively, while, as noted above, these interfaces are complemented with interfaces ContextReflection, ContextDiscoverable and ContextDataStoreCom, with the latter three facilitate aspects of the context distributor's operation in the overall architecture. The query mechanism serves the need for *on-demand* provision of context information, with the initiative being on the side of the interested application. The notification mechanism (also referred to as publish/subscribe) is suitable for repeating requests for context information where the interested application merely states the conditions under which it wishes to be notified of changes regarding the context information values. Under this scheme, the context consumer (i.e. the adaptation manager module of an m-commerce application) needs to be coded in a manner that can asynchronously receive and process incoming notification messages. Context information wrappers implement both the query and the notification mechanisms through interfaces that are uniform for all wrappers. Uniformity is a key requirement, since in this way applications may easily communicate with the wrappers, regardless of the wrapper implementation details.

1.2.1 The ContextQuery interface

The interface to the query mechanism has the form:

queryContext(timeSpecificaion, attributeList)

attributeList designates which attributes provided by the sensor are requested. This is required since context wrappers may be attached to context sources (physical sensors, logical sensors or users) that provide numerous attributes, only few of which are needed (e.g. a meteorological data sensor may provide information about temperature, humidity, etc., and we need only obtain information regarding temperature). Since *timeliness* is an important aspect of context information [11], the query mechanism allows the querying party to specify how "fresh" the context information is required to be through the *timeSpecification* designation. The available options for this designation are as follows:

- *QueryCurrent*: this specification instructs the wrapper to obtain a fresh value from the context source and return it. In some cases, obtaining a fresh value may not be pertinent (e.g. the name/surname of a user is not bound to change) and then the wrapper simply returns an appropriate value (as in the *Query Any* case).
- *QueryRecent*: this specification instructs the wrapper to *either* (a) obtain a fresh value from the context source and return it *or* (b) return the last value it has already read from the context source and cached into its local memory. The wrapper should thus implement an algorithm for deciding which the optimal choice is.
- *QueryAny*: this specification instructs the wrapper to *either* (a) obtain a fresh value from the sensor and return it *or* (b) return the last value it has already read from the sensor and cached into its local memory *or* (c) retrieve a value from the context information store and return it. Similarly to the *QueryRecent* specification, the wrapper should implement an algorithm for deciding which the optimal choice is.

In all types of requests described above, the client defines to the wrapper the attributes it requires and the wrapper returns an appropriate reply, such as the one depicted in Fig 4. This scheme decouples the querying mechanism from the context value obtainment implementation details, (e.g. interfacing to an RFID scanner, a floor sensor or a video image processor to detect the presence of an individual) and thus allows the application to be designed independently of the actual implementation of the sensing devices.

```
<ContextItem>
        <ContextAttributeName>Temperature</ContextAttributeName>
        <value>24.8</value>
        <metadata>
                <units>CelciusDegrees</units>
                <lastSensedTime>2010-04-08 12:32:11 EET</lastSensedTime>
        </metadata>
</ContextItem>
```

Fig 4 Reply to a queryContext request

1.2.2 The ContextNotification interface

The notification mechanism of context information wrappers is activated when the software component, which is interested in receiving notifications regarding a particular piece of context information, places a *subscription* for a notification produced by a context wrapper (flow 1 in Fig 5). Each such subscription is complemented with a *notification condition* which specifies the circumstances under which the particular subscriber wishes to receive notifications. Besides the current value of the context information element, the condition may refer to the previously observed value, useful for producing notifications when the change has exceeded a certain threshold (e.g. temperature – previousNotificationTemperature > 0.5); it may also refer to temporal information (e.g. produce a notification every hour, regardless of whether the value has changed) or to context information element metadata (e.g. check whether temperature is measured in Celsius or Fahrenheit degrees to set accordingly the notification threshold within the condition).

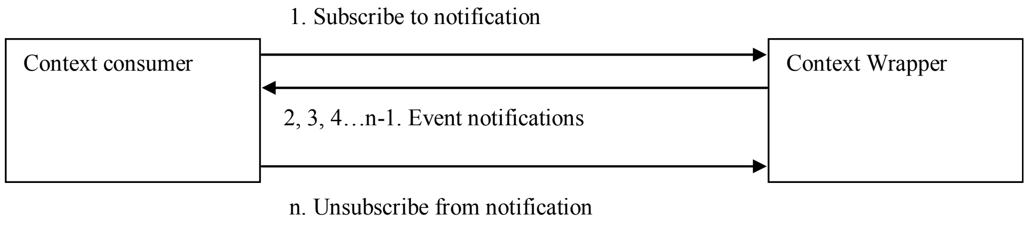
1. Subscribe to notification
Context consumer
Context Wrapper
2, 3, 4…n-1. Event notifications
n. Unsubscribe from notification

Fig 5. Publish/subscribe paradigm

Every time the wrapper detects that a notification condition is satisfied, it will send a notification to the consumer that has placed the relevant subscription (flows *2* to *n-1* in Fig 5). Finally, the context consumer may cancel its subscription through an *unsubscribe* request (flow *n* in Fig 5).

A notification service is therefore fully defined through the following elements: (i) the notification service name, (ii) the attributes it monitors and their types and (iii) the elements that can be used to form notification conditions, as well as the types of these attributes. According to this description, a notification service which relates to the user location may be as shown in Fig 6.

```
<Notification>
    <name>LocationUpdateNotification</name>
    <attributes>
        <attribute name=”location” type=”String”/>
        < attribute name=”identity” type=”integer”/>
    </attributes >
    <conditionElements>
        <conditionElement name=”location” type=”String”/>
        <conditionElement name=”previousNotificationLocation” type=”String”/>
        <conditionElement name=”identity” type=”integer”/>
        <conditionElement name=”currentTimestamp” type=”datetime”/>
        <conditionElement name=”previousNotificationTimestamp” ”type=”datetime”/>
    </conditionElements>
</Notification>
```

Fig 6 Example of location update notification

The interface of a notification service implements the publish-subscribe paradigm through the following operations [12]:

- *subscribe(n)*: allows a client to subscribe to notification *n*.
- *unsubscribe(n)*: allows a client to unsubscribe from a notification *n*, to which it has already subscribed.

- *advertise(n):* publicizes the availability of a notification type *n*, making it available for subscription to interested parties.
- *unadvertise(n):* revokes the publication of notification *n*'s availability, making it unavailable for further subscriptions.
- *sendNotification(n)*: checks the active subscriptions to notification *n* and sends the notification to all subscribers for which the respective condition evaluates to *true*.

When an interested party wants to register as a subscriber to a context information wrapper, it must specify (i) its identity (ii) and its location, i.e. the address at which notifications should be sent, (iii) the notification to which it subscribes, (iv) the attributes and the respective metadata which it wants to receive with each notification, and (v) the condition under which a notification should be sent to it. A simple condition includes (a) the property or the metadata name, (b) the comparison operator, and (c) the value against which the property or the metadata name will be compared. A condition may be either a simple condition or a number of conditions combined through logical operators (AND/OR/NOT). A condition specification may also involve arithmetic expressions. An example of a notification subscription condition is listed below:

```
<NotificationCondition>
        <attribute name=”identity”/>
        <operator comp=”=”/>
        <value val=”14”/> <!-- user id for user to be notified -->
</NotificationCondition>
```

Fig 7. Notification condition example

1.2.3 The ContextReflection interface

The ContextReflection interface allows context wrappers to be queried regarding the capabilities they offer and more specifically:

1. which context attributes it provides information on. For each attribute, a list of pertinent metadata is given, describing the attribute (e.g. a human-readable description), the value (e.g. units of measurement) and characteristics specific to the acquisition method (e.g. accuracy, period of value refreshment, minimum and maximum supported values). Context attributes may be queried through the *queryContextAttributes* method of the *ContextReflection* interface.
2. which notifications it publishes. For each notification, the information depicted in Fig 6 is returned. Notifications may be queried through the *queryNotifications* of the *ContextReflection* interface.

1.2.4 The ContextDataStoreCom interface

The *ContextDataStoreCom* interface includes all provisions for communicating with the data store for storing values obtained by the context source for further perusal or for querying already stored values when the algorithm employed by the *QueryAny* method indicates that such a value should be returned. In more detail, the ContextDataStoreCom interface encompasses the following methods:

1. *storeContextItemValue*, which accepts a context item value together with its respective metadata (cf. Fig 4) and stores it in the data store.
2. *retrieveContextItemValue*, which accepts a specification of the context item that needs to be retrieved (e.g. temperature, location etc) together with conditions on the attribute's value and/or metadata that it must hold (similarly to the *NotificationCondition* example in Fig 7). The method formulates and places the respective request to the context data store and returns the context data store's reply, which contains the attribute value and the respective metadata (cf. Fig 4).

The ContextDataStoreCom interface also encompasses methods for discovering the context data store and connecting to it. These methods are used internally by the *storeContextItemValue* and *retrieveContextItemValue* methods.

1.2.5 The ContextDiscoverable interface

The *ContextDiscoverable* interface allows for the context wrapper to be dynamically discovered by the respective modules within the context management architecture, and thus be subsequently used by interested context consumers. The *ContextDiscoverable* interface encompasses the following methods:

1. *registerToDiscoveryAgency*, which sends a message to the discovery agency announcing the existence of the context wrapper. The message contains information on the address at which the context wrapper can be reached and the attributes it provides. Further information on the attributes (the available metadata) and the notifications offered by the context wrapper may be obtained by any interested party by contacting the ContextReflection interface of the particular wrapper. The discovery agency should insert

the information into its repository and include information regarding the particular context wrapper into replies for queries requesting sources of the attributes that the context wrapper provides. The registration message may additionally contain information regarding the discovery characteristics of the context wrapper; for example, it may designate that a low frequency of *respondToContextConsumer* requests (see item 3, below) is desired to save energy and bandwidth.

2. *unregisterFromDiscoveryAgency*, which sends a message to the discovery agency, announcing that the particular context wrapper ceases its operation. The discovery agency should withdraw the information regarding the particular context wrapper from its repository and refrain from including information regarding the context wrapper into subsequent replies.
3. *respondToContextConsumer, which* supports the automated discovery of the context wrapper by the discovery agency. This method accepts and processes a null request (only the method name is specified) and responds to the message by a message containing only its address. With this information, the discovery agency may then examine if the context wrapper is already registered in its repository and if not, initiate a registration procedure through the *requestRegistration* method. The discovery agency may use the *respondToContextConsumer* request to detect context wrappers that have ceased their operation or are no longer reachable without having performed the respective unregistration.
4. *requestRegistration*. This method is called by the discovery agency to trigger the execution of the *registerToDiscoveryAgency* method. This method will be invoked by the discovery agency when it receives a response from a context wrapper that is not recorded into its registry.

1.3 The Context Interpreter

The context interpreter is the module that produces context information of higher level of abstraction, as opposed to context wrappers which only produce low-level context data. More specifically, it collects "primitive" information elements from the context distributor and the data store and applies to them inference procedures according to rules that have been defined. For instance, it may retrieve the GPS coordinates corresponding to the user's location to map it to a position on a specific road (e.g. "Motorway 5, 3rd kilometer") or determine if the user's location is "home," "office" or "on the move." The inference procedure may be performed using simple if/then rules or through more elaborate algorithms and techniques. The more widespread techniques involve ontology reasoning and machine learning. Ontology reasoning mainly comprises of producing new facts based on the already known facts and the information stored in the ontology in the form of classes, instances and relationships [13]. Machine learning techniques (e.g. Bayesian networks, decision trees) may be used for constructing static forecast models, which use low-level context information elements, in order to predict, for example, the user behaviour and automatically determine the user intention [14].

Context interpreters adhere to the context wrapper specifications. They *consume* context from context sources (context distributor and the data store) and make it available to other context consumers. However, since the input data is gathered from standardized sources, context interpreters' implementation may be greatly simplified since there is no need to write context-source specific code; instead, data gathering may be specified declaratively by simply listing the context sources some pertinent parameters (e.g. whether data will be retrieved according to the push or pull paradigm, what the polling frequency for the pull paradigm is).

According to the specification above, the full definition of a context interpreter includes (i) the information that will be interpreted (e.g. specific attributes) (ii) the context attributes that will be produced as output of the interpretation procedure and (iii) the procedure that will perform the interpretation and (iv) the notifications provided, if any.

Context interpreters implement the ContextQuery, ContextNotification, ContextReflection, ContextDataStoreCom and ContextDiscoverable interfaces, thus being ContextDistributor themselves and providing the services described in section 1.4.

1.4 The Context Information Discovery Agency

The context information discovery agency implements facilities for storing information about the context providers (context information wrappers, context information aggregators, context information interpreters), for locating them and for informing interested parties of how they can be contacted Additionally, it offers information about itself in order to be detectable from context providers. This functionality is accessible through the following operations:

- *addDiscoveredContextObject*: it adds information about a context information provider to the context information discovery agency's registry.
- *registerContextProvider*: this method intercepts the message sent by the *registerToDiscoveryAgency* operation of the context wrapper's ContextDiscoverable interface. As a response to receiving this message, the *registerContextProvider* method invokes the *queryContextAttributes* and *queryNotifications* operations of the context wrapper's ContextReflection Interface to gather information regarding the

attributes and notifications provided by the context wrapper and the pertinent metadata. When all this information has been collected, the *addDiscoveredContextObject* is invoked to insert the information in the agency's repository.

- *removeDiscoveredContextObject*: it removes the registered entry of a context information provider from the context information discovery agency's registry.
- *unregisterContextProvider*: this method intercepts the message sent by the *unregisterFromDiscoveryAgency* method of the context wrapper's *ContextDiscoverable* interface. As a response to receiving this message, the *unregisterContextProvider* method invokes the *removeDiscoveredContextObject* operation to remove the specific context provider from the agency's repository.
- *queryForDiscoveredContextObjects*: it allows interested parties to query the context information discovery agency about information regarding the context information providers in its registry.
- *respondToContextProvider*: this operation allows context information providers to locate the context information discovery agency (and subsequently register to it).

When a context information provider becomes active, it searches for the context information discovery agency (by broadcasting a query for the *respondToContextProvider* service) and then registers to it by invoking the *registerToDiscoveryAgency* operation. The details sent with the registration are (i) its ID, (ii) the address it can be reached at (e.g. if the communication is TCP/IP socket-based, the address will include the IP address and the port number), (iii) the attributes it provides and the related metadata and (iv) the notification services it offers. The context information discovery agency can itself invoke the *addDiscoveredContextObject* operation to register context information providers that have been discovered through a broadcast for the *respondToContextConsumer* service, which is implemented by context information providers.

When a context information provider terminates its operation, it should invoke the *unregisterFromDiscoveryAgency* operation of the context information discovery agency to remove itself from the context information discovery agency's registry. Context information providers may however terminate their operation abruptly (e.g. due to battery failure) and in these cases they cannot contact the context information discovery agency to perform the registry removal operation. In order to maintain its registry in an up-to-date state, the context information discovery agency periodically checks for the availability of the registered agents by broadcasting a *respondToContextConsumer* request and automatically unregisters context information providers that fail to respond to it.

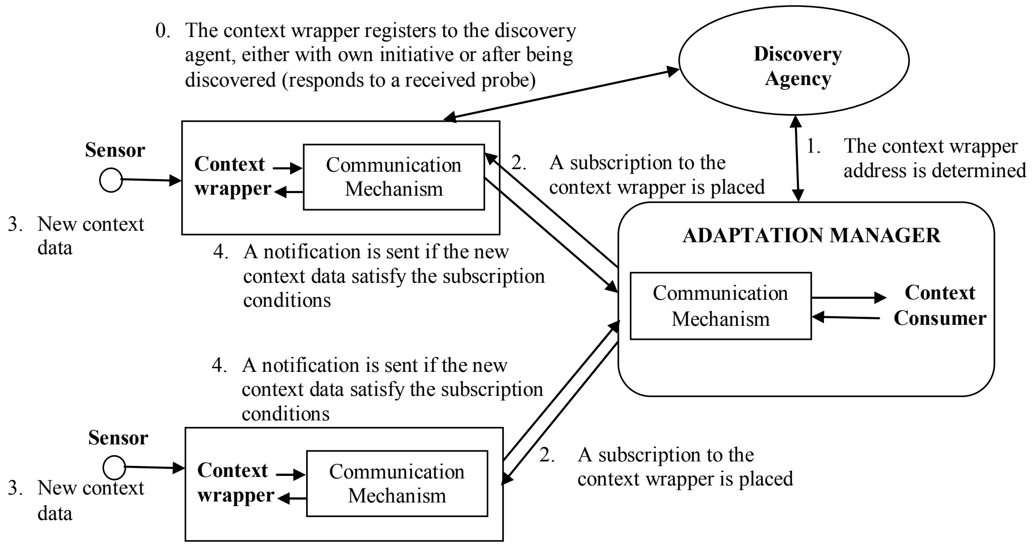
0. The context wrapper registers to the discovery agent, either with own initiative or after being discovered (responds to a received probe)
Discovery Agency
Sensor
Context wrapper
Communication Mechanism
2. A subscription to the context wrapper is placed
1. The context wrapper address is determined
3. New context data
ADAPTATION MANAGER
4. A notification is sent if the new context data satisfy the subscription conditions
Communication Mechanism
Context Consumer
4. A notification is sent if the new context data satisfy the subscription conditions
Sensor
Context wrapper
Communication Mechanism
2. A subscription to the context wrapper is placed
3. New context data

Fig 8 Example of an adaptation manager subscribing to multiple context wrappers

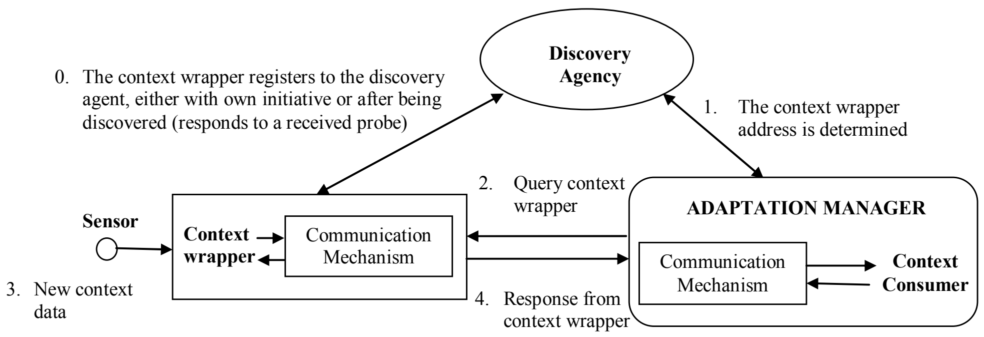
Discovery Agency
0. The context wrapper registers to the discovery agent, either with own initiative or after being discovered (responds to a received probe)
1. The context wrapper address is determined
2. Query context wrapper
Sensor
Context wrapper
Communication Mechanism
ADAPTATION MANAGER
Communication Mechanism
Context Consumer
3. New context data
4. Response from context wrapper

Fig 9 Example of an adaptation manager querying a context wrapper

Finally, context information consumers (adaptation managers, context information aggregators and context information interpreters) may invoke the *queryForDiscoveredContextObjects* interface of the context information discovery agency to locate the context information providers which make available some particular context information. Fig 8 illustrates the complete message sequence from the point that an adaptation manager of a context consumer queries the discovery agency for a context wrapper's address, up to the point that it receives the requested notifications (note that messages 3 and 4 may repeat multiple times). Fig 9 depicts the respective message sequence for the query/response paradigm (in this case, messages 2, 3 and 4 may be repeated multiple times).

1.5 The Context Information Store

The context information store allows for long-term storage of context information; this may be produced by any context information provider and once stored in the context information store may be later retrieved by context consumers. In this sense, the context information store plays the role of a buffer between context producers and context consumers, decoupling the context production from the context consumption time, while it also offers the potential to store large amounts of context data, which would be infeasible to do in other components. More specifically, the following uses are envisioned from the context information store:

1. context producers (context wrappers, information aggregators and context information interpreters) may store the information they gather from the sensors in the context information store and context consumers (adaptation managers, context information aggregators and context information interpreters) may retrieve it from there, in case the context producer is unreachable when the context data from it is required.
2. context wrappers may store the information they gather from the sensors in the context information store and retrieve it from there later to use it as a response to context queries, instead of querying the sensor again. This will be useful in a number of cases, such as the unavailability of the sensor (e.g. due to communication problems or battery failure), an attempt to implement a power-saving policy for the sensor, etc.
3. the context information store is a natural place to store large amounts of context information for performing tasks such as user behavior mining in order to optimize the CAMCA and/or deliver new services to the users.

The implementation details of the context information store, including storage format, policies for purging past information and query language (e.g. SQL or SPARQL) are beyond the scope of this paper.

1.6 Implementation Issues

Mobile commerce applications may be distinguished into three categories according to their architecture [15]. The first category includes applications that run exclusively on mobile devices and exchange data with a remote server (e.g. J2ME and Windows CE applications). The second category includes applications that run on some server and exchange only messages with the mobile device (typically SMS and MMS applications). The third category includes applications that run within a browser and exchange data with a remote server using a web protocol (HTTP, WAP, etc). According to Quah and Seet [16], the adaptation of these applications essentially comprises of taking into account the values of the context information elements to i) customize the data presented to the user (content adaptation) and/or ii) tailor the application's presentation properties (presentation adaptation) and/or iii) make the suitable modification of the application's functionality (functional adaptation). In order to achieve presentation and functional adaptation, context information must be available either when the application interface is generated (for browser-based applications or message-based applications) or at the location where the application is run (for "desktop-like" applications).

Regarding the first category of m-commerce applications (i.e. applications that run exclusively on the mobile devices), the interface is created at application development time, while the application is run later on the mobile device. On the contrary, for applications falling into the second and third category (message-based and browser-based, respectively), both the application interface generation and the application logic are hosted at the remote server and performed at run-time. Taking into account, however, the resource limitations of current mobile devices, the full-scale management and exploitation of context information at mobile device-side seems infeasible. Especially if numerous context information elements need to be taken into account and advanced interpretation techniques are required; the need for constantly updating the volatile elements of context information also implies increased communication costs and battery consumption, which are two additional deterring factors for adopting the mobile device-side adaptation. Therefore, the architecture presented here is primarily suitable for mobile applications of the second and third categories, where the remote server is mainly responsible for most tasks and the mobile device serves mostly as a presentation/user interaction apparatus. The proposed architecture can also be employed in applications falling in the first m-commerce application category, provided that the context information elements managed are few and the adaptation tasks do not require extensive resources.

It has to be noted here that context wrappers, which are responsible for capturing and delivering context information, *may* be hosted in mobile devices, in all three application categories. Thus, context information providers that supply information regarding the user (e.g. identity, location) or the mobile device (screen size, input capabilities etc) will naturally be accommodated in the mobile device. The mobile device may also host context information aggregators that capture data from context wrappers in its proximity (e.g. weather or traffic sensors). Context information from providers hosted in the mobile device will be transmitted to the central server, which will feed it accordingly to the relevant adaptation modules or deposit it in the context information store.

2 References

[1] Benou P., Vassilakis C. (2010) The Conceptual Model of Context for Mobile Commerce Applications. J ELECTRON COMM RES, Vol. 10, Vo. 2, pp. 130-165, Springer-Verlag.
[2] Henricksen, K., Indulska, J., McFadden, T., Balasubramaniam, S. (2005) Middleware for Distributed Context-Aware Systems. On the Move to Meaningful Internet Systems, Springer, LNCS 3760, pp. 846-863.
[3] Devaraju, A., Hoh S., Hartley M. (2007) A context gathering framework for context-aware mobile solutions .In Proceedings of the 4th international Conference on Mobile Technology, Applications, and Systems and the 1st international Symposium on Computer Human interaction in Mobile Technology, pp. 39-46.
[4] Di Zheng, Jun Wang, Yan Jia, Wei-Hong Han, Peng Zou, (2007) Middleware Based Context Management for the Component-Based Pervasive Computing. LECT NOTES COMPUT SC, Vol. 4610/2007, pp. 71-81.
[5] Kranenburg, H., Bargh, M.S., Iacob, S., Peddemors, A. (2006) A context management framework for supporting context-aware distributed applications. Communications Magazine IEEE, Vol. 44, Issue 8, pp. 67-74.
[6] Ceri, S., Daniel, F., Matera, M. (2007) Model-Driven Development of context-aware web applications. ACM Transactions of Internet Technology, Vol. 7, No. 1.
[7] Gellersen, H., Schmidt, A., Beigl, M. (2002) Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts. ACM J MOB NETW APPL, Vol. 7, No. 5, pp. 341 –351.
[8] Hoh, S., Devaraju, A., Wong, C. (2008) A Context Aware Framework for User Centered Services. 21st International Symposium Human Factors in Telecommunication.
[9] Kaikkonen, A., Kallio, T., Kekäläinen, A., Kankainen, A., Cankar, E. (2005) Usability Testing of Mobile Applications: A Comparison between Laboratory and Field Testing. Journal of Usability Studies, Issue 1, Vol. 1, pp. 4-16.
[10] Biegel, G., Cahill, V. (2004) A framework for developing mobile, context-aware applications. Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communication, pp.361–365.
[11] Benou, P., Bitos, V. (2008) Developing Mobile Commerce Applications. J ELECTRON COMM ORGAN , Vol. 6, No.1, pp. 63-78.
[12] Mühl, G., Fiege, L., Pietzuch, P. (2006) Distributed Event-Based Systems. Springer, 1st edition.
[13] Ye, J., Coyle, L., Dobson, S., Nixon, P. (2007) Ontology – based models in pervasive computing systems. The Knowledge Engineering Review, Vol. Issue 4, pp. 315-347.
[14] Frank, K., Rockl, M., Nadales, V., Robertson, P., Pfeifer, T. (2010) Comparison of exact static and dynamic Bayesian context inference methods for activity recognition. In Proceedings of Pervasive Computing and Communications Workshops, 2010 8th IEEE International Conference, pp. 189-195.