Reflective Middleware Solutions for Context-Aware Applications

Licia Capra, Wolfgang Emmerich and Cecilia Mascolo

Dept. of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK
{L.Capra|W.Emmerich|C.Mascolo}@cs.ucl.ac.uk

Abstract. The increasing popularity of wireless devices, such as mobile phones, personal digital assistants, watches and the like, is enabling new classes of applications that present challenging problems to designers. Applications have to be aware of, and adapt to, variations in the context of execution, such as fluctuating network bandwidth, decreasing battery power, changes in location or device capabilities, and so on. In this paper, we argue that middleware solutions for wired distributed systems cannot be used in a mobile setting, as the principle of transparency that has driven their design runs counter to the new degrees of awareness imposed by mobility. We propose the marriage of reflection and metadata as a means for middleware to give applications dynamic access to information about their execution context. We describe a conceptual model and an architecture that provide the basis of our reflective middleware. We demonstrate our ideas using a collaborative e-shopping case study that we developed with an industrial partner.

1 Introduction

Wireless devices, such as smartcards, mobile phones, personal digital assistants, watches and the like, are gaining wide popularity. These devices can be connected to wireless networks and software development kits are available that can be used by third parties to develop applications [?]. Applications on these types of devices, however, present challenging problems to designers. Devices face temporary loss of network connectivity when they move; they discover other hosts in an ad-hoc manner; they are likely to have scarce resources, such as low battery power, slow CPU speed and little memory; they are required to react to frequent and unannounced changes in the environment, such as high variability of network bandwidth, transfer to a different device, new location, etc.

When developing distributed applications, designers do not have to deal explicitly with problems related to distribution, such as heterogeneity, scalability, resource sharing, and the like. *Middleware* developed upon network operating systems provides application designers with a higher level of abstraction, hiding the complexity introduced by distribution. Existing middleware technologies, such as transaction-oriented, message-oriented or object-oriented middleware [?]

have been built adhering to the metaphor of the *black box*, i.e., distribution is hidden from both users and software engineers, so that the system appears as a single integrated computing facility. In other words, distribution becomes *transparent* [**?**].

These technologies have been designed and are successfully used for stationary distributed systems built with wired networks, but are they suitable for the mobile setting? The answer seems to be no. Firstly, the interaction primitives, such as distributed transactions, object requests or remote procedure calls, assume a high-bandwidth connection of the components, as well as their constant availability. In mobile systems, in contrast, unreachability and low bandwidth are the norm rather than an exception. Moreover, object-oriented middleware systems, such as CORBA [**?**], mainly support synchronous point-to-point communication with at-most-once semantics, while in a mobile environment it is often the case that client and server hosts are not connected at the same time. Secondly, and most notably, completely hiding the implementation details from the application becomes both more difficult and makes little sense. Mobile systems need to detect and adapt to drastic changes happening in the environment, such as changes in connectivity, bandwidth, battery power and the like. By providing transparency, the middleware must take decisions on behalf of the application. The application, however, can normally make more efficient and better quality decisions based on application-specific information. This is particularly important in mobile computing settings, where the 'context' (e.g., the location) of a device should be taken into account. We argue that the use of existing middleware technologies in mobile settings risks to penalize performance, to cause high operating costs, and may lead to unusability and non-scalable solutions [**?**].

In this paper, we propose the joint use of reflection [**?**] and metadata [**?**] in order to develop middleware targeted to mobile settings. Through metadata we obtain separation of concerns, that is, we distinguish what the middleware does from how the middleware does it. Reflection is the means that we provide to applications in order to inspect and adapt middleware metadata, that is, influence the way middleware behaves, according to the current context of execution. The principle of reflection, mainly investigated by the programming language community during the past years, is not new in the middleware community either, as shown by the many research projects in the field, such as OpenCorba [**?**], OpenORB [**?**], and dynamicTAO [**?**]. While they have mainly focused on reflective extensions to the basic mechanisms of CORBA, and therefore targeted to a wired environment, we exploit reflection to build a middleware for mobile computing settings instead.

The paper is organized as follows: in Section **??** we present an industrial case study that illustrates the inadequacy of current middleware solutions in a mobile setting and highlights the new requirements imposed by mobility. Section **??** introduces the basic concepts that have driven the design of our reflective middleware. Section **??** describes the reflective conceptual model in details and Section **??** illustrates how to map this model into a reflective architecture. In

Section **??** we discuss and evaluate our work and in Section **??** we conclude the paper and list future work.

2 A Case Study

In this section we present a collaborative electronic shopping system that we discussed, and partly developed, with Unipower, an industrial collaborator of our research group. Our goal is to show the inadequacy of the transparency principle and highlight the new requirements imposed by mobility.

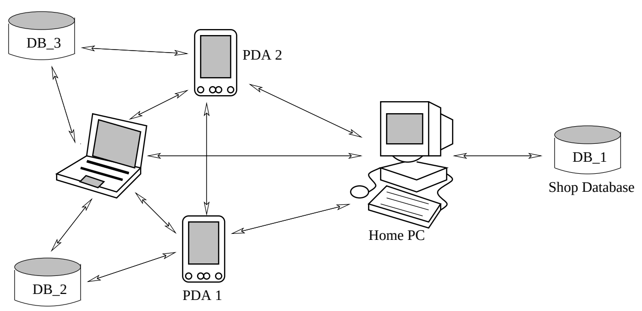

Fig. 1. Parties involved in the e-shopping system.

Fig. **??** illustrates our case study. Two main parties are involved: the shop, with its branches represented by the distributed database, and a set of customers, represented by PCs and mobile devices. We assume that each branch stores its product catalogue in its local database and makes it accessible to its customers for consultation at any time. As a possible set of customers we consider a couple, owning a PDA each and sharing a home PC. The couple does its weekly shopping electronically: they use the PC when they are at home and their PDAs while roaming around town.

There are two main operations that our application must support: consulting the product catalogue and determining an order. We now discuss the details of these operations.

Consulting the product catalogue. A PC and a PDA have different characteristics, especially in terms of battery lifetime, processing power, local memory and quality of the network connection. While it is feasible (at least in principle) for the client application running on the PC to connect to the network each time an access to a category or to a product in the catalogue is needed, this is not affordable in case of a PDA or a mobile phone, due to the high costs and low quality of the network connection. Even for a PC, going through the wire frequently decreases the quality of the service significantly, as a result of network latency.

In order to deal with this problem, the most frequently adopted solution is to replicate the product catalogue locally and update it whenever prices or the portfolio of the shop change. Even in this situation, PCs and PDAs exhibit rather different characteristics: while a PC generally has a huge amount of disk space that supports replication of the entire product catalogue, the PDA has a limited amount of memory, which inhibits replicating the full catalogue. Each time the mobile client application tries to access a product that has not been replicated locally, a request must be sent through the network to the PC or directly to a branch database, in order to gain access to it. To avoid performance degradation, we do not want to use these low quality links frequently, as they can be as slow as 9,600 bauds for GSM. Moreover, we would like to be able to continue working even when we are disconnected. It is therefore necessary to 'prefetch' the data the client wants to access. The question about *which* information should be replicated becomes then very important.

The replication policies adopted so far by many middleware systems include, among the others, LRU (Last Recently Used) data and MFU (Most Frequently Used) data: in all cases, it is the middleware that decides which information to replicate. If replication is transparent, the client application has no way to influence this choice, even though the application is the only one to know what it needs to access. We claim that, instead of letting the middleware guess what the client is going to need, the application has to be able to instruct the middleware on what it has to 'copy' locally and what it can instead access remotely ('link'). That is, we call for *replication awareness* instead of *replication transparency.*

Determining an order. Apart from the product catalogue, the application provides the abstraction of a shopping basket that is kept on the PC and is replicated on each PDA. The couple can do its weekly shopping independently of each other and synchronize the shopping basket only from time to time, or directly at the end of the week before submitting an order to the shop. As we want to submit a unique joint order, we need to reconcile the different (possibly conflicting) versions that the two family members have on their PDAs. This is not a trivial problem. Having two different shopping baskets does not necessary mean having conflicts to resolve (e.g., the parties may order products in different categories). Even when a conflict exists, there is no obvious way to solve it. We do not want the application designers to manually program the reconciliation task, as this is a general service necessary to a large set of applications. Instead, the middleware should enable the application to drive and tune the synchronization service provided by the middleware itself, using application-specific knowledge the middleware cannot have.

Once the shopping basket has been successfully reconciled, we may want to submit the order to the shop. This can be done from the PC at home, using a reliable and high bandwidth connection, or from a PDA while moving around, experiencing high variability in the quality of the network connection. The submission of the order is a critical operation; different protocols can be used for this purpose, for instance, two-phase commit or three-phase commit; while the second is the safest, it is also the heaviest. Once again, the middleware cannot opt for one protocol instead of the other transparently, by simply testing the condition of the network, as this decision strictly depends on the non-functional requirements of the application (e.g., reliability, performance, etc.).

Finally, while moving, it may happen that we want to submit an order and collect the products purchased within short time directly from the nearest branch of the shop. This could be done in a transparent way, with all the orders from all different customers being sent to a central database specially designed for this purpose, and, from there, redirected to the customer's nearest branch. In the presence of physical mobility, however, the nearest shop cannot be statically computed but needs to be determined at the time the user sends the order. The central database can then become a bottleneck, causing many delays. Moreover, we may not be able to connect to the central database due to a network failure, while still being able to reach one of its branches. A better solution would be to make the application aware of its location, locate the nearest shop, and then send the order there. In other words, the application needs to know its own physical location as well as the one of the component (the branch) providing the service it is looking for. Another foundation of traditional middleware systems, that is *location transparency*, may have to be replaced with *location awareness*.

As the example has highlighted, middleware solutions built adhering to the black-box metaphor are inadequate when applied to the mobile setting, as they do not allow applications to be aware of their execution context and to tune the behaviour of the middleware accordingly. In order to promote reusable and principled solutions, as opposed to ad-hoc ones, we now need to step back and redesign, keeping in mind the new requirements imposed by mobility.

3 Principles of Reflective Middleware

In this section, we abstract away from the particular example previously described and we introduce the basic principles that have driven the design of our reflective middleware.

Applications running on a mobile device need to be aware of their execution context. By context, we mean everything that can influence the behaviour of an application. Under this general term, we can identify two more specific levels of awareness, already encountered in our case study: *device awareness* and *environment awareness*. Device awareness refers to everything that resides on the physical device the application is running on; for example, memory, battery power, screen size, processing power and so on. We call these entities *internal resources*. Environment awareness refers to everything that is outside the physical device, that is bandwidth, network connection, location, other hosts (or services) in reach, and so on. We call these entities *external resources*.

On one hand, being aware of the execution context requires the designer to know, for instance, the location of the device, the hosts in reach, and, in general, any piece of information that is collected from the network operating system.

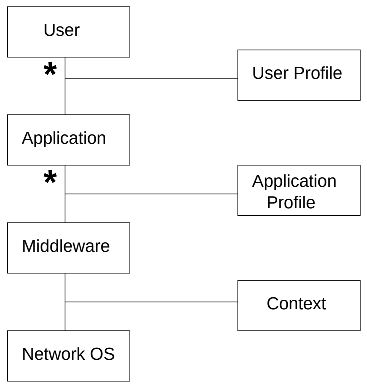

Fig. 2. User and application profiles.

On the other hand, we do not want the application designers to build their applications directly on the network OS, as this would be extremely tedious and error-prone; therefore, a middleware has to be put in place. The middleware must interact with the underlying network operating system and keep updated information about the execution context in its internal data structures. This information has to be made available to the applications, so that they can listen to changes in the context (i.e., *inspection* of the middleware), and influence the behaviour of the middleware accordingly (i.e., *adaptation* of the middleware).

Reflection and metadata are the means we rely on to build middleware systems that support context-aware applications. As Fig. **??** shows, there may be several applications running on the same middleware, and many different users using the same application. Each user may customize the application in many different ways; users can, for example, customize the task bar of the application interface using some icons instead of others; but they can also do more sophisticated things like asking the application to be silent when the user is in particular places (e.g., in a movie theatre, on a train, etc.), automatically disconnect from the network when the battery power is too low, etc. A user may also own more than one mobile device; we can therefore assume that the application can migrate from device to device, each of which may have different capabilities. In such a situation, the user could set up the application in order to display text-only information when running on a mobile phone, or disabling voice information when running on a PDA, and enabling all the functionalities when running on a laptop. In order to do so, the user sets up a *user-profile* that instructs the application on how to behave in different circumstances. From an application point of view, we call 'data' the subject of its own computation or, we could say, of its functional requirements (for example, the product catalogue in our case study). The user-profile is instead what we define as application metadata. The application filters out the settings it can manage alone in a context-independent way (e.g., layout of the task bar), and translates the other ones into an *application profile* that is then passed down to the middleware. This profile mainly relates

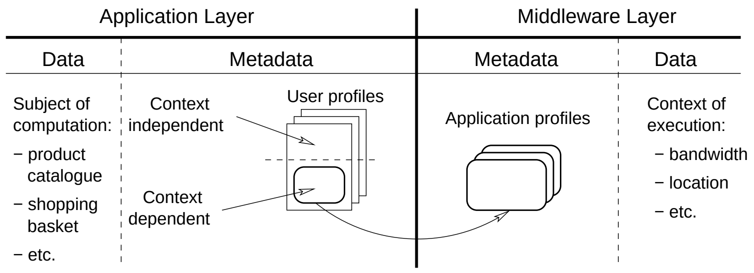

Fig. 3. Application and Middleware data/metadata.

to the non-functional requirements of the application; for example, referring to our case study, if reliability is an issue for the application, than the three-phase commit protocol is preferred to two-phase commit, when experiencing highly fluctuating bandwidth. From a middleware point of view, the context is its own data (e.g., value of the bandwidth, status of the network connection, status of the battery power, etc.), while the application profile is its own metadata (see Fig. **??**). From now on, it is the middleware that is in charge of maintaining a valid representation of the context, directly interacting with the network operating system; whenever a change in the execution context is detected, it consults its metadata to find out how the application has asked it to behave in such a configuration. Now the question is whether it is reasonable to assume that the application fixes its own profile once and for all at the time of installation and never changes it after. The answer is no. Both the needs of the user and the context change quite frequently, and we cannot expect the application designers to foresee all the possible configurations. We therefore need to provide the middleware with an initial profile, and then grant the application dynamic access to it. Here is where reflection comes into play. By definition [**?**], reflection allows a program to access, reason about and alter its own interpretation. The principle of reflection has been mainly adopted in programming languages, in order to allow a program to access its own implementation (see the reflection package of Java or the interface repository in CORBA). The use of reflection in middleware is more coarse-grained and, instead of dealing with methods and attributes, it deals with middleware data and metadata. Metadata store information about *how* the middleware has to behave *when* executing in a particular context. Applications use the reflective mechanisms provided by middleware to access their own profile, so that changes in this information immediately reflect into changes in the middleware behaviour.

There are many open questions that need to be answered; for example, which information do we encode exactly in the application profile and how? How can the middleware deal with the requests of many different applications running on it? We answer these questions in the following sections.

4 Reflective Conceptual Model

In this section we answer the questions about *what* information we need to encode in the application profile, that is, in the middleware metadata, and *how*.

The application profile is written by the application designer and then managed by the underlying middleware, that is, there must be an agreement between the two parts about the representation of the profile. We believe that the eXtended Markup Language (XML)[**?**], and related technologies (in particular XML Schema [**?**]) can be successfully used to model this information. XML has already been used as a standard for data formatting and exchange (e.g., SOAP [**?**]), as it offers a flexible framework for data structure definition and use. In our scenario, middleware defines the *grammar*, that is the rules that must be followed to write profiles, in an XML Schema; the application designer then encodes the profile in an XML document that is a valid *instance* of the grammar, according to the definition of valid XML documents given in [**?**]. Every change done later to the profile must respect the grammar, and this check can be easily performed using available XML parsers [**?**].

To understand what information to encode, we distinguish two different ways in which the application influences the behaviour of the middleware.

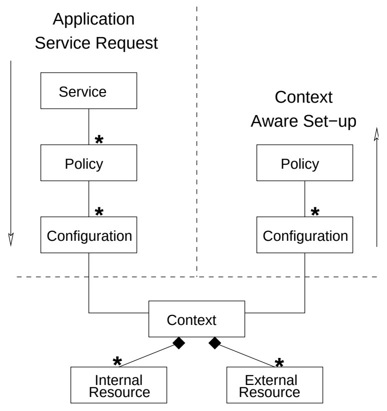

Fig. 4. Application profile.

1. *Changes in the execution context.* The application can ask the middleware to listen to changes in the execution context and react accordingly, independently of the task the application is performing at the moment. For example, the application may ask the middleware to disconnect when the bandwidth is fluctuating, or when the battery power is too low; to call back the application whenever we enter a particular location, etc. We establish an association between particular context configurations that depend on the value of one or more resources the middleware monitors, and policies that have to be applied, as shown on the right-hand side of Fig. **??**. The policies specified can be chosen among

a set of available behaviours the middleware provides, or can be call backs to the application. Fig. ?? and ?? illustrate a simple example of an XML document and corresponding XML Schema definition for this kind of information. As shown, a call back requires the specification of a reference to the application (e.g., an object reference `obj_ref_O`), an entry point (e.g., a method `method_m`), and a list of parameters. Whenever a change in the execution context causes a call back to be triggered, the method `method_m` is invoked on the object reference `obj_ref_O` with the specified list of parameters in input. Usually these parameters are XPath [?] expressions evaluated on the XML representation of the execution context kept by the middleware (Fig. ??); they are used to capture the current value of the resource examined.

```
<CONTEXT>
    <RESOURCE name="battery" value="value1"/>
    <RESOURCE name="location" value="value2"/>
    <!-- ... -->
</CONTEXT>
```

Fig. 5. XML encoding of the configuration context.

For the reflective principle, the middleware must grant applications dynamic access to both their profile and to the schema, in particular to the set of policies the middleware itself provides. In this way, we can both reconfigure the middleware to adapt to unpredictable situations, and extend the set of behaviours it provides with great flexibility.

2. *Service request.* The application can ask the middleware to execute a service; for example, to access some remote data it has not cached locally. There are many different ways a service can be provided; for example, the service 'access data' can be delivered using at least two different policies, 'copy' and 'link' (see Section ??). The circumstances under which an application may want to use them are different: a physical copy of data may be preferred when there is a lot of free space on the device, while a link (i.e., network reference) may become necessary when the amount of available memory prevents us from creating a copy, and the network connection is good enough to allow reliable read and write operations across it. Therefore, for every service the application may ask the middleware, the application profile specifies the policies that have to be applied and the requirements that must be satisfied in order to choose which of them to apply. These requirements are expressed in terms of the execution context (left-hand side of Fig. ??). Fig. ?? gives an example of how to express this information in the application profile in XML.

Particular services that middleware systems must provide to all the supported applications include trading and binding services. A *trading service* is put in place to find out which host provides a specific service requested by an application. In a mobile setting, hosts may come and leave quite rapidly; the services available when a host disconnects from the network can be completely different from the

```
<RESOURCE name="battery">
    <STATUS operator="lessEqual" value=x/>                 % context configuration
    <BEHAVIOUR policy="disconnect"/>                       % policy
</RESOURCE>

<RESOURCE name="location">
    <STATUS operator="equal" value=y/>                     % context configuration
    <CALLBACK>                                             % policy
            <APPLICATION_REF>obj_ref_O</APPLICATION_REF>
            <ENTRY_POINT>method_m</ENTRYPOINT>
            <PARAMETER>/CONTEXT/RESOURCE[@name="location"]/@value</PARAMETER>
    </CALLBACK>
</RESOURCE>
```

Fig. 6. XML encoding of a context aware set-up.

```
<xsd:element name="RESOURCE" type="ResourceType"/>
<xsd:complexType name="ResourceType">
    <xsd:attribute name="name" type="ResourceNameType" use="required"/>
    <xsd:element name="STATUS" type="StatusType"/>
    <xsd:choice>
            <xsd:element name="BEHAVIOUR" type="BehaviourType"/>
            <xsd:element name="CALLBACK" type="CallbackType"/>
    </xsd:choice>
</xsd:complexType>

<xsd:simpleType name="ResourceNameType" base="xsd:string">
    <!-- set of resources monitored by the middleware -->
    <xsd:enumeration value="battery"/>
    <xsd:enumeration value="bandwidth"/>
    <xsd:enumeration value="location"/>
    <!-- ... -->
</xsd:simpleType>

<xsd:complexType name="StatusType">
    <xsd:attribute name="operator" type="OperatorType"/>
    <xsd:attribute name="value" type="double"/>
</xsd:complexType>
<xsd:simpleType name="OperatorType" base="xsd:string">
    <xsd:enumeration value="equal"/>
    <xsd:enumeration value="less"/>
    <xsd:enumeration value="lessEqual"/>
    <!-- ... -->
</xsd:simpleType>

<xsd:complexType name="BehaviourType">
    <xsd:attribute name="policy" type="PolicyType"/>
</xsd:complexType>
<xsd:simpleType name="PolicyType" base="xsd:string">
    <!-- set of policies provided by the middleware -->
    <xsd:enumeration value="connect"/>
    <xsd:enumeration value="disconnect"/>
    <!-- ... -->
</xsd:simpleType>

<xsd:complexType name="CallbackType">
    <xsd:element name="APPLICATION_REF" type="xsd:string"/>
    <xsd:element name="ENTRY_POINT" type="xsd:string"/>
    <xsd:element name="PARAMETER" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>
```

Fig. 7. XML Schema definition.

```
<SERVICE name="accessData">
    <BEHAVIOUR policy="copy">
        <RESOURCE name="memory">
            <STATUS operator="greaterEqual" value=x/>
        </RESOURCE>
    </BEHAVIOUR>

    <BEHAVIOUR policy="link">
        <RESOURCE name="bandwidth">
            <STATUS operator="greaterEqual" value=y/>
        </RESOURCE>
        <RESOURCE name="memory">
            <STATUS operator="less" value=x/>
        </RESOURCE>
    </BEHAVIOUR>
</SERVICE>
```

Fig. 8. XML encoding of an application service request.

ones the host finds in the context when it reconnects. Therefore, on every host there must be a trader that keeps track of all the services provided by the hosts that are in reach at the moment. In general, there may be more than a provider of the same service; for example, if the service we are looking for is "access data x", there can be more than one host holding a replica of x in our neighborhood. In such a situation, the trader needs to choose which one to contact, and this decision can be taken using many different strategies (e.g., contact the closest host, contact the host on the cheapest link, etc.). Every application specifies (in its own profile) how the trading service must be delivered to it, that is, which policy the trader must apply when selecting service providers for the requests coming from this application.

Once the service provider to be contacted has been chosen, the middleware needs to decide which policy to apply to serve the request it is dealing with. If the application has not specified a particular policy, a *binding service* is invoked; the binder is in charge of checking the requirements related to each policy and deciding which one to adopt. Again, there may be circumstances where more than one policy can be followed; the selection is driven by the strategy (i.e., policy) specified by the application in its own profile under the voice "binding service" (e.g., use the policy that requires the least amount of resources, the one that provides the best quality of service, and so on).

5 Reflective Architecture

In this section we show how the conceptual model can be mapped into a reflective architecture.

As shown in Fig. **??**, the middleware maintains both its *data* and *metadata* in internal data structures. Data refer to a unique representation of the *context* of execution; we find here information about the actual values of all the internal and external resources that the middleware can monitor (e.g., memory, battery power, bandwidth, network connection, hosts in reach, and so on). Metadata

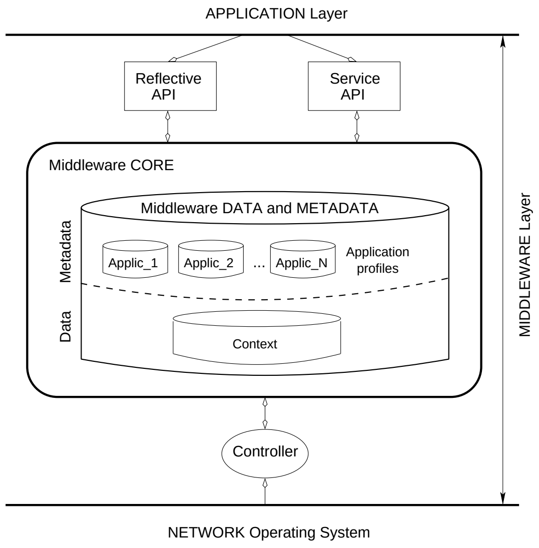

Fig. 9. Reflective Architecture.

refers to a set of application profiles, one for each application that runs on the middleware itself.

Data can be read and modified by the *Controller* component. Its task is to interact with the underlying network operating system in order to keep an updated configuration of the context. Whenever a change in the context happens, the middleware *CORE* looks up in the application profiles of running applications whether one or more of them have registered an interest in the changed resources, and triggers the corresponding actions.

Metadata can be created, read and modified by the application layer by means of the *Reflective API*. As we have seen in the previous section, we store application profiles as XML documents. These documents can be semantically associated to trees; the information stored can then be manipulated through the DOM (Document Object Model) API [**?**], which provides primitives for traversing, adding and deleting nodes to an XML tree, and the XPath [**?**] technology, which enables the addressing of specific points of an XML document. All the profiles are stored permanently by the middleware, but only the ones of running applications are actually 'active'. When a profile is initially passed down, and whenever an update is invoked, the CORE is in charge of verifying the consistency of the information it contains, before storing it permanently. For this purpose, the CORE runs a validating parser that parses the document and checks whether it is a valid XML instance of the grammar provided by the middleware to the application. XML Schema supports the specification of very detailed and complex grammars, so that the check performed by the parser can be extremely accurate. In case of an inconsistency, the update fails and an error message is reported to the application. The Reflective API can also be used in a sort of 'expert' mode to update the grammar (the XML Schema definition), that is, the set of services, the set of policies and the set of resources the middleware manages. It is the middleware CORE that is in charge of checking the consistency of the update; for example, if a new policy P is introduced, the code for it must be provided[1].

The *Service API* is used to call services provided by the middleware to the above applications. As we have seen, there are different ways an application can call a service:

`exec(service, provider, policy, service_specific_parameters)`, for example `exec(accessData, HostA, copy, X)`. If the application specifies both the service it requests, the provider and the policy to apply, the request is simply passed down to the middleware CORE that performs it: it first controls whether `HostA` is in reach and exports a replica of data `X`; it then checks that the specified policy is available for the requested service, and finally performs it. If any of these operations fails, an error message is reported to the application.

`exec(service, provider, service_specific_parameters)`. In this case the policy is missing, like in `exec(accessData, HostA, X)`. Once checked that `HostA` is actually in reach and that it exports a replica of `X`, a binding process that follows the rules imposed by the running application is started, in order to find out which policy must be applied to obtain the `accessData` service in this particular context. Then everything proceeds as before.

`exec(service, service_specific_parameters)`. Now both the policy and the service provider are missing; for example `exec(accessData, X)`. A trading process that follows the rules imposed by the running application is started, in order to look up if there exists an host that exports a replica of `X` in the current context; if this is the case, then a binding process is started to find out which policy must be applied and then everything proceeds as above. If something goes wrong, an error message is reported to the application.

6 Discussion and Related Work

We have described a middleware for context-aware mobile applications based on the principle of reflection and metadata. Through metadata, we achieve separation of concerns, that is, we distinguish what the middleware does from how the middleware does it. Reflection is then used to provide applications dynamic access to middleware metadata.

The principle of reflection has already been investigated by the middleware community during the past years, mainly to achieve flexibility and dynamic configurability of the ORB. OpenCorba [**?**], for example, is a reflective open broker that enables users to adapt dynamically the representation and the execution

[1] If everything is implemented in Java, the existence of a class P (to be dynamically loaded by the Java Class Loader) can be required.

policies of the software bus. This is achieved through two mechanisms: metaclasses which provide a better separation of concerns, in order to improve the class reuse, and a protocol which enables the dynamic changing of metaclasses, to allow adaptation of the system at run-time. This makes it possible to introduce new semantics to the initial model, such as replication, security, etc.

Blair et al. [?] have defined a "manifesto" for the next generation of middleware systems, claiming that they should be run-time configurable and allow inspection and adaptation of the underlying software. The approach they propose is based on components to structure the middleware platform, and on reflection to inspect and adapt the behavior of these components. The middleware appears to the application designer as a set of customizable components which can be tailored to the needs of the application.

Even though we adhere to the idea of using reflection to add flexibility and dynamic configurability to middleware systems, the platform developed in [?] and [?] to experiment with reflection was based on standard middleware implementations (i.e., CORBA), and therefore not suited for the mobile environment (see Section **??**).

Other middleware systems have been built to support mobility, without using the reflective principle. However, we observe that only partial solutions have been developed to date, mainly focused on providing support for location awareness on one hand, and for disconnected operations and reconciliation of data on the other hand.

Systems like Guide [?] and Cyberguide [?] provide services to users depending on their current location. Although important, we argue that the notion of context must not be limited to physical location, and our solution approaches a wider range of context information.

Bayou [?] targets database applications in mobile computing settings. The system handles disconnection and reconciliation in a very transparent way, exposing as little as possible to the application that has no way to influence the outcome of the reconciliation process. Although this vision might adapt to some scenarios, our case study has highlighted the need for a higher level of context awareness to target a broader range of applications.

In Odissey [?], a distributed file system is shared among mobile devices. Files are copied and their content is reconciled when the mobile hosts get in contact with each others using some form of application-specific adaptation. The abstraction of a file as the mobile data unit seems to be too coarse-grained in a context where bandwidth is so scarce and network connection so expensive. Furthermore, a file is nothing but a bytestream, a poor semantic structure that can only partially and with great difficulty be exploited to obtain application-specific reconciliation.

More recent systems attempt to cover issues such as cooperation and service availability. Again, being aware of what services and what hosts are around is important in mobile environments. In Jini [?] a global service look-up system is implemented as a global service, thus leading to centralized system problems in terms of scalability and fault tolerance.

Tuple space coordination primitives, initially suggested for Linda [?], have been employed in a number of mobile middleware systems such as Jini/JavaSpaces [?], Lime [?], and T Spaces [?], to facilitate component interaction for mobile systems. Although addressing in a natural manner the asynchronous mode of communication characteristic of ad-hoc and nomadic computing, all these systems are bound to very poor data structures (i.e., flat unstructured tuples), which do not allow complex data organization and therefore can hardly be extended to support metadata and reflection capabilities. We believe that XML, and in particular its associated hierarchical tree structure, allows semantically richer data and metadata formatting, overcoming this limitation.

7 Future Work and Concluding Remarks

The growing success of mobile computing devices and networking technologies, such as WaveLan [?] and Bluetooth [?], call for the investigation of new middleware that deal with mobile computing requirements, in particular with context-awareness. Our goal in this paper has been to outline a global model for the design of mobile middleware systems, based on the principle of reflection and metadata. The choice to use XML to represent metadata comes from our previous experience with XMIDDLE [?], an XML-based middleware for mobile systems that focuses on data reconciliation and synchronization problems and solves them exploiting application-specific reconciliation strategies. Our plan is to extend the previously built prototype to fully support the reflective model presented here. The middleware interaction with the network operating system to capture information about the context is possible, for example, using the Mobile Information Device Profile (MIDP) [?], that is part of the Java 2 Platform Micro Edition [?]. MIDP is a set of Java APIs which provide a runtime environment targeted to mobile information devices, such as cellular phones and PDAs. The MIDP specification addresses issues such as user interface, persistence storage, networking, and application model. We plan to exploit this technology in our prototype.

Other issues to be investigated are the followings. Conflicting policies: what happens if there are two applications A_1 and A_2 running at the same time, and application A_1 asks the middleware to disconnect from the network when bandwidth is lower than x, while A_2 wants to disconnect only for values of $y \ll x$? At the moment we do not deal with conflicting policies, assuming that in every moment there is always at most one application used by the end-user, while all the others may be 'idle'; we therefore give a sort of priority to the setting of the 'running' one. To remove this limitation, we could imagine the existence of an "administrative user" that instructs the middleware on how to behave when facing conflicting requests from different applications, giving, for example, priority to connection instead of disconnection, in case A_2 is performing some critical tasks while the user is focusing on A_1.

Another major problem is security. We can give restricted access to local data associating a set of potential users to each service exported by the application, but how can we protect the system from reflection itself? Reflection is a technique for accessing protected internal data structures. The use of XML Schema could help preventing the users from setting-up middleware behaviours in a malicious way, if the assumption that the reflective capability is provided for secure uses only (application-specific profiles derived from the ones written by the users) holds. However, reflection could cause security problems if malicious programs break the protection mechanism and use the reflective capability to change the grammar of the middleware along with the profiles. We plan to investigate this issue further.

Acknowledgements

We would like to thank our industrial partner Unipower, and in particular Ewan Harrow and Chiara Cardamone for providing the case study illustrated in the paper, Anthony Finkelstein for pointing out the importance of metadata, and Andrea Savigni for the helpful discussions while preparing the paper.