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. In this paper, we argue that middleware 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. Finally, we describe a conceptual model that provides the basis of our reflective middleware.

1 Introduction

Recent advances in wireless networking technologies and the growing success of mobile computing devices, such as laptop computers, third generation mobile phones, personal digital assistants, watches and the like, are enabling new classes of applications that present challenging problems to designers. Devices face temporary and unannounced 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 changes in the environment, such as new location, high variability of network bandwidth, etc.

When developing distributed applications, designers should 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 [4] 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 fixed networks, but they do not appear to be suitable for the mobile setting. 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, 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 [2].

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.

2 Principles of Reflective Middleware

In this section, 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. 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, error-prone and lead to non-portable applications. Instead a middleware should be used to solve these issues. 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).

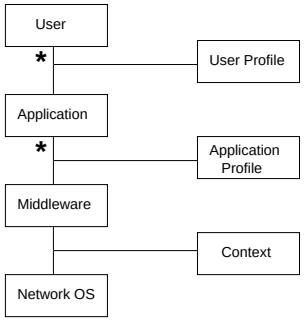

Fig. 1. User and application profiles.

Reflection and metadata are the means we rely on to build middleware systems that support context-aware applications. As Fig. 1 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. 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 (e.g. a product catalogue for an e-shopping application). 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. 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. 2). 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 mid-

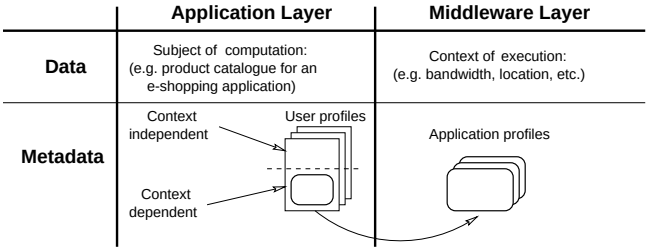

Fig. 2. Application and Middleware data/metadata.

dleware with an initial profile, and then grant the application dynamic access to it. Here is where reflection comes into play. By definition [3], 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.

3 Reflective Conceptual Model

The last section has left us with an open question: *what* information do we need to encode in the application profile, that is, in the middleware metadata, and *how*? We now provide an answer.

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)[1], and related technologies (in particular XML Schema) can be successfully used to model this information. 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. 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.

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

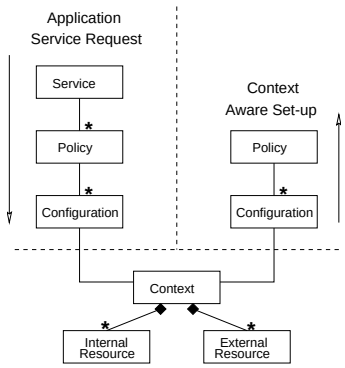

Fig. 3. Application profile.

fluctuating, or when the battery power is too low. 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. 3. Fig. 4 illustrates a simple example of an XML document for this kind of information.

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

Fig. 4. XML encoding of a context aware set-up.

Middleware interacts with the underlying network operating system in order to keep an updated configuration of the context. Whenever a change in the context happens, it 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.

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' (i.e., a physical copy of the bunch of data is created locally) and 'link' (i.e., a network reference to the master copy is created). 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 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 re-

quirements 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. 3). Fig. 5 gives an example of how to express this information in the application profile in XML.

```
<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. 5. XML encoding of an application service request.

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 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).

For the reflective principle, middleware must grant applications dynamic access to their profiles: whenever a profile is modified, the middleware 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. Also the grammar, that is, the XML schema, can be updated and the middleware is in charge of verifying the consistency of the updates: for example, if a new policy P is introduced, the code for it must be provided[1]. 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.

4 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. Examples include OpenCorba, dynamicTAO, the work done by Blair et al., etc. Even though we adhere to the idea of using reflection to add flexibility and dynamic configurability to middleware systems, the platforms developed to experiment with reflection were based on standard middleware implementations (i.e., CORBA), and therefore not suited for the mobile environment.

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 (e.g., Nexus and Teleporting), and for disconnected operations and reconciliation of data (e.g., Bayou and Odyssey).

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.

[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.

5 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 [5], 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.

Other issues to be investigated are the followings. Conflicting policies: what happens if two applications ask the middleware to behave differently when executing in the same context? What if the same application requires conflicting behaviors when changes related to different resources happen at the same time (e.g., "disconnect when battery is low" vs. "connect when bandwidth is high")? All these questions are currently under investigation

Another major problem is security. Portable devices are particularly exposed to security attacks as it is so easy to connect to a wireless link. Reflection seems somehow to worsen the situation. Reflection is a technique for accessing protected internal data structures and it could cause security problems if malicious programs break the protection mechanism and use the reflective capability to disclose, modify or delete data. Security is a major issue for any mobile computing application, and therefore proper measures need to be included in the design of any mobile middleware system. We plan to investigate this issue further.

References

1. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.
2. L. Capra, W. Emmerich, and C. Mascolo. Middleware for Mobile Computing: Awareness vs. Transparency (Position Summary). In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.
3. F. Eliassen, A. Andersen, G. S. Blair, F. Costa, G. Coulson, V. Goebel, O. Hansen, T. Kristensen, T. Plagemann, H. O. Rafaelsen, K. B. Saikoski, and W. Yu. Next Generation Middleware: Requirements, Architecture and Prototypes. In *Proceedings of the 7^{th} IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 60–65. IEEE Computer Society Press, December 1999.
4. W. Emmerich. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering - 22^{nd} Int. Conf. on Software Engineering (ICSE2000)*, pages 117–129. ACM Press, May 2000.
5. Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. An XML-based Middleware for Peer-to-Peer Computing. In *Proc. of the International Conference on Peer-to-Peer Computing (P2P2001)*, Linkopings, Sweden, August 2001. To appear.