An experience in adaptation in the context of mobile computing

Nabil Kouici, Denis Conan, and Guy Bernard

GET / INT, CNRS UMR SAMOVAR
9 rue Charles Fourier, 91011 Évry, France
{Nabil.Kouici, Denis.Conan, Guy.Bernard}@int-evry.fr

Abstract. Mobile computing with hand-held devices such as personal digital assistants and mobile phones is becoming an alternative to classical wired computing. In such environments, a disconnection is a normal event and should not be considered as an interruption of service freezing the application. As surveyed in the literature, there is much work dealing with mobile information access that demonstrates that the *laissez-faire* approach (adaptation to mobility performed only by the application) and the transparent approach (adaptation performed solely by the middleware) are not adequate, thus leading to the *collaboration* strategy in which both the application and the middleware participate to the adaptation. In this paper, we describe our experience in developing an approach to specify the adaptation of distributed component-based applications and in designing a platform that adapts applications.

Key words: Adaptation, mobile computing, component orientation.

1 Introduction

With the evolution of wireless communication, mobile computing with hand-held devices such as personal digital assistants and mobile phones is becoming an alternative to classical wired computing. In such environments, a disconnection is a normal event and should not be considered as an interruption of service freezing the application. We make the distinction between two kinds of disconnections: voluntary disconnections when the user decides to work on their own for saving battery or communication costs, or when radio transmissions are prohibited as aboard a plane; and involuntary disconnections due to physical wireless communication breakdowns such as in an uncovered area or when the user moves out of the reach of base stations. We also consider the case where the communication is still possible but not at an optimal level, resulting from intermittent communication, low-bandwidth, high-latency, or expensive networks. As a consequence, the mobile terminal may be *strongly connected* (connected to the Internet via a fast and reliable link), *disconnected* (no network connection to the Internet), or *weakly connected* (connected to the Internet via a slow link).

More and more, the distributed application's entities can spread over fixed terminals [14], or over fixed and mobile terminals [18]. In the first case, the

distributed entities present on the mobile terminals are very often lightweight entities such as client's GUI which access data and computation servers installed in fixed hosts. In the second case, a mobile terminal can be a client for servers and can be a server for other hosts. This last case was less frequently studied in mobile environments because of the limited capacity of mobile terminals and because of the difficulty in implementing these applications with traditional object-oriented, database-oriented, or file-oriented design and programming paradigms.

In addition, as the development of distributed applications converges towards component-based applications designed with component-oriented middleware such as EJB, CCM and .Net, new opportunities appear to better address the application complexity by separating the functional and the extra-functional[1] concerns.

In this paper, we present our experience in developing an approach to specify the adaptation of distributed component-based applications for the continuity of service during disconnection, and in designing a platform that adapts the application to resource availability variations for detecting and preparing disconnections.

The remainder of this paper is organised as follows. Section 2 gives the motivations and related work, and introduces the three types of adaptation that we used, namely static, dynamic and auto-adaptation. Before the detailed report on the experience, Section 3 overviews the solution to disconnection management. Next, Sections 4, 5 and 6 develop how the solution is built based on static, dynamic and auto-adaptation, respectively. Finally, Section 7 summarises the adaptation requirements elicited and presented during the paper, and concludes the paper.

2 Motivations and related work

The need to keep working while being disconnected raises the problem of data and code availability. Our approach aiming at solving this problem is to adapt distributed execution to the characteristics of mobile environments. According to [17], the adaptation to mobility can be performed by the application (*laissez-faire* strategy), by the middleware (*transparent* strategy), or by both the application and the middleware (*collaboration* strategy). As surveyed in [7], there is much work dealing with mobile information access which demonstrates that the *laissez-faire* and the transparent approaches are not adequate. We then let the end-user intervene during the development of the application (via the architect) and during the execution by expressing preferences. The first intervention is through the addition of use cases specific to disconnection management: for instance, the end-user wants to make the choice between a few full-fledged functionalities and many more but degraded ones, or wants to give priorities when it is possible to do things a better than usual. The second end-user's intervention

[1] In this paper, the extra-functional concerns, when they are provided by a middleware, are called middleware services.

is during the execution. To this aim, contextual information such as the connectivity mode of the mobile terminal is displayed to the user. The reason for this display is that the end-user wants to somewhat change their behaviour during disconnections. Furthermore, as already mentioned, the end-user wants to make choices in terms of functionalities accessed during disconnections.

As stated in [3, 4], in addition to be prepared before the execution during application's development (*static* adaptation) or triggered by actors[2] during the execution (*dynamic* adaptation), adaptation can also be automatically performed by the middleware that reacts to some changes in the context of the application during execution (*auto-adaptation* adaptation). This latter adaptation consists in transparent switching between the various configurations of the application specified in the static adaptation. These configuration changes are triggered and controlled by the middleware which auto-adapts according to the needs of the application and of the context. [2] analyses requirements of applications and the system to cope with dynamically changing execution environment. In our approach, the end-user is also involved in the process of eliciting the requirements for the auto-adaptation.

In addition, auto-adaptation is performed thanks to *reflection* [15]. The principle is to allow software part to introspect and adapt its own functioning according to the available resources. Reflection also leads to a better separation between functional and extra-functional concerns. Reflection is already used in middleware design to achieve reconfiguration and adaptation required by mobile computing [1, 5]. There are mainly two kinds of reflection. The *behavioural* reflection is concerned with the reification of computations and their behaviour: for example, the dispatching of requests and the addition of pre- or post-treatments. The *structural* reflection is concerned with the underlying structure of the application (object, component...): for example, the capability of representing the structures of components using meta-data. In our work, we intensively use reflection.

In component-oriented middleware such as CCM, EJB and .Net, the extra-functional concerns are limited in terms of their number and their type. Several approaches for the integration of extra-functional concerns have been proposed. The most used are Aspect-Oriented Programming (AOP) [9] and the component/container paradigm [19]. In the first approach, the code implementing the extra-functional concern (called *aspect*) is developed independently and weaved throughout the implementation of the functional concerns. In the second approach, the component only contains functional concerns (the business logic) and the container provides the execution environment. Extra-functional concerns are handled and enforced by the container, using standardised frameworks and techniques such as code generation. Many works have been carried out in the integration of extra-functional concerns into containers. In that area of interest, the most studied extra-functional concerns are persistency, transactional support, security, and distribution. [13] integrates the management of the quality of service into EJB containers, [16] integrates transactional policies into CCM

[2] Systems or end-users external to the application.

containers, and [8] proposes a reflective transaction service management which uses behavioural and structural reflection to allow new transaction services to be installed and used according to the needs of the application. However, the subject of our study, disconnection management, is rarely considered in component-oriented middleware.

3 Overview of the solution to disconnection management

The principal of our solution is to cache the server entity of the remote host on the mobile terminal and use it during disconnection according to the concept of disconnected operation [10]. For that very reason, a local proxy of the remote component, called a disconnected component, achieves the same functionalities as the component in the remote server, but is specifically built to cope with disconnection and weak connectivity. The solution is then twofold. Firstly, the distributed application must be built in such a way that it specifies the behaviour while being disconnected. This is accomplished by using some meta-data to specify application's components and functionalities: which components or functionalities can be cached? And which ones must be present in the mobile terminal for the disconnected mode? Secondly, the adaptation during execution must choose the policies specified at the software architecture's design time according to the execution context and the decisions of the end-users. To this aim, we organise the architecture of containers so that they orchestrate the middleware services specifically designed for detecting disconnection events and for caching components according to the application's profile, which can be dynamically overloaded by the end-users.

4 Static adaptation

We have introduced a meta-model for designing applications that deal with disconnections. This meta-model is based on meta-data that define an application profile. The disconnectability meta-data indicate whether a component residing on a remote server can have a proxy component on the mobile terminal (the disconnected component). If this is the case, the original component is said to be disconnectable. Software architects set the disconnectability meta-data since they have the best knowledge of the application semantics. Furthermore, disconnectability implies design constraints that the developers must respect. Next, the necessity meta-data indicate whether a disconnected component must be present on the user terminal. Clearly, the necessity applies only on disconnectable components. The necessity is specified both by application's developers and end-users. The former stake-holders provide a first classification in developer-necessary and developer-unnecessary components, and the latter stake-holders can overload a developer-unnecessary component to be user-necessary at runtime. Finally, the priority meta-data indicate the priority between components.

However, end-users are only aware of application functionalities and unaware of components which are used to perform these functionalities. Thus, we define

a service as a set of components that interact with each others to achieve a functionality. The application as a whole may be regarded as a set of services which are accessed by users through a GUI. Thus, we define two types of interactions: intra-service (between components in the same service) and inter-services (between services). However, the local use of a service during a disconnection may require the presence of others services in the cache. Solving this issue leads to the determination and the computation of dependencies between services [11]. These dependencies are presented within a directed graph where nodes denote services and edges denote the *"use"* dependency which is annotated with the necessity meta-data. In addition, service availability in disconnected mode implies the presence of some components which are used for achieving this service. Thus, by analogy, component dependencies are also drawn within the dependency graph where nodes denote components and edges denote dependencies between components.

The development process is based on the "Façade" design pattern [6] and the "4+1" view model [12]. The "Façade" design pattern allows to simplify the access to a set of related components by providing a single entry point, thus, reducing the number of components presented to end-users. The "4+1" view model makes possible the organisation of the software architecture in multiple concurrent views (use cases, logical, process, development, and physical). Each one addresses the concerns of some of the various stake-holders of the distributed application. In addition, it helps in separating the functional and extra-functional concerns.

5 Dynamic adaptation

The use of the "Façade" design pattern during application development reduces the number of components presented to the GUI, thus simplifying the design of the GUI. During execution, the GUI can present to the end-user the list of services offered by the application and their corresponding meta-data (disconnectability, necessity, and priority). The end-user uses this list to overload the necessity of some unnecessary services at it suits. These overloads lead to a propagation of the meta-data intra- and inter-services. More details about necessity propagation are given in [11]. The role of the initial application profile is the identification of the minimum set of services (and thus components) that must be cached at launching time. In fact, the middleware refuses to start the adaptation on the mobile terminal if there is not enough memory space in the cache for these components. In consequence, a change in the necessity at runtime provokes a dynamic management of the content of the cache. Two important issues exist in managing the cache. The two mechanisms of the cache management, deployment and replacement, take into account these meta-data and use reflection to dynamically introspect components and modify their meta-data. This is where structural reflection is used in the dynamic adaptation. Therefore, the deployment mechanism load new components when the end-user tags service as user-necessary, and the replacement mechanism uses the priority of

user-necessary and unnecessary services when there is not enough memory size, evicting firstly less-priority unnecessary components.

6 Auto-adaptation

In our approach, the auto-adaptation is performed using a specific container architecture and the application transparently benefits from the middleware services provided by the platform.

In the component/container paradigm, the communication between the container and the component is done through interfaces: *Internal interfaces* used by the component developer and provided by the container to assist in the implementation of the component's behaviour; *call-back interfaces* used by the container and implemented by the component, either through generated code or directly, so that the component can be deployed into the container. The communication infrastructure between components is controlled by the container through entities called *controller*. In most of the component models, the container offers at least two controllers. The first one acts as a *pre-request interceptor* intercepting all incoming requests and the other one acts as a *post-request interceptor* intercepting all outgoing requests. For disconnection management, we add five controllers in the container: A local connectivity detector to detect disconnections, an access to the cache management service, an access to the logging service, and an access to the reconciliation management service. Each controller is related to a middleware service.

Each component is executed within the container. In this container, all the incoming (*resp.* outgoing) requests of the component are intercepted by the pre-request (*resp.* post-request) interceptor which interacts with the other controllers for the disconnection management. This is where behavioural reflection is used.

7 Conclusion

In summary, the paper reported an experience in using adaptation in mobile computing for dealing with disconnection management. Table 1 summarises the adaptation requirements elicited and presented during the paper. The table has three dimensions: the type of reflection (structural *vs.* behavioural), the type of adaptation (static *vs.* dynamic *vs.* auto-adaptation), and the different requirements' stake-holders (gathered into three groups: architect, developer, and middleware for itself).

Our position is to claim that before being adapted during execution, an application must be modelled and its variation points clearly defined in the software architecture. This is particularly of utmost importance for extra-functionalities that require a collaboration strategy such as disconnection management for which the different stake-holders (from the architect to the end-user) intervene by giving preferences. The modelling leads to an application's profile that is used as an input and modified in a controlled manner by the middleware and by the end-user during execution.

	Static adaptation	Dynamic adaptation	Auto-adaptation
Structural reflection	*Architect*: Provision of an initial application's profile, *e.g.* UML tagging for saying which components must/may be cached *Developer*: Insertion in the containers of interposition for cache management and connectivity detection, and design and development of disconnected components	*End-user*: Change (overload) of application's profile for cache management through a GUI	*Middleware*: Transparent deployment and replacement of disconnected components in the cache according to the meta-data initialised in descriptors and overloaded by the end-users
Behavioural reflection	*Architect*: Definition of contextual information, *e.g.* what is strong, weak or null connectivity, and specification of adaptation policies, *e.g.* deployment and replacement strategies for cache management *Developer*: Insertion in the containers of orchestration of middleware services, here cache management and connectivity detection	*End-user*: Display of connectivity information as an iconic image, and voluntary disconnection through a GUI when the end-user decides to work on their own for saving battery or communication costs, or when radio transmissions are prohibited as aboard a plane	*Middleware*: Transparent switching between local and remote invocation, and detection of connectivity information for involuntary disconnection

Table 1. Adaptation requirements with the different stake-holders.

To conclude, we outline to open issues. Currently, the designs of the disconnectable components for the remote host and their corresponding disconnected counterparts for the mobile terminal are completely distinct. How can it be envisioned that the design of the latter components is just a specialisation, a parameterisation... of the former ones? Could we use aspect weaving at design time for this? Dealing with this issue is necessary before foreseeing the possibility to adapt component-based legacy applications to extra-functionalities that require a collaboration strategy. In addition, a practical limitation of the current platform is that full-fledged containers —*i.e.*, composed of all the possible extra-functional properties— are very big to be loaded and deployed on the mobile terminal. How can we dynamically add extra-functional "aspects" at runtime?

References

1. A. Al-bar and I. Wakeman. A Survey of Adaptative Applications in Mobile Computing. In *Proc. ICDCS Workshop on Smart Appliances and Wearable Computing*, pages 246–251, Mesa, Arizona, USA, Apr. 2001.
2. C. Becker and G. Schiele. Middleware and Application Adaptation Requirements and their Support in Pervasive Computing. In *Third International Workshop on Distributed Auto-Adaptive and Reconfigurable Systems*, Providence, Rhode Island, USA, May 2003.
3. E. Bruneton. *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties.* PhD thesis, INPG, Grenoble, France, 2001. In French.

4. C. Canal, J.-M. Murillo, and P. Poizat. Coordination and Adaptation Techniques for Software Entities. In J. Malenfant and B. Ostvold, editors, *ECOOP Workshop Reader*, volume 3344 of *LNCS*, pages 133–147, Oslo, Norway, June 2004.
5. L. Capra, G. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting Reflection in Mobile Computing Middleware. *ACM SIGMOBILE Mobile Computing and Communications Review*, 1(2):34–44, 2002.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
7. J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):117–157, June 1999.
8. R. Karlsen and A. Jakobsen. Transaction Service Management: An Approach Towards a Reflective Transaction Service. In *Proc. 2nd Middlware International Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, June 2003.
9. G. Kiczales, J. Lamping, M. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, 1997.
10. J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Pacific Grove, USA, May 1991.
11. N. Kouici, D. Conan, and G. Bernard. Caching Components for Disconnection Management in Mobile Environments. In *International Symposium on Distributed Objects and Applications, DOA*, Larnaca, Cyprus, Oct. 2004.
12. P. Kruchten. Architectural Blueprints: The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
13. A. Meguel. Integration of QoS Facilities into Component Container Architectures. In *Proc. Fifth IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, pages 394–401, Washington, DC, USA, May 2002.
14. L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity ofr Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain resort,CO, Dec. 1995.
15. N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a Reflective Component-based Middleware Architecture. In *Proc. Workshop on Reflection and Metalevel Architectures*, Sophia Antipolis, France, June 2000.
16. R. Rouvoy and P. Merle. Abstraction of Transaction Demarcation in Component-Oriented Platforms. In *Proc. 4th ACM/IFIP/USENIX International Middleware Conference*, volume 2972 of *Lecture Notes in Computer Science*, pages 305–323, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.
17. M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc. 15th Symposium on Principles of Distributed Computing*, pages 1–7, Philadelphia, USA, 1996.
18. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou: A Weakly Connected Replicated Storage System. *Proc. 15th Symposium on Operating Systems Principles*, 1995.
19. M. Volter. Server-side Components—A Pattern Language. In *Proc. Sixth European Conference On Pattern Languages of Programs*, Irsee, Germany, July 2001.