Seamless Context Adaptation on a Service-Oriented Framework

Dana Popovici, Mikael Desertot, and Sylvain Lecomte

UVHC, LAMIH UMR 8201 CNRS,
University Lille North of France
59313 Valenciennes, France
firstname.surname@univ-valenciennes.fr

Abstract. This article describes an easy, efficient way to manage context-aware applications with the help of metadata. We rely on CATS, our proposition for an application framework embedded on mobile devices. It is designed to host applications conforming to the SOA principles for achieving a flexible and dynamic architecture. Our framework provides non-functional capabilities for context management and for the adaptations required at context changes. In this article we focus on the use of iPOJO handlers and the advantages they bring to the OSGi technology.

1 Introduction

Mobile devices such as smartphones and tablets are becoming more and more part of our daily lives. In the past years they have known a great success and also a great evolution. These devices are meant for personal and frequent use, with a multitude of interesting and helpful applications (notes, maps and navigation, weather, email, etc.). The mobile devices follow users on their trips, assisting them along the way. We wish to improve the functioning of the devices through context-awareness and flexible applications.

Users move from one place to another, causing their applications to run in different contexts. Moreover, some places can have specific applications, like shops, museums, car parks, etc. Our goal is to provide a simple way for users to benefit from these specific applications and in the same time have their own applications adapt to the context changes. For better understanding, let us take as an example the Vespa [3] application for information sharing between drivers. On one hand, it shares all kinds of information: accidents, emergency brakings, emergency vehicles passing by, etc. On the other hand, it is also concerned with parking places, a different type of information, as it can cause competition between the drivers. This is a good example for the importance of context: if only a few users are in the same vicinity, a free parking place can be announced to all cars; if a greater number of users are present, the free place should be reserved for a single driver to avoid competition (see [4]); if the user is next to an indoor car park, he should request a parking place from the server of the car park. Thus, a single application has multiple ways of functioning, depending on the context.

C. Borcea et al. (Eds.): MobilWare 2012, LNICST 65, pp. 207–220, 2013.

In our previous work [16,15], we have proposed an application framework called CATS, hosting transportation applications that accompany users on the move. CATS is the execution environment for service based applications, offering management capabilities on top, for context-awareness and adaptation. Thanks to this framework, applications can be designed by dividing their functionalities into modules. For a same functionality, we can provide multiple implementations, each suited for a different context situation. As such, Vespa has been adapted for the CATS framework, with multiple implementations for the parking service.

This article describes our approach for achieving context-awareness and dynamic adaptation through the use of iPOJO Handlers. It is an efficient and non intrusive solution that allows applications to be developed in a clean manner while keeping the framework light. We evaluate our proposition through a series of tests on several Android devices.

2 Related Work

There are two important issues related to our work: the context and the architectures that allow for flexible and adaptable applications. First of all, what is context and how should it be used? If we start from the rather general definition given by Dey and Abowd [6] we should include "everything" that could influence the behavior of the applications as context information. We can cite some surveys on modeling and processing context information, [10,17] who show the different strategies used in research. From our point of view, the context should be modeled and used once for all transportation applications, as they run on the same device, for the same user. We have described the context elements that affect applications in the transportation domain in our previous work [5].

Identifying what context is and how it influences our applications resolves only half of the problem. The second half concerns the reaction to changes. How do our applications modify their behavior? It seems clear that their architecture should be as modular as possible, providing an easy way of changing parts of an application when the context imposes it. In a related work, [14] proposes a Dynamic Software Product Line to create applications using the most suited components, taking context into consideration. They describe a context-aware framework using sensors [2]. In this solution, applications have predefined configurations that are chosen with respect to the execution context. Solutions for context adaptation are also available with Composite Capability/Preference Profiles (like [13]), but they relate rather to the adaptation of content and not that of the functionalities. Finally, works like [1] introduce middleware to consider context adaptation for applications. This framework in particular targets the assembly of distributed applications whereas we consider the assembly of standalone context-aware applications embedded on a mobile device.

We would like to go a step further, by allowing to download and install new services while the application is still running. This provides more flexibility and adaptability to the applications. To the best of our knowledge, there is no literature concerning the download and installation of application components (services) "on the fly" for mobile devices involved in transportation applications.

The service-based approach has also been employed by [12] in their work for an autonomic management system. Our work too nears the concepts of autonomic computing through the desire to provide a framework with self-management capabilities. The concept of autonomic computing has been stated for the first time almost 10 years ago, one of the first works to mention it being [11]. For now the vision is not fully attained, as indicated in [7].

3 Context and Context-Awareness

The context is one of the main concerns when building applications nowadays, especially for mobile users. From the developers point of view, it is important to define all context elements and situations that influence the one application he is writing. From our viewpoint, our framework CATS must be able to support all context elements for a multitude of applications. This is why we propose a simple generic structure to represent any Context Element (CE) (Table 1).

Table 1. Representation of a Context Element (CE)

CE	
Name	- *the unique name of the context element*
Type	- *the type of information it contains*
Value	- *the value of this element at a time being*
[Unit]	- *(optional) the unit of measurement*
[Category]	- *(optional) a category from a classification*

A Context Element is represented by a unique name. It could be of great help to have one or more ontologies describing the Context Elements, to avoid giving different names for the same element or the same name for different elements. There could be an ontology for the transportation domain, another for the CE related to the device, and so on. However, it is not the scope of this paper to discuss ontologies, we only retain that a unique name is required for each CE.

We have judged necessary three types of Context Elements: *boolean*, *discrete* and *continuous*. A "boolean" element will only have two possible values, true or false. This type of Context Element describes mostly a resource that is available or not, like the Wifi or the GPS signal. An unavailable resource represents an important context situation that probably needs an adaptation, so it should be represented in our framework. A "discrete" Context Element is related to a context situation represented through discrete values. For example, if we would like to represent the type of road for a driver, we could differentiate between "city", "highway", "car park" and others. At last, a "continuous" element is one that can be characterized through a numeric value, like the speed at which the user is moving. Two optional pieces of information can be added to the description of a Context Element. The Unit, representing the unit of measurement for elements of type "continuous". A Category can also be specified, based on some

classification of the elements. For instance, some Context Elements are related to the *device/hardware*, like the GPS module or the Wifi module, while others are related to the *environment*, like the number of neighbors.

Context-awareness implies that applications react to the changes in their context. Because the user is on the move, the environment changes frequently, and so do many important elements like the communication networks, the number of neighbors, the type of road, etc. The context should be evaluated repeatedly during the functioning of the context-aware applications. We propose to use some lightweight modules called Context Monitors, to evaluate the state of the context. Each Monitor should handle a single Context Element and either have a configurable evaluation frequency or expose a method to force the evaluation. Context-awareness can be achieved thanks to the Monitors which detect changes when they occur (or sufficiently fast after) and notify the interested applications.

4 Application Composition

Applications are composed of multiple functionalities, which can be divided into independent modules. First, we can identify the functionalities that are common to most applications, like for example positioning. When possible, it is more interesting to have a single piece of code handling the localization, rather than having multiple applications implement similar code. Second, each part of an application providing a certain functionality can be implemented in multiple ways. For each computation we can choose the most appropriate way to do it.

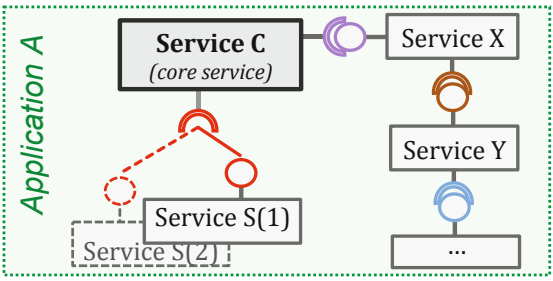


Fig. 1. Application example for the CATS Framework

In order to achieve the separation of functionalities, we chose to follow the principles of Service-Oriented Architecture (SOA). We use applications built out of services: a "core service" representing the business logic (the central part) of the application and several other services implementing different functionalities, which are used by the "core service", as represented in Fig. 1. An application is the assembly of multiple services. Like explained in [9], a service is an interface representing the contract between the service providers and clients. The service

providers are objects accessed via direct method invocation. This way, we can have the same functionality with different implementations, each one adapted to a certain context situation. We call "equivalent services" the different implementations of the same interface, *Service* $S(1)$ and *Service* $S(2)$ in our example.

We say that *Service* S is a Context Dependent Service, as it depends on the context of execution. As such, if the user is in one context situation, *Application* A should use *Service* $S(1)$, the first implementation of S. If the user is in another context situation, A should use $S(2)$, the second implementation. To make this possible, each of the implementations must define its dependencies on Context Elements, with a representation similar to that given in Section 3. For each CE that $S(1)$ or $S(2)$ depends on, the services must provide the corresponding informations. There is a difference with respect to the representation of the CE, notably in what concerns the value of the Element. A Context Dependent Service must thus describe for which value of the CE it is supposed to work best. If it depends on a CE of the type "boolean", then the service depends on a resource and will work only if the resource is available. For example a positioning service might depend on the GPS signal and not work if it is not available. If the CE is of type "discrete", the service must specify the value for which it works. In the case of a "continuous" CE, the service can specify an interval of values for which it works. For example a service might have an implementation adapted for a low average speed, 0-50 km/h, and a second implementation for an average speed of over 50 km/h. With the help of the Context Monitors, the CATS framework detects when the values of the Context Elements change, and can thus bind the suitable implementation of each service dynamically.

5 CATS Framework and VESPA

The CATS Framework, introduced in our previous work [16], is the execution environment for multiple service based applications, as well as the management modules that allow for context-awareness. There are several advantages to the use of our framework. First, it allows to share services between applications. The positioning service is one of the best examples, as most transportation applications will use it. A second advantage is that the context is managed by the framework, allowing applications to be lighter and to concentrate on the functional parts. Moreover, a context change can concern more than one application, so when it is handled is to the benefit of several applications. For example, when the GPS signal becomes unavailable, the management modules will find an equivalent service as a replacement. Fig. 2 shows an overview of the CATS Framework with two applications that share the "Position" service. On the right side we represent the management modules: the Context manager, handling the context-related information; the Execution manager, dealing with the execution of the services; the Trader, handling the download of new services.

Vespa consists of a core and several other services, from which only a few are represented here. As explained in the introduction, on of the functionalities proposed by Vespa concerns the ad-hoc management of parking places. A first

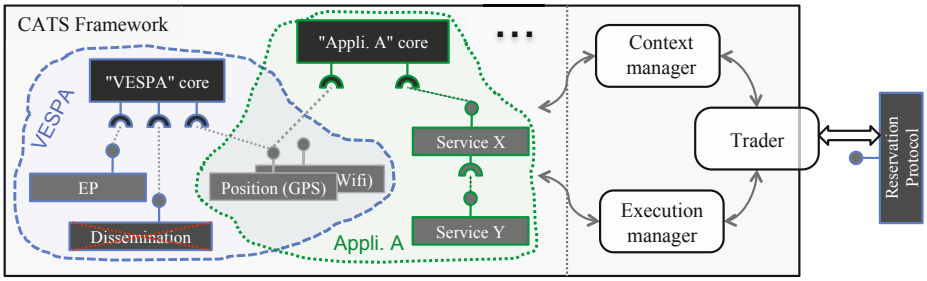


Fig. 2. VESPA and other applications on the CATS Framework

protocol consists in disseminating the information of a free place (using a special protocol that avoids flooding the network). If there are many interested neighbors, this solution can prove to be inefficient, as there would be a high number of cars trying to get the same place. The "Dissemination" service must be stopped and replaced with the "Reservation Protocol", in order to reserve the place for a single driver among the interested ones. Fig. 2 shows the "Dissemination" service being stopped, and the "Reservation Protocol" service being downloaded by the Trader to be installed and started on the CATS Framework. We note that the two services mentioned here are both implementations of the "Parking" service, and their alternative use is based on the number of neighbors.

6 Prototype with iPOJO

We have developed our prototype as an Android Activity which embeds the Felix Framework and iPOJO [1] [8]. The CATS Framework has been constructed on top of **Felix** 3.2.2, an OSGi implementation by Apache released in May 2011. It is a certified platform[2], conforming to the OSGi specification, Release 4 Version 4.2 from March 2010. We used the implementation of iPOJO version 1.8.0 from January 2011. The CATS Framework is an execution environment for multiple applications compatible with both Android and OSGi. The applications are built of modules, which are bound at the execution. Each application must have its own component handling the display (GUI), and may use the services available on the framework. The management modules are implemented as services running on the platform and oversee the non-functional capabilities of our framework. Besides the management modules that were already described in our previous work [16] (Context manager, Execution manager and Trader) we introduce in this paper a set of iPOJO Handlers for the CATS framework.

The Execution Manager oversees the execution of the services. It must know all the Context Dependent Services that are available on the CATS framework and can decide to start or stop services based on notifications from the Context

[1] `http://felix.apache.org/site/apache-felix-ipojo.html`
[2] `http://www.osgi.org/Specifications/Certified`

Manager on context changes. If there is no suitable version of a service, i.e. one that is adapted to the current context situation, the Execution Manager must call the Trader to search for a replacement.

The Context Manager can evaluate the state of the context on demand, but also on a continuous basis, when certain elements need to be monitored. It keeps track of all Context Elements, being informed by the Monitors when changes occur or requesting the Monitors to reevaluate the context. It then informs the Execution Manager of the changes.

The Handlers

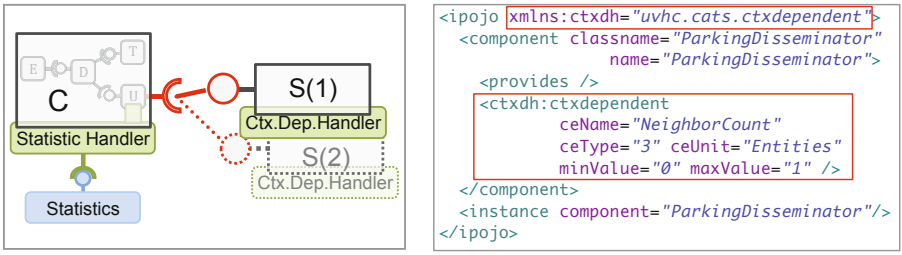
There are several advantages to the iPOJO component model. One of them is the use of handlers to manage non-functional concerns like the binding of components by injecting the needed code inside the services. Moreover, iPOJO is extensible, it allows developers to create their own handlers for specific functionalities for their framework. Another advantage is the development of simple components as plain old Java objects. The component's metadata can all be set in an XML file, having thus a complete separation of the functional code (shown in Fig. 3(b)). Furthermore, it allows us to reuse code that has been written for other purposes. For example, we can adapt a piece of code measuring the state of a resource periodically, and use it as a Context Monitor. The modifications imply simply adding metadata for the iPOJO information and for the handler to be plugged. In the following we present shortly the handlers we use for CATS.

The Context Monitor Handler is used to link the Monitors to the Context Manager. It reads the information about the Context Element that is monitored: name, type, value field, [unit, category]. The handler intercepts all modifications of the value field and updates the Context Manager. This way, the Context Manager is updated with the most recent context state and can detect changes.

The Context Dependency Handler is plugged to the Context Dependent Services in order to register them with the Execution Manager. It relies on the description of the CE that each service implementation must provide, as seen in Section 4. The Execution Manager has knowledge about all implementations of a certain service and about the context conditions in which to use each of them.

The Statistic Handler is used for measurement purposes, as it detects the state changes of client components. It then registers the time when a component has been invalid because of a missing dependency, allowing us to measure the impact of switching between components at runtime.

In Fig. 3 we present an example with a set of components and their associated handlers. The component C is a client requiring as a provider the Context Dependent Service S. The two implementations of S, S1 and S2, have the Context Dependency Handler plugged, in order to inform the framework of the execution conditions that they need. Based on the values read by the Context Monitors, the Execution Manager will decide which of these services can execute. If none of them is suited with the current context, C will not be able to function, as its dependency will be unsolved. The Statistic Handler is plugged on C and registers



(a) Client C & provider S (b) Metadata of service S

Fig. 3. Test configuration with a component C and a Context Dependent Service S

the unavailability periods, when C's dependency is not satisfied. Fig. 3(b) shows the metadata needed by Service S to declare the Context Dependency Handler. The service specifies which handler is plugged and the metadata related to the Context Element that it depends on (inside the red square). Other metadata includes the "<provides />" tag, showing that S is a provider, the name of the service ("ParkingDisseminator") and the creation of an instance.

7 Evaluations

We have evaluated the execution of our framework and the adaptation of applications at context changes with the configurations described above. We have used two types of phones and a tablet: Sony Ericsson Xperia ray - running on Android 2.3.4, Samsung Galaxy 551 - running on Android 2.3.6 and the HTC Flyer tablet - running on Android 3.2.1.

The goal of the evaluations is to asses the adaptation time when the context changes. For this reason, we consider the moment when the change has been detected by the Context Manager and wish to see how long it takes until the suited service is ready for use, as well as the impact this action has on the application. We considered the Vespa application, which uses a parking service to advertise free parking places to other vehicles and get information about free places from the other entities. The parking service is context dependent and has several ways of negotiating the parking places between vehicles, based on the number of neighbors. Here are the elements used in the evaluation:

- **Context Element {***Name* = NeighborCount; *Type* = Continuous *(coded as the integer "3" in Fig. 3(b))*; [*Unit* = Entities; *Category* = Environment;]**}**. In this case, the value is an integer greater than or equal to 0, representing the number of neighboring devices.
- **Context Monitor for "NeighborCount"** is a service that evaluates the number of one-hop neighbors every 15 seconds. For testing purpose, the service has been modified to return the same series of numbers repeatedly, to force the same service exchanges.

- **Vespa**, an application that uses inter-vehicles communication to share information about the traffic. It requires a Parking Service.
- **Parking Service**, a Context Dependent Service with the following implementations, depending on the CE NeighborCount
  - **DPS** *Dissemination Parking Service*: a vehicle liberating a parking place broadcasts the information to the surrounding vehicles. This version should be used if NeighborCount $\in \{0, 1\}$.
  - **RPS** *Reservation Parking Service*: the vehicle liberating a place advertises it and reserves the place for one of the interested vehicles. This version of the Parking Service works when NeighborCount $= 2$.
  - **DPSv and RPSv** *versions of the two previous services*: work similarly to the DPS or RPS services, for NeighborCount $\in \{3, 4, 5, 6, 7\}$.

We note that the context conditions for each Parking Service here have been chosen for experimentation purposes only. For the final implementations of DPS and RPS a careful study should be carried out considering the NeighborCount (number of neighbors) that represents the switching point. Up to a certain limit, the information of a free parking place can be disseminated without causing competition. After this limit, the place should be subjected to reservation.

The experimentations have been carried out with either two or six equivalent services present on the devices. The same set of tests have been executed for DPS and its versions, then for RPS and its versions, allowing us to compare the impact of a "heavier" service. The DPS versions have a size of around 5 kB and don't launch any threads, so they can be considered as "light" versions of the Parking Service. The RPS versions are around 15 kB and launch a thread, so we consider them as a "heavier" version of the Parking Service. The goal of our experimentations is to measure the reaction time from the detection of a context change until the application is adapted, i.e. the current Parking Service, which is now inappropriate with respect to the new context, is stopped and a new one is bound, complying to the new context conditions. All the services used for testing are registered in a given order in the Execution Manager and processed one at a time when started or stopped. As a consequence, it matters when the services are switched on and off: for short periods of time, we can either have two equivalent services working or none working. The Vespa application can thus stop working due to the missing dependency.

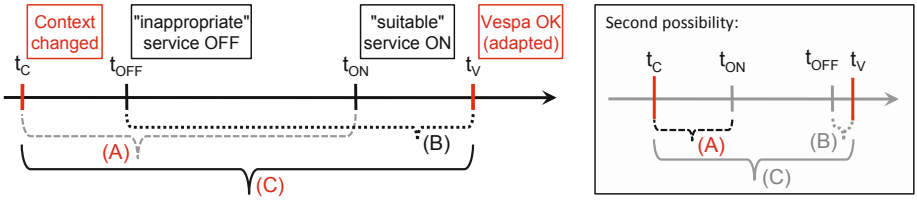


Fig. 4. Experimental measurements

Fig. 4 shows the events taking place during the execution of the Vespa application, when the context changes. We consider t_C, the moment when the context change is detected, t_{OFF} when the current service is switched off, t_{ON} when the most suitable service is switched on (and ready to use) and t_V, when the Vespa application is adapted by having the new service bound. Based on the order in which the actions take place, t_{OFF} and t_{ON} can be in any order. In the experiments, we have measured the different time periods: Time (A) - from the detection of the context change to when the suitable service is started and ready to use; Time (B) - the period in which Vespa has been unavailable because of the missing dependency; Time (C) - from the detection of the context change until Vespa is adapted and operational again. Depending on the order in which the services are handled, Vespa might not become unavailable, making it impossible to measure Time (B) and Time (C). This behavior is due to the iPOJO framework, which manages the bindings of services when there is more then one available. We can thus optimize the behavior of our framework by ensuring that the suitable service is switched on before switching off the other one. Nevertheless, our goal here was to measure the time from the detection of the change until the adaptation was achieved, represented by the Time (C), so no optimization has been done.

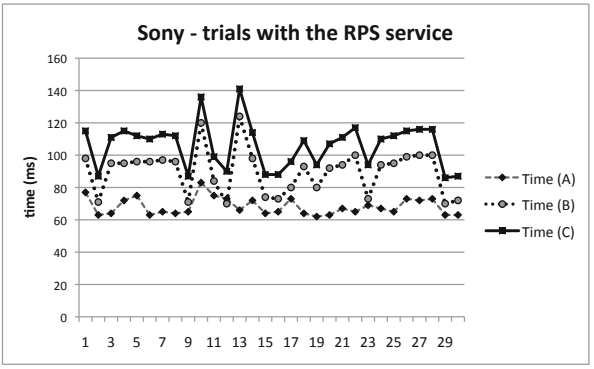


Fig. 5. Times (A), (B) and (C) when switching between versions of the RPS service

The first type of experimentation is intended to study the three times presented above over a set of 30 trials. Fig. 5 shows the times (A), (B) and (C) on the Sony phone in the case when the inappropriate service is stopped before switching the new one on, causing Vespa to stop functioning. We can notice that $t_V > t_{ON}$: the time that iPOJO requires from the detection of the new service until it is bound is always superior to 0. Therefore, we can say for sure that Time (C) is greater than the other two times, as it can be seen also in Fig. 5. For Time (A) and Time (B) there is no rule of which is greater than the other, but several of our simulations have shown Time (B) to be slightly greater than (A). We can notice from the figure that Time (C) follows the same path as

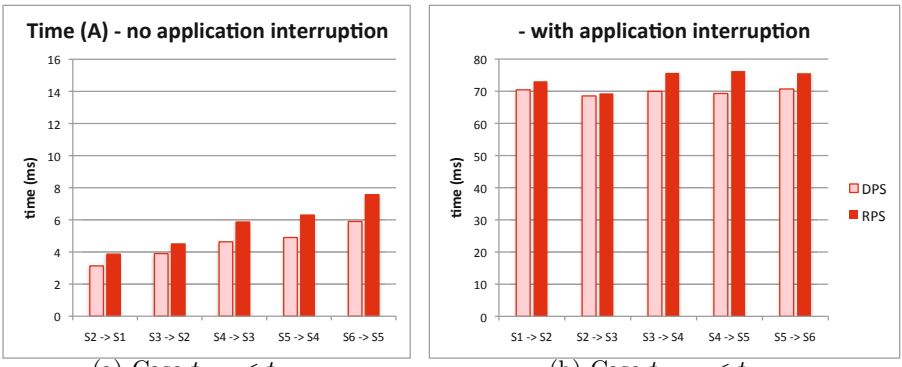
Time (B), which can be explained when looking at the significance of the two: $Time(C) = t_V - t_C$; $Time(B) = t_V - t_{OFF}$; $Time(C) - Time(B) = t_{OFF} - t_C$ The fact that the difference between these two time spans is almost constant implies that the time needed to switch off the inappropriate service varies very little. The averages and the standard deviations of these measurements are presented in Table 2 and show that the complete adaptation of the application in case of a context change takes about 106 ms.

Table 2. Average and standard deviation for Times A, B and C, with the RPS

	Time (A)	**Time (B)**	**Time (C)**
Average (ms)	68,16	90	106,26
Standard deviation	5,34 ms (7,83%)	14,09 ms (15,65%)	14,17 ms (13,33%)

A second type of experimentation has been performed with a twofold goal: first to estimate the influence of having more than one alternative service, and second to estimate the impact of services with different complexities. For this purpose, we have used six equivalent services, either versions of DPS (the "light" implementations) or versions of RPS (the "heavier" implementations). In the results that we present, the services are called $S1, S2, ..., S6$ and represent either the six versions of DPS, or the six versions of RPS. The indexes indicate the position of the service in the list of the Execution Manager, allowing us to deduce the overhead introduced by the number of equivalent services. We have imposed the context conditions such that the services were switched either from $S1$ to $S6$ or the other way around. A switch indicated as $S_{i+1} \rightarrow S_i$ implies that S_i is started to replace S_{i+1} (which is stopped right after that). This is the case were Vespa continues to function without noticing the service switching. The results for this case are presented in Fig. 6(a). A switch indicated as $S_i \rightarrow S_{i+1}$ implies that S_i is stopped before starting S_{i+1}, causing Vespa to be interrupted while its dependency is unresolved. The results of this case can be seen in Fig. 6(b). We note that the transitions $S_1 \rightarrow S_6$ and $S_6 \rightarrow S_1$ are different, because they cause the opposite behavior as the other ones. For a more clear view of the results, these two transitions are not presented, but their values are consistent with the rest of the experimentations. In the first case, when S_1 is stopped, S_6 is started in 68,4 ms and 71,63 ms for DPS and RPS respectively. In the second case, S_1 is started in 4,3 ms and 4,46 ms respectively.

In Fig. 6 we examined the time it takes from the detection of a context change, until the suitable service is started (i.e. the service that works best for the new context situation). Two different aspects were taken into consideration here: the number of equivalent services and the complexity of the services. In 6(a) we notice a clear influence of the position of the service in the list of equivalent services. The further it is in the list, the longer it takes until it is completely switched on. This value increases steadily from 3,1 ms to 5,9 ms and from 3,8 ms to 7,5 ms for the DPS and RPS respectively. We can also observe a difference between the lighter DPS and the slightly heavier RPS which is a little longer to



(a) Case $t_{ON} < t_{OFF}$ (b) Case $t_{OFF} < t_{ON}$

Fig. 6. Time (A) on the Sony phone - "light" vs. "heavier" service

start. The tests have been carried out on the other two testing devices, the HTC tablet and the Samsung phone. They have shown similar results with respect to the increasing tendency based on the number of services and their complexity. In Fig. 6(b) we present the case where the inappropriate service is switched off before starting the new one. Because it is the same thread that stops and starts the services, an overhead is induced and it takes around 70 ms until the suitable service is ready to use. In this case, the influence of the number of services and their order in the list isn't obvious any more. Nevertheless, the services have a rather uniform behavior, with the average values varying between 68,5 ms and 70,7 ms for DPS and between 69,3 ms and 76,2 ms for RPS. For each transition, the "heavier" RPS is still slightly longer to start than DPS. These results allow us to conclude that service number and complexity do influence the adaptation time, but within reasonable bounds. They are also an indication of an easy way to optimize our framework, by simply fixing the order of events: first the suitable service is switched on, and only after that the inappropriate one is switched off.

In order to have an overview of the times (A), (B) and (C), as well as the differences between the testing devices, we present the average results of these experimentations in Fig. 7. It is important to prove that our solution is efficient on different phones running the Android operating system, and that the CATS framework behaves in a similar way on all of them. Of course, the Android version and the supporting architecture have an important influence on the execution time, but the results rest consistent.

From the results presented in Fig. 7, Time (C) is the most important one, showing the total time of adaptation. As expected, the best performance is achieved with the HTC Flyer tablet, which is able to switch the services and adapt an application in little over 90 ms. The difference between the Time (A) and Time (C) is given by the time needed by iPOJO to bind the new service to the application. The Samsung phone, the oldest of the devices, is the least performant, while still providing an acceptable result: 145 ms in average for a complete adaptation of the application. The results that have been described here represent the average results of 30 trials for each test.

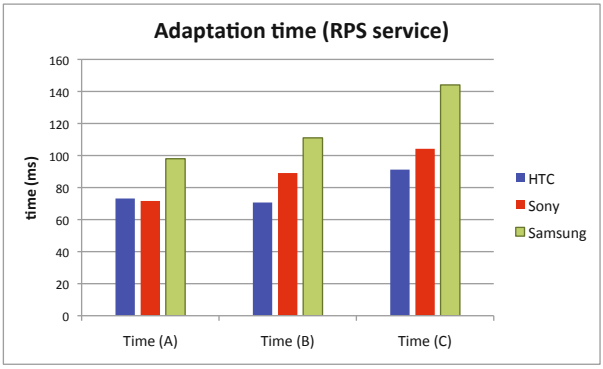


Fig. 7. From context change to an adapted application

8 Conclusion

In this paper, we propose an approach based on iPOJO handlers for our embedded application framework called CATS. This framework is dedicated to mobile devices such as smartphones, offering an execution environment for transportation-oriented applications which conform to the SOA principles. We provide adaptation by switching between equivalent services, based on context criteria. As such, an application will use the service that is adapted to the situation it is in. In this paper we have evaluated the time necessary from the detection of a context change until an adapted service is ready to use. An important part of the non-functional operations are carried out by the iPOJO handlers.

We have introduced a series of handlers to help manage the framework during the process of context detection and application adaptation. A first type of handler works with the Context Monitors, who update regularly the value of a certain element. The use of a handler has the great advantage of being able to reuse code with almost no modifications. Indeed, suppose that an existing piece of code is used to read a certain value (battery level, speed, etc.). In order to transform this into a Monitor, we must only describe the metadata of the Context Element and plug the appropriate handler to it. A second type of handler is used to cope with context dependency while keeping the service development as clean as possible. A service describes the non functional elements with the help of metadata, leaving the rest to the handler. At last, a handler was used for testing measurements, as it detects the invalidation and validation of the applications when dependencies are not resolved.

In this paper, the adaptation of applications was examined from various points of view and with different scenarios. Based on the order in which the stopping and starting actions are performed, on the number and on the complexity of the services, the adaptation time can vary, but remains reasonably fast. In order to optimize the performance of our framework, we can ensure that a context change is handled by first switching a new service on and only then switching the inappropriate one off.

References

1. Capra, L., Emmerich, W., Mascolo, C.: Carisma: Context-aware reflective middleware system for mobile applications. IEEE Trans. Softw. Eng. 29(10), 929–945 (2003)
2. Conan, D., Rouvoy, R., Seinturier, L.: Scalable Processing of Context Information with COSMOS. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 210–224. Springer, Heidelberg (2007)
3. Delot, T., Cenerario, N., Ilarri, S.: Vehicular event sharing with a mobile peer-to-peer architecture. Transportation Research Part C: Emerging Technologies 18(4), 584–598 (2010)
4. Delot, T., Cenerario, N., Ilarri, S., Lecomte, S.: A cooperative reservation protocol for parking spaces in vehicular ad hoc networks. In: 6th International Conference on Mobile Technology, Applications and Systems (Mobility Conference 2009), pp. 1–8. ACM Digital Library (September 2009)
5. Desertot, M., Lecomte, S., Popovici, D., Thilliez, M., Delot, T.: A context aware framework for services management in the transportation domain. In: 2010 10th Annual International Conference on New Technologies of Distributed Systems, Tozeur, Tunisia, pp. 157–164 (2010)
6. Abowd, G.D., Dey, A.K.: Towards a Better Understanding of Context and Context-Awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999)
7. Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the vision of autonomic computing. Computer 43, 35–41 (2010)
8. Escoffier, C., Hall, R.S., Lalanda, P.: ipojo an extensible service-oriented component framework. In: IEEE International Conference on Service Computing (SCC 2007), Salt Lake City, USA, pp. 474–481 (2007)
9. Hall, R.S., Pauls, K., McCulloch, S., Savage, D.: Osgi in Action: Creating Modular Applications in Java. Manning Publications (2010)
10. Hoareau, C., Satoh, I.: Modeling and processing information for context-aware computing: A survey. New Gen. Computing 27(3), 177–196 (2009)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
12. Maurel, Y., Diaconescu, A., Lalanda, P.: Ceylon: A service-oriented framework for building autonomic managers. In: IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, pp. 3–11 (2010)
13. Mukhtar, H., Belaid, D., Bernard, G.: User preferences-based automatic device selection for multimedia user tasks in pervasive environments. In: 5th Internat. Conf. on Networking and Services, p. 43. IEEE Computer Soc. (2009)
14. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. In: 13th International Software Product Line Conference SPLC 2009, vol. 1, pp. 131–140 (August 2009)
15. Popovici, D., Desertot, M., Lecomte, S., Delot, T.: A framework for mobile and context-aware applications applied to vehicular social networks. In: Social Network Analysis and Mining, pp. 1–12, 10.1007/s13278-012-0073-9
16. Popovici, D., Desertot, M., Lecomte, S., Peon, N.: Context-aware transportation services (cats) framework for mobile environments. International Journal of Next-Generation Computing 2(1) (2011)
17. Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing (2004)