Kalimucho: Contextual Deployment for QoS Management

Christine Louberry[1], Philippe Roose[2], Marc Dalmau[2],

[1] ASCOLA-LINA, Ecole des Mines de Nantes, 4 rue A. Kastler,
44300 Nantes, France
christine.louberry@mines-nantes.fr
[2] T2I-LIUPPA, IUT de Bayonne, 2 allée Parc Montaury,
64600 Anglet, France
{philippe.roose, marc.dalmau}@iutbayonne.univ-pau.fr

Abstract. Recently, the increasing use of mobile technologies leads to face with new challenges in order to satisfy users. New systems deal with three main characteristics: context changes, mobility and limited resources of such devices. In this article we try to address such requirements using QoS-driven dynamic adaptations of application deployment. We are particularly interested in distributed applications QoS management facing with hardware limitations and mobility of devices, user requirements and usage constraints. We propose a service-based reconfiguration platform named Kalimucho. It implements a contextual-deployment heuristic to find a configuration matching context and QoS requirements. Kalimucho was tested with the Osagaia/Korrontea component model and several devices; the results confirm that Kalimucho provides a satisfying execution time to adapt applications.

Keywords: QoS management; context-awareness; contextual deployment; dynamic adaptation; software component.

1 Introduction

The increasing use of mobile technologies leads to face with new challenges in order to satisfy people using mobile devices. As they can now take their devices anywhere, people wish to use their favorite applications as well as at home. Moreover, they wish that applications could be automatically customized according to their location, light, weather or other environmental oriented context. However, we have to deal with three main characteristics of such systems: context changes, mobility and limited resources.

The aim of this article is to answer to user needs and changes of the environment using QoS-driven dynamic adaptations of application deployment. We are particularly interested in QoS management of distributed applications facing with hardware limitations and mobility of devices, user requirements and usage constraints.

In this article, we illustrate our work with the sample use case of a tourism-centered application: the visit of a museum. Three visitors use the application running on a mobile device (a smart phone for example). The museum provides a server

hosting a video information service (fig. 1). We consider that the best QoS is to broadcast a color video. The visitor #3 moves in the museum. The platform is advised of this change and consequently estimates the quality of the video service provided to the user #3. First, the bandwidth is low but the device still reach the server. To ensure the continuity of the service, one solution is to reduce the number of transmitted data. The server transforms the color video in a black and white one. Secondly, the device cannot reach the server. The platform has to look for a new route to reach the device #3 and ensure the continuity of the service. For example, the visitor #2 device, which also uses the video service, can be a relay for the device #3.

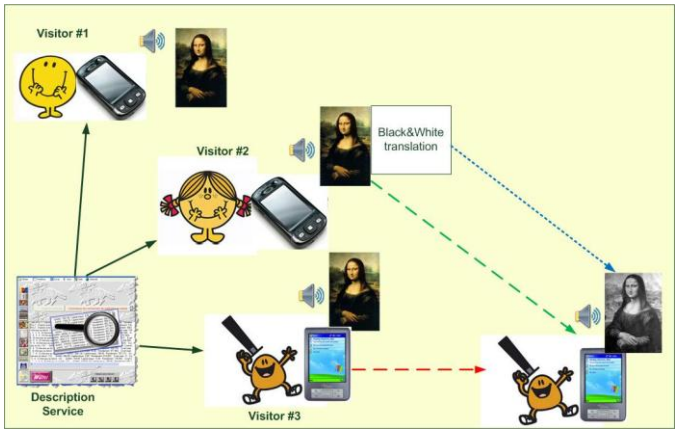

Fig. 1. Visit of a museum application

Discovering a new route to broadcast information is a classic problem, many protocols as AODV [5] can be implemented. Nevertheless, such protocol just manages physical routing constraints. To propose more interesting solutions not only based on technical criteria of feasibility, we use a supervision platform distributed on each device enabling to have a complete and global knowledge of the application. Hence, because the color video is already broadcasted to the users #1 and #2, it is possible to use one of these two users as a relay for the user #3, which cannot reach the video service anymore. However if the relay device does not have enough energy available, the broadcast of a color video is not possible. We can thus propose to install a black and white conversion component on the relay device in order to transmit the adapted video more suitable to the energy available on the device. The energy consumption depending on the quantity of data, the conversion into black and white allows to reduce the quantity of data to transmit to the device #3 and thus to preserve its energy. Furthermore distributing services allows network load balancing. Obviously such a solution requires a good knowledge of the whole application in terms of services in order to choose the best deployment. Our approach using dynamic reconfiguration and redeployment allows proposing reliable solutions from both points of view of the infrastructure and the QoS.

In the remainder of this paper, we first discuss related work (Section 2). Section 3 presents the definition of context and QoS we use in this paper. In Section 4, we present our QoS model, which addresses the utility of an application and its durability. Then, in Section 5, we describe Kalimucho, our reconfiguration platform that provide contextual-deployment of applications in order to meet QoS requirements. Finally, Section 6 presents an experimentation of Kalimucho with the Sunspot platform and we conclude and discuss directions for future work (Section 7).

2. Related work

Routing protocol is the most common solution when dealing with QoS in distributed or mobile environment [6][11]. Routing protocols allow discovering reliable routes, ensuring the continuity of service and the best bandwidth. However, such protocols are not sufficient to face with heterogeneity and context changing as well as high-level routing decision. Following this report, several works proposed software architecture to manage QoS in mobile and limited systems. Such approaches address the global problem of context-awareness to manage QoS and adapt applications.

Music [13] project propose a context-aware middleware to adapt application in mobile systems. "*Planning-based adaptation refers to the capability of adapting an application to changing operating conditions by exploiting knowledge about its composition and Quality of Service (QoS) meta-data associated to the application components*"[8]. Music considers that applications are developed with a QoS model such as utility functions. Applications are described in several variations where components are associated to QoS meta-data. The planning process chooses variations in order to maximize utility. Music is a device-oriented approach. For each adaptation, an adaptation domain is defined around the device requesting adaptation where the adaptation planner can act. Moreover, distribution plans are associated to each variation that limits the range of solutions. QuAMobile [15] is a generic QoS-aware software architecture for multimedia distributed applications. It is based on two concepts: QoS-oriented components monitoring QoS and a service planner. As Music, this service planner allows composing dynamically a configuration according to QoS requirements. This approach shows how it is important to model QoS and describe components and devices in order to provide a suitable configuration. However none of these works tackle the problem of distributed deployment. Some works such as Carisma and AxSeL [3] provide contextual deployment solutions. As well as optimize resources consumption, they allow driving deployment according to physical and/or logical dependencies. Nevertheless no one includes network communication cost in adapting applications.

3. Context and QoS

A unique definition of what the context is does not exist. The context is application domain dependant. This article addresses QoS management through context adaptation in mobile applications. Hence, context has to include particularities of such

applications: mobility, limited devices, customization according location, environment. We refer to the definition of Schilit and Teimer [14] and one of David and Ledoux [7]. Such definitions point out that the context and the QoS are linked: context changes can be seen as a QoS evolution. However, all context changes do not have the same consequence. Thus, we define three categories of context (fig. 2): user, usage and execution. The user context refers to user preferences, what service user wants to use. The usage context refers to application constraints. There are functional specifications, which define what the application must do and must not do. The execution context refers to hardware (CPU, memory, energy) and networks capabilities. These entities are traditionally monitored in context-aware systems. The context is a measure of the QoS of applications. Usually used in networks to measure the performance of transmissions according to quantitative criteria such as delay, gigue or rate of error, QoS cannot only be based on a network and hardware criterion [9]. The consideration of the users' point of view is necessary but not enough for the QoS evaluation when dealing with constraint devices. Indeed, using limited mobile devices implies to optimize the energy consumption and the way to provide applications so that the offered service fits within environment constraints and can be used for a long time.

To achieve this goal, we wish to act at the three levels (fig. 2). At the infrastructure level, we have to guarantee the *continuity of service*, whatever the evolutions of the infrastructure are and despite hardware or network failure. At the application level, we have to guarantee the *durability* of the application. Indeed, the use of limited devices raises the problem of their lifetime. Finally, at the user level, we have to guarantee the respect of user needs to provide a useful application (*utility*).

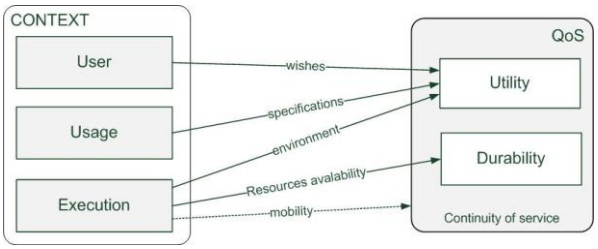

Fig. 2. QoS and context interactions

Continuity of service. Considering application QoS, the main objective is to guarantee the continuity of service despite the hardware, software and network failures. Furthermore we have to face with the heterogeneity due to the use of several types of devices, as well as hardware fails like the battery level of mobile devices.

Application durability. We wish to guarantee the continuity of service of applications running on limited devices. One solution consists in maximizing the lifetime of the application. A device without energy causes the disconnection of all the services running on it and consequently may compromise the continuity of service. [1] points out that network exchanges consume more energy than computation. Therefore, solutions based on service mobility can minimize transmissions and maximize lifetime.

Application utility. We defined the usage constraints as the functional specifications of the application that the system has to respect. For example, in the application of visiting a museum, the designer can express constraints as follows:

- When the visitor comes into a room where the conference is, avoid providing the information service with sound to not disturb the visitor.
- When it is close time, activate the guide service to drive visitors to the exit door.

4. A two dimensional QoS model

In order to make the decision to reconfigure, we have to measure the application QoS. We aim at providing useful application running as long as possible. In mobile systems we have to meet the user needs and to maximize the lifetime of devices. To minimize energy consumption, most of approaches try to choose the most suitable components according to their CPU and energy consumption. Actually, network communications on mobile devices consume 90% more energy than data processing [1]. Energy consumption is closely linked with network distribution.

Our approach does not limit lifetime management with optimizing resources consumption. We also try to optimize network balancing. We propose a two dimensional QoS model allowing optimizing utility and durability of an application.

4.1 Utility

Utility is represented as a classification of configurations. Each configuration has a mark first determined by the designer. This classification changes according to the context. Usage constraints correspond to rules, which change the utility of a set of configurations. For example, when sound is higher than 70dB, we avoid providing the sound features so that the user does not support noise disturbance. This kind of constraint is translated as an Event-Condition-Action rule:

If (sound > 70) throws *beginConference*

Event: [*beginConference*, sound, « - », 0.2]

We associate an event to a feature (*sound, video, etc.*), an operator to increase or decrease the utility mark and a coefficient. In our example, we decrease the utility of a configuration providing sound when *beginConference* event occurs. Such a rule can change the classification placing concerned configuration in a bottom position.

4.2 Durability

As utility functions minimize impact of several factors in systems [2], we use two functions to evaluate durability of an application. The first one aims at minimizing the impact of the resources consumption when deploying components on devices. Then, the second one aims at minimizing the network balancing of a deployment.

Durability depending on the resources consumption. Each component and device is represented with a 3-uple: consumption/availability of CPU (C), memory (M) and energy. Each value is expressed in percentage between 0 and 1. A component consumes no energy (CPU or memory), 0, or 100% energy of a device, 1. When a configuration is deployed, we calculate the influence of each component supported by a device (component C on device H (1)).

$$(\text{C on H}) = (C_H - C_C, M_H - M_C, E_H - E_C) . \quad (1)$$

However, we have to distinguish components that seem equivalent. For example, A(0.5, 0.2, 0.2), B(0.2, 0.5, 0.2) and C(0.2, 0.1, 0.6). If we calculate the average of the three values, we obtain 0.3 for all these components. If we deploy A on a device P, A will consume much energy and P will be rapidly out of order. B consumes much memory and P will not be able to support other component anymore. C consumes much CPU, which can slow computation. To resume, the discriminatory factor relates to the deployment impact of a component on a device and we use the min of the three values as the durability of deploying C. We name this equation QoS_RC:

$$QoS\_RC(C\ on\ H) = max(0, min(C_H - C_C, M_H - M_C, E_H - E_C)) . \quad (2)$$

If we deploy several components on a device, we add all the values of components and we finally calculate the durability of a configuration:

$$QoS\_RC(configuration) = \min\left(QoS\_RC\ (A, B\ on\ P), QoS\_RC\ (C\ on\ H)\right) . \quad (3)$$

This method to calculate QoS is not optimal because we choose a configuration that consumes little energy on two devices instead of choosing a configuration that consumes much energy on one device. This method exhausts devices little by little instead of completely exhausts devices one by one. Nevertheless, our goal is to maximize the lifetime of devices, so we want to avoid breaking up a device.

Durability depending on the network consumption. We calculate the durability depending on network consumption (QoS_NC) with a similar method than QoS_RC. For a particular deployment, we know each device supporting each component and we know all network links between components.

In a previous work, we propose a design method describing each component and device in an ID card [12]. In these ID cards, a device knows the theoretical bandwidth (BWTh) of every networks it can reach (Wi-Fi, Bluetooth, Zigbee, etc.). In our applications, software components and data flows are encapsulated into containers (called Osagaia for software components and Korrontea for data flow containers). They are composed of three entities, Input Unit, Output Unit and Control Unit, which are the main information source for the platform. When a unit detects an evolution of its context, it raises an event informing the platform and the platform can query information from containers (section 5). According to the Korrontea container, we can monitor the bandwidth of connectors during execution and calculate the average bandwidth of a connection between two devices. Then we can calculate the available bandwidth between two devices, H1 ad H2:

$$BW_{H_1H_2} = BWTh_{H_1H_2} - BWAv_{H_1H_2} . \quad (4)$$

Finally, ID cards of components indicate the output bandwidth that a component produces. So, we can apply a simple method to know the output bandwidth of a device is to sum the output bandwidth of every component it supports:

$$QoS\_NC = max\left(0, \frac{BW_{H_iH_j} - \sum^{output\ bandwidth\ of\ components\ between\ H_iH_j}}{BWTh_{H_iH_j}}\right). \quad (5)$$

4.3 QoS evaluation

We can represent applications as configuration graphs [10] where one application can be realized by one or several configurations with different QoS. For each application, configurations are classified according their decreasing utility. The Utility QoS is first determined by the designer depending on the application domain. Then the usage context defines some rules modifying the configurations QoS.

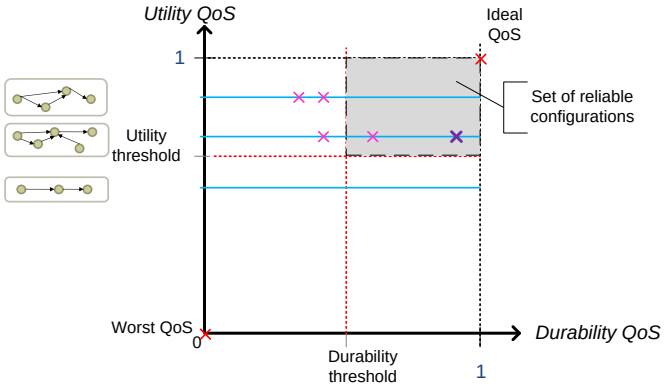

Fig. 3. QoS evaluation

For example, when sound level is higher than 70dB, we propose to avoid configuration providing sound. Hence, we apply a coefficient, which reduces QoS of such configuration, consequently, classification is modified. This classification is the base for QoS evaluation. We represent the QoS of a configuration (and its deployments) in a two dimensional diagram where X-axis represents the Utility and Y-axis represents the Durability. We place Utility and Durability thresholds, which define a set of reliable configurations (fig. 3). Such thresholds can be modified during execution in order to enlarge this set and achieve a deployment. We study each configuration of the classification from top to bottom and calculate durability of their deployments. While the QoS of a deployment is outside the boundaries, we test another deployment or configuration until a deployment meets QoS requirements. If there are two reliable deployments as in figure 2, we take the one with the best QoS.

The following paragraph presents Kalimucho, a reconfiguration platform that implements this QoS model through a heuristic for contextual deployment.

5 Kalimucho

In this article, we choose to address QoS management in mobile and constraint systems by dynamic reconfiguration of applications. Hence, we use component-based applications, which provide flexibility and modularity. They are based on a component model, Osagaia and a connector model, Korrontea [4]. These models encapsulate each component and connector in a suitable container, which allows to monitor context (activity, QoS, delay, etc.) and to control component/connector life cycle (start, stop, connection, disconnection, migration). These containers can be controlled through a control unit that is an interface with the platform. So, we can act directly on components and connectors to reconfigure applications.

To provide such quality in mobile applications, we propose a distributed QoS-aware platform: Kalimucho (https://kalimucho.dev.java.net/). Kalimucho consists in five collaborating services, distributed to all devices supporting the application in order to have a global knowledge of the system. This platform is able to trigger the context changes and to modify the structure and the deployment of the application using five basic actions: add, remove, connect, disconnect and migrate component/connector. This set of five services allows Kalimucho ensuring the four following objectives: (1) it must first be able to capture the context. Events can come from the application or the platform itself. It must have mechanisms to capture these events, interpret them and take the appropriate decision to adapt the application. These mechanisms correspond to the application monitoring and are provided by the Supervisor service. (2) When the reconfiguration decision is taken, the platform must be able to propose a reliable deployment. So it must know all the software components and devices available and test whether a deployment meets the QoS requirements. It is carried by the Reconfiguration Builder service. (3) The reconfiguration involves moving, adding and removing components. To maintain an effective application, we must ensure the reliability of the network connections between devices supporting the application. Kalimucho therefore needs a service to maintain the network topology of the application. This is the Routing service. (4) Finally, our applications can be used with any device. All devices do not have the same hardware and software resources. We must manage this heterogeneity between devices. Components must be able to run on any device. We use a component model where each component/connector is encapsulated in a container suited to the device, which free it of all non-functional properties. Lifetime of containers is limited to the use of the component. They are created when the component is installed and destroyed when the component is removed. The container must then be adapted to the device. The platform needs a service able to create specific containers depending on the component and the device. This is the Container Factory and the Connector Factory.

Depending on events, Kalimucho provides two reconfiguration processes. First, it can migrate a service: it tries to differently deploy the components of the current service. Secondly, it can deploy a new configuration of a service: it evaluates a set of configurations respecting utility and tries to find a deployment respecting durability. To find such a deployment, it implements a heuristic for contextual deployment.

5.1 Heuristic for contextual deployment

This heuristic aims at finding a configuration and its deployment meeting the context constraints and the QoS criteria: Utility and Durability. This heuristic is composed of two parts. The first one evaluates the utility criterion whereas the second one searches for a deployment optimizing durability. It plays the following scenario: For each configuration where the utility is between 1 and the utility threshold, our heuristic searches for a deployment. This deployment has to ensure durability between 1 and the durability threshold.

To limit the energy consumption, we propose to minimize the number of network links. Since there are location dependencies for some components, we try to group them on imposed devices. When they cannot support component anymore, we try to place component on a device located on paths between imposed devices (Fig.5).

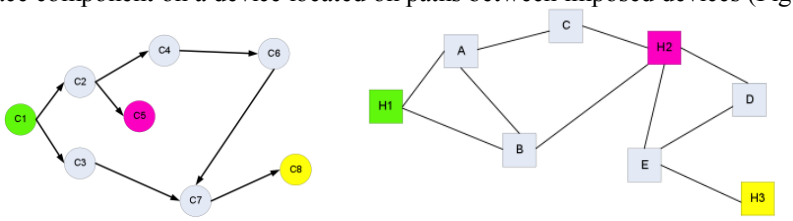

Fig. 4. (a) Graph of the "Text" configuration. (b) Network Graph

This solution is based on a device ranking (table 1), which is itself based on two concepts: component weight and device weight. We can represent configuration as an oriented graph (fig. 4(a)). Then we define paths between components that are fixed with particular devices (location dependence). We define the weight of a component as its minimal rank in such path. It is the minimal distance from the component to a fixed-component. Fixed-components are always set to rank 0. We apply the same definition to define the weight of a device but network is represented as a non-oriented graph (bidirectional paths).

From this ranking, we try to deploy each configuration until one meets QoS requirements. In order to find rapidly a deployment, we improve the initial ranking of devices with other criteria: type of a component, energy, CPU and memory:

- ***Type of component***: This ranking aims at using a maximum of non-limited devices. So, we distinguish 3 types of devices: Fixed, CDC[1] and CLDC, according to Sun Microsystems's J2ME standard classification. CLDC devices are the most limited such as mobile phones and wireless sensors.
- ***Energy***: Energy available on devices is an important criterion because it directly addresses the durability of devices and consequently the durability of the application. It may imply more reconfigurations than CPU or memory-use changes.
- ***CPU***: CPU workload is a less important criterion than energy. However, a high workload can slow the computing of components.
- ***Memory***: Memory availability has no impact on the durability of the application at this moment. It will impact when we would add component on a device.

1 According to Sun Microsystem devices type description

Table 1. Ranking of devices.

Device	H1	H2	H3	A	C	B	D	F	E
Impact	0	0	0	1	1	1	1	1	2
Type	Fixed	CDC	CDC	Fixed	Fixed	CDC	CLDC	CLDC	CLDC
Energy			0.8				0.2	0.3	
CPU				0.9	0.75				
Memory									

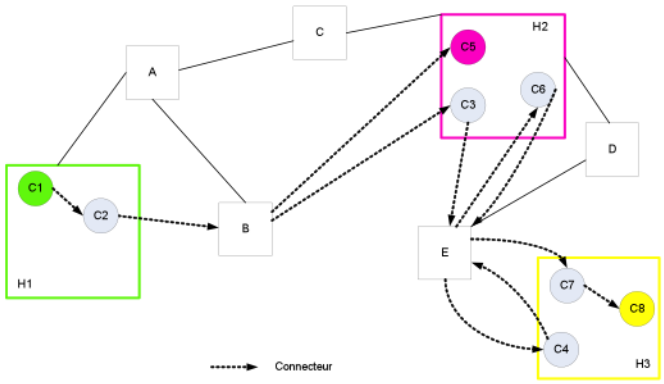

Fig. 5. Result of deploying the "Text" configuration

According to this ranking, the heuristic uses a recursive approach to calculate a deployment for each component:

1. We rank components, in a list CL, according their impact.
2. We select the first component where impact == 1
   a. If it does not exist, it means that all components have been placed.
3. We rank devices, in a list DL, according the criteria of table 1.
4. We calculate the impact of placing this component on the first device of DL.
   a. If (QoS_RC >= QoS_RC threshold) then we place the component on this device and we invoke the recursive heuristic since step 1. If the placement refers to the last component of CL, we can calculate the QoS_NC of the deployment.
   b. If (QoS_RC < QoS_RC threshold) then we go back to step 4 with the next device in DL if there is one. Else, this recursive call fails and implies a new calculus to place again the previous component.

In case of failure, we can update the QoS threshold in order to select new configurations, which can provide a deployment meeting QoS requirements. If the QoS threshold update cannot provide such a deployment and the heuristic has tested all the configurations, it means that the application cannot be adapted.

As an example, we try to deploy the "Text" configuration represented in figure 4(a) on the network represented in figure 4(b). The "Text" configuration is composed of 8 components, C1 to C8. Location dependences have been defined: C1 is placed on H1, C5 and C6 are placed on H2 and C8 is placed on H3. The result of the heuristic in

figure 5 confirms that the components are grouped on the devices already used in order to limit the number of networks links and the energy consumption.

6. Experimentations

We developed a prototype of Kalimucho running on a netbook, a PDA and two SunSPOT platforms. This implementation of Kalimucho can deploy and reconfigure components-based applications using batch files. It can also capture contextual elements necessary for decision-making. The Supervisor service relates, as an alarm, the context changes and the user requests to adapt the application. To do so, the *Supervisor service* is able to monitor the state of a device (memory, CPU, battery), to monitor the state of a component or a connector (QoS, activity, connections, etc.) and to relay statements to other distributed platforms. The Reconfiguration Builder service is able to create or remove components/connectors, migrate components, connect or disconnect a component input and delete or duplicate component output and send commands to other platforms.

Kalimucho is distributed on all devices supporting the application. We implement specific versions according to the type of device. Thus we find in our prototype:

- Kalimucho for fixed devices such as PCs and laptops (about 260Ko);
- CDC Kalimucho for devices such as smart phone (about 356Ko);
- Kalimucho for CLDC devices such as sensors Sun Spot (about 169Ko). We also implemented an Android version.

Finally, the devices do not use the same network, so we have to provide tools to relay information through the several networks. In our prototype, the PC and the Android phone use Wi-Fi although SunSpots use Zigbee. To enable the Sunspots to communicate with others devices, a base-station plugged into the PC runs as a gateway.

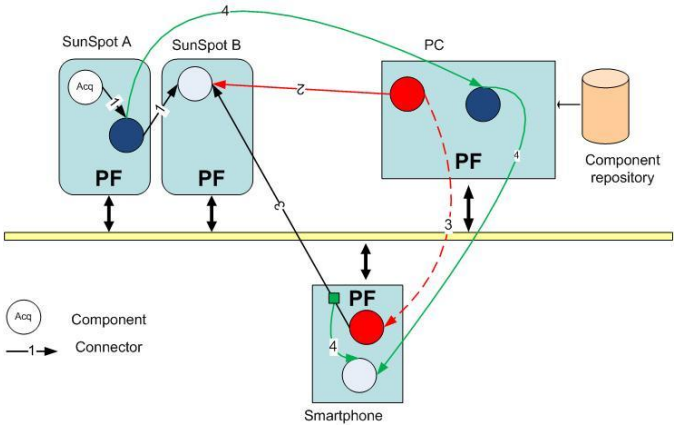

Fig. 6. Display application

We propose a scenario that where we can use the five actions providing by the platform and measure execution time of these actions. This is an application

displaying the angle captured by a SunSpot. This application is initially composed of 3 components: capture, processing and display (Figure 6). The Supervisor service of Kalimucho captures the activity of components and connectors, the QoS and the resources of devices (CPU, energy, memory). First, we replace the processing component to reverse display (1). Then, we add a new component on the PC to choose color display (2). We migrate the color component on the smart phone (3). We have to care about the network links. Hence, the smart phone has to share information with a SunSpot through the PC (gateway). Finally, we reconfigure the application to display on the smart phone too. We duplicate the output from color component to send information to the display components on the SunSpot and the smart phone (4). Table 2 resumes the execution time of all the command we used in this scenario. We can notice that the most execution times are in the order of millisecond, which is acceptable for such limited devices.

Table 2. Execution of Kalimucho commands on a SunSpot sensor and a Nexus One mobile phone.

Command	Execution time in ms	
	SunSpot	Android
Create a component	70 to 170 ms	450 to 750 ms (upload of a 2kB byte code)
Delete a component	20 ms minimum Depends on time to end the component	15 ms. Depends on component activity
Create a connector	Internal: 70 to 110 ms Distributed: 100 to 190 ms on device receiving the command, 30 to 120 ms on the other	Internal: 3 to 15 ms Distributed: 10 to 100 ms
Delete a connector	Internal: 60 to 80 ms Distributed: 100 to 260 ms on device receiving the command, 30 to 120 ms on the other	Internal: 3 to 15 ms Distributed: 3 to 25 ms
Disconnection or reconnection of an input	20 to 60 ms	2 to 7 ms
Duplication of an output	20 to 80 ms	2 to 7 ms
Read QoS of a container	80 ms	
Read state of a container	70 to 80 ms	
Read state of a device	70 to 90 ms	
Migrate a component	90 to 230 ms	650 to 750 ms (upload of a 2kB byte code)

7. Conclusion and future work

Pervasive computing is becoming a reality. Nowadays, people want to use application anywhere with its mobile device. Due to the mobility, people want applications to be adapted according to context changes. This brings new challenges to traditional

applications. As said in [16], applications should be context aware because of limited resources of the devices and the variability of the execution context. Most approaches deal with energy consumption providing planning-based adaptation or contextual deployment. However, these approaches only consider CPU and energy consumption, no one considers network communication cost.

So we propose a QoS model allowing to guaranty the utility of an application and to maximize its lifetime. The durability is a fundamental notion with mobile devices because a high quality application is useless if it just runs for a few time. Utility measures the adequacy of the provided application to user needs and application specifications. Durability measures the lifetime of the application according to resources and network consumption. Then we propose Kalimucho, a contextual deployment platform. It implements the QoS model through a recursive heuristic. We test Kalimucho on several platforms such as SunSpot and Android. Execution times of Kalimucho commands, in the order of millisecond show that response time is acceptable for limited devices.

However, it still exist an obvious limit to our approach. Although QoS can be adapted dynamically, it is based on static measure of resources consumption from components. Then, the heuristic to select a deployment only computes the QoS of the service where the reconfiguration event occurred, and not the whole application. When reconfiguring, we offer the possibility to get the best QoS for one service. The reconfiguration of one service does not imply the reconfiguration of another one. But, the modification of the application (its deployment) has consequences on the execution context because it modifies the charge of devices and the network traffic. So, a reconfiguration of one service may induce the raise of events that will trigger new reconfigurations.

Future works focus on the design and test of this configuration choice heuristic. We must specifically work on:

- Does the platform have to manage priorities on events in order to manage quicker reaction for some of them?
- When an event is managed, do we have to manage those waiting or ignore them? Doing a reconfiguration modify the context and consequently some events produced before this configuration may be obsolete.
- The QoS model manages utility QoS and durability. The importance between these to criteria depends on the application. For example, a video surveillance application needs to give priority to durability whereas the one presented for museums visits gives priority to utility QoS in order to produce good quality information corresponding to users' demands.

Acknowledgment

This work was funded by the National Research Agency under MOANO project.

References

1. Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. Computer Networks, 38(4) :393–422, 2002.
2. Mourad Alia, Viktor S. Wold Eide, Nearchos Paspallis, Frank Eliassen, Svein O.Hallsteinsen, and George A. Papadopoulos. A utility-based adaptivity model for mobile applications. In AINA Workshops (2), pages 556–563. IEEE Computer Society,2007
3. Ben Hamida, Le Mouel, Frénot, and Ben Ahmed. A graph-based approach for contextual service loading in pervasive environments. In Proceedings of the 10th International Symposium on Distributed Objects and Applications (DOA'2008), Lecture Notes in Computer Science, Springer Verlag, Monterrey, Mexico, November 2008.
4. Bouix, Roose, and Dalmau. The korrontea data modeling. In Ambi-Sys'08 : Proceedings of the 1st international conference on Ambient media and systems, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
5. Chakeres, I. D., Belding-Royer, E. M.: AODV Routing Protocol Implementation Design. In Proceedings of the International Workshop on Wireless Ad Hoc Networking (Tokyo, Japan, March 2004). WWAN'04 Springer, 2004.
6. Chen, T. w., Tsai, J. T., and Gerla Y. Qos routing performance in multihop, multimedia, wireless networks. In In Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC, pages 557–561, 1997.
7. David, P., C., and Ledoux, T.: Wildcat: a generic framework for context-aware applications. In Sotirios Terzis and Didier Donsez, editors, MPAC, volume 115 of ACM International Conference Proceeding Series, pages 1–7. ACM, 2005
8. Floch, Hallsteinsen, Stav, Eliassen, Lund, and Gjørven. Using architecture models for runtime adaptability. IEEE Software, 23(2) :62–70, 2006.
9. Franken, L. J. N., Haverkort, B. R.: Quality of service management using generic modelling and monitoring techniques. Distributed Systems Engineering. 4, 1 (1997), 28–37
10. Laplace, S., Dalmau, M., Roose, P.: Kalinahia: Considering quality of service to design and execute distributed multimedia applications, In NOMS, pages 951-954IEEE, 2008.
11. Lin, C. R., Liu, J. S., Qos routing in ad hoc wireless networks. IEEE Journal On Selected Areas In Communications, 17(8) :1426–1438, 1999.
12. Louberry, C., Roose, P., Dalmau, M.: QoS Based Design Process for Pervasive Computing Applications, ACM Mobility 2009, Nice, France (2009)
13. Rouvoy, Barone, Ding, Eliassen, Hallsteinsen, Lorenzo, Mamelli, and Scholz. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Software Engineering for Self-Adaptive Systems, vol. 5525, pages 164–182. Springer, 2009.
14. Schilit, B., N., and Theimer, M, M.: Disseminating active map information to mobile hosts. IEEE Network, 8(5): 22–32, 1994
15. Sten Lundesgaard Amundsen, Ketil Lund, Carsten Griwodz, and P˚ al Halvorsen. Qos-aware mobile middleware for video streaming. In EUROMICRO-SEAA, pages 54–61.IEEE Computer Society, 2005.
16. Zheng D., Wang J., Jia Y., Han W., Zou P. 2007. Deployment of Context-Aware Component-Based Applications Based on Middleware. UIC. Volume 4611 of Lecture Notes in Computer Science. Springer