MUSIC: an Autonomous Platform Supporting Self-Adaptive Mobile Applications

Romain Rouvoy, Mikaël Beauvois
University of Oslo
P.O.Box 1080 Blindern, 0316 Oslo, Norway
rouvoy@ifi.uio.no, beauvois@ifi.uio.no

Laura Lozano, Jorge Lorenzo
Telefónica I+D
Parque Tecnológico Boecillo, 47151 Valladolid, Spain
e.st@tid.es, jorgelg@tid.es

Frank Eliassen
University of Oslo
P.O.Box 1080 Blindern, 0316 Oslo, Norway
frank@ifi.uio.no

ABSTRACT

With the continuous improvement of device capabilities, mobile applications are becoming not only context-aware, but also self-adaptive. This new trend of applications is capable of tuning their behavior depending on changes observed in the surrounding environment. However, these adaptations are often restrained by the static features of the supporting middleware, which is not able to exploit the opportunities offered by the environment in order to improve the quality of the adaptations. The MUSIC platform offers a modular support maximizing the user satisfaction by adapting dynamically mobile applications. Beyond this support, the MUSIC platform can itself be adapted towards the evolving environment (e.g., location, network connectivity) in order to satisfy both the user requirements and the device properties. In this paper, we focus on three complementary approaches for adapting the MUSIC platform, namely i) the distribution of the adaptation process among the neighboring devices, ii) the assignment of the devices responsibilities, and iii) the adaptation of the platform components. Thus, MUSIC leverages the adaptation of mobile applications by changing autonomously the middleware characteristics.

Categories and Subject Descriptors
D.4.7 [OPERATING SYSTEMS]: Distributed systems

Keywords
Self-Adaptive Middleware, Planning-based Adaptation

1. INTRODUCTION

The current trends in software for mobile devices include the development of context-aware and self-adaptive applications. These applications are able to tune their behavior depending on the evolution of their execution context. This tuning is supported by an adaptation middleware, which is capable of reconfiguring the application whenever the device’s surroundings change. However, these adaptations remain restrained by the middleware capabilities, which are often statically defined and centralized.

The MUSIC platform offers a modular support maximizing the user satisfaction by adapting dynamically mobile applications, such as InstantSocial [5]. Beyond this support, the MUSIC platform can itself be adapted towards the evolving environment in order to satisfy both the user requirements and the device properties. This support is enabled by the use of a common component model for modeling both the mobile applications and the middleware platform.

In this paper, we focus on three complementary approaches for adapting the MUSIC platform, namely i) the self-distribution of the adaptation process among the neighboring devices, ii) the self-assignment of the devices responsibilities, and iii) the self-adaptation of the platform components. Thus, MUSIC leverages the adaptation of mobile applications by changing and distributing autonomously the middleware capabilities.

The rest of the paper is organized as follows. We first introduce the MUSIC platform, which acts as the reference platform for experimenting the autonomous organization of the adaptation process (cf. section 2). Then, we describe how platform profiles can be used for distributing the MUSIC platform among a set of colocated mobile devices (cf. section 3). We present how this support can be extended to organize dynamically the distribution of profiles (cf. section 4) as well as their self-adaptation (cf. section 5). Finally, we conclude on this experience (cf. section 7).

2. FOUNDATIONS

This section introduces the principles of planning-based adaptation (cf. subsection 2.1) and their implementation in the MUSIC platform (cf. subsection 2.2).

2.1 Principles of planning-based adaptation

Planning-based adaptation refers to the reconfiguration capability of an application to changing operating conditions by exploiting knowledge about its composition and Quality of Service (QoS) metadata associated to its constituting services [4]. We therefore consider applications that are developed with a QoS-aware service model. The QoS model associated with a ubiquitous application defines all the reasoning dimensions used by the planning-based adaptation middleware to select and deploy the service implementations that contribute to provide the best utility. The utility of an application grows when its constituting services better fulfill user preferences while optimizing device resource consumption. Figure 1 describes the QoS-aware service model we use to describe self-adaptive applications. This model is composed of two parts: a structural model and a reflection model. On the one hand, the
The structural model describes the structure of a service-oriented application and provides a similar abstraction to the Service Component Architecture (SCA) assembly model [7]. In particular, a Service Type describes not only the service capabilities, but also a set of Properties. Provided and required service types that are causally connected are grouped as a Service Port. A set of Service Ports can be implemented by an Atomic or a Composite Realization. A composite realization encloses a set of stereotyped Roles and Connectors, which are themselves associated to service ports and service type, respectively.

![Diagram of the service meta-model](Image)

**Figure 1:** Description of the service meta-model.

On the other hand, the reflection model isolates the adaptation capabilities of an application. In particular, each service plan reflects a particular component realization and its potential explicit dependencies towards other service types as well as its implicit dependencies towards the hosting platform (e.g., run-time environment type and version), which are reflected as dedicated service types. Then, planning refers to the process of selecting services that make up an application configuration providing the best possible utility to the end user. This process can be triggered during several steps of the application life-cycle, such as during the deployment of the application or at run-time if the execution context suddenly changes. When such an adaptation process is triggered for a particular service type, the planning middleware iterates over the service plans associated to the service type. For each service plan, it resolves the plan dependencies—i.e., the required service types—and invokes the Predicted Properties for evaluating the suitability of the configuration to the current execution context. The predicted QoS properties are input to a normalized utility function that computes the expected utility of a composition of plans making up an alternative application configuration [3]. This approach enables the QoS model used by the planning framework to be customized to handle various properties (e.g., performance, monetary cost), while the predicted properties can be configured to support complex heuristics (e.g., QoS negotiation protocols).

### 2.2 Architecture of the MUSIC platform

Figure 2 illustrates the architecture of the MUSIC Platform, which supports the planning-based adaptation principles. The adaptation process is coordinated by the Adaptation Manager, which is in charge of triggering the planning procedure and controlling the reconfiguration process in response to context changes. The Adaptation Reasoner supports the planning procedure by operating a generic reasoning heuristics that exploits metadata included in the available plans. The plan repository provides an interface IPlanResolver for the planner to retrieve the service plans associated with a given service type during planning. The planner may request plans that are compatible with a given service type, at which point the plan repository will search for matching service types. Any additional metadata on the required service type will help the plan repository to exclude plans and filter the search space [3]. Thus, the planner resolves recursively the service plans dependencies to build a service configuration. Configurations are automatically discarded by the planner if explicit or implicit dependencies remain unresolved (e.g., no platform is available for deploying the component). Then, the heuristics ranks the service configurations by evaluating their utility with regards to the service objectives. This evaluation is achieved by computing the offered properties using the QoS predictors associated to each plan contained in the selected service configuration and retrieved from the component QoS Manager.

![Diagram of the MUSIC platform](Image)

**Figure 2:** Architecture of the MUSIC platform.

The reconfiguration process is handled by the Configuration Executor and consists of taking the set of plans selected by the planner and reconfiguring the service. Before deploying the service configuration selected by the planner, the configurator brings the current service into a quiescence state, by suspending the execution of its contained services. If the plan describes a service instance, then the configurator connects this instance to other services, using the Service Binding. If the plan describes a composite or an atomic service, then the service should be created and deployed by the Service Factory using the blueprint description enclosed within the plan.

Plans are typically published to (and discarded from) the plan repository by i) applications, ii) service development tools using the interface IPlanManager, or iii) other plan repositories available remotely, and can thus trigger the planner for re-planning of the application if needed (e.g., the discarded plan was associated to a running service). The adaptation of a service can also be triggered by context changes detected by the Context Manager.

Our current implementation of the MUSIC Platform is based on the OSGi technology.¹ This choice is motivated by the modularity and the dynamicity of the OSGi model, while existing implementations, such as Concierge [8], exhibit a reasonable memory footprint for resource-constraint devices (80 kB).

Building on the above description, the following three sections discuss three complementary approaches for achieving self-adaptation of the MUSIC platform.

### 3. DISTRIBUTING THE PLATFORM

The first approach we propose for achieving self-adaptation of the MUSIC platform consists in dynamically organizing the adaptation process depending on the platform services available in the landscape. This approach exploits the capacity of the MUSIC platform to discover the services that are available in the landscape [10]. This support is realized by i) exporting adaptation-

¹OSGi Alliance: http://www.osgi.org
related mechanisms as services (cf. subsection 3.1), ii) defining adaptation middleware profiles (cf. subsection 3.2), and iii) exploiting the adaptation services made available (cf. subsection 3.3).

3.1 Exporting the platform as services

The implementation of the MUSIC platform with OSGi enables the adaptation components to be published as remote services by using the R-OSGi framework [9]. R-OSGi acts within MUSIC as a binding framework and uses a pure Java implementation of the Service Location Protocol (SLP) as a discovery service advertising the availability of exported services. When considering the architecture depicted in Figure 2, several OSGi components making up the MUSIC platform can be exported as remote services. Since each MUSIC component isolates a specific step of the adaptation process, publishing these components can offer several opportunities for distributing the adaptation process:

- **Context Manager** publishes the capabilities of the device (e.g., battery left, memory available) as context information. This information can be used by other devices to introspect the capabilities of the device and complete potentially missing context information (e.g., geographical location);
- **Kernel** publishes the alternative configurations that can be used to reconfigure a ubiquitous application. The plans returned by the repository can be used to reconfigure an application with a recent update of the configuration;
- **Adaptation Reasoner** enables other devices to use its planning heuristics to determine the best configuration of an application. As the MUSIC platform supports various planning heuristics (exhaustive, Bellman-Ford, constraint satisfaction algorithms [2, 3]), the other devices can exploit the associated services depending on their objective (optimal configuration, resource-saving configuration, etc.);
- **Configuration Executor** enables other devices to deploy components on the device. This enables other devices to save memory and improve performances by deploying part of the application components remotely.

3.2 Selecting the platform profile

Each OSGi component of the MUSIC platform can be exported separately depending on i) the capabilities of the device and ii) the willingness to share the device resources. Based on that, the MUSIC platform can be instantiated with all of the OSGi components or only a subset of them if the device is constraint. The minimal configuration of the MUSIC platform is basically composed of the Kernel, which isolates the MUSIC run-time environment. Based on this minimal configuration, a lightweight profile—without the Adaptation Reasoner—can be used for a resource-constrained mobile device, while a comprehensive profile can be deployed on an Internet tablet. The profile describes the MUSIC platform configuration—i.e., the list of services remotely available—of a device involved in the adaptation domain. The adaptation domain of a given device groups all the surrounding devices that can be exploited to perform an adaptation. Thus, the exported services are tagged with a particular property that describes the adaptation domain identifier and thus control the advertisement of the MUSIC platform services to mobile devices sharing the same identifier—i.e., the same adaptation domain in this case. The value of this property enables the partition of the adaptation domain depending on different policies. For example, the policy can consist in using an encrypted key as adaptation domain identifier in order to make the exported services visible to the devices owning the same key.

3.3 Integrating the platform services

The exploitation of the adaptation services available in the landscape consists in discovering alternative services for adapting the application, selecting the most suitable adaptation service, and then invoking the selected service. If the discovery process is achieved by using an SLP discovery agent, the selection process relies on the service filtering features of the OSGi framework, while the remote invocation is based on the R-OSGi framework. First, the SLP discovery agent is configured with a particular adaptation domain identifier as a service filter. Then, whenever an adaptation service is discovered in the landscape, a local proxy is registered in the OSGi service registry [9]. The component Adaptation Manager connects to the service Context Manager (local or remote) and automatically registers a context listener using the OSGi whiteboard model. When an adaptation is triggered by a context change, the Adaptation Manager connects to the Adaptation Reasoner to start planning a new application configuration. When the new configuration is planned, the Adaptation Manager then delegates to the Configuration Executor the deployment of the application. Thus, thanks to the integration of the R-OSGi framework within the MUSIC platform, the Adaptation Manager can exploit the loose coupling feature of the OSGi framework to select and invoke adaptation services in a transparent manner as depicted in Figure 3.

![Figure 3: Distribution of the adaptation process.](image)

Although efficient, this approach suffers from two limitations: i) the overall organization of profiles is not coordinated to ensure that at least an instance of each service is available in the adaptation domain, and ii) the profiles are not able to evolve when the execution context changes. In section 4 and 5, we introduce solutions for leveraging these two limitations.

4. DYNAMIC ASSIGNMENT OF PLATFORM PROFILES

This approach proposes a protocol to setup the distributed MUSIC platform by assigning at run-time specific profiles to each device of an adaptation domain in order to optimize the overall performance of the system. This protocol encloses an extensible list of profiles (cf. subsection 4.1), and the mechanisms used for the dynamic assignment (cf. subsection 4.2).

4.1 Definition of platform profiles

As the MUSIC platform targets mobile and changeable environments, it encourages the support of a dynamic platform configu-
ration to optimize the scarce resources of mobile devices. This dynamic configuration embraces two different issues: i) the conformance to the adaptation domain, and ii) the assignment of a profile to each device in the adaptation domain. The distributed nature of the MUSIC platform enables to share the workload among the adaptation domain devices by assigning a profile to each one as introduced in section 3.

A device can play two main roles in an adaptation domain: master or slave. The master, hosted by the user device, organizes and controls the adaptation domain, while a slave offers resources to the adaptation domain and remains under control of the master. The master is mandatory and unique; however there might be none or several slaves in the same adaptation domain. A slave can provide resources for applications and/or for the platform services; in the latter case, the slave relieves the master of hosting such services and the overall performance can be improved. For the time being, we have identified the following set of relevant platform profiles:

- **Standalone Master (SM):** the adaptation domain is composed exclusively by the master device and there is no slave device. This situation is typical when the network is disabled. All the platform components are deployed in the master device;
- **Full Master (FM):** this profile is very similar to SM although, in this case, there are slaves in the adaptation domain;
- **Light Master (LM):** this master profile imports the adaptation reasoner service from a slave device. The master device gets rid of this highly computational task;
- **Application Slave (AS):** a slave device with this profile offers resources to host application components and services. This profile is required in the MUSIC distributed applications;
- **Adaptation Reasoner Slave (ARS):** this slave device computes the best configuration for the applications in the adaptation domain on behalf of the Light Master profile;
- **Context Slave (CS):** this kind of device interacts with the distributed context information in order to provide new context information (also known as context provider) or to infer additional context information (also known as context reasoner);

The assignment of profiles is realised by the master device in order to simplify the approach, because the master is unique and thus a negotiation protocol is not required for assigning this responsibility. Two main rules are considered during this assignment: i) only one master is permitted in the adaptation domain—i.e., Standalone Master, Full Master, and Light Master profiles are incompatible—and ii) an Adaptation Reasoner Slave profile is mandatory if the device plays the Light Master profile. Thus, Table 1 lists the platform components required for each profile.

### Table 1: Platform components required per profile.

<table>
<thead>
<tr>
<th>Profile Assigner</th>
<th>SM</th>
<th>FM</th>
<th>LM</th>
<th>AS</th>
<th>ARS</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptation Manager</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Adaptation Reasoner</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Config. Executor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Context Manager</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kernel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 2: QoS model of the required resources per profile.

<table>
<thead>
<tr>
<th>SM</th>
<th>FM</th>
<th>LM</th>
<th>AS</th>
<th>ARS</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>400</td>
<td>550</td>
<td>350</td>
<td>70</td>
<td>300</td>
</tr>
<tr>
<td>CPU</td>
<td>400</td>
<td>400</td>
<td>200</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td>Battery</td>
<td>30</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Network</td>
<td>0</td>
<td>80</td>
<td>100</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

#### 4.2 Assignment of platform profiles

The **Profile Assigner** is deployed on the master device and is responsible for the consistency of the adaptation domain.

**Discovery of devices.** By using the discovery mechanisms described in section 3, the master device discovers the devices belonging to the adaptation domain. The resources of the discovered devices are retrieved from the service reference associated to the device or by the remote Context Manager services. This resource metadata helps the Profile Assigner to decide the most suitable distributed configuration. There are some important considerations: i) the application models determine the maximum number of slaves required in the adaptation domain, ii) some devices are not discovered, but known from the master as part of the adaptation domain, and iii) a device (discovered or not) can provide specific resources or services or be constrained to one profile.

#### Utility evaluation

Utility functions are a relevant mechanism to evaluate the most adequate configuration of the adaptation domain. The best device to perform a profile scores the highest utility regarding the set of resources defined in the QoS model and required to support this profile and the device’s resources:

\[
\text{Utility}_{\text{profile}} = \begin{cases} 
0 & \text{if } \forall q \in \text{QoS} : q_{\text{avail}} < q_{\text{req}} \\
\sum_{q \in \text{QoS}} \left( \frac{q_{\text{avail}}}{q_{\text{avail}} + q_{\text{req}}} \right) & \text{otherwise}
\end{cases}
\]

**Profile assignment process.** The Profile Assigner performs the following steps:

1. **Selection of master device profile.** The utility function\(^3\) is applied to the master device for the 3 master profiles: Standalone Master, Full Master or Light Master. If the Standalone Master profile is the final choice, there is no slave in the adaptation domain and the process is completed;

2. **Assigning the predefined profiles.** The devices, which are statically configured to perform a specific profile, maintain this profile. It might be preconfigured by the application models or by the publication of the resources provided by a device;

3. **Selection of the Adaptation Reasoner Slave.** If the master was assigned a Light Master profile, it is required an Adaptation Reasoner Slave. If it was not a predefined profile, the utility function is applied among the available devices and the device with the highest utility value is selected. If it was not possible to find a device with an utility higher than 0, the master device becomes Full Master;

4. **Assigning the application slaves and context slaves.** The required number of application slaves and context slaves is obtained from the application models associated to the adaptation domain. If they do not belong to the preconfigured profiles, the utility function is evaluated among the rest of the discovered devices in order to cover the requested number of slaves.

At the end of the process, only the assigned devices belong to the adaptation domain.

#### Platform reconfiguration

The **Profile Assigner** component propagates each profile by the Kernel component imported from each device of the adaptation domain. The platform is reconfigured by using two standard OSGi services: i) **Declarative Services** which activates or deactivates the platform components depending

---

\(^3\)The utility function for the master profile could be enriched by including additional QoS dimensions—i.e., the network cost or user preferences.
on the assigned profile; each platform component declares these profile dependencies, and ii) Configuration Executor which enables a finer configuration of each platform component; the Kernel might modify the properties of a platform component to adapt it to the new profile.

**Resource arbitration.** Another important issue related to the profiles assignment is the device resources arbitration of the adaptation domain. In order to avoid that the Profile Assigner appropriates all the available resources of a device, a mechanism is necessary to lend and withdraw resources inside the adaptation domain. It can be easily accomplished by defining policies based on different aspects, such as i) the importance of the master to a device, ii) the other adaptation domains the device belongs to, iii) connectivity issues, and iv) work load.

**Triggering of a self-adaptation.** Policies also specify when the Profile Assigner has to trigger an adaptation and how to carry out this adaptation. The events triggering an adaptation are:

- **A device performing a profile becomes unavailable:** the Profile Assigner tries to replace this profile or reconfigures the adaptation domain if no substitute is found;
- **Profiles conflict:** the Profile Assigner decides which device performs the profile;
- **Performance degradation:** in this case, it will be necessary to reassign profiles according to the policies, in order to enhance the performance of the adaptation domain.
- **A node becomes available:** if the node has sufficient resources, the Profile Assigner triggers a profile reassignment;

## 5. ADAPTING THE PROFILES

The objective of this approach is to support the self-adaptation of the platform profiles that are assigned. This is achieved by considering the MUSIC platform as a self-adaptive context-aware application. This means that the MUSIC platform can dynamically reconfigure itself to replace existing platform components or deploy new ones depending on context changes (cf. subsection 5.1). This support is realized by exploiting the existing planning-based adaptation middleware while applying some platform-specific rules (cf. subsection 5.2).

### 5.1 Examples of platform adaptations

As introduced in section 2, MUSIC uses the same component model to design both the self-adaptive context-aware applications and the supporting middleware platform. This originality thus enables the MUSIC platform to consider itself as an adaptation target. By doing so, the platform is able to optimize its behavior depending on application requirements and context changes. For the time being, the relevant adaptations we identified refer to i) the replacement of a component realization, ii) the re-organisation of a composite component, and iii) the deployment of application-specific middleware components.

**Replacing a component realization.** As introduced in Figure 2, the component Adaptation Reasoner has been designed to support different heuristics for selecting the best application configuration (brute-force and greedy algorithm [3], constraint programming [2], etc.). Each heuristics is implemented as a specific realization of the component Adaptation Reasoner (cf. Figure 4).

Each realization is qualified by a set of QoS dimensions that reflect the non-functional behavior of the heuristics. These dimensions are summarized in Table 3 and are used as a basis for selecting the planning heuristics to execute depending on the current context. For example, the following utility function can be used to select the adaptation reasoner realization that provide a robust and exact solution under a linear processing time, while minimizing the memory consumed during this process:

$$ Utility_{AR}(mem\_avail) = \begin{cases} 0.0 & \text{if mem} - \text{avail} \leq \text{mem} \text{exact} \cdot \text{proc} \times 100 \\ \text{mem} & \text{otherwise} \end{cases} $$

![Figure 4: Possible realizations for the Adaptation Reasoner.](image)

Table 3: Specification of the reasoning QoS model dimensions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>rob</td>
<td>Heuristics gives a robust solution</td>
<td>(0), yes (1)</td>
</tr>
<tr>
<td>exact</td>
<td>Heuristics gives an exact solution</td>
<td>(0), yes (1)</td>
</tr>
<tr>
<td>proc</td>
<td>Heuristics processing time</td>
<td>(0), exponential, linear (1)</td>
</tr>
<tr>
<td>mem</td>
<td>Memory required by the heuristics</td>
<td>0–100</td>
</tr>
</tbody>
</table>

**Reconfiguring a composite component.** Another example of platform adaptation consists in adding new components for supporting an additional feature. In [10], we present exhaustively the extension of the MUSIC platform to support the planning of Service-Oriented Architectures (SOA). This feature is isolated as a component SOA Composite, which is responsible for discovering, negotiating, and binding with remote services available in the landscape. In terms of adaptation, the MUSIC SOA Platform depicted in Figure 5 represents an alternative configuration of the MUSIC Platform introduced in Figure 2. The MUSIC platform can therefore switch between configurations supporting or not the discovery of remote business services when needed (e.g., the battery level becomes too low). Furthermore, the concrete realization of the component SOA Composite has to be selected by the MUSIC platform to enable the interaction with the appropriate service technology (e.g., Web Services, CORBA, JavaRMI, OSGi) [10].

![Figure 5: SOA configuration of the MUSIC platform.](image)
Deploying application-specific platform components. Finally, MUSIC supports also the dynamic deployment of application-specific platform components. This type of adaptation has been applied to the deployment of context sensors (e.g., resource sensors, application sensors) and run-time environments (e.g., Smalltalk, Fractal, Groovy, or AspectJ execution supports) [6]. These platform components are selected and deployed within the Context Manager and the Kernel in coordination with the application components via the definition of implicit dependencies (cf. subsection 2.1).

5.2 Support for the platform self-adaptation

This reflective support for self-adaptation requires to take into account specific issues related to bootstrapping, planning, and reconfiguring the platform.

Bootstrapping the MUSIC platform. This step corresponds the initial deployment of the platform, which can be operated in two ways. A minimal MUSIC platform can be instantiated and triggered to plan and deploy a new configuration of the platform, which fits to the device properties. A remote MUSIC platform can be requested to plan and deploy an optimal configuration of the MUSIC platform for the current device.

Planning a new platform configuration. When triggered by a context change, the adaptation reasoner requests the plans associated to the MUSIC platform components and builds the alternative platform configurations. The heuristics compares the respective utility functions and selects the configuration with the best score. For example, the heuristics may choose between a MUSIC platform configuration providing the SOA support and a simple planning heuristics realization, or a configuration with an efficient planning heuristics implementation, but no SOA support.

Reconfiguring the MUSIC platform. The reconfiguration of the MUSIC platform can be operated in two different ways, similarly to its bootstrapping. If the local Configuration Executor executes the configuration of the platform, then the Kernel and the Configuration Executor cannot be replaced by the reconfiguration as these components are central to the platform. To leverage this limitation, the Adaptation Reasoner can trigger a remote Configuration Executor to operate the reconfiguration, thus allowing a comprehensive reconfiguration of the MUSIC platform.

6. RELATED WORK

Related work has been done in the research of adaptable mobile systems. SATIN [11] is a middleware for adaptive applications where logical mobility paradigm is applied to redistribute components when a reconfiguration takes place. In MUSIC, we rather propose profiles assignment: a less complex and less consuming resources solution that avoids code transfers at run-time.

Other research related work [1] carries out adaptation by analogy with control systems. When an adaptation is triggered, a services dependency model is used to test several designs until a successful design is reached. Introducing utility functions the best configuration is selected according to the available resources but avoiding this heavy test and fail approach. We aim for a simpler approach to middleware self-adaptation, feasible to be deployed in a resource constrained device and with minimal impact in the existing architecture, only adding a new component, the Profile Assigner.

7. CONCLUSION

Nowadays mobile applications are not only context-aware but also self-adaptive. Self-adaptation thus enables the mobile applications to evolve dynamically depending on changes observed in their environment. Nevertheless, we observe that these adaptations are often constrained by the capabilities of i) the supporting platform and ii) the mobile device.

MUSIC is a middleware platform supporting the adaptation of mobile applications, but opposed to existing platforms, the MUSIC platform is able to adapt itself for improving the adaptation quality. In this paper, we describe three complementary approaches used by the MUSIC platform to optimize its behavior according to the surrounding environment and the application requirements. This first approach enables the publishing and the discovery of platform services available in the adaptation domain. This enables MUSIC to distribute the adaptation process across a set of platform profiles. These profiles are dynamically organized by the second approach, which assigns each required platform profile to a mobile device available in the adaptation domain. Finally, the third approach enables the MUSIC platform to reconfigure the profile it is assigned by itself, thus offering new alternative services for distributing the adaptation process. In a near future, these approaches will be validated by the industrial pilot applications of the MUSIC project.

Acknowledgements

The authors thanks to MiNEMA and partners of the MUSIC project for valuable comments; and in particular Ulrich Scholz (European Media Laboratory GmbH) and Svein Hallsteinsen (SINTEF). This work was partly funded by the European Commission through the project MUSIC (EU IST 035166), see http://www.ist-music.eu.

8. REFERENCES