A Pluggable and Reconfigurable Architecture for a Context-Aware Enabling Middleware System

Nearchos Paspallis[1], Romain Rouvoy[2], Paolo Barone[3], George A. Papadopoulos[1], Frank Eliassen[2], and Alessandro Mamelli[3]

[1] University of Cyprus, P.O. Box 20537, 1678 Nicosia
Cyprusnearchos@cs.ucy.ac.cy, george@cs.ucy.ac.cy
[2] University of Oslo, P.O. Box 1080, 0316 Oslo
Norwayrouvoy@ifi.uio.no, frank@ifi.uio.no
[3] HP Italy, Via G. Di Vittorio 9 20063 Cernusco sul Naviglio, Italy
paolo.barone@hp.com, alessandro.mamelli@hp.com

Abstract. Context awareness is a core feature of modern mobile and ubiquitous computing systems. Although it has not reached its full potential yet, one can already observe significant activity in the area of software engineering for supporting the development of context-aware applications. An example of such an activity is the MUSIC project, which proposes a middleware featuring a generic and reusable context management system. This paper describes the pluggable architecture of this system, and explains how it advances the state of the art through its support for context heterogeneity and better resource utilization. The former is achieved with the use of a novel architecture, which enables the separation of low-level, platform-specific context plug-ins from higher-level application-specific ones. The improved resource utilization is achieved through intelligent activation and deactivation of context plug-ins based on the needs of the active applications. The proposed approach has been experimentally evaluated and the results indicate that it significantly improves the resource utilization in context-aware applications, especially when deployed on lightweight mobile devices.

Keywords: Context Awareness, Middleware, Architectures.

1 Introduction

Context awareness has always been considered as an important aspect of modern mobile and ubiquitous computing systems. In the past few years, the software engineering community has committed significant efforts in the design and development of context-aware applications [1, 3, 4, 5, 8]. Some of these efforts have focused on the design of reusable middleware [10, 11, 12, 13], aiming to automate much of the required development effort. An example is the *Mobile Users in Ubiquitous Computing Environments* (MUSIC) project [14] which attempts to support the development of context-aware, self-adaptive applications by providing results in two main directions. First, a development methodology is provided, supported by software development tools allowing the design and implementation of context-aware, self-adaptive applications using the *Model Driven Development* (MDD) paradigm. Second, it provides a middleware implementation which is deployed between the *Operating System* (OS) layer of the target device and the context-aware applications, with the purpose of automating much of the context-awareness and self-adaptation aspects of the latter by enabling reusability of the corresponding mechanisms.

One of the main features of the MUSIC middleware is its provision of a generic, context management system, capable of handling arbitrary types of context, and being able to connect to arbitrary numbers and types of context providers and consumers. This paper describes the architecture and the features of the MUSIC context system, with a special emphasis on its pluggable architecture. It is argued that as a result of its pluggability, the proposed architecture provides a high degree of context heterogeneity along with high resource utilization. These features are highly important in the area of mobile and ubiquitous computing because, typically, the devices involved in such environments are lightweight mobile and embedded devices, and consequently vary in both technologies and capabilities.

The rest of the paper is organized as follows. Section 2 provides the foundations and describes the architecture of the MUSIC middleware with emphasis on its context system. Then, in Section 3 we present the pluggable architecture along with the mechanisms responsible for the dynamic resolution and activation of the plug-ins. Following that, Section 4 presents the approach and the results of the experimental evaluation of the architecture. Finally, Section 5 discusses related work and compares our approach with it before the paper is concluded with Section 6.

2 Foundations of Context-Awareness and Self-adaptation

By context-aware, we refer to systems that are capable of sensing their context and reacting based on their knowledge on it. In our approach, we follow one of the most cited definitions of context, provided by *Dey* in [1]: "*[context is] any information that can be used to characterize the situation of an entity; [where] an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*". This definition was chosen as it is general enough while at the same time it encapsulates the essence of mobile and pervasive computing environments: the most important context is that which characterizes the immediate environment where the user and a [computing] system interact, including the user and the system themselves.

As context-aware systems are capable of reacting to changes, they are considered important by mobile users and users in ubiquitous computing environments, because of the high variability which naturally characterizes them. Systems are capable of reacting to context stimuli in many ways. From a software perspective, a common method of response is the *software adaptation* [2]. Generally, software systems react by means of parameter-based or compositional adaptation, where the former refers to changes to a specific parameter of the application and the latter refers to more comprehensive changes to its composition, possibly by altering its architecture and replacing part of its functionality [8].

To enable self-adaptation, context-aware systems typically implement a control loop where the situation—*i.e.*, the context—is periodically evaluated and the adaptations are decided autonomously—*i.e.*, the user is not involved in the decision control loop. Different approaches exist for the implementation of such control loops, many of which build on results from the artificial intelligence domain: *action*, *goal,* or *utility function* [3]. For the purposes of this paper, it is assumed that utility functions are used in the control loop to assign a fitness score to each of the possible adaptation options by evaluating the context conditions [4]. That score—*i.e.,* the utility—is used to rank all the adaptation options and thus to select the most appropriate one. Eventually, when an adaptation is selected—*i.e.,* because a variant providing a higher utility than the one currently in use is found—the system is reconfigured accordingly.

One of the most important novelties of this approach is that it enables the design of context-aware, self-adaptive applications with reusable components, by separating the concern of designing the functional aspects of the application from the concern of supporting context-aware, self-adaptive behavior [5].

2.1 Open Context Management System

The MUSIC middleware architecture defines a number of components interacting with each other to seamlessly enable the self-adaptive behavior of the deployed applications. These components include the *context management*, the *adaptation reasoner*, and the *life-cycle management* (which is based on OSGi). This paper focuses on the design of the *context management* component, with emphasis on its pluggable architecture.

One of the main goals of this architecture is to allow the development of context-aware, self-adaptive systems with a consistent, easy to follow methodology. In this regard, we designed an architecture that allows the development of the context sensing parts of an application independently of its context consuming parts. With this approach, when a context aware application is designed, its functional logic is defined independently of its context sensing logic. While designing the business logic of the application, its context dependencies need only be defined explicitly in a context query, or implicitly through a utility function—*i.e.*, defined in its composition plan [8]. The actual context sensing and processing can be defined independently via plug-ins, which the middleware is responsible to manage. These context plug-ins are annotated with properties describing their *required* and *provided* context types. Thus, it is possible to dynamically and autonomously resolve context plug-ins and also to seamlessly manage their life-cycle—*i.e.,* through underlying middleware.

The design of the context architecture is based on a mutual context space, where data is stored and accessed via well-defined interfaces. The exact formatting of the context data in the context space is specified by a context ontology, which guarantees that independent context providers and consumers can seamlessly interoperate within the architecture. This is achieved, because the context ontology provides information not only of the semantics of the abstracted context, but also of their representation—*i.e.*, how they are modeled as data-structures. The MUSIC context model is described in details in [6]. In this architecture, the context plug-ins are dynamically attached to the mutual context space, and they push their sensed context data into the space according to their internal logic. Optionally, the plug-ins can also access context data directly from the context space in order to process it and generate higher-level context data in return. Explicit access to the context data is provided by means of well-defined interfaces enabling both synchronous and asynchronous access to the context data. These interfaces are complemented with a *Context Query Language* (CQL) [7], which provides a powerful and expressive, SQL-like language for specifying and accessing context data. Finally, the context space is attached to a context storage—*i.e.,* a database—which allows for storing and accessing of historical context data. To prevent this storage from overflowing, its content is periodically garbage collected to allow replacement of old and obsolete context values by new ones.

2.2 Middleware-Supported, Context-Aware and Self-adaptive Applications

As already explained, the MUSIC middleware provides support for the deployment of component-based, context-aware and self-adaptive applications. In this respect, the applications do not explicitly encapsulate any adaptation logic nor do they include any adaptation mechanism themselves. Rather, the *adaptation reasoning* (decision making) and *reconfiguration* (parameter-based and compositional reconfiguration) are automatically handled by the underlying middleware. This is achieved by means of specially annotated, *composition plans*.

As it is explained in [8], plans in the form of architectural models can be used to model the possible adaptations that an application can undergo while at the same time maintaining its functional properties. These plans formalize, in a recursive manner, the possible ways in which an application can be composed. Thus, the middleware automatically computes a set of possible adaptations (in the form of component-compositions) that each application can apply. This process can be triggered during several steps of the application life-cycle, such as during the deployment of the application or at runtime if changes in the execution context require so.

In MUSIC, this decision-making process is driven by the *Quality of Service* (QoS) metadata associated to the application components [4, 8]. MUSIC therefore considers applications that are developed with a QoS-aware component model, which defines all the reasoning dimensions used by the adaptation reasoner to select and deploy the component implementations providing the best utility. The utility of an application increases when its components better fit the user preferences while minimizing the resource consumption. Thus, this utility can be considered as a function that measures the *fitness* of a plan with regards to the current contextual conditions. Once this evaluation is applied to all possible alternatives, the adaptation reasoner selects the one that provides the highest utility. A detailed example of how utility functions are used to compute the fitness of alternatives based on their context is presented in [8].

The MUSIC middleware follows a layered-approach for interpreting context changes into adaptation decisions, as illustrated in Fig. 1. At the lowest level, the hardware, context is retrieved using physical context sensors or OS calls. For example, the *memory* and *CPU availability* are computed in coordination with the OS, which typically maintains such statistical information. Other context types, such as the *GPS coordinates*, are typically retrieved from hardware-based sensors (*e.g.*, a GPS Module attached to the device in the case of this example).

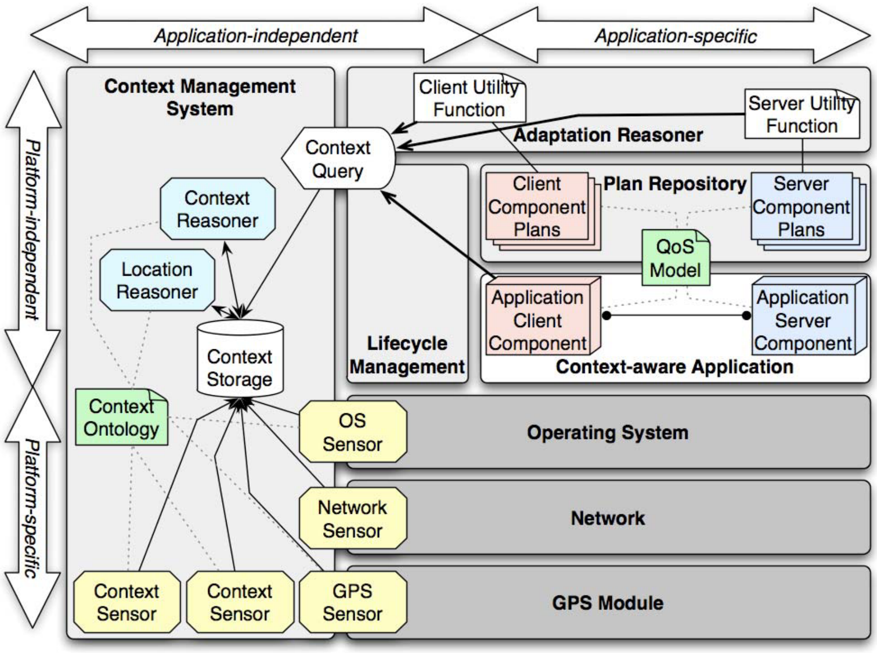

Fig. 1. Layers of Context Reasoning in Self-Adaptive Applications

In order to make that information accessible and understandable to the rest of the middleware, specialized Context Sensor plug-in components are used. These sensors can in essence be thought of as wrappers around the low-level sensing machinery or code, similar to the software drivers used to enable interfacing with PC peripherals. The MUSIC middleware supports the control and introspection of context plug-ins via the Lifecycle Management module and by implementing a special IContextPlugin interface, respectively. The activation and deactivation of sensors is typically delegated further down to the actual physical sensor. For example, in the case of an OS Sensor, which periodically gets updates on memory availability, deactivation would cause it to stop poking the OS, thus saving on CPU cycles. Alternatively, when a hardware sensor—*i.e.*, a GPS Sensor—is involved, disabling it results to the complete shut down of the device, thus saving battery power. The introspection refers to the capability of accessing metadata associated to context plug-ins. These metadata provide information concerning the context types and the QoS the plug-ins provide and require. As it will be discussed in the following paragraphs, introspection is an essential requirement for enabling the intelligent and autonomous activation and deactivation of the sensors.

Further up in the context system's hierarchy are the Context Reasoners. These are specialized plug-ins, which do not directly interact with low-level resources—*i.e.*, OS and hardware—but rather they compute higher-level information by processing the provided low-level context data—*i.e.*, collected from sensors. For example, a context reasoner could process the location context data provided by a GPS sensor along with the entries in the users' agenda to assess their occupation (*e.g.*, driving, sleeping, attending a lecture, etc.).

Similar to context sensors, the context reasoner plug-ins can be introspected and also they can be deactivated and activated on demand to optimize their resource consumption. Furthermore, they are also associated to metadata—*i.e.*, provided and required context types and QoS—enabling the middleware to assess when they are needed and when they are not. The two main differences between context reasoners and context sensors is that the former deal primarily with higher-level context information—*i.e.*, the user is driving versus the user is located at coordinates [x, y]—and that the former also do not have any dependencies on hardware or software, platform-specific components.

Further up the layered architecture of Fig. 1 are the *Utility Functions*. These are artifacts used by the Adaptation Reasoner to compute the fitness of a component, a composition or an application to some given context conditions. As such, the utility functions are typically expressed as mathematical formulas (or small programs), which combine the values of specific context types with QoS metadata of the relevant components in order to compute numerical values. Normally, the utility functions depend on higher level context types, but in some cases they can also depend directly on lower-level context types.

The utility functions are at the higher level of adaptation reasoning, available to the middleware. They are right above the Context-aware Application in the presented hierarchy, but the applications are themselves unaware of both the adaptation logic and the utility functions. This separation of concerns is achieved by the Plan Repository, which maintains metadata associated to the application components.

From the developers' perspective, the adaptation reasoning is designed and implemented throughout the layers of the presented hierarchy. Although the context reasoners could theoretically be kept outside the adaptation reasoning loop, in practice they are at the core of the adaptation decisions. The reason is that they enable interpretation of low-level context data to higher-level context data which can more directly influence the adaptation decisions. Most importantly, the context reasoners are application-oriented, and thus platform independent, as opposed to context sensors which are generally tightly coupled to the underlying hardware. This separation between platform-independent context reasoners and platform-specific context sensors is one of the main advantages of this architecture, as it will be argued in the following sections.

3 Design of a Pluggable Context Architecture

The architecture of the MUSIC context management system is based on two main abstractions: *context sensors* and *context reasoners*. Both of them correspond to pluggable code artifacts implementing the `IContextPlugin` interface. This allows for introspection and life-cycle control using the *Inversion of Control* pattern. In more detail, this interface specifies methods for activating and deactivating the individual plug-in components, as well as for accessing its associated metadata (even when inactive). Typically, the activation and deactivation methods delegate the events to the underlying machinery (*e.g.*, in the case of hardware sensors) or detach themselves from a thread (*e.g.*, in the case of a reasoner periodically probing a context value).

The plug-ins' metadata are defined at design-time from the context sensor (and reasoner) models. Most notably, the plug-in metadata reflect information on the context types provided and possibly required by the corresponding plug-in. In addition, QoS metadata can also be associated to plug-in components in a way similar to the one used for application components (*e.g.*, freshness, accuracy, resource consumption, etc.). The plug-ins are packaged in self-contained files (typically JAR archives) and, optionally, along with context-aware, self-adaptive applications. This is particularly useful in the case of application-specific context reasoners, because the latter are platform-independent (just like the context-aware applications) and thus can be packaged once and be reused multiple times on different devices (cf. Fig. 1).

At deployment-time, the plug-in components are automatically identified and their metadata are registered within the context management system. Besides the provided context types of the plug-in, the context system also evaluates its context needs. This is required, because in the case of context reasoners, which dynamically synthesize complex context information based on more elementary types, it has to be verified that the needed context types are available. This is achieved using a context dependency resolution algorithm.

Finally, the context needs of the deployed applications are continuously evaluated at runtime by the context management system. Based on the running context needs, the required context plug-ins are activated, while the rest of them are deactivated to minimize the resource consumption. As new context-aware applications are deployed, started, stopped and removed, the Adaptation Reasoner communicates the updated context needs to the context system which dynamically activates and deactivates the deployed plug-ins accordingly. The exact data-structures and algorithms involved in this process are further described in the following subsections.

3.1 Resolution and Activation Mechanisms

The pluggable architecture of the context system is based on a fundamental idea: the context information is provided by *context plug-ins* and consumed by *context consumers*. Both of these entities can be deployed dynamically and there are no guarantees concerning the availability of either. Even further, the plug-ins and the context consumers can be developed by different teams with no prior knowledge of each other. In this case, consistency over the format and the semantics of context types is achieved by means of a common ontology as described in [6].

To handle the dynamic availability of context plug-ins and context consumers, the context system monitors both and react on events involving changes to their availability. For instance, when a new context plug-in is installed or an existing one removed, the context system must react accordingly. Similarly, when new context consumers—*i.e.,* context-aware applications—are started or existing ones are stopped, the context system must also react to ensure that the appropriate context plug-ins are activated and deactivated accordingly. More accurately, the context system attempts to reconfigure the set of active plug-ins only when the needed context types change.

To handle the dynamic and autonomic resolution and activation of the context plug-ins, we define two algorithms. The invocation of the first algorithm, the *dynamic* *resolution*, is triggered by changes to the availability of context plug-ins: when new plug-ins need to be installed or existing ones should be removed. It can be assumed that changes to the availability of context plug-ins at runtime are reasonable because, often, the developed context-aware, adaptive applications are bundled with their required context reasoner and perhaps even their needed context sensor plug-ins. The invocation of the second algorithm, the *dynamic activation and deactivation* of the plug-ins is merely depended and triggered by changes to the context needs. Typically, the context needs change when new applications are started or existing ones stopped.

Table 1. Basic data-structures and algorithms used in the resolving mechanism

Resolution mechanism

Basic data-structures

[all plug-ins] – contains all known (*i.e.* installed) context plug-ins; this set is updated before the invocation of the algorithm

[resolved] – a subset of the [all known plug-ins] set, it contains only those plug-ins that have been found to be resolved; this set is updated during the invocation of the algorithm

[offered] – a mapping of all offered context types to a list of resolved plug-ins offering them

Triggered by

Changes to the [all plug-ins] set

Algorithm

/* first make sure that all resolved plug-ins are included in the [resolved] set */

```
changes-detected ⇐ true;
1. while changes-detected do
  changes-detected ⇐ false;
  1.1. for each p in [all plug-ins] do
    1.1.1. if requiredContextTypes(p) ⊆ [offered] then
      [resolved] ⇐ [resolved] ∪ {p}
      [offered] ⇐ [offered] ∪ {offeredContextTypes(p) → p}
      changes-detected ⇐ true;
    end if
  end for
end while
```

/* next make sure that no unresolved plug-ins are included in the [resolved] set */

```
changes-detected ⇐ true;
2. while changes-detected do
  changes-detected ⇐ false;
  2.1. for each p in [resolved] do
    2.1.1. if not requiredContextTypes(p) ⊆ [offered] then
      [resolved] ⇐ [resolved] - {p}
      [offered] ⇐ [offered] - {offeredContextTypes(p) → p}
      changes-detected ⇐ true;
    end if
  end for
end while
```

The first algorithm, depicted in Table 1, is used to resolve the context plug-ins. A resolved plug-in is one which either has no context dependencies—*i.e.,* context requirements—or all its dependencies are provided by some other resolved plug-ins. The resolution mechanism defines three data-structures: the *all plug-ins* set, the *resolved* plug-ins set, and the *offered* context types map. The first is a set containing all known—*i.e.,* installed—plug-ins. The second set is a subset of the first one, and it contains all the plug-ins that are found to be resolved. In practice, the outcome of the resolution algorithm is the update of this set. The last data-structure is a map. It contains mappings of the context types *offered* by resolved plug-ins to the list of the corresponding plug-ins offering it. As the same context types might be offered by multiple plug-ins, this data-structure provides quick access to all of them.

When a new context plug-in is installed, or an existing one uninstalled, the event is reflected in the [all plug-ins] set, which consequently triggers the plug-in resolution algorithm. This algorithm iterates in two phases where first all the plug-ins are tested to make sure that the resolved ones are included in the [resolved] set and, second, all the plug-ins in the [resolved] set are tested to ensure that no unresolved instances are included in it. As the addition of a new plug-in in the [resolved] set can affect whether other plug-ins are resolved as well, this step is repeated until no changes are observed (steps 1, 1.1, and 1.1.1). Similarly, when a plug-in is removed from the [resolved] set, it might cause other plug-ins to become unresolved. So this process is also repeated until no further changes occur (steps 2, 2.1, and 2.1.1). When the algorithm ends, it is guaranteed that all the resolved plug-ins, and only those, are in the [resolved] set.

Table 2. Basic data-structures and algorithms used in the activation mechanism

Activation and deactivation mechanism

Basic data-structures
[offered] – a mapping of all offered context types to a list of resolved plug-ins offering them
[needed] – a mapping of all needed context types to a list of the corresponding plug-ins
[active] – a set of all the plug-ins that need to be active

Triggered by
Changes to the [needed] context types set

Algorithm
```
/* first make sure that all needed plug-ins are included in the [active] plug-ins set */
1. for each $C_T$ in [needed] do
   1.1 for $p$ in select(offered[$C_T$]) do
      [active] $\Leftarrow$ [active] $\cup$ {$p$}
      [needed] $\Leftarrow$ [needed] $\cup$ {requiredContextTypes($p$) $\rightarrow$ $p$}
   end for
end for
/* next make sure that no unneeded plug-ins are included in the [active] plug-ins set */
2. for each $p$ in [active] do
   2.1. if {offeredContextTypes($p$) $\rightarrow$ $p$} $\cap$ [needed] == $\emptyset$ then
      [active] $\Leftarrow$ [active] - {$p$}
      [needed] $\Leftarrow$ [needed] - {requiredContextTypes($p$) $\rightarrow$ $p$ }
   end if
end for
```

The next mechanism, depicted in Table 2, maintains that only the plug-ins that are actually needed by the deployed applications are activated. To achieve this, the context system maintains a list of the currently needed context types which is updated dynamically. Changes to this set trigger the activation and deactivation algorithm.

The basic data structures used in this mechanism are three: the *offered* plug-ins set which is the same described in the resolution mechanism, the [needed] map, which is used to map the needed context types to the actual components requiring them—*i.e.,* both applications and plug-ins—and the [activated] set, which is a subset of the [resolved] set described in the resolution mechanism, and which includes the plug-ins which were decided to be activated. The outcome of the corresponding algorithm is the update of the [activated] set. Although not explicitly stated in Table 2, the context system activates and deactivates the plug-ins as they enter and exit the [active] set.

The activation algorithm runs in two phases as well. First, it makes sure that at least one context plug-in is included in the [active] set for each needed context type. The selection is done using a special operation: *select*. As it will be discussed in the implementation section, this operation can be as trivial as a picking a random or all the matching plug-ins, or it can be more complex enabling more intelligent selection of the context plug-ins based on their attached QoS metadata. For instance, if two plug-ins are available for offering location data, then the selection might be based on their accuracy, on their battery consumption, or a combination of the two. Once a plug-in is selected, it is added to the [active] set, and an appropriate mapping from its required context types is added to the [needed] map. In the second phase, all the plug-ins in the [active] set are tested against the [needed] map. If no needed context types are matched to a specific plug-in, then the latter is removed from the [activated] set and the corresponding mappings of its required context types are removed from the [needed] map.

3.2 Implementation Issues

As part of the *MUSIC middleware*, two implementations were provided for the proposed architecture. The first one is a proof-of-concept implementation, where the plug-ins are handled as individual components. The resolution of the plug-ins takes place within the context system, along with the selection of the plug-ins to be activated (line 1.1 of Table 2). For the latter, a brute force approach is used, where *all* the plug-ins offering the required context type are selected for activation. In this case, it is expected that when multiple plug-ins are used to offer data of the same context types, they act in a complementary way and the context system automatically augments the sensed data to form mutually accepted values.

The second implementation is more elaborate as it leverages the existing planning framework of the MUSIC middleware to allow more granular selection of the active plug-ins. Fig. 2 illustrates an overview of the MUSIC middleware architecture, and depicts how the planning-based adaptation is supported. As described in Section 2, the Planner component supports the planning procedure by operating a generic reasoning algorithm that exploits metadata included in the available plans. In particular, the plans are composed based on their type compatibility to describe alternative application configurations. Then, the reasoning algorithm ranks the application configurations by evaluating their utility with regards to the application objectives. This evaluation is achieved by computing the offered properties using the Property Predictors associated to each plan contained in the selected application configuration and retrieved from the Plan Repository.

The Plan Repository component provides an `IPlanBroker` interface for the Planner to retrieve plans associated with a given component type during planning. The Planner may request plans that are compatible with a given variation point, at which point the Plan Repository will search for matching component types. Any additional metadata on the required component type will help the Plan Repository to exclude plans and filter the search space. Plans are typically published to (and discarded from) the Plan Repository by applications and component development tools using the interface `IPlanRepository`, and can thus trigger the Planner for re-planning of the application if needed (*e.g.*, when a discarded plan is associated to a running component).

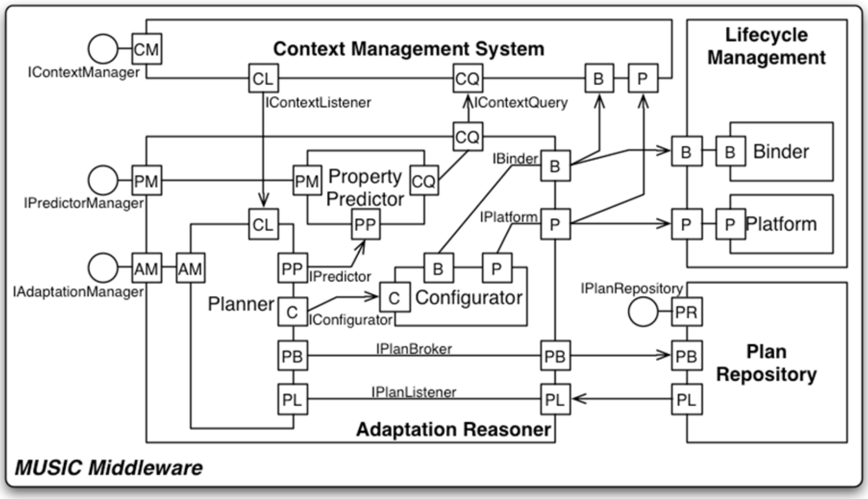

Fig. 2. Architecture of the Pluggable Context Middleware in MUSIC

The *reconfiguration process* is handled by the Configurator component and consists of taking the set of plans selected by the component Planner and reconfiguring the application. Before deploying the application configuration selected by the reasoning engine, the component Configurator brings the current application into a quiescence state, by suspending the execution of its contained components. Then, if the component described by a plan is in the *running* or *deployed* state, the associated component instance is configured for the variation point and connected to other components using the component Binder. If the component is in the *described* state, then the component is preliminary instantiated and deployed by the component Platform using the component implementation description associated to the plan. The result of the reconfiguration (*e.g.*, reference of the deployed instance) is automatically reflected into the selected plans. Thus, the MUSIC planning-based middleware offers a modular and extensible approach for adapting applications built with various types of component models. In particular, the concept of plan can be derived to support heterogeneous context plug-ins and their associated metadata. Furthermore, the components Platform and Binder provide sufficient abstractions for supporting different types of component. In this second implementation, the interfaces `IPlatform` and `IBinder` are also implemented by the Context Management System component for supporting the life-cycle management of context plug-ins and their dynamic deployment and activation. Thus, the context plug-in component metadata is reflected as a plan, which can be hosted by the Plan Repository and exploited by the Adaptation Reasoner to control the deployment and activation of context reasoners and sensors. This means that the resolution algorithm we introduced is implemented by the Planner component, while the activation algorithm is implemented by the Configurator component. Thus, based on the result of the adaptation process that has been triggered by context changes, the Configurator interprets the selected plans and interacts with the Context Management System to resolve and activate (resp. unresolved and deactivate) the context plug-ins. Thus, this implementation unifies the process of adapting both application and technical components in order to achieve common objectives (*e.g.*, reduce the memory consumption) in a consistent manner. In particular, conflicts during context plug-ins resolution are handled by the Planner component, which selects the context plug-ins providing the best utility to the application. Furthermore, this homogeneous planning mechanism opens up for support of *Context as a Service*, by benefiting of the *Service-Oriented Architecture* (SOA) support provided by the MUSIC middleware [14]. This means that context sensors provided by the execution environment (*e.g.*, a location sensor provided by a WiFi router) can be discovered and exploited by the context management system.

4 Experimental Evaluation

In order to evaluate the efficiency of the proposed algorithms, we implemented a prototype solution based on the MUSIC context manager and a set of context plug-ins (both real and simulated). This context manager is used by an application executing a scenario where the context changes dynamically. While the scenario progresses, we measure the performance of the algorithms by monitoring the resource consumption (battery and memory usage) in the device. The scenario is initially executed with the algorithms disabled, which results in all installed plug-ins being always active. Then, the scenario is repeated with the dynamic plug-in mechanism enabled which results to the dynamic activation of the needed plug-ins only.

Our evaluation focuses on PDA devices because resource consumption is more critical for this type of resource-constrained devices. We selected an HP iPAQ 6910 handheld running Windows Mobile 5.0, because it is equipped with both Wi-Fi and Bluetooth adapters, a GPS receiver, and GSM phone capabilities. This equipment allows us to implement and test a rich set of real plug-in sensors in addition to a set of simulated ones. The exact approach is described in the following subsections.

4.1 Implementation of the Context Plug-Ins and Experimental Setup

In order to implement the simulated scenario, we have defined the following set of real and simulated context plug-ins:

- *Bluetooth sensor plug-in* (real) allows to switch on and off the Bluetooth adapter,
- *Wi-Fi sensor plug-in* (real) allows to switch on and off the Wi-Fi adapter,
- *GSM sensor plug-in* (real) allows to switch on and off the phone functionality,
- *Location sensor plug-in* (real) allows to switch on and off the GPS receiver of the device and to retrieve information on the current geographical position,
- *RFID sensor plug-in* (simulated) simulates the detection of an RFID tag,
- *Light sensor plug-in* (simulated) simulates the detection and measurement of the ambient light of the environment where the device resides,
- *Weather reasoner plug-in* (simulated) based on the location info (*e.g.* provided by the GPS plug-in) and on Internet access (provided by the Wi-Fi adapter and controlled by the Wi-Fi plug-in), it simulates the invocation of a web service providing weather forecast for a given location.

The context plug-ins were implemented using the Java language, but some operations (*e.g.* Bluetooth, Wi-Fi and Phone activation or screen brightness adjustment) required low-level access to the underlying operating system. Hence, we leveraged the JNI technology which allows invocation of methods contained in native libraries from inside Java code. We implemented a native library using the C++ language, which wraps the calls to the required native methods and links them to the corresponding Java code, according to the JNI specification. Also, since the focus was on PDAs, a suitable runtime environment was selected to execute the Java bytecode. We selected the CrE-ME 4.0 for PDA *Java Virtual Machine* (JVM) from NsiCom Ltd, which is based on the JDK 1.3.1 specification and supports the J2ME CDC Personal Profile.

4.2 The Scenario: A Tourist Visiting a City Assisted by a PDA Computer

The scenario selected for our experimentation consists of a sequence of actions performed by a tourist while visiting a city. The tourist uses a handheld PDA device, which hosts the context middleware described earlier, along with an application, which defines a set of context needs and reactions to context changes. The reaction consists of adapting the device configuration to the new context for the objective of optimizing the resource utilization and maximizing the user satisfaction. This scenario is inspired and highly related to one of the main pilot applications proposed by the MUSIC Consortium to demonstrate the capabilities of the middleware.

The scenario is split in phases called "scenes", having a pre-defined duration. The next subsection lists the *scene description* and the *context changes* that happen in the environment for each scene. Moreover, the *adaptation* column describes how the application adapts to changes when the plug-in mechanism is enabled. In order to evaluate the effectiveness of the proposed mechanism, we repeated the same scenario having the plug-in mechanism disabled. In this case, all the components needed by the application—*i.e.,* GPS, Bluetooth, and Wi-Fi adapters, RFID reader, Light sensor, and Phone—were always switched on and the screen brightness was set to max.

4.3 The Simulation Application

The simulation of the scenario is driven by an application that reproduces the context changes and the user actions according to the scenes described below. When the application is launched with the context plug-in mechanism enabled, all the steps involving plug-in resolution and activation are performed, and the application reacts to the context changes by performing the corresponding adaptations. On the other hand, when the context plug-in mechanism is disabled, all needed low-level resources are switched on at start-up, and no adaptation is performed.

Scene 0 (duration: 95 sec): The tourist leaves the hotel for a tour of the city. He starts the application on the PDA. As the application requires RFID, GSM and light sensors, the associated plug-ins are resolved and activated.

Scene 1 (duration: 5 min): The tourist is walking in the street and the weather is sunny. As the ambient light changes from *normal* to *intense*, the screen brightness is automatically set to *maximum* (better contrast).

Scene 2 (duration: 10 min): The tourist reaches an area providing a Wi-Fi network. He now walks in a shady alley and ask to be notified periodically of weather forecast. As the ambient light changes from *intense* to *normal*, the screen brightness is set back to *medium* (better contrast and lower power consumption). At the same time, the application requires weather information and the weather plug-in is thus resolved and activated.

Scene 3 (duration: 30 sec): The tourist enters a metro station and the RFID reader detects a tag providing the following information about the environment: "*Station DUOMO*", no Wi-Fi, no Bluetooth kiosk, and no GSM coverage. Thus, the Wi-Fi, GPS, and Phone adapters are switched off and the weather plug-in is deactivated while the Bluetooth adapter is switched on.

Scene 4 (duration: 20 min): The tourist waits in the metro station. As the light sensor detects a change due to the station lights, the screen brightness is set to *low* in order to save battery.

Scene 5 (duration: 30 sec): The tourist gets off the couch and walks towards the exit. The RFIT reader detects a tag providing the following information: "Station LORETO", Wi-Fi and GSM available, no Bluetooth kiosk. Thus, the Wi-Fi and GPS adapters are turned on, the weather plug-in is activated and the phone is switched on.

Scene 6 (duration 30 sec): Now the weather outside turns to cloudy. As the light sensor detects this change, the screen brightness is set to *medium*.

Scene 7 (duration 10 min): The tourist walks around the city guided by its GPS, receiving weather forecast on a display whose brightness is set to *medium* and having its GSM turned on.

Scene 8: The tourist reaches the hotel and quits the application. As a result, all the context plug-ins required by the application are now deactivated to release the device resources.

In addition to the tasks described above, the simulation application performs the measurements of the memory and battery consumption in a separate thread. It does so by polling the underlying operating system at a given interval (*i.e.* every 5 seconds) to retrieve the resources status. In this way, the remaining battery charge, the actual memory usage (in terms of bytes and memory load as a percentage), the timestamp of the measurement and the corresponding scene name are all saved in a file.

4.4 Experimental Results

As already explained, our goal was to evaluate the efficiency of the context plug-in mechanism. We repeated the experiment twice: once with the plug-in mechanism enabled and a second one where it was disabled. All the experiments were initiated with the battery fully charged and without other applications running, in order to maintain as an identical execution environment as possible.

The graphs of Fig. 3 depict the trends in memory use (in terms of absolute memory used in Megabytes and also in terms of system memory load percentage) as a function of the scenario scenes. As shown in the graphs, the execution with the context plug-in mechanism enabled was less memory consuming in almost all the scenes. Table 3 shows a summary of the numerical values measured during the experiment. Enabling the context plug-in mechanism has saved an average of 21.29% of the total memory used compared to when the context plug-in was disabled. In terms of memory load, the average gain was 17.44%.

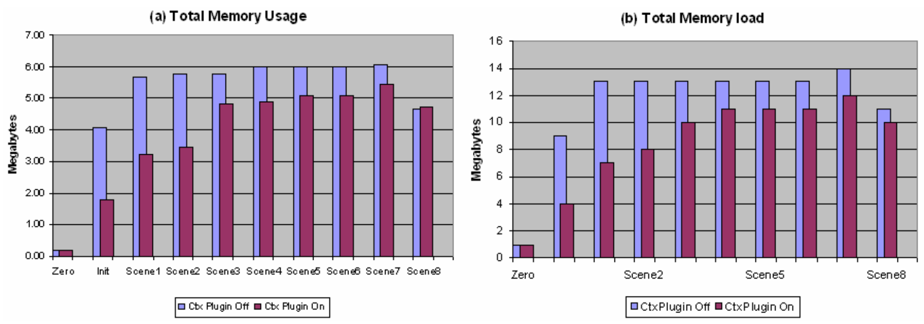

Fig. 3. (a) Total Memory Usage, and (b) Total Memory Load

The graphs of Fig. 4 depict the battery load during the execution of each step of the scenario. The first graph shows the trend of the total battery consumption, while the latter illustrates the relative battery usage for each scene. The execution with the context plug-in mechanism enabled was proven to be more energy efficient in all the scenes.

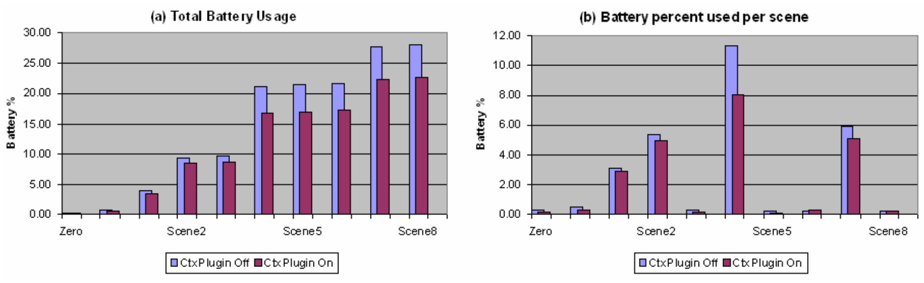

Fig. 4. (a)Total Battery Usage, and (b) Battery usage per scene

As depicted in Table 3, the activation of the plug-in mechanism has saved 20.95% of the total battery consumption. In absolute terms, for the used device, it means that 6% of the total battery capacity was saved. Similar results are also reflected in the relative battery usage per scene, where the average gain was 20.68%. The following table depicts the experiment measurements in detail.

For the execution of the experiments we used the first implementation described in section 3.2, where the selection of the activated plug-ins is not dependent on their QoS properties. As only one plug-in has been defined per required context type, this selection does not have a significant effect on the measurements.

Table 3. Results of the measurements obtained during the experimentation. TMU(MB): Total Memory Usage (in Megabytes); TML(%): Total Memory Load (in percentage); TBU(%):Total Battery Usage (in percentage); BPUPS (%): Battery Percent Used Per Scene (in percentage)

	TMU (MB)			**TML (%)**			**TBU (%)**			**BPUPS (%)**		
Scene	*P_Off*	*P_On*	Δ%	*P_Off*	*P_On*	Δ%	*P_Off*	*P_On*	Δ%	*P_Off*	*P_On*	Δ%
Zero	0.19	0.19	0.00	1	1	0.00	0.30	0.19	37.84	0.30	0.19	37.84
Init	4.07	1.80	55.76	8	4	50	0.83	0.54	34.57	0.48	0.32	32.18
Scene 1	5.67	3.21	43.42	12	7	41.7	4.00	3.43	14.27	3.13	2.86	8.59
Scene 2	5.74	3.45	39.82	12	8	33.3	9.41	8.40	10.80	5.38	4.92	8.38
Scene 3	5.74	4.82	16.07	12	10	16.7	9.75	8.63	11.49	0.29	0.19	34.07
Scene 4	5.99	4.88	18.53	12	11	8.33	21.07	16.73	20.62	11.28	8.06	28.56
Scene 5	5.99	5.06	15.46	12	11	8.33	21.38	16.87	21.09	0.27	0.10	64.63
Scene 6	5.99	5.06	15.53	12	11	8.33	21.69	17.22	20.59	0.27	0.30	-12.76
Scene 7	6.06	5.45	9.99	13	12	7.69	27.64	22.33	19.22	5.92	5.07	14.36
Scene 8	4.63	4.71	-1.69	10	10	0	27.96	22.65	18.98	0.25	0.27	-9.09
	AVG Δ%:		***21.29***	AVG Δ%:		***17.44***	AVG Δ%:		***20.95***	AVG Δ%:		***20.68***

5 Related Work

Many works exist in the literature proposing various models, methodologies and architectures for the creation of context-aware applications. The Context Toolkit [1] is one of the first works in the area of context management. In this framework, the users program interpretation and aggregation functionality in monolithic blocks, one interpreter and one aggregator per application which must encapsulate the entire context processing logic. However, the management of system resources consumed by context sensors and reasoners is not addressed.

The context manager of Draco [10] is organized around a database and an ontology broker. The component-based approach is chosen for its ability to dynamically adapt the context management system to changing conditions of applications' requirements and context devices. The objective is to (un)deploy on demand functional context management components, such as filtering, history, or transformation. However, the adaptation process in Draco is driven by the objective of saving storage space, but does not support the description and the management of context dependencies.

Similar to our work, the COSMOS framework [11] defines an architecture where individual components (called context nodes) are composed to implement the desired logic of a context-aware application. From a point of view, the context nodes correspond to context plug-ins in our architecture. Also similar to our approach, the context nodes are responsible to infer higher level context information from data gathered at lower architectural layers. Like our approach, the COSMOS approach aims to allow the development of context-aware applications where the context-awareness logic is separated from the business logic. However, in COSMOS the context nodes are merely logical units of composition, while in MUSIC the plug-ins correspond to deployable components. This implies that in our case plug-ins can be dynamically activated and deactivated in a way which controls resource consumption. From an architectural point of view, COSMOS also differs in the sense that the context nodes are directly connected to each other, while in MUSIC all the plug-ins are attached directly to the context system, which acts as a hub to route context notifications as needed.

RCSM [12] is an object-oriented framework where each context source (users, sensors, operating system, remote hosts) is separated. But, the authors do not tackle the issues of the synchrony of the treatments or of the control of system resources for context management. PACE [13] presents a similar architecture in which context data are stored in a database. The meta-data (temporality, quality, etc.) are added either to context data or to relations between them. The authors indicate clearly that they did not have a look at issues, such as scalability or performance.

6 Conclusions and Future Work

The main contribution of this paper is the specification of a pluggable and extensible context architecture that allows the dynamic activation and deactivation of context plug-ins on demand. It has been showed that this mechanism improves the resource utilization of the target devices, which is quite important in the case of small mobile and resource-limited devices. Another contribution of the proposed architecture is that it promotes component-oriented software engineering by separating the task of developing context providers—*i.e.,* sensor and reasoner plug-ins—from the task of developing context consumers—*i.e.,* context-aware applications. In this way, the developers design reusable context plug-ins by specifying the required and provided context types. These types can be specified in an unambiguous manner using the provided context ontology [6]. On the other hand, the developers of context-aware applications do not need to be aware of the details of the implementation of context providers. Rather, they only need to define the context types needed by their applications and handle the provision of these context types as a separate task—*i.e.,* reuse existing plug-ins or develop custom ones when needed. In the case of context-aware, self-adaptive applications, context needs can be expressed either directly or indirectly—*i.e.,* annotated in the context queries [7] and utility functions [4] of their corresponding plans.

Although not implemented yet, the proposed approach can naturally facilitate the development of context plug-ins using the MDD paradigm. It is an ongoing process of the MUSIC Consortium to specify an MDD-based methodology for the development of context plug-ins, automating the adoption of relevant patterns and the reuse of common mechanisms. We also plan to extend the plug-in metadata with extra information describing additional features such as accuracy per context type, cost of sensing (*e.g.*, in terms of CPU and memory consumption, as well as on battery drain). This would allow for more intelligent management of the available resources and the context plug-ins, especially in the case where multiple providers are available for the same context types and the system needs to choose only one.

Acknowledgments. The authors acknowledge the partial financial support given to this research by the EU (6th Framework Programme, contract number 35166).

References

1. Dey, A.K.: Understanding and Using Context. Personal and Ubiquitous Computing 5(1), 4–7 (2001)
2. McKinley, P.K., Masoud Sadjadi, S., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
3. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility Functions in Autonomic Systems. In: 1st International Conference on Autonomic Computing (ICAC), New York, NY, USA, May 2004, pp. 70–77. IEEE, Los Alamitos (2004)
4. Alia, M., Eide, V.S.W., Paspallis, N., Eliassen, F., Hallsteinsen, S., Papadopoulos, G.A.: A Utility-based Adaptivity Model for Mobile Applications. In: 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW), Niagara Falls, Ontario, Canada, May 2007, pp. 556–563. IEEE, Los Alamitos (2007)
5. Paspallis, N., Papadopoulos, G.A.: An Approach for Developing Adaptive, Mobile Applications with Separation of Concerns. In: 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, IL, USA, September 2006, pp. 299–306. IEEE, Los Alamitos (2006)
6. Wagner, M., Reichle, R., Khan, M.U., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A Comprehensive Context Modeling Framework for Pervasive Computing Systems. In: Meier, R., Terzis, S. (eds.) DAIS 2008. LNCS, vol. 5053, pp. 281–295. Springer, Heidelberg (2008)
7. Reichle, R., Wagner, M., Khan, M.U., Geihs, K., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A Context Query Language for Pervasive Computing Environments. In: 5th IEEE PerCom Workshop on Context Modeling and Reasoning (CoMoRea), Hong Kong, March 2008, pp. 434–440. IEEE, Los Alamitos (2008)
8. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2), 62–70 (2006)
9. Alia, M., Hallsteinsen, S., Paspallis, N., Eliassen, F.: Managing Distributed Adaptation of Mobile Applications. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 104–118. Springer, Heidelberg (2007)
10. Preuveneers, D., Berbers, Y.: Adaptive Context Management using a Component-based Approach. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 14–26. Springer, Heidelberg (2005)
11. Rouvoy, R., Conan, D., Seinturier, L.: Software Architecture Patterns for a Context-Processing Middleware Framework. IEEE Distributed Systems Online (DSO) 9(6) (June 2008)
12. Yau, S., Karim, F., Wang, Y., Wang, B., Gupta, S.: Reconfigurable Context-Sensitive Middleware for Pervasive Computing. IEEE Pervasive Computing 1(3), 33–40 (2002)
13. Henricksen, K., Indulska, J., McFadden, T., Balasubramaniam, S.: Middleware for Distributed Context-Aware Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, pp. 846–863. Springer, Heidelberg (2005)
14. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., Stav, E.: Composing Components and Services using a Planning-based Adaptation Middleware. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 52–67. Springer, Heidelberg (2008)