MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments

Romain Rouvoy[1], Paolo Barone[2], Yun Ding[3], Frank Eliassen[1], Svein Hallsteinsen[4], Jorge Lorenzo[5], Alessandro Mamelli[2], and Ulrich Scholz[3]

[1] University of Oslo, 0316 Oslo, Norway
rouvoy@ifi.uio.no, frank@ifi.uio.no
[2] HP Italy, 20063 Cernusco sul Naviglio, Italy
paolo.barone@hp.com, alessandro.mamelli@hp.com
[3] European Media Laboratory GmbH, 69118 Heidelberg, Germany
yun.ding@eml-d.villa-bosch.de,
ulrich.scholz@eml-d.villa-bosch.de
[4] SINTEF ICT, 7024 Trondheim, Norway
svein.hallsteinsen@sintef.no
[5] Telefónica I+D, 47151 Valladolid, Spain
jorgelg@tid.es

Abstract. Self-adaptive component-based architectures facilitate the building of systems capable of dynamically adapting to varying execution context. Such a dynamic adaptation is particularly relevant in the domain of ubiquitous computing, where numerous and unexpected changes of the execution context prevail. In this paper, we introduce an extension of the MUSIC component-based planning framework that optimizes the overall utility of applications when such changes occur. In particular, we focus on changes in the service provider landscape in order to plug in interchangeably components and services providing the functionalities defined by the component framework. The dynamic adaptations are operated automatically for optimizing the application utility in a given execution context. Our resulting planning framework is described and validated on a motivating scenario of the MUSIC project.

Keywords: Adaptation planning, component-based architectures, self-adaptation, service-oriented architectures.

1 Introduction

With the emergence of ubiquitous computing, common future scenarios will consist in people moving around carrying mobile devices, which they use extensively to assist both leisure and business related tasks. This will not only involve interactions with services provided through the Internet, but also with services directly provided by devices available in the surrounding environment.

For developers of mobile applications this is a very challenging scenario. Users' movements in ubiquitous computing environments cause frequent and unexpected changes in the execution context of their applications. For example, a mobile device is frequently roaming, and its applications have to be dynamically adapted to remain useful under new network conditions. Such an adaptation requires the detection of context changes, but also the selection of an application configuration that maintains a satisfactory *Quality of Service* (QoS) in the new context. Furthermore, when services become a part of the ubiquitous environment, both the availability and the quality of the services on which the applications depend becomes a concern of the application developer. There is therefore a need to dynamically discover services both when they become available and when they disappear. Also, such applications need to embed logic enabling them to reason about how and when to use a service available in the surrounding, to select among service alternatives when there are more than one available, and to adapt when a service disappears. Such a self-adaptation process is generally complex and costly to implement. To achieve self-adaptation, developers can use programming language features, such as conditional expressions, parameterization, and exceptions. However, these approaches introduce complexity by intertwining adaptation and application logic. Also, they make software evolution difficult. Conversely, approaches that use application independent middleware approaches for adaptation relieve the applications from adaptation concerns [1].

In the MUSIC project, we follow the latter approach by seeking to separate the self-adaptation concern from the business logic concern and delegate as much as possible of the added complexity related to self-adaptation to generic middleware. The adaptation process relies on the architecture model of the application, which specifies its adaptation capabilities and its dependencies to context available at runtime. In MUSIC, an application is modeled as a component framework, which defines the functionalities that can be dynamically configured with conforming component implementations. Thus, the purpose of an adaptation-planning framework is to evaluate the utility of alternative configurations in response to context changes, to select a feasible one (*e.g.*, the one with highest utility) for the current context and to adapt the application accordingly.

In this chapter, we propose a comprehensive extension of the MUSIC platform and planning framework we initially sketched in [2]. Currently, MUSIC only supports the adaptation of component-based architectures. The proposed extension enables the self-adaptation of mobile and ubiquitous applications in the presence of *Service-Oriented Architectures* (SOA). The planning middleware evaluates discovered remote services as alternative configurations for the functionalities required by an application. This means that the extended planning framework can support seamless configuration of component frameworks based on both local and remote components as well as services. In particular, components and services can be plugged in interchangeably to provide the functionalities defined by the component framework. In case of services, the planning framework deals directly with *Service Level Agreement* (SLA) protocols supported by the service providers. In addition to that, we introduce in this chapter a support for advertising services and associated service levels, in order to satisfy dynamically incoming service requests. Hence, MUSIC applications can use the MUSIC platform to share services with the environment.

In the remainder of this chapter, we first describe in section 2 the MUSIC approach to planning-based adaptation for component-based applications. In section 3, we introduce a motivating scenario for the support of SOA for self-adaptive applications in a ubiquitous environment, as well as derive a set of requirements. Section 4 exposes the MUSIC support for consuming and providing services in ubiquitous environments. Section 5 describes the integration of SOA into the MUSIC platform from an implementation perspective, while section 6 provides a preliminary validation of our approach by discussing how the requirements derived in section 3 are met by the proposed design. In section 7, we discuss related work before concluding and pointing out further work in section 8.

2 The MUSIC Approach to Self-Adaptation

Planning-based adaptation of a component-based application refers to the capability of a system to adapt to changing user needs and operating conditions by exploiting knowledge about its composition and *Quality of Service* (QoS) characteristics of its constituting components [2,3,4,5,6]. In MUSIC this knowledge is provided in the form of a *QoS-aware model* (cf. Figure 1), which describes the abstract composition, the relevant QoS dimensions and how they are affected when varying the actual component configuration. This model is exploited by the adaptation middleware to select, connect, and deploy a configuration of Component Realizations providing the *best utility*. The utility measures the degree of fulfillment of user preferences while optimizing device resource utilization [1,3]. The model describes the abstract composition as a set of Roles collaborating through Ports, which represent either functionality provided to or required from collaborating components. Properties and property predictor functions associated with the ports define how the QoS properties and resource needs of components are influenced by the QoS properties of the components they depend on. A port has a Type defining the functionality represented by the port in terms of interfaces and protocol. Component realizations implement ports and a component realization can be used in a role if the ports match (same type). Component realizations are Atomic or Composite. A Composite Realization is itself an abstract composition and allows for recursive decomposition. Constraints are predicates over the properties of the constituting components of a composition, which restrict the possible combinations of component realizations (*e.g.*, configuration consistencies) [3,7].

The model is represented at runtime as *plans* within the middleware. A plan reflects a component realization and describes its ports and associated property

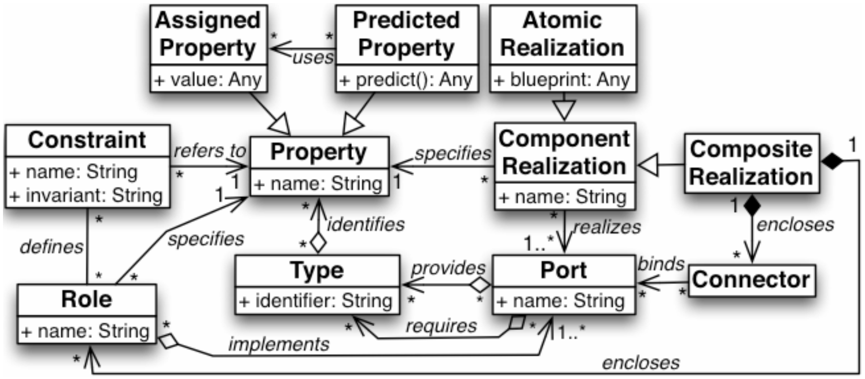

Fig. 1. Description of the MUSIC meta-model

predictors as well as *implicit dependencies* on the hosting platform (*e.g.*, platform type and version). In the case of an atomic component realization, it also contains a reference to the class, which realizes the component. In the case of a composite realization, the plan describes the internal structure in terms of roles and ports and the connections between them. Variation is obtained by describing a set of possible alternative realizations of the roles.

Then, *planning* refers to the process of selecting the components that make up an application configuration providing the best possible utility to the end-user. This process will be triggered at start-up of the application and at run-time when the execution context suddenly changes. When such an adaptation process is triggered for a particular type, the planning middleware iterates over the plans associated to the roles. For each plan, it resolves the plan dependencies and evaluates the configuration suitability to the current execution context by computing the Predicted Properties. The predicted properties are input to the normalized utility function that computes the expected utility of the evaluated application configuration [1,2,3,4,5]. The utility function of an application is provided by the developer and is typically expressed as a weighted sum of dimensional utility functions where the weights express user preferences (*i.e.*, relative importance of a dimension to the user). A dimensional utility function measures user satisfaction in one property dimension.

An example model for an application assisting traveling on public transportation is shown in figure 2. It is described as a collaboration of five roles. GUI presents a graphical user interface on the device. Main embeds the application logic and binds the different functionalities together. Main interacts with Route to find the shortest route and the estimated travel time. It also uses Map to get localized maps and Location to get the current location. The QoS properties used in the model are specified in table 1. Property predictors for the application, specified as functions of the properties of the components it consists of, are associated with the composition in figure 2. The utility function assumes that the user always prefers high accuracy and low battery consumption, while the relative weighting (w_acc, w_bat) will be extracted from the user profile by the middleware.

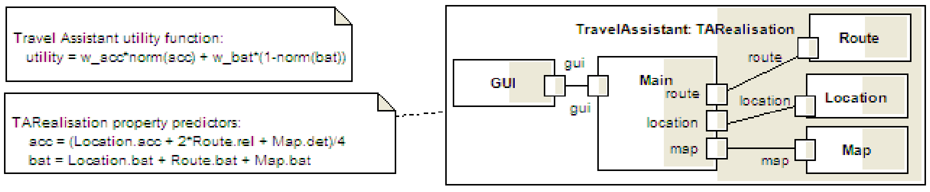
Travel Assistant utility function:
utility = w_acc*norm(acc) + w_bat*(1-norm(bat))
TARealisation property predictors:
acc = (Location.acc + 2*Route.rel + Map.det)/4
bat = Location.bat + Route.bat + Map.bat
TravelAssistant: TARealisation
GUI
gui
gui
Main
route
location
map
route
location
map
Route
Location
Map

Fig. 2. Example model for a TravelAssistant application

Table 1. Relevant QoS properties for the TravelAssistant application

Property	Description	Value range
acc	Accuracy	1-10
det	Level of detail of map	1-10
rel	Reliability of estimated travel time	1-10
bat	Power consumption of a component or link	0-∞

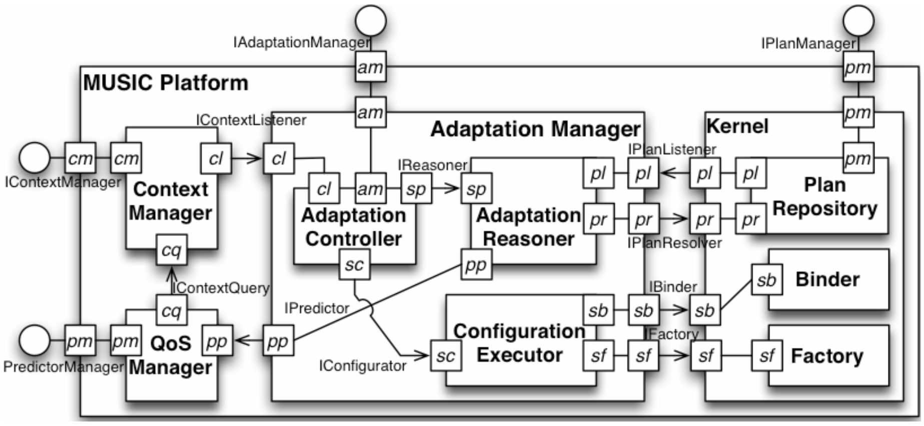

Fig. 3. Architecture of the MUSIC platform

The middleware manages a collection of active applications and seeks to maximize the overall utility, which is computed as a weighted sum of individual application utilities. The weights in this case express application priorities of the user.

Figure 3 depicts the component-based architecture of the MUSIC platform. The planning is typically triggered by context changes detected by the Context Manager. The Adaptation Controller coordinates the adaptation process. The Adaptation Reasoner supports the execution of the planning heuristics, which is driven by metadata included in the plans [4]. The Plan Repository provides an interface *IPlanResolver* to the adaptation reasoner allowing for the recursive retrieval of plans associated to a given port. Any additional metadata on the required types will help the plan repository to exclude plans and thus drastically reduce the exploration space [4,6]. The adaptation reasoner builds a valid application configuration and discards those whose dependencies are unresolved. Then, the heuristics ranks the application configurations by evaluating their utility based on the computation of the predicted properties, whose values are retrieved from the QoS Manager.

The *reconfiguration process* is handled by the Configuration Executor, which uses the set of plans selected by the planner to reconfigure the application. This requires the collaboration of the components, which must implement a reconfiguration interface allowing the middleware to bring them to a state where they can be safely replaced and transfer their state to an alternative component.

3 Challenges of Ubiquitous and Service-Oriented Environments

The term *service* is perhaps one of the most over-used and confusing terms in the software industry as analyzed in [9]. Typically, services are defined as *functionalities* or capabilities provided by a software system to other software systems or to a human user [10]. In the context of SOA, services are provided by independent service providers, which instantiate the providing software on their computers and advertise the services they provide using standardized mechanisms, such that they can be discovered and bound dynamically by consumers which need them. A fundamental concept of service-orientation is the standardized *service contract* [11], which is used to express the service semantics and capabilities. Service QoS properties are normally negotiated between the service provider and the *service consumer*, and are described as part of the service contract as a *Service Level Agreement* (SLA). A *service level* is used to describe the expected performance (*e.g.*, response time and availability) and properties such as billing, termination terms, and penalties in case of a violation of the SLA [12]. A SLA can either be created after selecting a fixed service level offer among several pre-defined offers or, in more complex cases, after a customization via a negotiation process. An SLA may be valid for a limited period or may be terminated explicitly. During *SLA provisioning*, the provider monitors the service QoS and adapts its resources to avoid *SLA violations*. The consumer may also perform monitoring to avoid trusting the provider blindly.

The platform presented in the previous section focuses on component based self-adapting systems. When mobile devices move around in ubiquitous computing environments they experience a dynamic service landscape and additional requirements to self-adaptation arise which require extensions of the platform. To investigate these issues, we consider the following scenario of Paul who is on his way to meet a friend, assisted by applications on his mobile device. First, we introduce several situations that Paul encounters and explain how he and his device react. Then, we explain the requirements that enable such flexibility.

3.1 Example Scenario: Paul on His Way to Meet a Friend

Paul has been at a concert in Paris. Now, he is taking the subway to a friend to see her new home and to tell her about the show. His MUSIC-enabled mobile device is WiFi-, UPnP-, and GPS-enabled. It provides several applications, among them a service-based version of the *TravelAssistant* from the example and a media-sharing application.

The *TravelAssistant* assists Paul with route planning, ticket vending, detects traveling delays, and notifies Paul if he is affected by such delays. The media-sharing platform, called *InstantSocial* [8], appears as a web site. However, instead of relying on a central Internet server, it is served by a composition of services scattered across nearby devices. As more users participate, this platform becomes more robust, the number of shared content items increases and it may become more attractive for the users. As soon as a critical mass of users leaves, it stops operating.

Scene 1. The scenario starts with Paul entering the Paris subway. He wants to plan the journey to his friend, which requires a route service for the subway as well as a location and a map service for the remaining trip. RATP, the operating company, offers a route service for public transportation and a map service of Paris at two QoS levels: *basic* and *premium* quality. Via UMTS, there is also access to a commercial service of high quality, though for a higher monetary cost. Paul requests services of high quality and his device chooses the cheap premium service of RATP.

Scene 2. With his *TravelAssistant*, Paul devises his journey and buys a ticket. As regular traveler, he has an electronic pass. Upon approaching a validation post, his device detects it, Paul's pass is checked, and the entrance gate opens automatically.

Scene 3. Inside the train, Paul thinks of searching for further pictures of the concert. He starts the *InstantSocial* application, which configures itself according to the other *InstantSocial* instances in the vicinity. His device notifies Paul about the presence of a matching media-sharing group. He joins and a moment later his display shows a selection of pictures, each representing a collection of shots of interests. He browses through the content, selects the ones he likes, and begins to download.

Scene 4. During the trip, there is an incident in the metro, blocking the planned itinerary. The travel assistant notifies Paul and proposes an alternative metro route with a different final station. Unfortunately, planning the remaining trip is not as smooth as desired: RATP reserves a large share of its bandwidth to guide the emergency personnel and declines to offer the high-quality map service. Furthermore, he cannot use GPS because the system's satellites are out of sight. As best solution, Paul's device chooses the external high-quality services, despite the higher cost.

Scene 5. Now, Paul is in a train with fewer visitors of his concert. Due to the decreased robustness, *InstantSocial* adapts it focus from *sharing* to *collecting* pictures. The other instances tend to do the same such that the combined media platform weakens. Finally, Paul is notified about the poor quality and he terminates *InstantSocial*.

Scene 6. After leaving the subway, the GPS module starts working and his device guides him through the streets. Some minutes later, he arrives at his friend's home in time with a device full of impressions to share.

3.2 Requirements for Planning-Based Adaptation

During Paul's journey, the applications on his device make flexible use of a variety of services and protocols nonetheless remains operational through various context changes. In particular, *TravelAssistant* and *InstantSocial* depend on external services that are dynamically chosen and used. Each *InstantSocial* instance also offers services to other instances. All these examples of flexibility require middleware support, which is provided by the design presented in this chapter.

Scene 1 shows a service selection process depending on QoS. The use of an UPnP-based service in scene 2 demonstrates the need for alternative connection protocols and services. Scene 3 demonstrates the degree of flexibility required: an *InstantSocial* instance is a combination of local and external services; it is able to offer and may use services at different QoS levels. The actual composition of the instance at a specific time has to be decided at runtime. Scene 4 features a willful reduction of a QoS level by the provider of an external service. It results in an adaptation to an alternative service provider, although the original provider is still offering the service, too. In scene 5, the device has to cope with an unplanned service termination by the sudden disappearance of *InstantSocial* instances. Furthermore, it demonstrates the deliberate termination of services by the user. Thus, to support scenarios of the kind presented above, we need to extend the platform to deal with the following SOA requirements:

- Dynamic discovery of services,
- Dynamic binding and change of binding to service providers,
- Negotiation of service level agreements and detection of violations,
- Hosting and publishing of services.

4 Supporting Service-Oriented Architectures within MUSIC

The interpretation of the term *service* presented in the previous section relates naturally to the port concept in the conceptual model presented in section 2. Thus we can accommodate services in the conceptual model simply by considering that ports represent services provided by or required by components, that services are described by types, and that service levels are described by properties. However, the middleware must be extended in several ways to cope with the challenges derived above. The remaining of this section introduces the *consumer-* and the *provider-side* support offered by the MUSIC platform in order to enable the seamless integration of services made available in a ubiquitous computing environment.

4.1 Consuming Services within MUSIC

In SOA-based computing environments, an application typically uses one or more services, which possibly depend on further services and so on. Thus, a large number of computers owned and administrated by different organizations may potentially be involved. This problem is aggravated when we deal with several applications running concurrently. Thus, optimizing utility over the entire set of involved computers is likely to be intractable both from a technical and administrative point of view. Therefore, we have to delineate the scope of an adaptation to be more tractable. To this end, we introduce the notion of *adaptation domain* and the distinction between *internal* and *external* services.

An *adaptation domain* is a collection of MUSIC platform instances controlled by one *adaptation manager*. It includes one distinguished node (*e.g.*, a handheld device), which represents a permanent binding to a user. This node acts as the *nucleus* around which the adaptation domain forms dynamically as *auxiliary* nodes come and go. The movement of nucleus nodes or changes in connectivity due other phenomena causes the dynamic evolution of an adaptation domain. Adaptation domains may overlap in the sense that auxiliary nodes may be members of multiple adaptation domains. This adds to the dynamics and increases the complexity because the amount of resources the auxiliary nodes are willing to provide to a particular domain may vary depending on the needs of other served domains. The user of a nucleus node may start and stop applications or shared components, and the set of running components is adapted by the adaptation manager according to these user actions and context changes, taking into account the resource constraints.

Clearly, it makes a difference whether a role is bound by instantiating a component implementation running in the adaptation domain where a system is built (*private instance*), by using a service provided by a component instance already running there (*internal service*), or by connecting to a service provided by a third party (*external service*). In the first two cases, the adaptation manager building the system must provision the resources and has control of the provided service level. In the latter case, the service level is outside the control of the adaptation manager, and it is necessary to negotiate an SLA with the service providers in order to compare the suitability of services by different providers and weight against deploying an internal service. External services may be provided by other adaptation domains or by third party providers (also referred to as *external non-MUSIC services*).

Discovery of Services and Service Levels. Providers make their services accessible to third parties according to specific *discovery protocols*. The MUSIC platform supports an extensible set of discovery protocols allowing the detection of services available in the service landscape. The discovery of a service triggers the retrieval of its *service description*, which includes information on the service capabilities, semantics, and possibly the offered service level(s) or QoS properties in form of an *agreement template*. The service description and, if available, the related agreement template are then converted to service plans, each one reflecting an alternative realization for the service level.

Negotiation of Service Level Agreements. The planning phase involves the evaluation of the available plans, for selecting the composition optimizing the utility of the applications running on the device. The utility depends on the QoS properties predicted by the services, whose value can be *static* or *dynamic*. Static properties consist of fixed values that do not change over the time. Dynamic property values can change according to the current status of the service. Evaluating the actual QoS values for such properties requires a process of negotiation with the service provider. The current MUSIC negotiation protocol is inspired by the WS-Agreement specification [13] (for both the definition and the creation/monitoring of SLAs), where the provider enriches the service description with an agreement template and the consumer fills in the template to create and submit an *agreement offer*. The offer creation is driven by *Service Level Objectives* (SLO), which are conditions defined at application or configuration level and act as pre-defined criteria for negotiating an SLA contract. Once the provider has accepted the offer, the agreed property values are reflected in the plan.

Provisioning of Service Level Agreements. Whenever a service available in the landscape is selected for use as a result of the adaptation reasoning, the MUSIC platform instantiates *service proxies*. These Proxies act as local representatives of the remote services and encapsulate the communication protocol necessary to access them in a location-transparent way. They are created by a *binding framework*, which provides dedicated proxy factories. Each factory supports a particular communication protocol to export or import a service. During the binding phase, the SLA contract associated with the selected plan is provisioned and enforced by the involved parties, which includes the reservation of computing resources and the deployment of SLA monitoring facilities [11,15,16].

Monitoring of Service Level Agreements. For the purpose of SLA monitoring, the service proxy is instrumented with appropriate monitoring mechanisms according to the content of the SLA contract (*e.g.*, response delay, result quality). Both parties are responsible for checking the status of the *agreement* and for taking proper actions in case of violation of the agreement. Thus, after the creation of an *agreement,* the MUSIC middleware, at any given time, must be able to check the current state of the agreement itself. When an agreement is not fulfilled anymore, the MUSIC middleware must terminate it and trigger a new adaptation process in order to detect a new set of available services and to select among them the best candidate to replace the one breaking the contract. SLA-enabled service providers handle the state model of an agreement and of its constituting terms, and make them accessible to consumers in form of readable properties of the agreement.

On the consumer side, the MUSIC middleware architecture is responsible for checking the state of an agreement according to pre-defined policies (*e.g.*, at given intervals or when detecting that the expected performance of a service is degrading). By querying the service provider for the agreement state, it is possible to detect whether the agreement has been violated or not. In case of violation, the consumer terminates explicitly the agreement by invoking a *terminate* operation on the *provider side* (since there might be costs associated to the usage of the service), and discards the related service plan, hence triggering a new adaptation process.

4.2 Providing Services within MUSIC

Hosting both applications and components providing services to the outside world in an adaptation domain complicates the adaptation reasoning. In addition to the user owning the device, there are also external service consumers, which may have conflicting needs (expressed in the SLA). Fortunately, the utility function approach lends itself quite naturally to cope with such situations. Our solution is to treat shared components providing services to external clients in the same way as applications and equip them with their own utility function, computing the degree of fulfillment of active SLAs. Using the weights, the overall utility function balances the utility to the owner of the device against the utility to service clients. This information about user preferences is included in user profiles.

Another difficulty is related to property prediction. For shared services, the resources needed by the component to guarantee a certain QoS often depend on the number of consumers. Hence, property predictor functions for shared services must take this into account.

Publishing of Services and Service Levels. By publishing its description using the discovery protocols supported by the MUSIC platform, a service running on a node can be made available to other nodes within the adaptation domain. Each service description encloses the service type as well as an *agreement template* describing the static QoS properties that are provided by this service. QoS dimensions referring to dynamic properties of the application are unbound in order to be fixed at a later time depending on the capabilities and the processing load of the hosting node.

Negotiation of Service Level Agreements. The MUSIC platform supports the negotiation of agreements by playing the role of a service provider. Whenever a service consumer selects one of the published services, the MUSIC platform receives an *agreement offer* for consuming this service. The MUSIC platform applies the negotiation heuristics to decide whether to accept or reject this offer by taking the current resource availability into account. This heuristics predicts the impact of accepting the offer with regards to agreements that have been already accepted. If the resulting impact does not trigger any violation of previous agreements, the MUSIC platform creates an *agreement*, which keeps track of the negotiation process.

Provisioning of Service Level Agreements. When a service consumer requests an internal service, the MUSIC platform checks that the requested service refers to an accepted agreement. Then, the *binding framework* instantiates a service skeleton—*i.e.*, a local representative of the service consumer—which reflects the ongoing agreement and implements one of the supported communication protocols (*e.g.*, SOAP or RMI). Invocations received via the service skeleton are delegated to the service instance locally deployed on the node.

Monitoring of Service Level Agreements. Depending on the negotiated properties agreed in the agreement, the service skeleton is instrumented with context sensors, which are responsible for monitoring the agreement. The MUSIC platform provides a library of sensors for observable properties (*e.g.*, invocation latency) as part of its context middleware. If one of the sensors detects a violation in one of the dimensions of the agreement, it notifies the MUSIC platform about this violation, which results in the notification of the service consumer and the termination of the agreement.

5 Realizing the Support for Service-Oriented Architectures

This section describes the extension of the MUSIC platform in order to support the SOA principles as well as the realization of the MUSIC reference implementation.

5.1 Architecture of the Service-Oriented MUSIC Platform

To support the above-mentioned SOA principles [11], we have integrated new components into the MUSIC Platform (cf. Figure 4, the composite component SOA Support). As MUSIC is independent of a particular technology, various implementations of these components can be developed (*e.g.*, Web Service, CORBA, RMI, or UPnP).

More specifically, the Service Discovery is responsible for publishing and discovering services using different discovery protocols. The Remoting Service is responsible for the exporting of services at the service provider side, and for the binding to these services at the service consumer side. Whenever a service is exported, it is enabled to accept requests from (remote) service consumers. Each *service description* defining the provided functionalities and containing the necessary information for the consumer to access the service[1] can be published by the service discovery. If the service provider offers additional guarantees for the published services, *agreement templates* are published in addition to the *service description*.

The service discovery supports the dynamic registration of *discovery listeners*. A *discovery listener* can have interest for particular services and can enforce customized policies to handle them. For example, the *Remote Platform Discovery Listener* is particularly interested in finding remote instances of the MUSIC platform in order to provide information about the MUSIC platforms connected to the applications. The *SLA Discovery Listener* is interested in finding services accompanied with an SLA support. Upon the discovery of services, the service discovery notifies the registered *discovery listeners* by passing them the service descriptions. Since plans are the base for the Adaptation Manager to perform planning-based adaptation, the *discovery listeners* create service plans based on the *service descriptions* and the *agreements* negotiated by the SLA Negotiation. Plans for remote services are generated whenever

[1] For example, the service consumer needs the remote service URL in order to access it. In case of a RMI-based binding, this URL would be `rmi://localhost:8080/EchoService`. While, in case of a Web Service, the URL is the location of the WSDL document, *e.g.*, `http://localhost:8090/axis2/services/EchoService?wsdl` for the *Echo Service*.

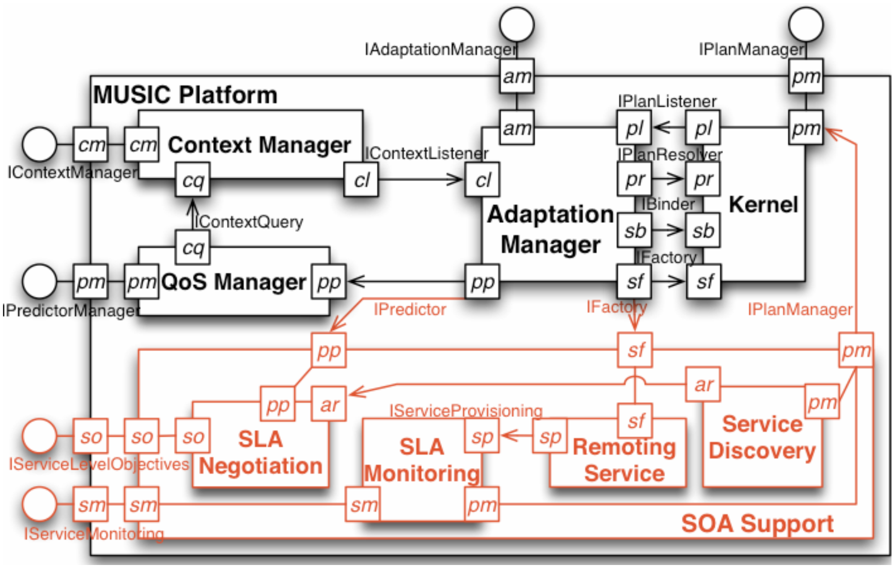

Fig. 4. SOA configuration of the MUSIC platform

services are discovered; hence plans are available when the adaptation manager triggers an adaptation at a later time. Plans are automatically discarded and removed from the Plan Repository whenever remote services disappear or for some reason become unavailable to the middleware.

The distributed instances of the MUSIC platform form a federation such that the service discovery on different platforms can interact with each other. Hence, MUSIC platform A can be aware of a service, which is published using a protocol supported by MUSIC platform B and not supported by A. If the remoting service on platform A supports the appropriate communication protocol, A is able to bind to that service which it would not able to discover alone.

Agreement templates can be either static or allow for dynamic negotiation [12]. Furthermore, a service may be offered at a pre-defined set of service levels. When the service discovery detects such a service, it first generates an abstract service plan enclosing structural and behavioral metadata related to the service. Then, in order to reflect the alternative service levels the service discovery publishes an extended version of the service plan for each service level into the plan repository. Such a service level plan inherits the metadata of the service from the abstract service plan and extends it with the additional QoS properties described by the particular service level (*e.g.*, service accuracy and cost).

The adaptation manager is then able to compare each available service level when applying the reasoning heuristics. Since service negotiation is a time critical factor for an efficient planning process, it should be resolved as soon as possible. In MUSIC, the negotiation is generally performed during service discovery for static QoS properties (*e.g.*, service cost) described by the service levels. The resulting static QoS property values are included into the service plan such that the predicted properties can automatically report them at a later time. However, in presence of a flexible service level [11,13], the negotiation becomes dynamic, meaning that the SLA is negotiated during the planning process. Dynamic negotiation is particularly required when the adaptation manager needs to reason about up-to-date QoS properties (*e.g.*, current service accuracy). In this case, the predicted properties, when evaluated by the reasoning heuristics, delegate the negotiation of the requested property to the SLA negotiation. The negotiation protocol is driven by SLOs, which are pre-defined criteria for negotiating SLA [15].

The Configuration Executor generally iterates over the plans composing the new configuration in order to reconfigure the application. As described in section 2, the configuration executor distinguishes between plans which refer to available services and plans which refer to services that are not available yet. In order to benefit from remote services, the configuration executor now faces a third case: If the plan refers to a remote service available in the environment, the configuration executor uses the Remoting Service to generate a specific component that will act as a *service proxy*. A service proxy is a local representative of the remote service. In particular, it implements the service type described by the application components and encapsulates the communication necessary to access the remote service. By invoking the service proxy, a service consumer interacts with the remote service in a location-transparent way—*i.e.*, as if the remote service is a local one.

The remoting service supports the dynamic integration of binding frameworks. During the binding phase, the SLA associated with the selected plan is provisioned and enforced by the involved parties. For the purpose of monitoring, the service proxy is instrumented with appropriate monitoring mechanisms by the component SLA Monitoring according to the content of the SLA (*e.g.*, response delay, result quality). The SLA monitoring is responsible for checking the status of the agreement for taking proper actions in case of its violation.

As an example of performing SLA monitoring in ubiquitous environments, the service proxy implements a disconnection detection algorithm. This disconnection support is inspired by the principles of *ambient programming* [17]. When loosing the connection to a remote service, the proxy stores the incoming service requests in a queue and returns a non-blocking *future object* to the application. The future object includes actions that are triggered whenever the connection is resolved to process the result of the request. If the connection is lost for a long period, the service proxy terminates the agreement via the component SLA negotiation. Subsequently, the SLA monitoring removes the associated service level plan from the plan repository to trigger an adaptation of the application. During the reconfiguration process, the request queue is transferred to the new component (or service proxy) that will be selected and deployed by the middleware.

5.2 Implementation of the Service-Oriented MUSIC Platform

The reference implementation of the MUSIC platform is based on the architecture described in section 5.1. The selection of the framework, which the reference implementation of the MUSIC architecture is built upon, has been made to meet the most relevant requirements for the MUSIC platform. They are, in particular, open source framework, multi-platform support, suitability for resource-constrained devices, and SOA support. Therefore, we selected OSGi to leverage the MUSIC platform.

OSGi (`http://www.osgi.org`) defined itself as the dynamic module system for Java, is a service-oriented component-based framework. The success of OSGi may be attributed to its relative simplicity, efficiency, openness, and portability. Multiple open source implementations of OSGi are available. Since its initial design, OSGi targets resource-constrained devices. Some existing implementations, such as *Concierge* [18], exhibit a reasonable memory footprint for resource-constraint devices (80 kB). Furthermore, some initiatives, such as the *OSGi Mobile Specification (JSR-232)* [19], the *Eclipse eRCP project* [20] or the *Sprint Titan platform* [21], propose OSGi for hosting applications in mobile devices. OSGi offers a class-loading mechanism to dynamically load/unload modules (bundles in the OSGi terminology). This feature is particularly interesting to support the plug-ability of the MUSIC architecture. Plug-ability is required to tackle the heterogeneity in communication and service discovery technologies. It also allows the integration of an extensible set of customized context sensors and adaptation algorithms.

SOA is the cornerstone of both OSGi and MUSIC. The SOA implementation in OSGi is simple and efficient, based on fast Java method invocations and a service registry, which provides mechanisms to react on the appearance and disappearance of services (essential in mobile environments). However, OSGi lacks of distribution support because OSGi services only communicate within one Java VM. The Service Discovery and the Remoting Service jointly extend OSGi with transparent distribution support and provide an abstraction to dynamically incorporate realizations of different discovery and communication protocols.

The Service Discovery delegates requests for publishing and discovery to protocol-specific implementations of service discovery, which are plugged into the platform as OSGi services implementing the interface *IServiceDiscoveryFactory*. Currently, the *Service Location Protocol* (SLP) protocol based on *jSLP* [22] and the *Universal Plug and Play* (UPnP) protocol based on the *UPnP bundle of DomoWare* [23] are supported by the MUSIC platform.

The Remoting Service supports plug-ability in a similar way. Each protocol-specific realization implements the interface *IExportFactory* or *IRemoteBindingFactory,* and is registered as a service to the service registry. Currently, the remoting service has support for exporting and binding services using sockets messaging and UPnP. The Web Service support is under development by a lightweight SOAP engine and small footprint HTTP server [24]. We create dynamic service proxies with the code generation library *CGLIB* [25], and attach communication protocol-specific interceptors to the service proxies. The instrumentation of a service proxy with SLA monitoring mechanisms will be realized by adding monitoring interceptors to the proxy (*e.g.*, to measure the response time).

MUSIC has chosen a set of lightweight frameworks and protocols to offer the best balance between performance in mobile devices and application requirements. The preliminary implementation of the *TravelAssistant* has demonstrated the good behavior of the MUSIC platform in a handheld device.

6 Discussions

As a preliminary validation of our approach, in this section we present a walk-through of how the middleware would behave in the scenario described in section 3.1. Table 2 presents the realizations available for the different services with property predictors for the relevant properties. In addition to the properties defined in table 1, we also introduced cost, which is very relevant for 3rd party services, and extended the utility function as follows: *utility=0.6*norm(acc) + 0.1*(1-norm(bat))+ 0.3*(1-norm(cost))*. Based on this extended model we computed the utilities of the various configurations in different scenes. Table 3 shows the utility of the best configurations in different situations during the scenario.

In the first three scenes of the scenario, the composition *i)* using the RATP Location, Map, and high quality Route services predicts the highest utility and is therefore chosen. In scene 4, the high quality RATP Map service breaks its SLA. The service proxy observes this and notifies the component SLA Monitoring, which terminates the agreement and triggers a re-planning. The Adaptation Manager predicts that using the commercial Map service instead now yields the highest utility and asks the Configuration Executor to reconfigure the application's service binding. This includes generating a corresponding service proxy. In scene 6, the device's GPS discovers the satellites and publishes the associated service plans into the Plan Repository. As this service provided by a local component is free and accurate, the adaptation manager predicts its use to have the highest utility and reconfigures accordingly.

Table 2. Services defined in the TravelAssistant application

Service	Description	Provider	Level	Property predictors
Location	Locates the device geographically	RATP		cost=0, acc=5, bat=1
		Local component using the builtin GPS		cost=0, acc=7 if GPS signal, 0 otherwise, bat=3
Map	Provides a map of a limited area	RATP	basic	cost=0, det=1, bat=2
		RATP	detailed	cost=5, det=9, bat=4
		3rd party		cost=9, det=9, bat=4
Route	Computes best route and estimated travel time	RATP	basic	cost=0, rel=1, bat=1
		RATP	reliable	cost=5, rel=7, bat=1

Table 3. Some alternative configurations and utilities of the TravelAssistant

Configuration			*Utility*		
Location	Map	Route	Scene 1	Scene 4	Scene6
RATP	RATP detailed	RATP reliable	**0,64**	-	-
RATP	3rd party	RATP reliable	0,56	**0,56**	0,56
builtinGPS	3rd party	RATP reliable	-	-	**0,58**

The *InstantSocial* application appears to the user as a centralized application, while under the hood, each user runs its own IS instance in its own adaptation domain. The multi-user behavior emerges from the interactions among the IS instances services—*i.e.*, each IS instance offers services and uses services offered by the others. The utility function determines the composition and behavior of an individual instance depending on the local resource situation and the QoS of the used services, and therefore indirectly also on the composition and resource situation of the other instances. Thus, the user-visible shape of *InstantSocial* appears according to size and quality of the instances in the collection.

The composition of IS describes three roles: *browser proxy* (BP), *presentation* (P), and *content repository* (CR). The *content repository* component is responsible for maintaining an inventory of available content in all the participating devices and providing access to it. CR instances act both as consumers and providers of the *membership* service. When a new CR instance is created, it will use the membership service provided by an existing instance to become included in the common distributed content repository, and later it may provide this service to another new instance. CR instances also implement partial replication of content to ensure a certain stability of the federated repository even if participants leave. *Presentation* components monitor the content repository in order to find relevant content elements, according to user preferences. They present lists of relevant contents and selected content elements to the BP component. *Browser proxy* components execute as demons and invoke the built-in browser to present the user interface when *InstantSocial* is in the foreground.

7 Related Work

Adaptive Service Grids (ASG) [26] and VieDAME [27] are initiatives enabling dynamic compositions and bindings of services for provisioning adaptive services. In particular, ASG proposes a sophisticated and adaptive delivery lifecycle composed of three sub-cycles: *planning*, *binding*, and *enactment*. The entry point of this delivery lifecycle is a *semantic service request*, which consists of a description of what will be achieved and not which concrete service has to be executed. VieDAME proposes a monitoring system that observes the efficiency of BPEL processes and performs service replacement automatically upon performance degradation. Compared to our planning-based middleware, ASG and VieDAME focuse only on the planning per request of service compositions with regards to the properties defined in the semantic service request. Thus, both approaches do not support a uniform planning of both components and services as our planning-based framework for ubiquitous applications does. However, our planning-based middleware can be extended to integrate ASG and VieDAME adaptive services and thus support the dynamic enactment of service workflows.

Menasce and Dubey [28] propose a QoS brokering approach in SOA. Consumers request services from a QoS broker, which selects a service provider that maximizes the consumer's utility function with regards to its cost constraint. The approach assumes that service providers register with the broker by providing service demands for each of the resources used by the provided services as well as cost functions for each service. The QoS broker uses analytic queuing models to predict the QoS values of the various services that could be selected under varying workload conditions. This approach is of interest both from the viewpoint of a consumer and a provider. While the client is relieved from performing service discovery and negotiation, the provider is given support for QoS management. This approach, however, requires the client device to be able to access the broker, which might not be possible in ubiquitous environments. Our approach differs in that we consider the offered properties as alternatives to determine the best application configuration and allow the client to adapt to the service landscape.

CARISMA is a mobile peer-to-peer middleware exploiting the principle of reflection to support the construction of context-aware adaptive applications [29]. Services and adaptation policies are installed and uninstalled on the fly. CARISMA can automatically trigger the adaptation of the deployed applications whenever detecting context changes. CARISMA uses utility functions to select application profiles, which are used to select the appropriate action for a particular context event. If there are conflicting application profiles, then CARISMA proceeds to an auction-like procedure to resolve (both local and distributed) conflicts. Contrary to MUSIC, CARISMA does not deal with the discovery of remote services that can trigger application reconfigurations. However, the auction-like procedure used by CARISMA could be integrated in the MUSIC middleware as a particular negotiation protocol.

The conceptual models of both SeCSE (`http://secse.eng.it`) and PLASTIC (`http://www.ist-plastic.org`) focus on service-oriented systems. Inspired by the SeCSE model, the PLASTIC model extends it by introducing new concepts, such as context, location, and service level agreements. The MUSIC and the PLASTIC model have in common that both combine SOA and component-based software development. However, the MUSIC conceptual model uses a component-centric approach, while the PLASTIC model uses a service-centric approach.

Finally, *R-OSGi* extends OSGi with a transparent distribution support [30] and uses *jSLP* to publish and discover services [22]. The communication between a local service proxy and the associated service skeleton is message-based, while different communication protocols (*e.g.*, TCP or HTTP) can be dynamically plugged in. In contrast to *R-OSGi*, the discovery and binding frameworks of MUSIC are open to support a larger range of discovery and communication protocols.

8 Conclusion and Perspectives

In this paper we have introduced the design of a QoS-driven generic planning framework for self-adaptive mobile applications, which seamlessly supports and blends component-based and service-based configurations. In particular, we have shown that the framework is able to adapt to changes in a landscape of ubiquitous remote services that dynamically come and go, and where the offered service qualities vary. The framework exploits these changes to maximize the overall utility of applications. To that aim, the paper has shown how the planning middleware discovers remote services and evaluates them as alternative providers for the functionalities required by an application. The planning framework deals directly with SLA protocols supported by the services to negotiate the best QoS for the end-user. The current MUSIC platform has already implemented the binding and discovery of services with a range of well-known technologies, while the SLA support is currently under development.

As a preliminary validation of our approach, the paper also explained how the planning framework handles a use case in which the *TravelAssistant* and the *InstantSocial* applications of a mobile user exploit ubiquitous services, such as location, map, and content services, to improve their utility whenever such services become available. The *TravelAssistant* has successfully validated the service binding and discovery, and will be enhanced in future releases. *InstantSocial* will be developed by the end of the MUSIC project (`http://www.ist-music.eu`).

Acknowledgements. We would like to thank our partners of the MUSIC project for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

References

1. Mascolo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware. In: Gregori, E., Anastasi, G., Basagni, S. (eds.) NETWORKING 2002. LNCS, vol. 2497, pp. 20–58. Springer, Heidelberg (2002)
2. Rouvoy, R., et al.: 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)
3. Geihs, K., et al.: A comprehensive solution for application-level adaptation. Software: Practice and Experience (2008)
4. Brataas, G., et al.: Scalability of Decision Models for Dynamic Product Lines. In: Int. Work. on Dynamic Software Product Line, DSPL (2007)
5. Floch, J., et al.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2) (2006)
6. Lundesgaard, S.A., et al.: Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 76–89. Springer, Heidelberg (2007)
7. Khan, M.U., Reichle, R., Geihs, K.: Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications. IEEE Distributed Systems Online 9(7) (2008)
8. Fraga, L., Hallsteinsen, S., Scholz, U.: InstantSocial – Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware. EASST Communications 11 (2008)
9. Baida, Z., et al.: A shared service terminology for online service provisioning. In: 6th Int. Conf. on Electronic commerce (2004)
10. Sassen, A., Macmillan, C.: The service engineering area: An overview of its current state and a vision of its future. European Commission. Network and Communication Technologies, Software Technologies (2005)
11. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, Englewood Cliffs (2006)
12. Dan, A., Ludwig, H., Pacifici, G.: Web service differentiation with service level agreements. IBM White Paper (2003)
13. Andrieux, A., et al.: Web Services Agreement Specification (WS-Agreement), Open Grid Forum Recommended Specification (2005)

14. Flores-Cortés, C.A., Blair, G.S., Grace, P.: An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. IEEE Distributed Systems Online 8(7) (2007)
15. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management 11(1) (2003)
16. Morgan, G., et al.: Monitoring Middleware for Service Level Agreements in Heterogeneous Environments. In: 5th Int. Conf. on e-Commerce, e-Business, and e-Government (I3E), Poznan, Poland, vol. 189 (2005)
17. Dedecker, J., et al.: Ambient-Oriented Programming. In: Companion of the 20th Ann. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2005)
18. Rellermeyer, J.S., Alonso, G.: Concierge: a service platform for resource-constrained devices. In: 2nd Eur. Conf. on Computer Systems (EuroSys). ACM, New York (2007)
19. JCP. OSGi Mobile Specification (JSR-232), `http://jcp.org/en/jsr/detail?id=232`
20. Eclipse. Embedded Rich Client Platform, `http://www.eclipse.org/ercp`
21. Sprint. Sprint Titan, `https://developer.sprint.com`
22. Rellermeyer, J.S., Kuppe, M.A.: jSLP, `http://jslp.sourceforge.net`
23. Demuru, M., Furfari, F., Lenzi, S.: DomoWare, `http://domoware.isti.cnr.it`
24. Equinox. OSGi HTTP Server, `http://www.eclipse.org/equinox/server/http_in_equinox.php`
25. Baliuka, J., et al.: Code Generation Library (CGLIB), `http://cglib.sourceforge.net`
26. Kuropka, D., Weske, M.: Implementing a Semantic Service Provision Platform — Concepts and Experiences. Wirtschaftsinformatik Journal (1), 16–24 (2008)
27. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: 17th Int. Conf. on World Wide Web (WWW). ACM, New York (2008)
28. Menasce, D., Dubey, V.: Utility-based QoS Brokering in Service Oriented Architectures. In: Int. Conf. on Web Services (ICWS) (2007)
29. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Trans. on Software Engineering 29(10) (2003)
30. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)