Hosting and Using Services with QoS Guarantee in Self-adaptive Service Systems

Shanshan Jiang[1], Svein Hallsteinsen[1], Paolo Barone[2], Alessandro Mamelli[2], Stephan Mehlhase[3], and Ulrich Scholz[3]

[1] SINTEF ICT, Postboks 4760 Sluppen, 7465 Trondheim, Norway
shanshan.jiang@sintef.no, svein.hallsteinsen@sintef.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
stephan.mehlhase@eml-d.villa-bosch.de,
ulrich.scholz@eml-d.villa-bosch.de

Abstract. In service-oriented computing, the vision is a market of services with alternative providers offering the same services with different cost and quality of service (QoS) properties, where applications form and adapt dynamically through dynamic service discovery and binding. To ensure decent and stable QoS to end users and efficient use of resources, it is required that both client applications and service implementations are able to adapt both their internal configuration and their binding to other actors in response to changes in the environment. To this end, service level negotiation and agreements (SLA) are important to ensure coordinated end to end adaptation. In this paper we propose a solution based on the integration of an SLA mechanism into a compositional adaptation planning framework and describe a simple yet powerful implementation targeted for resource constrained mobile devices. As validation we include a case study based on a peer-to-peer distributed mobile application.

Keywords: Service level agreement, service level negotiation, self-adaptation, service-oriented architecture, adaptation planning.

1 Introduction

In service-oriented computing, the vision is that systems providing functionality to end users form dynamically through service discovery and binding at runtime. This is supported by a service "market", where alternative service providers offer different service levels (SL) for the same services and where service offers appear and disappear and change dynamically. Service level agreement (SLA) serves to establish terms and conditions, especially SL guarantees, between service providers and service consumers, and thus allows systems to control the SL provided to end users.

In our work on self-adaptation in mobile and ubiquitous computing environments, we have advocated a combination of component oriented and service oriented adaptation. Service consumers and providers adapt dynamically both their internal component configuration and their service bindings in order to optimize the utility to the end

F. Eliassen and R. Kapitza (Eds.): DAIS 2010, LNCS 6115, pp. 15–28, 2010.

users as well as ensuring efficient utilization of resources. The coordination of the adaptation of part systems is facilitated by service level negotiation and agreements. In the context of the MUSIC project (http://www.ist-music.eu) we have created a development framework based on this approach, including both modeling and middleware support.

The principles of this approach to self-adaptation have already been presented and discussed in several publications [1,2,3,4]. The contribution of this paper is to explain the adopted service level negotiation mechanism and how it is integrated with the component level adaptation apparatus to achieve the coordinated adaptation we are seeking. To validate our design we have used the MUSIC framework to implement a peer-to-peer media sharing application, allowing users to dynamically form communities and create and comment a common media collection. By analyzing the design and behavior of this application we can demonstrate that our solution works as intended.

The paper is organized as follows: Section 2 presents the MUSIC approach to self-adaptation. Section 3 describes the design and implementation of the MUSIC Negotiation Framework as well as its integration into the adaptation framework. Section 4 presents the InstantSocial case study and demonstrates how SLAs are considered in the adaptation reasoning both at the provider side and the consumer side. Section 5 discusses related work before concluding the paper.

2 Adaptation Framework

The MUSIC approach is an externalized approach to the implementation of self-adaptation where the adaptation logic is delegated to generic *middleware* working on the basis of *models* of the software and its context represented at runtime [1,2].

These models understand *systems* as collections of collaborating *components*, modeled as *compositions* with typed roles and connectors. A *connector* models collaboration between two components, where one provides a service to the other. A *role* models a component providing services to or requiring services from other components. A component is either atomic, or a composition itself, thus allowing hierarchic decomposition. A composition may delegate the provisioning or consumption of a service to the level above it by leaving the appropriate end of the connector unbound.

To build system instances according to the above model, we need to find components which conform to the roles in the composition specification, instantiate these components, and connect the component instances according to the composition specification.

Typically there will be several component *variants* matching a role, differing in terms of a set of *varying properties.* This is modeled by *property predictor* functions associated with components. Property predictor functions are expressions over the context, the resources and the properties of collaborating components, and in the case of composite components, also the properties of the constituting components.

Varying properties typically model variation in extra functional properties (i.e., QoS properties) and resource needs, but may also represent variation in functionality. Thus, by selecting components with different varying properties we can build systems with different properties from the same system model and we can modify the properties of a running system by replacing one or more components.

A system has a *utility function*, which expresses how well suited a given configuration is in a given situation based on the predicted values for the varying properties. The utility function is an expression over the predicted properties of the system and the properties of the current context. The adaptation middleware aims to adapt the running systems so as to maximize the overall utility.

In Service-Oriented Architecture (SOA) based computing environments, systems are typically distributed with *part systems*[1] deployed on a potentially large number of computers owned and administered by different organizations. Part systems represent end user applications or service providing components, or both. The goal of the *adaptation planning* is to select appropriate variants that can be used in a composition to optimize the overall utility. However, optimizing utility over the entire set of computers involved is likely to be intractable both from a technical and an administrative point of view. Therefore we limit the scope of system models and the optimization of the utility to part systems, and rely on dynamic service discovery and binding to connect part systems, and on service level negotiation between them to ensure coordinated adaptation. The *adaptation planning process* considers components installed on its computer to populate the roles of the part system model, service providers located by dynamic service discovery to bind dependencies on external service providers, and takes into account service level agreements with consumers of provided services. Serving external clients consumes resources, and therefore whether or not to publish a service outside the part system or to accept new clients, is also decided at runtime by the adaptation middleware.

System models are represented at runtime as *plans.* A plan contains the details and the QoS properties (in the form of property predictors) of a certain realization. The dependency on an external service is represented as a special kind of plan called service plan, with an associated set of *plan variants* representing the available service providers.

MUSIC provides generic middleware to support running and adapting applications created using the above models. Obviously, components and services will have to be designed to be dynamically replaceable, and handle the transfer of state between variants where necessary.

The MUSIC middleware is based on a pluggable architecture and implemented based on an OSGi framework [2], where it is convenient to extend the architecture by plug-ins. The initial architecture proposed has been modified and extended during the implementation process for incorporating the SLA mechanism.

Figure 1 gives a simplified view of the MUSIC middleware. Plans and plan variants are stored in the *Plan Repository* in the *Kernel*. The *Adaptation Manager* handles the adaptation planning process for a part system, which is triggered basically by context changes detected by the *Context Manager*, and by plan changes in the plan repository. The *Adaptation Controller* coordinates the adaptation process. The *Adaptation Reasoner* supports different planning heuristics using metadata provided by the plans. The Reasoner builds valid application configurations by solving their dependencies and ranks the configurations by evaluating their utility based on the computation of the predicted properties. The *Configuration Executor* handles the reconfiguration process

[1] In MUSIC a part system may actually span several nodes. However, since this is transparent to the SLA mechanism, we do not explain it further in this paper.

using the plans selected by the Reasoner. The *Communication* provides basic support for SOA in distributed environment. The *Discovery Service* publishes and discovers services based on *service descriptions* using different discovery protocols. Whenever a service is discovered which matches a service dependency of a part system running in a node, a corresponding service plan variant is created in the plan repository. The plan variants are removed from the plan repository whenever the provider disappears or retires the offer. The plan variants will also be updated when the QoS properties provided by the services are changed. The *Remoting Service* is responsible for the binding and unbinding of services. At the service provider side, it exports services hosted by the provider (i.e. enable them to accept service requests), and at the service consumer side, it provides bindings (i.e. remote access) to the discovered remote services.

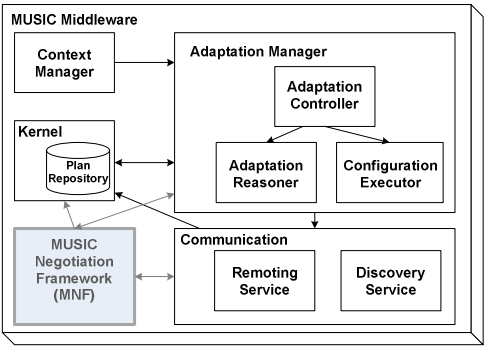

Fig. 1. Simplified architecture of MUSIC middleware

In the following we will discuss the design and implementation of the *MUSIC Negotiation Framework* (MNF, depicted in grey), and its integration with the adaptation framework. The MNF is responsible for service level negotiation and violation handling (cf. Sect. 3.3). It interacts with the Adaptation Manager and the Communication to realize the adaptation process integrated with SLA mechanism.

3 Integrating SLA with the Adaptation Framework

Current state-of-the-art work for SLA specification is WS-Agreement [5], a proposed recommendation of the Open Grid Forum. It specifies the general structures and terms for SLA definition and a simple single round negotiation protocol. We have selected WS-Agreement as a starting point for our work. However, WS-Agreement is technically too heavy for resource constrained mobile devices. Therefore, we adopt a custom, lightweight implementation. Below we present the overall approach for the integration work and then describe the main extensions in detail.

3.1 Requirements for the SLA Mechanism

In order to achieve coordinated adaptation of part systems, a service provider needs to know about its consumers (e.g. who and how many) and what they need (e.g. service

level) and incorporate such information into the adaptation process. We have identified a set of requirements for the SLA mechanism in our context:

(i) To allow providers to take into account the needs of the current consumers when adapting. The provider should consider the QoS requirements from the consumers (typically as required service levels) and the number of consumers when allocating resources. Such information should be reflected in the utility function so that it can be integrated into the utility-based adaptation reasoning.
(ii) To allow providers to notify consumers if they change the service level as a result of adaptation.
(iii) To allow the propagation of service level changes throughout the network of providers and consumers.
(iv) To give providers the flexibility of withdrawing a service offer, while maintaining the provisioning of the service to current consumers in order to avoid being overloaded.

For the server side of the mechanism we have focused on service exchange between peer nodes, typical of collaboration oriented mobile applications, and not specialized service provider nodes. This is also reflected in the case used for validation. However the client side of our solution may also exploit services offered by specialized service provider nodes not using the MUSIC technology.

3.2 Overall Description of the Approach

Below we give an overview of how services, service level negotiation, agreement and monitoring are integrated into the adaptation process in MUSIC:

1. *Service publication and discovery*: In MUSIC, services are advertised with the service levels predicted by the current variant of the service providing application. The advertisement mechanism delivers service descriptions to all the MUSIC nodes which are interested. Such service description contains information needed to properly specify and locate the service, together with a set of properties describing the current service level offered. The service level advertised consists of the predicted property values associated to the component providing the service. When a service is discovered at the consumer side, the advertised service level is used to create a service plan variant in the plan repository, which can be later evaluated by the Adaptation Reasoner when computing the utility of the available compositions. There can be multiple service plan variants for a service plan corresponding to alternative providers that a MUSIC node in the SLA-Consumer role (cf. Sect. 3.3) can select from.
2. *Service selection and negotiation*: The Adaptation Reasoner selects the most appropriate service offerings among a set of different service providers and different service levels available, each considered as a variant, by deciding if the variant with its service level can contribute to the composition configuration that gives the highest overall utility. If a service variant is selected by the reasoning process, a negotiation process is initiated towards the provider with an offer created based on

the selected service level. If negotiation is successful, an SLA will be created and the service will be provisioned with the guaranteed service level. If negotiation fails[2], the Reasoner selects another variant and re-negotiates. The negotiation process thus provides the adaptation planning with a mechanism to bind to the appropriate service provider with guaranteed service levels.

3. *Service monitoring*. In a ubiquitous service environment, the provided service levels may be dynamically changed. We use a simplified mechanism (cf. Sect. 3.3) to check the conformance of SLAs according to the predicted property values defined in the property predictors leveraging the MUSIC planning mechanism.
4. *Service violation and re-negotiation*. A service violation discovered by service monitoring will trigger the re-adaptation process of the Adaptation Manager. The violated SLA will be terminated and the Reasoner may select another available service variant and initiate the negotiation process.

3.3 The MUSIC Negotiation Framework

The MUSIC middleware can, at the same time, play two separate roles with respect to the negotiation process: It can provide negotiable services to remote nodes (*provider-side* negotiation) and use negotiable services provided remotely (*consumer-side* negotiation). *Provider-side* negotiation consists of evaluating an offer coming from a remote node and, possibly, creating an SLA between the parties; *consumer-side* negotiation consists of creating and submitting a request for reaching an SLA towards a service provider. All the SLAs reached as a result of the negotiation process must be properly monitored to verify their compliance with the agreement terms over the time. The service level negotiation and monitoring capabilities in MUSIC are provided by a custom, lightweight negotiation model, called *MUSIC Negotiation Framework (MNF*) and implemented by the *SLA Manager*. The internal components of the SLA Manager and their relationship are depicted in Fig. 2 and briefly described in the following. For a detailed description of interfaces and behavior, readers may refer to [6].

The *SLARepresentation* allows the creation of a MUSIC-specific, internal representation of an SLA describing the terms of the agreement, the actors involved and their roles, the associated QoS, the SLA state, etc. Once created, an SLARepresentation is stored into the *SLARepository*, a component which collects all the SLAs created by the MUSIC middleware (both when acting as an SLA-Provider and an SLA-Consumer). In addition, it allows other MUSIC middleware components to register as listeners for repository change events happening in the repository.

The *SLAMonitor* constantly monitors the QoS of an offered service and checks it against an SLA reached with a service consumer. In MUSIC, we adopt a simplified mechanism called *provider-side* SLA monitoring: The provider checks SLA conformance at the end of the adaptation planning process, which is based on the predicted QoS values calculated from the property predictors defined in the service plan variants;

[2] Due to the time gap between the adaptation reasoning and the negotiation, negotiation may fail in cases like the provider disappears, changes its service level, or can not accept more consumers. However, as the reasoning time is short, the probability of such failure is small.

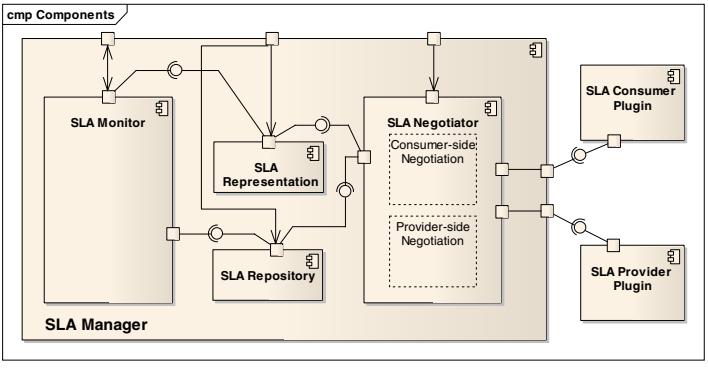

Fig. 2. Structure of the SLA Manager

the consumer relies on the provider for SLA monitoring by periodically checking the SLA state with the provider[3].

The *SLANegotiator* performs all the steps enabling consumer-side and provider-side negotiation, described at the beginning of this section. The negotiation logic is supported by corresponding SLA plug-ins that implement specific service level negotiation protocols. The current MNF has available plug-ins for a MUSIC internal negotiation protocol, which is a customized version of WS-Agreement with single round negotiation.

In order to integrate MNF into the MUSIC framework (cf. Fig. 1), two additional actions are performed at the end of the planning process:

- For all SLA-enabled service plan variants selected by the Adaptation Manager, an SLA negotiation process is triggered for each service. If the negotiation for a service fails, the corresponding service plan variant will be invalidated and a re-adaptation process is triggered.
- The Adaptation Manager invokes the SLA Manager to check SLA states for all active SLAs provided by the MUSIC node as a mechanism for provider-side SLA monitoring.

3.4 Discussion

The current MNF implementation and its integration into the MUSIC adaptation framework fulfills the first three requirements listed in Sect. 3.1. For (i): The number of consumers is the same as the number of SLAs and can be easily obtained from the MNF. The QoS requirements of the consumer are reflected in the service offer submitted by the consumer during service level negotiation. Both information can be

[3] A common approach for service monitoring from the literature is to use context sensors to gather data about service level metrics and parameters of the provided service at the consumer side. We adopt the simplified mechanism for provider-side SLA monitoring so as to eliminate the needs for consumer-side monitoring. However, context sensors can be readily integrated into the MUSIC framework due to the extensible plug-in architecture. See Sect. 0 for explanation of the rationale for this approach.

included in the utility function for adaptation reasoning. For (ii): The provider updates the SLA states when there is a change in the offered service level and the consumer can detect such change by periodically checking the SLA states with the provider. For (iii): The propagation of service level changes is realized by leveraging the service discovery and the SLA monitoring mechanisms. Requirement (iv) is currently unsupported, but we have designed a mechanism based on special flags that can realize it.

The integration implementation has leveraged MUSIC specific features in SLA monitoring to simplify the processing and improve the performance. Firstly, since the MUSIC adaptation framework uses predicted property values for reasoning (i.e., evaluated property values based on the property predictors at the given context), we use the predicted property values when performing provider-side monitoring. In addition, the consumer relies on the provider for SLA monitoring. This mechanism eliminates the need for additional context sensors to collect real-time QoS data both at the provider side and at the consumer side. Secondly, as any cause for service property changes will trigger an adaptation planning process, it is sufficient to check the property values at the end of the planning process. These mechanisms allow for a practical, lightweight implementation for mobile devices.

Although our implementation is MUSIC specific, it is quite flexible due to the plug-in architecture. Our current MNF implementation provides plug-ins for the MUSIC internal negotiation protocol. However, by delegating the negotiation logic to plug-ins, alternative negotiation protocols and technologies can easily be incorporated into the MUSIC framework. For example, the MUSIC internal protocol assumes that a service provider will publish only the current service level. To work with non-MUSIC nodes using a negotiation protocol [7] that provides alternative service levels in the service description, plug-ins for that negotiation protocol can be implemented on MUSIC nodes. Such plug-ins must create a service plan variant for each alternative service level, such that they are considered as different variants in the adaptation reasoning, and negotiate the selected service level with the provider.

Because our simplified monitoring mechanism relies on the provider for SLA monitoring, it implicitly requires the consumer to trust the provider. If such assumption does not hold in a dynamic environment, the consumer can use context sensor plug-ins to provide consumer-side SLA monitoring as adopted by other SLA frameworks.

4 InstantSocial Case Study

InstantSocial (IS) [3] is a media sharing platform for transient user groups that allows members to tag, to comment, and to search for text and images. IS has three design goals: (i) Maintaining a peer-to-peer network yielding high connectivity, (ii) providing access to a large number of media despite varying availability of devices, and (iii) balancing the load upon multiple resource-limited devices. IS accomplishes these goals by building on the previously described SLA capabilities of the MUSIC middleware. The following design extends and refines the IS version described in [3].

4.1 Design and Utility Function of InstantSocial

Figure 3 shows the design of the InstantSocial application: Variant configurations, components, their properties and associated property predictors. Table 1 lists the

property types and their descriptions. InstantSocial is divided into two parts: The user interface (UI) and the content repository (CR). The content repository instance holds the media items and their associated data, e.g., comments and tags. This component provides and consumes two different services: The context access service (*ca*) and the routing service (*rs*).

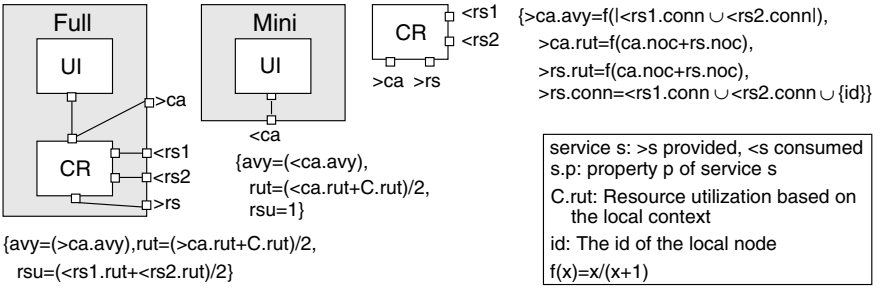

Fig. 3. Design of the InstantSocial application

The content repository is the computationally most demanding part of the InstantSocial application. InstantSocial can run in two variants to allow the reduction of resource usage, if necessary. In the *Full* variant, the application consists of both parts described above, whereas in the *Mini* variant InstantSocial does not start a content repository. Instead, it uses the content access service provided by a Full instance in its proximity. Each variant configuration is characterized by three properties: *Availability* (*avy*), as a measure for how much media is accessible in the current configuration; *resource utilization* (*rut*), indicating how much the node is in use; and *routing service utilization* (*rsu*), indicating how much the consumed routing services are under load.

The quality of the *ca* service is characterized by two properties: The *availability* (*ca.avy*) and the *resource utilization* (*ca.rut*). The *ca.avy* property serves as an indicator for the amount of media that is accessible through the service. The *ca.rut* property signalizes how much the providing node is currently used; it depends on the number of consumers (*noc*) for the provided *ca* and *rs* services.

InstantSocial instances use a routing service *rs* to interconnect. Each instance hosts one such service and consumes two (*rs1* and *rs2*). The *rs* service provides means to route messages through the network and builds an overlay network on the nodes. Note that the overlay network uses directed links between nodes, as opposed to standard network protocols like TCP. The quality of the *rs* service is determined by a *resource utilization* (*rs.rut*) property and a *connectivity* (*rs.conn*) property. The *rs.rut* property is an indicator for how much the routing service is currently used. The *rs.conn* property indicates the number of other nodes reachable by using this provider.

In MUSIC, the utility is a value between 0 and 1. The described properties are mapped into this interval by using the function $f(x)$ (cf. Fig. 3). The utility function of a configuration is defined as: $utility = c_1 \cdot avy + c_2 \cdot (1-rut) + c_3 \cdot (1-rsu)$, where c_1, c_2, and c_3 are relative weights of the properties and they sum up to 1. As media availability is of high importance of InstantSocial, *avy* should be dominant. For the following scenario, we use c_1=0.6, c_2=0.3, and c_3=0.1. The *rut* property of a service provider depends on

the number of its service consumers and thus enables its utility function to consider the consumer's needs during adaptation. The *rsu* property is used to select *rs* providers which have less workload. Therefore, it regards the needs (workload) of the provider and helps to spread service uses, thus preventing the overload of individual nodes.

Table 1. List of the property types of InstantSocial instances

Property type	Description	Value range
rut	Measure for the resource utilization of a node	0..1
rsu	Measure for the routing service utilization	0..1
avy	Measure for the availability of media items	0..1
conn	Connectivity of the node	Set of nodes

4.2 The InstantSocial Scenario

The following scenario – Andy is travelling home after visiting a concert – demonstrates the previously described design.

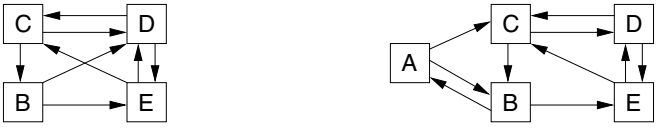

Fig. 4. Network layout before (left) and after (right) Andy joins the InstantSocial network

Scene 1: Andy visited a Björk concert and now sits in the train on his way back home. Betty, Chris, David, and Erika, also Björk fans, were already in the train. All of them have their InstantSocial instances (B, C, D, and E, respectively) running in Full mode and they already built the network depicted in Fig. 4 (left) when Andy enters the train. The arrows point from an *rs* service consumer to the provider. In the depicted initial situation, the connectivity property (*avy*) of all nodes has the same value because each node can reach the media of all others. However, during network changes, this property differs among the nodes. Each node has its own value of the resource utilization property (*rut*): For example, the services provided by B are used by one consumer only (*rs.rut* = 0.5), while D's services are consumed by three nodes (*rs.rut* = 0.75). Because the routing service utilization property (*rsu*) of a node depends on *rut* of other nodes, this property differs between the nodes.

When Andy starts his instance A, the middleware finds the services provided by the nodes that already established the network. After the best combination of services is identified (Table 2), the nodes negotiate SLAs for the routing services. After A arbitrarily connects to B and C, the *rs.conn* property of A's provided *rs* service is updated. Note that in order to avoid numerous re-adaptations, only the *rs.conn* property is subject to the SLA. In other words, if *rs.rut* of a consumed service changes then the SLA is not violated. Note that the resource utilization (*rut*) is independent of *rs1* and *rs2*, and therefore it is not included in Table 2. In this scene, its value is 0, as the services of A have no consumers and there are plenty of local resources available.

Table 2. Utility of node A based on the its choice of routing services

rs1	*rs2*	*avy*	*rsu*	utility
B	**C**	**0.8**	**0.5833**	**0.5216**
	D	0.8	0.6250	0.5175
	E	**0.8**	**0.5833**	**0.5216**
C	D	0.8	0.7083	0.5091
	E	0.8	0.6667	0.5133
D	E	0.8	0.7083	0.5091

Scene 2: We assume that B is the first node noting this change to its plans. Therefore node B re-adapts and its utility function indicates that it is better to disconnect from D in order to connect to A. This decision is based on the higher availability property of the variants that include A. After negotiating with and connecting to A, B also updates its *rs.conn* property, which leads to SLA violations on all nodes using B. After all re-adaptations have settled, the network reaches full mutual reachability again, now including node A (Fig. 4, right).

Scene 3: On Chris' mobile the resources start getting too low to run a Full configuration. Because Chris is still browsing through some media items, the adaptation middleware reconfigures his application into the Mini mode and therefore has to choose a content access provider. Candidates are the *ca* services of A, B, D, and E. These nodes still provide the same availability but different resource utilization (*ca.rut*), as this latter value depends on number of consumers: The node E has two consumers while A, B, and D have only one each. Therefore, C is free to choose from the latter and finally connects to A.

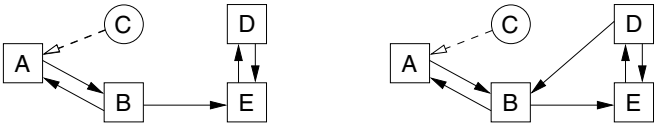

Fig. 5. Network layout after node C has switched to Mini mode (left) and after the resulting re-adaptation of node D (right)

Scene 4: After C re-configured into the Mini mode (Fig. 5, left), D and E are no longer able to reach A and B. By chance, D notices the removal of C first. The resulting re-adaptation has to select two nodes among A, B, and E to which D connects. All variants using E have high utility, so this connection is maintained. The choice between A and B is based on their *rsu* property. Both have one consumer of their routing service, but A has another consumer: The provided content access service is now used by C. The resulting higher *rut* value of A leads to a higher *rsu* value for variants that include A. Therefore, D connects to B and again reaches all the nodes in the network (Fig. 5, right).

4.3 Experiences Gained with the MUSIC Approach

The adaptive behavior of the system described in the previous section is quite complex and relies on the dynamic balancing of multiple and partly conflicting concerns across a number of users and computers. Nevertheless, the approach described in this paper keeps modeling such systems relatively simple, because a) the separation of the application logic from the adaptation logic decreases complexity significantly, b) the property predictor and utility functions provides a natural way to express the decision logic, and c) the dynamic service discovery and service level negotiation support ensures the necessary coordination between the involved nodes.

In our approach a service providing component advertises one service level at any time, the one predicted by the model in the given context. This approach appears to be appropriate in peer-to-peer oriented scenarios like our case study. However, in more client-server oriented scenarios, a server might want to advertise different service levels and prioritize requests in accordance with the agreed service level. Since MUSIC has been perceived primarily as a client-side technology, we have not focused on such requirements. However the consumer side of the MNF allows MUSIC nodes to discover and use services provided by non-MUSIC nodes, which may behave in this way, as already discussed in Sect. 3.4.

Another limitation of our approach is that the SLAs may be overly strict. Changing the service property of a MUSIC hosted service always causes a violation of all its SLAs. In some cases, the consumer might prefer a looser SLA and be notified only if the provided service level was decreased or moved outside given bounds. Consider for example the property *rs.conn* defined above. If a node is added to this set, the change does not have to constitute an SLA violation. The desired semantics of the agreement between provider and consumer in this case is that a particular set of nodes is reachable through the provided service. Consequently, extending this set does not violate this condition. Of course, a proper utility function will choose the same service again, so that the application is not re-configured and for the user everything stays the same. However, preventing SLA violations in such cases would reduce the resources spend on adaptation reasoning.

In summary, the proposed SLA architecture is sound and applicable. In particular, it is lightweight compared to existing approaches like WS-Agreement, and thus is more suitable for mobile devices.

5 Related Work

Several SLA specifications have been proposed targeting for software-based SLA negotiation, such as WSLA [7] and WS-Agreement [5]. WS-Agreement is the most mature one and we have selected it as the starting point for our work. There exist also several SLA frameworks proposed by projects, such as SLA@SOI [8], BREIN [9], BEinGRID [10], and AssessGrid [11]. These SLA frameworks with their software implementations, however, do not specifically target for self-adaptive service systems, nor do they consider specific resource constraints on mobile devices. In fact, current SLA implementations, e.g. based on WS-Agreement, are technically heavy and not suitable for resource constrained mobile devices.

Several works have addressed self-adaptation supported by generic middleware, similar to the MUSIC approach. CARISMA [12] focuses on adaptation of middleware level services. Planning consists of choosing among predefined rule-based adaptation policies using utility functions and resolving policy conflicts using an auction-like procedure. CARISMA does not support dynamic service discovery that can trigger application reconfiguration and the rule-based policies do not consider prediction of non-functional properties. However, the auction-like procedure used by CARISMA could be integrated to the MUSIC middleware as a particular negotiation protocol.

The self-adaptation techniques proposed by Rainbow [13] are also similar to MUSIC. Rainbow uses component-based architecture model and adaptation strategies based on situation-action rules are scored using *utility preferences* specified for the quality dimensions, where the adaptation manager selects the highest scoring strategy.

QuA [14] is a QoS-aware adaptation framework also based on utility functions. It calculates predicted quality using predictors and specifies quality requirements and adaptation policies using utility functions that map quality prediction to a scalar value. QuA has been applied to support self-adaptive SOA applications by integrating both interface layer and application layer mechanisms providing cross-layer adaptations [15]. However, the QuA middleware has no prototype implementation and does not focus on mobile applications.

Genie [16] adopts a similar approach to self-adaptation using component framework and architecture models to support runtime adaptability. It is however not service-oriented and has no dynamic service discovery and SLA support.

As far as we know, these self-adaptive systems do not provide an SLA mechanism for adaptation targeted for mobile domain. We are unaware of other work that uses SLA as a mechanism to achieve coordinated end-to-end adaptation. We therefore consider our integration of SLA mechanisms into the adaptation framework and a fully working reference implementation for mobile devices an essential contribution for ensuring QoS-aware and guaranteed self-adaptation.

6 Conclusions and Future Work

In this paper we have described how we use and integrate an SLA mechanism in an adaptation framework for self-adaptive service systems in order to allow service providers to take into account the needs of their clients in their adaptation logic and thus achieve coordinated end-to-end adaptation. This approach has been implemented in the MUSIC adaptation framework. As a preliminary validation of the implemented solution, a case study is included, demonstrating how it is exploited in the peer-to-peer mobile application InstantSocial to achieve coordinated dynamic adaptation of a set of collaborating application instances on different devices. Initial test shows that the application behaves as described. Performance measurements are currently in progress using several trial applications in addition to InstantSocial.

We intend to improve the implementation by extending some capabilities of the framework, which are currently provided at a proof-of-concept level. To simplify implementation, the compliance of a service level with an offer is currently based on an exact match between the values of the QoS required and provided. We plan to introduce a more complex reasoning for handling flexible logic conditions on the QoS

terms to be compared, such as “greater than”, “less then”, “between”, etc. We plan to improve the current implementation to fulfill the last requirement mentioned in Sect. 3.1, i.e., to support the flexibility of selective SLA creations. In addition, we intend to provide additional plug-ins that enhance the MNF by interacting with negotiation protocols different from the MUSIC-specific one.

Acknowledgements. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

References

1. 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)
2. Rouvoy, R., et al.: MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In: Cheng, B.H.C., et al. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 164–182. Springer, Heidelberg (2009)
3. Fraga, L., Hallsteinsen, S., Scholz, U.: InstantSocial – Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware. EASST Communications 11 (2008)
4. Hallsteinsen, S., Jiang, S., Sanders., R.: Dynamic software product lines in service oriented computing. In: 3rd Int. Work. on Dynamic Software Product Lines, DSPL (2009)
5. Andrieux, A., et al.: Web Services Agreement Specification (WS-Agreement). Open Grid Forum Recommended Specification (2005)
6. Barone, P.: D4.3 System design of the MUSIC architecture. MUSIC deliverable (2009)
7. 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)
8. SLA@SOI project, `http://sla-at-soi.eu/`
9. BREIN project, `http://www.eu-brein.com/`
10. BEinGRID project, `http://www.beingrid.eu/`
11. AssessGrid project, `http://www.assessgrid.eu/`
12. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Trans. On Software Engineering 29(10) (2003)
13. Garlan, D., et al.: Rainbow:Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
14. Gjørven, E., et al.: Self-adaptive systems: A middleware managed approach. In: Keller, A., Martin-Flatin, J.-P. (eds.) SelfMan 2006. LNCS, vol. 3996, pp. 15–27. Springer, Heidelberg (2006)
15. Gjørven, E., Rouvoy, R., Eliassen, F.: Cross-layer Self-adaptation of Service-Oriented Architectures. In: MW4SOC 2008, pp. 37–42. ACM, New York (2008)
16. Bencomo, N., Blair, G.: Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. In: Cheng, B.H.C., et al. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 183–200. Springer, Heidelberg (2009)