Utilising Alternative Application Configurations in Context- and QoS-Aware Mobile Middleware

Sten A. Lundesgaard, Ketil Lund, and Frank Eliassen

Simula Research Laboratory, Network and Distributed Systems,
P.O. Box 134, N-1325 Lysaker, Norway
{stena, ketillu, frank}@simula.no
http://www.simula.no/departments/networks

Abstract. State-of-the-art dynamic middleware uses information about the environment in order to evaluate alternative configurations of an application and select one according to some criteria. In the context of applications sensitive to Quality of Service, we have identified the need for a platform independent description of configurations that includes non-functional behaviour, and that allows handling of a large number of application configurations. In this paper, we present a modelling principle and a service plan concept, which together represents such a description. The modelling principle and plan concept extend state-of-the-art with i) a model of the alternative configurations that ensure a minimum of reconfiguration steps; ii) a specification that contains information elements of the configuration, dependencies to the environment, and QoS characteristics; and iii) a platform independent specification. In the paper, we also perform a qualitative assessment of our approach, and we describe a proof-of-concept implementation.

1 Introduction

The mobile domain represents a dynamic heterogeneous environment that constitutes a considerable challenge to application developers. Dynamic middleware for component-based applications is one answer to this challenge. These middleware platforms provide a traditional run-time environment, and, in addition, employ late-binding and reflection principles for dynamic (re)configuration of applications (e.g., OpenORB [1] and UIC [2]). However, today's state-of-the-art solutions within this field have some important shortcomings that need to be solved.

First, existing approaches force the application developer to explicitly specify all alternative application architectures. With many different combinations of hardware and software in heterogeneous environments, this gives a large number of different configurations and computationally intensive reconfiguration steps.

Second, dynamic middleware solutions for components mainly combine context-awareness with reconfiguration mechanisms, ignoring the Quality of Service (QoS) characteristics of the different application configurations. As a consequence they fail to support applications where QoS characteristics are critical, such as applications for streaming and conferencing.

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 228 – 241, 2006.

Third, in current state-of-art solutions, the specifications of the application configurations are defined for a particular middleware. Thus, the specification and its information elements are specialised for a specific set of tasks and platforms, making reuse difficult.

In this paper, we present a service modelling principle and a service plan concept that together provide a solution for these shortcomings. To briefly illustrate the principle and concept, we use the life-cycle of an application (see Figure 1). At design time, the service modelling principle is utilised for de-composition of the application into service compositions, atomic services, and alternatives thereof. The service plan concept provides the artefacts needed to deploy specifications of the alternative application configurations. We refer to the process of identifying and choosing an application configuration as *planning*. Service plans enable the middleware to perform planning, by providing information about the alternative application configurations, any dependencies to context elements (runtime environment, communication technology, storage facilities, etc.) and the resulting QoS characteristics for resources available (processing load, data rate, memory usage, etc.). For configuration of the application service plans provide the composition and parameter configurations of the components. When reconfiguration is needed, service plans provide meta-data about the running application and holds references to both composition and single components.

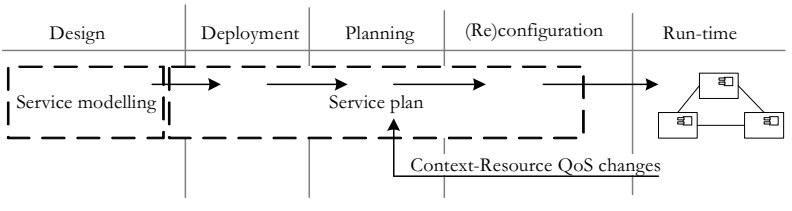

Fig. 1. Life-Cycle

The remainder of this paper is structured as follows. In Sect. 2 we present the service modelling principle, the service plan concept, and how to apply these in the different life-cycle phases. Sect. 3 assesses the principle and concept using qualitative criteria. In Sect. 4 we describe the implementation of our context- and QoS-aware mobile middleware and the proof-of-concept work related to this paper. Sect. 5 discusses related work. Finally, Sect. 6 presents our conclusions and future research.

2 Enabling Alternative Application Configurations

We start by presenting the service modelling principle and the service plan concept, before describing how these two make it possible to utilise alternative application configurations in a context- and QoS-aware middleware.

2.1 Service Modelling Principle

In Service Oriented Computing (SOC) an application is viewed in terms of service levels of abstraction [3], with resource services at the lowest level up to aggregated

services at the highest level. This view is also the foundation of our service modelling principle, where we order the service levels by introducing a service model. At the highest level, denoted the *service level*, the service is offered to the user (or client software). This service is divided into a composition of sub-services of a finer granularity, at the *sub-service level* in the service model. These sub-services may again be divided into service compositions. When a service can not be decomposed any further it is considered an atomic service. At the *atomic service level* the implementation of a service is a self-contained software component.

In SOC, semantic representations of components are used to make the software suitable for late-binding [3], i.e., components are bound together during (re)configuration. This is also one of the reasons why semantic descriptions, referred to as *service types*, are included in our service modelling principle. Another reason is the separation this establishes between the design of the application behaviour and the alternative application configurations. Each service in the service model therefore has a type, and all alternative implementations of a service must conform to the same service type. This enables us to model alternatives at all service levels in one single service model. Thus, we avoid one architecture model for each application configuration. The variations between these alternatives are not constrained by the service modelling principle, but by the deployable artefacts and the middleware. In our work we have three types of variations: service compositions, parameter configurations, and components.

Figure 2 exemplifies both the decomposition of an application and the alternative implementations of the service types. In the figure, the service types *j1* and *jl2* have two alternative implementations each.

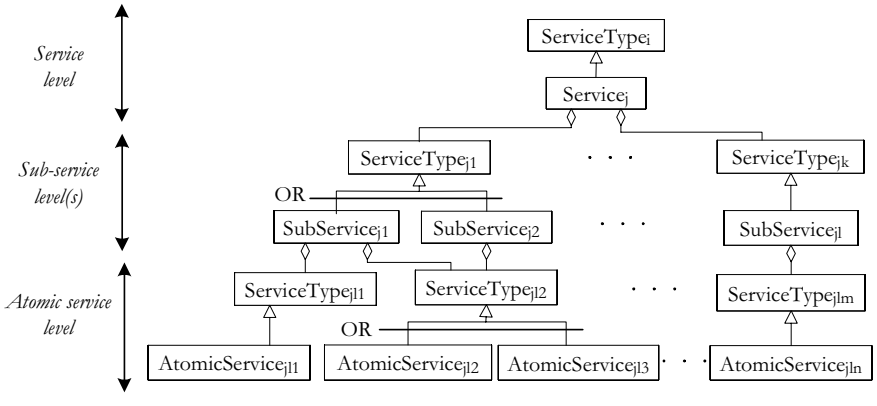

Fig. 2. Service Model

2.2 Service Plan Concept

As described above, the service model allows for alternative implementations of services at all abstraction levels. The dynamic mobile middleware use run-time information about the environment choose the most suitable among these application configurations. Therefore, we need to be able to specify dependencies to context

elements in the environment and the QoS characteristics of the application configuration, in addition to the software architecture and behaviour [4].

The semantics of a service is represented as a *service type*. We introduce the *service plan* as the association between a service type and one implementation of that type, and, if applicable, with a particular parameter configuration. Thus, the service plan concept effectively bridges the gap between the layered service model of an application and the corresponding running application. By specifying application configurations and providing information about the alternatives, the service plans enable the dynamic middleware to identify and choose between configurations. While existing approaches use specifications defined for particular middleware and therefore are specialised for a specific task, the service plan is defined at a conceptual level to ensure that the result is middleware independent.

As can be seen from the conceptual model in Figure 3, a service plan contains five information elements: i) *Service* is the name of the service type of which the plan specifies the implementation, ii) *Implementation* specifies either a component or a composition of service types, iii) *Dependencies* to context elements in the environment and their properties, iv) *ParameterConfiguration* lists values for configuration of the implementation, and v) *QoSCharacteristics* of the specified implementation. An important attribute of our service plan concept is the support for compositions of service types. A service composition can in turn be part of another service composition, enabling the specification of the application as a recursive structure of service types.

Together, these elements provide a complete specification of a service implementation, separated from the implementation itself, and independent of any middleware. There are, however, some challenges associated with a platform independent specification of dependencies and QoS characteristics. In particular context and resource models that define the semantics and classifications of context elements and resources are needed. These models are applied when modelling the properties and QoS characteristics of context elements and resources in the environment and designing the context and resource managers for the middleware. However, this is a separate challenge that we discuss in [5].

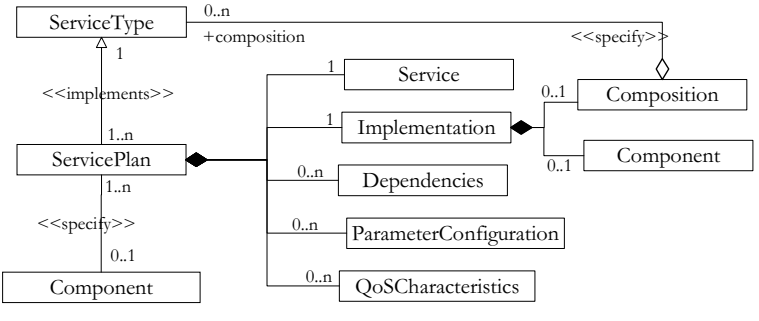

Fig. 3. Conceptual Model

2.3 Application Life-Cycle

In the following sections we outline how to utilise the service model and service plans in the different phases of the application life-cycle (illustrated in Figure 4), from design to (re)configuration. Where appropriate, we make recommendations regarding design and technologies considered useful for a mobile middleware.

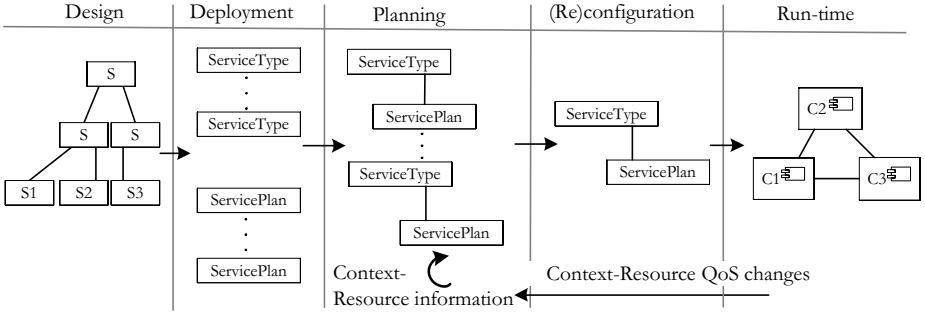

Fig. 4. Life-Cycle

2.3.1 Design

In the design phase, alternative configurations with context dependencies and QoS characteristics are identified. There are two approaches. Decompose the application, starting at the service level, into a hierarchy of services with alternatives (top-down approach). Alternatively, start at the bottom with atomic services and compose these into sub-services, which again may be combined to other services (bottom-up-approach). Both approaches result in one service model with the abstraction levels of the application, semantic representations of all services that form the application, and the alternative implementations that constitute the application configurations.

For each implementation, any dependencies to context elements must be identified. A context model is recommended to ensure that the middleware interprets the specified dependencies correctly. Furthermore, suitable descriptions of the QoS characteristics are required, which typically involves specifying mapping functions between different QoS levels. QoS and resource models should be applied to ensure a correct interpretation by the middleware.

2.3.2 Deployment

After the alternative application configurations have been defined in one model, service types and service plans are prepared for deployment. For the service type, one may use Web-Service Description Language (WSDL), as this format has certain desired properties: an open standard, readable, and supported by most software engineering tools. For the same reasons, service plans should also be deployed as text files, such as the eXtensible Mark-up Language (XML).

In the text file, information elements that constitute the service plan should be presented as an ordered tree, in order to make it easy for the middleware to read the data and move up and down inside the file. An example of a tree structure for service plans

is shown in Figure 5. Below the root there are child nodes that divide the tree into five branches, one for each of the information elements specified in the conceptual model (see Figure 3). Out of these five branches, only *ServiceType* and *Implementation* are mandatory, since these are required when publishing the service. The *QoSCharacteristic* node has four children, which reflect the QoS modelling method that we apply to define the QoS characteristics of a service (exemplified in [17]). Another QoS modelling principles may result in other nodes. The *Implementation* node is somewhat different from the others, as it is the only parent with two alternative children, *composition* or *component*. Composition is used when the implementation is a service composition, while component specifies a single component.

During deployment, service types and plans are loaded and interpreted. This functionality can be implemented in the middleware in different ways. For instance, the loader can be activated from an operations and management console and only upload service types and plans in certain catalogues. Alternatively one may choose to confine loading to a predetermined set of alternative service configurations, by using a configuration file for the loader.

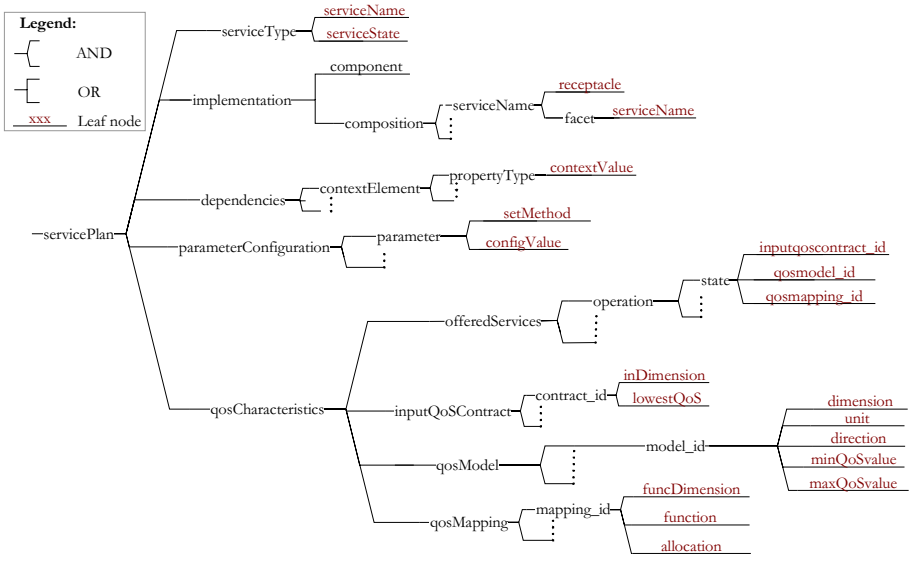

Fig. 5. Ordered Tree Structure for Service Plans

2.3.3 Planning

When a user requests access to an application, the context- and QoS-aware middleware uses the deployed service types and plans to identify the alternative configurations suitable for the current environment, and to choose the one that provides satisfactory QoS. For synthesising the alternative configurations from the information inside the service plans, we suggest the *service configuration* pattern (see Figure 6). The service configuration is asked to resolve itself, using the type of the requested service, together with a plan for that type, as input. This is done for each alternative

plan, resulting in one service configuration object for each alternative configuration of the application. In case of a service composition, the service configuration analyses the connections between the receptacle and facet ports of the service *implementation*. From this the service configuration creates the *next level* of service types and service plans, shown in Figure 6.

Service configurations that can not execute in the current environment are filtered, by checking the specified *dependencies* against context information. Next, the service configurations are compared by using the *QoS mapping* functions at each level inside the service configurations. There are different methods available, such as comparing predicted end-to-end QoS for similar dimensions or applying utility functions. Furthermore, in case of several possible configurations with satisfactory QoS, the middleware can either maximise utility/QoS or minimise the resource load.

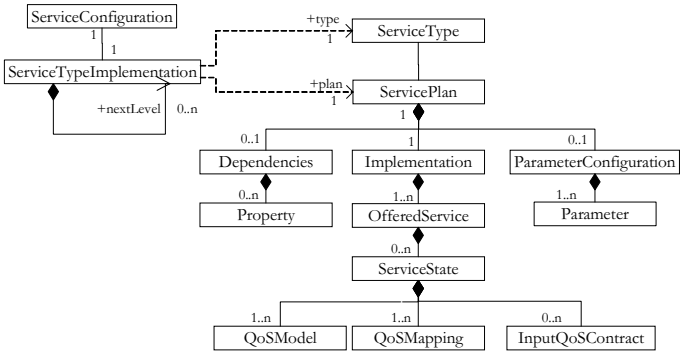

Fig. 6. Service Configuration

2.3.4 Configuration and Reconfiguration

After instantiation, the selected service configuration, together with the associated service plans, change roles from specifying one possible configuration of the application to specifying the architecture, behaviour, parameter configuration, dependencies, and QoS characteristics of a running application. Thus, during reconfiguration the service plan concept is a meta-level model of the application configuration. This model is casually connected to the application, so any changes made to the meta-level causes corresponding changes in the application. The causal connection is enforced by the middleware. For reconfiguration this is useful, since it makes the implementation open for inspection without having to involve the components.

If context dependencies or QoS requirements are violated, the middleware will replan the service, and if another service configuration meets the QoS requirements, the middleware reconfigures the application. Within the middleware, a reconfiguration manager, or equivalent, handles the reconfiguration. There are many ways to design the reconfiguration manager and the causal connection between a service plan and the corresponding base level composition/component, but this is a separate challenge not discussed here.

3 Qualitative Assessment

In Sect. 1, we presented three major issues that need to be addressed when using dynamic component middleware as platform for QoS-sensitive applications. In this section, we go through each of these issues, and describe how they are met by our service modelling principle and service plan concept.

Configuration manageability. There are two facets of this issue. First, to be able to achieve and maintain both functional properties and QoS in a dynamic heterogeneous environment, a large number of alternative application configurations must be designed and deployed. Second, the required changes should be as few as possible when reconfiguring, i.e., avoid solutions that require that the entire application has to be replaced. The service modelling principle and service plan concept address the manageability issue by combining service layers and three (re)configuration methods (composition, parameter configuration, and component implementation) into one service model. Service layers make it easy for application developers to design the alternatives at a suitable abstraction level, from which, when deployed, a range of different alternative application configurations can be derived. For instance, a sub-service with three alternative implementations, which again consist of sub-services that have nine alternative implementations, a total of 27 alternative compositions can be derived. All these alternatives are represented in one single service model, so we avoid individual architecture models for each configuration. The three (re)configuration methods address the manageability issue, by giving the application developers full control of the composition, parameter configurations, and component implementations. Together with the service layers, these reconfiguration methods ensure that the dynamic middleware only reconfigures the parts of the application that is required, and without always having to perform computational intensive reconfiguration steps associated with changing the entire component composition.

QoS-awareness. Existing dynamic middleware platforms for component based applications do not combine context- and QoS-awareness. Hence, we require a specification with information elements that puts the middleware in a position to identify application configurations that can execute in a particular context and to choose one of these based upon the end-to-end QoS characteristics (assuming the middleware has context information about the environment and information about the resource QoS characteristics). The service plan has information elements for specifying both dependencies to context elements in the environment and QoS characteristics of the implementation. When applied to the implementations, service plans can specify context dependencies and QoS characteristics at all levels in the service model. The service plan concept is technology independent, but we recommend XML for deployable files and a design pattern for synthesising the application configurations (explained in Sect. 2). Both are extensible, which makes it easy to combine the service plan concept with any context model, QoS model, or mapping functions between QoS level.

Platform Independence. To be able to deploy the application and the alternative configuration on different context- and QoS-aware middleware solutions, the specification must be platform (i.e., middleware, network, and operating system) independent. This requires that one adhere to the fundamental principle of separation of concern. In particular, specifications of application configurations and their QoS

characteristics must be independent of the functional code, and the transitions between the phases in the life-cycle, e.g., design to deployment, deployment to planning, and planning to (re)configuration, use technologies that are independent of middleware and programming language. The service modelling principle gives the required separation between the semantic representation of the service and the specification of the alternative implementations. Furthermore, the service plan provides the specification of the implementation at a conceptual level, making it easy to implement and utilise throughout the application life-cycle.

4 Implementation

In order to show that the service modelling principle can be used together with existing software modelling principles, we have applied it to a component-based video streaming application. It was important to make the modelling realistic, and not merely an exercise. Thus, a scenario of the complete system including the environment, user mobility, and different terminal types was prepared (refer to [17] for a complete description). In the scenario, video is pre-encoded in different combinations of frame rate, resolution, and colour scheme. Figure 7 shows the service model (simplified) of the video streaming application, which has alternative implementations at both sub-service and atomic service levels (i.e., it utilise the three configuration methods).

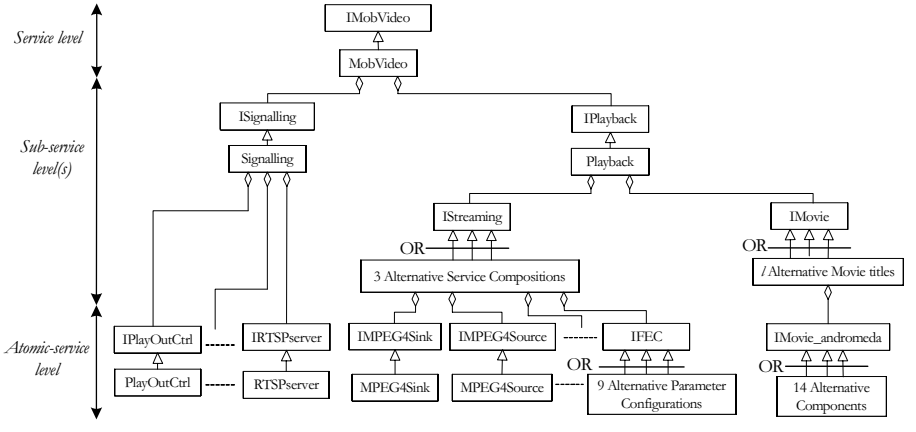

Fig. 7. Service Model of a Video Streaming Application

Even though our implementation has only one video title, the service model specifies a total of 266 alternative application configurations. After converting the service model into deployable artefacts, 39 service plans are derived. Compared to solutions which employ one architecture model for each configuration, our service model principle reduces the number of specifications by a factor of six. Furthermore, the combination of service levels and service plans divides the complex task of modelling

context dependencies and QoS characteristics into small, manageable pieces. Thus, we avoid the explicit specification of all configurations, whether it is as one large or multiple individual specifications. A task that proved to be difficult was defining the QoS mapping functions. We found that this requires benchmark test results of components running on different classes of hardware, operating system, run-time environment, and network.

Both service type and service plan are included in the architecture of our context- and QoS-aware middleware; QuAMobile (QUality of service-Aware component Architecture for MOBILE computing) [6][7]. The core of the architecture, depicted in Figure 8, has hooks for domain specific plug-ins *service planner*, *context manager*, *resource manager*, *configuration manager*, and *reconfiguration manager*. To test the ability of the dynamic middleware to choose an application configuration suitable for a particular environment, we have designed a graphical test tool where the characteristics of the environment are defined. For entering user QoS requirements, a Web-based interface is hooked up to QuAMobile through an Applet and a façade to the *service context*. In our implementation, service types are deployed as WSDL-files and service plans as XML-files. During loading service plan XML-files are interpreted, and information from the tree (illustrated previously in Figure 5) is extracted using the Java Document Object Model open-source software, rel. 10 beta. Created components are placed in the *repository*, while types and plans are published in the *broker*.

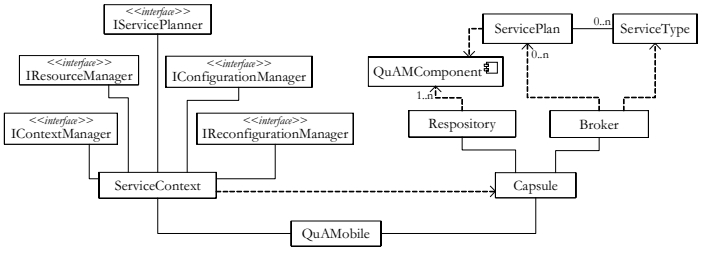

Fig. 8. QuAMobile Core

During planning the deployed service plans and the abstraction levels of the service model come into practical use. The planning phase commences when a service request with the name of the service type and the user's QoS requirements are sent from the presentation layer (Web-pages and Java applets) to the business layer (where QuAMobile resides), and to the service planner. The implementation uses the service configuration pattern illustrated in Figure 6. An instance of the service configuration class synthesises the alternatives by resolving the application from the top. At any point where it detects alternative service plans for the same service type, it clones itself and asks each new service configuration object to continue resolving from the level that has been resolved till now. Synthesized service configurations that can not execute in the current environment are filtered, by checking the specified context dependencies against information from a shared context and resource data model [5].

Next, the QoS characteristics are calculated by using mapping functions, which map between different QoS levels (resource-application, application-application, and

application-user). The QoS mapping functions have variables, such as resource QoS, context properties, and other QoS mapping functions. Together with the mapping function, these are processed by our QoS calculation software, which is based on the Java Expression Parser, rel. 2.3. Figure 9 shows examples of QoS mapping functions. In the implementation, QoS prediction starts at the atomic service level (bottom-up calculation). Predicted QoS is stored inside the service configuration object. After predicting the end-to-end QoS characteristics, the service planner removes service configurations that have service implementations that do not meet the specified *min-QoSValue* and *maxQoSValue* (see Figure 5). The last task of the planning process is to check the predicted QoS against the user QoS requirements, which in QuAMobile are expressed in dimensional utility functions.

```
<mapping_id>startUpTimeplay
   <funcDimension>startUpTime</funcDimension>
   <function>(((tmpeg4 SMALLEREQUAL 1000) = 5);
                    .  .  .
             ((tmpeg4 SMALLEREQUAL 10000) = 2);
             ((tmpeg4 SMALLEREQUAL 31840) = 1))</function></mapping id>
```

a)

```
<mapping_id>tmpeg4
   <funcDimension>startUpTime</funcDimension>
   <function>tmpeg4source+trtpTrans+2*tfec+tprefetchStart+12*tmpeg4sinkEmptyB</function>
</mapping id>
```

b)

```
<mapping_id>trtpTrans
   <funcDimension>delay</funcDimension>
   <function>vbitRate/(20*RrtpTrans)</function></mapping_id>
```

c)

Fig. 9. Example of QoS mapping functions: a) user QoS to utility, b) application to user QoS, c) context-resource to application QoS

The chosen service configuration is forwarded to the *configuration manager*. If/ when the resource monitors or context sensors detect changes in the environment, the planning phase is restarted, and if successful, a list of components to delete, create, and new bindings, are forward to the *reconfiguration manager*.

5 Related Work

When developing applications for dynamic middleware, Architecture Description Languages (ADLs) can be used for specifying the functional aspects of the application configuration. A number of ADLs are available; see the discussion of ADLs provided by Medvidovic et al [8]. Some of these ADLs, such as ACME [9] and Darwin [10], support hierarchical composition, but they need to be extended with support for alternative configurations at different abstraction levels. Furthermore, ADLs are, in general, a design-time artefact, and not intended for managing run-time adaptation.

There are ADLs that have been extended with reconfiguration steps, e.g., Plastik [11] and Rainbow [19]. Both use an extension of ACME which enables the application developer to specify (one) application configuration and a set of conditions under

which reconfiguration shall take place. A compiler converts the ACME specification to platform specific, executable files. Compared to Plastik and Rainbow, the service plan concept provides more information, e.g., parameter configurations and dependencies to context elements. Furthermore, the service plan concept assumes that the middleware has logic for deciding which part to reconfigure. In Plastik and Rainbow, this is decided prior to run-time using action policies/strategies.

To ensure that the application behaviour is maintained during reconfiguration, specifications of the reconfiguration steps have been developed (see survey by Bradbury et al [4]). These specifications assume that one particular application configuration is running, ignoring the need to find an initial configuration that can execute in an arbitrary environment. The service modelling principle and service plan concept address this weakness by providing specifications of alternative application configurations that can be used for both initial configuration and reconfiguration.

The QoS of an application is strongly dependent on the characteristics of the available resources. Thus, a specification must include the QoS requirements at resource level. However, users do not relate to resources; their perception of quality is subjective, e.g., sound quality, video contrast, or cost. Therefore, QoS requirements to an application are often specified at the user level and mapped down to resources. Such specifications are commonly referred to as QoS specification languages (see survey by Jin et al [12]). The XML-based Hierarchical QoS Mark-up Language (HQML) [13] and the Component Quality Modelling Language (CQML) [14] are two examples of QoS specification languages. HQML was designed for distributed Web-based multimedia applications, and uses XML-tags for the partitioning and for information elements. The XML file is associated with the Web application, and accessed by a Web-client with a plug-in that interprets the information elements inside the XML file. CQML, in addition to specifying user and application QoS requirements, can be utilised in UML-based analysis models of the application, enabling both model-driven QoS-awareness and run-time QoS interpretation and mapping. A principle difference between QoS specification languages and our service plan is that the service plan is developed as a concept for the application life-cycle. If considering only deployment, there are similarities between a service plan and HQML or CQML specifications, because they both relate resources to the user level. With respect to information elements, the service plan has support for specifying context dependencies, a feature that is generally lacking from QoS specification languages [12].

There are examples of research projects that address all phases of the application life-cycle and QoS-awareness, e.g., $2K^{Q+}$ [15] and Quality Object (QuO) [16]. $2K^{Q+}$ provides a QoS software engineering environment for specifying alternative component compositions and their QoS characteristics, which are then compiled for running on the $2K^{Q}$ middleware. Part of this environment is the QoSTalk [13] graphical programming and consistency checking tool, which uses the HQML QoS specification language. This engineering environment employs a platform dependent compiler, i.e., it produces executable code for (re)configuring of the application. Furthermore, it requires that the middleware has probing facilities that can measure QoS and resource usage for a test-run of the application configuration. Results from the QoS probing are fed into the compiler and, thus, set the conditions for reconfiguring the application. The QuO framework relies on a suite of description languages for specifying QoS. Specifications are compiled to executable code, which is used for monitoring QoS and

controlling the interaction between distributed objects across a CORBA middleware. Our service plan concept and QuO specifications serve the same purpose, but a service plan has more information, is platform independent, and thus represents a more flexible solution.

Most existing approaches assume that the target environment is known at design time. One example of a middleware that is not based on this assumption is the Context-Aware Reflective mIddleware System for Mobile Applications (CARISMA) [18]. It uses application profiles to (re)configure the middleware. If a mobile device is used in an unforeseen environment, the application can adapt to the profile, and thereby change the behaviour of the middleware. Thus, the application profile is a dynamic specification, while the service plan is static in order to enable predictable (re)configurations. The service plan concept supports unforeseen environments, by allowing alternative service plans of the same application.

6 Conclusions

This paper focuses on how dynamic middleware for the mobile domain can utilise alternative application configurations. We have identified three issues that are important to address in order to achieve such platform-based configuration, and that current state-of-the-art solutions fail to target: first, the heterogeneity of both hardware and software within this domain means that the developer must be provided with means to manager a large number of alternative configurations; second, QoS characteristics of the different configurations must be specified, to enable the middleware to select among the alternative configurations; and third, to enable separation of concerns, and thereby reuse, the specifications must be platform independent.

Our approach for handling these three issues is based on a service modelling principle for designing a large number of variants, and a service plan concept used to connect service types to implementations of the types. Service plans also specify the QoS characteristics and context dependencies of the implementations. Using the life-cycle of an application we have presented a qualitative assessment of our approach, and demonstrated how the service model and the service plans, together, cover all phases of the life-cycle. Finally, we have described our implementation of the principle and the concept in our dynamic mobile middleware called QuAMobile, which serves to demonstrate the feasibility of our approach.

Currently our work is on the design of the causal connections from the service configuration object and the service plans, down to the running component instances, and we are studying the integration of the service plan into a software engineering tool.

References

1. Coulson, G., Blair, G., Clarke, M., and Parlavanzas, N.: The design of a configurable and reconfigurable middleware platform. Distr. Computing Journal, Vol. 15 (2002), 109-126
2. Roman, M., Kon, F., and Campbell, R.: Reflective Middleware, From Your Desk to Your Hand. IEEE Distributed Systems Online, Vol. 2, No. 5 (2001)
3. Huhns, M.N., and Singh, M.P.: Service-Oriented Computing: Key Concepts and Principles. IEEE Internet Computing, Vol. 9, Issue 1 (2005), 75-81

4. Bradbury, J., Cordy, J., Dingel, J., and Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specification. Proc. of the ACM SIGSOFT International Workshop on Self-Managed Systems (2004), 28-33
5. Amundsen, S., and Eliassen, F.: Combined Resource and Context Model for QoS-aware Mobile Middleware. Accepted for the 19th International Conference on Architecture of Computing Systems, (2006)
6. Amundsen, S. Lund, K., Eliassen, F., and Staehli, R.: QuA: Platform-Managed QoS for Component Architecture. Proc. of the Norwegian Informatics Conference (2004), 55-66
7. Solberg, A., Amundsen, S., Aagedal, J., and Eliassen, F.: A Framework for QoS-aware Service Composition. Proc. of ACM International Conference on Service Oriented Computing, ACM Press (2004).
8. Medvidovic, N., and Taylor, R.N.: A Framework for Classifying and Comparing Architecture Description Languages. Proc. of the 6th European Software Engineering Conference (1997), 60-76
9. Garlan, D., Monroe, R., Wile, D.: ACME: Architectural Description of Component-based Systems. Foundations of Component-based Systems, Cambridge University Press (2000), 47-68
10. Magee, J., and Kramer, J.: Dynamic Structure in Software Architectures. Proc. of ACM SIGSOFT'96: 4th Symposium on the Foundations of Software Engineering (1996), 3-14
11. Batista, T., Joolia, A., and Coulson, G.: Managing Dynamic Reconfiguration in Component-based Systems. Lecture Notes in Computer Science, Proc. of the 2nd European Workshop on Software Architecture, Vol. 3527 (2005), 1-17
12. Jin, J., and Nahrstedt, K.: QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy, IEEE Multimedia Magazine, Vol. 11, No.3 (2004), 74-87
13. Xiaohui, G., Nahrstedet, K., Yuan, W., Wichadakul, D.: An XML-based Quality of Service Enabling Language for the Web. Journal of Visual Language and Computing, Special issue on Multimedia Languages for the Web, Academic Press, Vol. 13, No. 1 (2002)
14. Aagedal, J.: Quality of service support in development of distributed systems. Ph.D. thesis, University of Oslo (2001)
15. Wichadal, D., Nahrstedt, K., Gu, X., and Xu, D.: $2K^{Q+}$: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework. Lecture Notes in Computer Science, Proc. of the ACM International Conference on Distributed Systems Platforms, Vol. 2218 (2001), 373-394
16. Loyall, J., Bakken, D., Schantz, R., Zinky, J., Karr, D., Vanegas, R., and Anderson, K.: QoS Aspect Languages and Their Runtime Integration. Lecture Notes in Computer Science, Proceeding of the 4th International Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, Vol. 1511 (1998), 303-318
17. Amundsen, S., Lund, K., Griwodz, C., and Halvorsen, P.: Scenario Description –Video Streaming in the Mobile Domain, Technical report (2005), http://www.simula.no/~stena/techReports/ScenarioDescription/ScenDesc_MobVideo_B1.pdf
18. Capra, L., Emmerich, W., and Mascolo, C.: CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, Vol. 29, No. 10 (2003), 929-945
19. Garland, D., Cheng, S-W., Huang, A-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Computer, Vol. 37, No. 10 (2004), 46-54