SERVICE PORTABILITY FRAMEWORK FOR INTEGRATED COMMUNICATION ENVIRONMENTS

Dmytro Zhovtobryukh[1] and Veikko Hara[2]
[1]Department of Mathematical Information Technology, University of Jyväskylä, 40014 Jyväskylä, Finland; [2]TeliaSonera Corporation

Abstract: Future services must become intelligent to meet the high demands of pervasive computing environments. But until pervasive systems with their ambient intelligence supersede conventional mobile computing environments, it is quite a challenge to incorporate context-awareness and adaptability in services currently available. Such step would bring outstanding flexibility and ubiquity to contemporary mobile computing systems and semantically rich web environments. This paper presents a distinct vision of portable service provisioning which elaborates the concept of a portable service by proposing a dynamic reconfigurable service application design based on context-aware infrastructure support.

Keywords: service portability, service adaptation, network interoperability, context awareness, semantic ontology, industrial Semantic Web environment

1. INTRODUCTION

Current services are designed to operate in specific communication environments. The approach is rather awkward as far as bringing ubiquity into computing and communication to prospective customers is concerned, especially taking into account the growing tendency to integrate modern communication systems. But regardless of whether two adjacent network systems are interconnected or not, the operating service application in the majority of cases should be immediately terminated and reinitiated as the user's terminal crosses the border between the two systems and reassigns its connection to the new one

in its range. This is a problem due to the lack of appropriate infrastructure support enabling seamless operation of service application throughout multiple network systems. However, even where it exists, such infrastructure support does not spread on more than two systems. Needless to say that beside apparent performance issues this, beyond doubt, distracts potential users. Therefore it is highly desirable to provide customers with services that roam among interconnected network systems and even various devices, in an effectively transparent and continuous manner. To achieve that, pervasive computing environments, richly endowed with ambient intelligence, would be of great assistance, but unfortunately they are yet a long way from being widespread. Nonetheless, certain steps towards more universal service provisioning and seamless service consumption can be made already now.

What should be dealt with in the first place is the rigidity of services. They should not be oriented towards specific environments, but should be flexible so that they can be consumed by different users, through diverse communication systems and with various devices. Pervasive (ubiquitous) computing[1,2,3] paradigm addresses this issue by decoupling services, applications, devices and users from each other and viewing them as completely independent entities. They are no longer firmly tied together, but have their own functions and objectives and interact with one another when needed. In particular, applications are seen as special entities that perform specialized tasks on users' behalf. Appropriate infrastructure support allows them to be highly customizable and personalized according to users' needs, roam freely between various devices, adapt to changing environmental conditions and be independent of the underlying communication technology. Similar infrastructure support would be a desirable amendment to modern computing systems as well. Not only would it increase service reusability and improve users' perception, but also it would bring current communication standards closer to each other and alleviate the further escalating problem of network interoperability.

This article describes a vision of what is adequate infrastructure support for present-day interoperating communication environments. We propose a reflective context-aware infrastructure for building, rapid prototyping and dynamic adaptation of portable service applications. Some specific details about the proposed service provisioning framework are omitted or not addressed yet and left for

further study. However, we believe this article gives a good idea of portable services and of network interoperability problem.

Context-aware computing is nowadays one of the hottest research fields in communications because adaptive intelligent applications are currently in great demand. Numerous researchers and research groups actively develop middleware infrastructures for adaptable context-aware service applications. They seek fresh and robust solutions for next generation computing and networks. However, the majority of proposed infrastructures have limited application due to their orientation on the future communication standards or on highly specific practical implementations, such as smart spaces. For example, Chen and colleagues[4] work on development of smart meeting room system called EasyMeeting that relies on the agent-based context-aware middleware infrastructure Cobra. Gu and colleagues[5] develop an interesting context-aware architecture, which is based on the Open Service Gateway Initiative and is utilizing semantic ontology reasoning, for smart-home environments. Hewlett-Packard's project Cooltown is focused on a Web-based infrastructure for context-awareness[6]. ContextToolkit features a programming approach for modeling and rapid prototyping of context-aware applications[7]. Some other related research activities are described by Chen and Kotz[8]. In contrast to them this article portrays an infrastructure that can be applied to a wide spectrum of contemporary communication environments on a rather wide scale and it specifically focuses on service provisioning and delivery. We feel that the application of our vision to Semantic Web environments is currently one of the most challenging, since semantically rich Web services have to be context-sensitive to become really intelligent.

The article is organized as follows. After this introductory part we present our vision of the Service Portability framework in Section 2. In Section 3 we will speak in detail about service adaptation patterns and introduce the sketch design of our context-aware middleware infrastructure. Section 4 focuses on the perspective of a particular application of the presented vision in an industrial Semantic Web environment. Finally, Section 5 concludes the paper discussing the lessons learned and motivating our future work.

2. SERVICE PORTABILITY FRAMEWORK

The main design goal of the service portability framework is to set up a base for seamless provision of any service through any type of environment. By environment it is meant here a particular combination of physical surroundings and computing environment. The latter is the most important aspect because operational characteristics of each environment mainly depend on the present computing facilities and deployed communication standards. By seamless service provision we mean that services are delivered to users in a continuous and distraction-free manner regardless of any changes that may occur in an environment during an active service session. Though such service provisioning paradigm is quite a challenge, and may be unattainable in practice, the objective is to make services as independent of environments as possible. The main idea is not new: to distinguish between two service instances, one of which is unique and as generic as possible, and the other one a highly specific implementation of the service. No matter how the potential environments differ, it should be ensured that the service is always presented by the unique global instance. This instance should always stay unchanged to ensure that the service remains the same while its specific implementations are customized with respect to the requirements of concrete environments.

In order to achieve this goal, it is necessary to introduce a special service provisioning architecture that would separate the global unique service instance from its actual implementation for a specific environment. (Similar ideas were initially proposed by Banavar and collegues[2]). Such architecture should manifestly adhere to two-phase principle of service provisioning, where the individual phases are virtually independent and concerned with generic and specific service instances respectively.

It is plain from the definition that the two-phase service provisioning comprises two distinct phases. The first phase is *service creation*. It consists of design and deployment of the global instance of the service. Let us call this unique instance a *generic service*. "Generic" here means that on this stage the service definition is devoid of any specific properties related to environments where the service is intended to be used. The closest practical analogue of a generic service is a Web service, which is relieved from low-level details and is advertised by semantic service description that is

practically a collection of metadata describing a service. Despite the similarity we intentionally distinguish a Generic service from a Web service to eliminate any association of our conceptual view of a service with its concrete technological realizations for now.

The other phase is *service delivery*. It embodies a construction of a service application from the generic service created during the service creation phase. The service application is already a specific implementation of the service. And since it is aimed for use within a particular environment, all the necessary properties and functionalities are incorporated into it. In other words, every single application of the same service specifically accommodates to the requirements, properties and restrictions of the concrete environment. It must be noted that at this stage, while the application might be altered, the service still remains unaltered. Having been constructed and launched, the application should continuously stay tuned to the requirements until its termination. However, there is a tendency of environmental conditions to progressively change, which makes the process of delivering service applications more complicated.

The concept of service portability implies not only that every generic service may obtain a multitude of differing applications in various environments, but also that during service delivery environments and their properties may dynamically change. As long as such changes render service applications inadequate or even inoperative, the applications should be adapted at run-time not to disrupt pending service sessions.

The function of applications' adaptation is performed by a specific middleware for service portability (as well as by the applications themselves or in collaboration between the applications and the middleware). We propose a context-aware reflective middleware architecture for service portability. Context-awareness plays an important role here as it is a good foundation for application adaptation frameworks and as it allows an even treatment of environmental conditions along with other contexts. Reflectivity is a vitally important property for such middleware, since it allows capturing dynamics, making the architecture viable in a highly dynamic mobile environment.

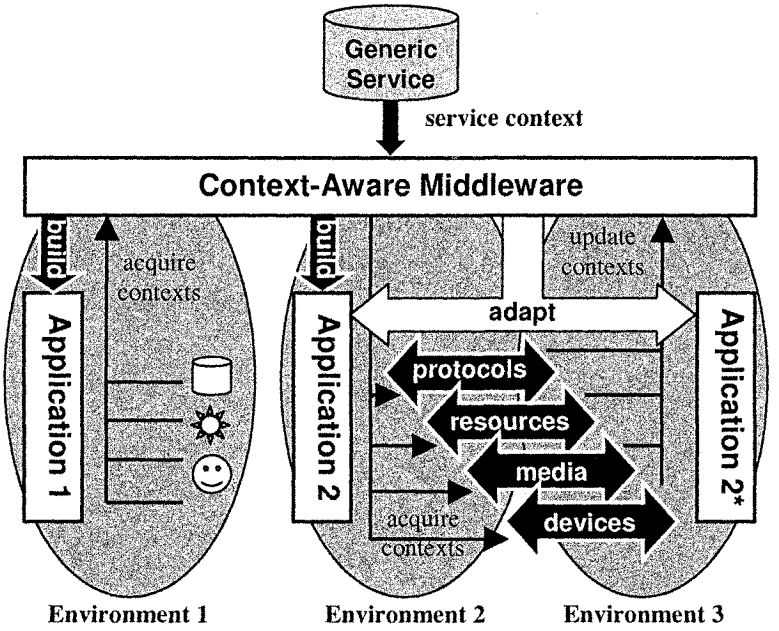

Figure 1. General view of the service portability framework

Figure 1 shows the service portability framework in a general setting. When a user or some application invokes a service from the service provider, the context-aware middleware receives a service context of the corresponding generic service located in the provider's network. The middleware also analyzes various context information acquired from the target environment, and builds a new service application based on the results of the analysis made. As it can be seen from Figure 1, in this way the middleware can produce different service applications (Application 1 and Application 2 in the figure) for different environments from the same generic service. All the necessary information influencing the application being built is obtained from within the context acquired from the target environment. However, the environment, where the application is running, can suddenly change. For example, user may move to another device, or hand over to another environment with different transmission medium and protocol stack, some resources in the environment may vanish or their levels may change. Though this list is far from complete, all these changes (shown in Figure 1 with thick black arrows) essentially mean that the environment has changed. All of them are retrieved by

the middleware in the form of context at run-time. At this point the application may turn inoperative, since the environment has changed. To prevent that the middleware has to perform a run-time adaptation of the application (or the application can adapt itself, if it is able to do that). The middleware detects context changes, analyzes them and performs an adaptation procedure on the application.

2.1 Service creation

As it was already noted above, at the service creation stage of service provisioning procedure a generic service is being designed. The obvious objective of this design effort is to supply the service definition with all the necessary data to make the service instance unique, unambiguous and easily interpretable. The generic service instance should be unique in the sense that it should be the only one for the purpose and distinguishable from any other service, i.e. it would not be confused with any other similar service. At the same time the service definition should not be overloaded with any redundant details, which might, in the worst case, turn the service useless for certain environments. A generic service basically contains only metadata about the content of the service and about its operational requirements. Properties related to network standards, protocol stacks, data formats, transmission characteristics, and device capabilities are application-specific and should be implemented within an actual service application. In Web Service Architecture such metadata is called service description[9]. For example, it may provide a link to the service content source, list restrictions on capabilities of end terminal equipment, store some authentication information, etc. Metadata contained in a generic service is to be used during the service delivery stage to construct an appropriate service application which would deliver the service to an end user in the most consistent and reliable way.

Such generic service design is beneficial from several points of view. First of all, it allows maintaining a unique globally available service instance, which is easily recognizable and is not likely to be confused with any other service entities. Secondly, this instance is really generic, which means that it does not possess any implementation-specific properties. Being generic, it does not shrink the range of possible specific implementations of the same service and increases potential reusability of the service. Finally, due to separation

of the service and its application, the generic service design allows service porting between diverse environments without any alterations of the service itself.

One more attractive opportunity offered by the generic service design is service composition. In such service provisioning framework two or more services can be jointly delivered by a single application in a rather simple fashion. However, complex interaction of two services at run-time would still present a quite complicated problem.

2.2 Service delivery

Service delivery phase begins with the construction of the service application and ends with its termination. This phase can be logically split into two sub-phases: load-time and run-time. At load time, i.e. build time, the application is constructed from the generic service and is brought into conformity with initial conditions (context) ascertained from the environment. This way, the application is particularly adjusted to operate on top of certain terminal capabilities, protocol stack, resource level, etc. Thus, it is guaranteed that the newly launched application does not fail to perform correctly at startup. At run-time the application runs as normal, but it adapts or gets adapted whenever essential environmental changes occur. Essential environmental changes can be understood as changes in the environment that may influence or even adversely affect the application's performance and operability. Run-time adaptation of applications leads to preservation of service session continuity, which is indispensable for service quality and robustness.

In order to create applications in a rather automatic manner, i.e. without real human designer's intervention, a special framework for application design is needed. An attempt to create a theoretical foundation for such automatic service application design was made in our previous work[10]. In essence, this research work proposes a service reference model for characterization of services with respect to different functional layers, which exist in the system and implement certain functionalities related to services. The main concern of the elaborated model is to create accurate layering of service-related functionalities so that they could be developed relatively independently. What is this needed for then?

Current services are vertical services. This means that they are tightly built into applications that deliver them. They are developed

with "all-in-one" principle in mind. All the service-related functionalities from user interface to content rendering are rigidly built into the service instance. The service is seen as a monolith having indivisible structure, which is rather implicit than explicit. Such approach is inflexible in the sense that the service is adaptable only to the extent that is stipulated already at the stage of construction. To illustrate this comment, one can imagine, for example, a GSM short message service (SMS). SMS is designed to exchange text messages, it cannot exchange images and videos. To provide these new possibilities a completely new service called multimedia messaging service (MMS) was created. The restriction to text type of content is implicitly built into SMS, and it cannot be modified to accommodate other types of content. Instead a completely new service needs to be created. Another example is mobile telephony. A user cannot normally cross the border between two different communication networks (by different we mean different network standards) without interruption to the active call, because the application, which delivers calling service to the user, is not designed to be operated on top of a different protocol stack.

An alternative view on the service design focuses on transition from rigid vertical services to horizontal ones. Horizontal services are those provided by the environment. They bring certain system functionality to applications. For example, transport service provides such functionalities as packet transmissions, end-to-end security, traffic management, etc. Applications utilize these functionalities to achieve their specific goals. However, whenever some of the available horizontal services change (like in the last example with GSM telephony), applications are no longer capable of pursuing their own goals, having been intentionally designed to operate on top of a narrow set of horizontal services. In the modern communications world with its strong integration trends such a state of affairs appears increasingly unacceptable. Instead, horizontal services should be effectively used to alleviate the efforts on application design and to make service applications more flexible.

According to our idea, the structure of an application should be modular. It should comprise several functional layers similar to those described by Zhovtobryukh and Kohvakko[10]. Each layer may contain multiple modules that perform specific functions. However, these modules should not be necessarily predetermined at the stage of application construction. Instead, the structure has to be flexible so

that modules can be added, removed or substituted at any time. They are to be tied together by some sort of unified interface, which would allow for certain variety of structural alterations. These modules are called *service primitives* and should be mostly provided by the environment where the service application happens to be operating. The role of the modules is not to perform a certain functionality, but to utilize corresponding functionality of horizontal services available in the environment, and to transform this functionality into the form which the application needs.

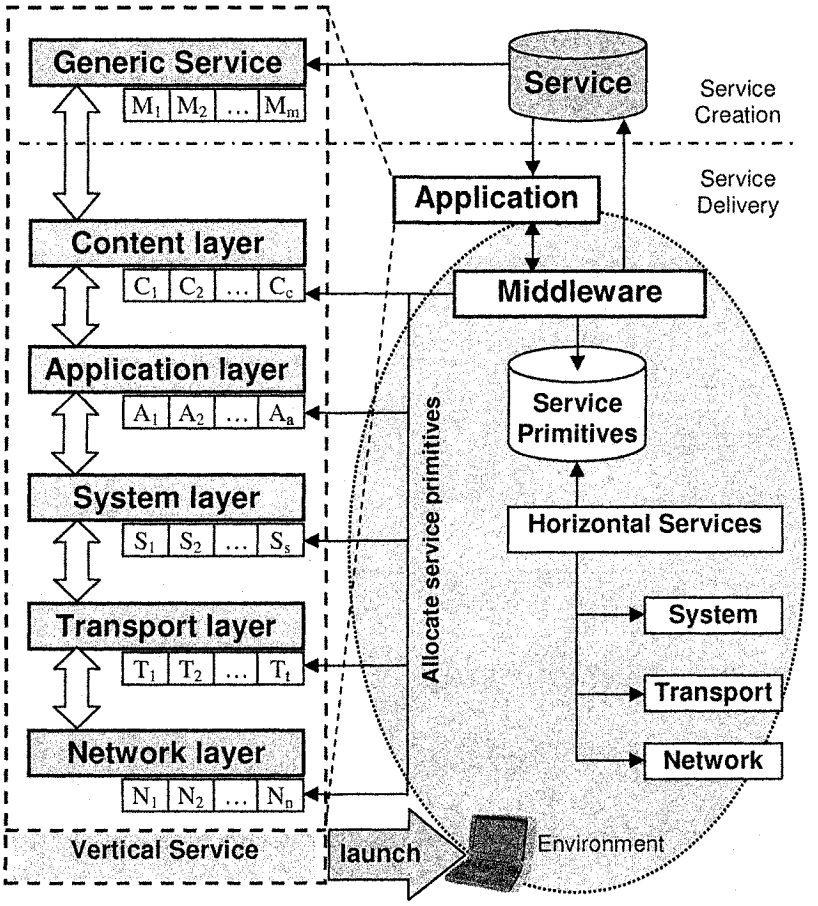

Figure 2. Two-phase service provisioning architecture

Let us explain how this works in a greater detail. Figure 2 depicts the scheme of dynamic assembling of a vertical service application

from service primitives available in a concrete environment. To illustrate this scheme in a rather practical way let us use the following scenario.

Steve is at home and is going to leave soon for a lecture which takes place at the university campus. But he knows that a televised football match he has been waiting for the whole day is to start in 5 minutes. Steve takes his handheld device, which has a wireless local area (WLAN) connection to the Internet and through the web interface of the corresponding TV channel's site invokes a video streaming session to watch the broadcast from the match on the way to the university. In the provider's network there is a generic service that provides a connection to the TV broadcast server. The middleware sends a query to this generic service and gets the service context of the video streaming service. This context data contains the address of the TV broadcast server, authentication information for establishing a connection to it and some technical requirements that should be met in order to receive the video streaming service. After that the middleware acquires the context from the environment, namely the device capabilities from the profile of Steve's handheld, characteristics of the active wireless link, traffic management schemes supported by the underlying system architecture, protocol stack used in the WLAN network, etc. Having analyzed this context information, the middleware starts the process of application construction. It locates necessary service primitives with respect to the results of context analysis made and assembles them to get an application. For example, the middleware may obtain video and audio content primitives from the broadcast server to make the application capable of processing and outputting streaming video and audio. It also obtains, from the local environmental repository, the primitive which allows the application to receive packets of streaming traffic type. It is important that the local WLAN network enables horizontal transport service that supports the streaming type of traffic with appropriate Quality-of-Service (QoS) policies. Similarly, the necessary primitives for all the layers shown in Figure 2 are found. Once all the primitives are allocated the vertical service application is finalized and launched on Steve's handheld device. Steve takes his device with him and watches the broadcast on the way to the university.

Thus, a vertical service no longer has a fixed structure. Instead, it is adjusted to concrete environmental requirements by adding

appropriate service primitives. This way a multitude of specific implementations of the same generic service can be created.

Portability of a service in an active state, i.e. a service being delivered to a user at that moment, is achieved in a very similar way. At the application's run-time the environment may suddenly change due to the user's switchover to another environment. Having found itself in another type of environment, the application may turn inoperative, which means that some of the currently equipped service primitives are, perhaps, no longer suitable to deliver the service through the new type of environment. Therefore, the application must, accordingly, be adapted to correctly operate in the new environment. The adaptation procedure, which is to be applied in this case, is essentially a substitution of invalid service primitives with valid ones. New primitives are located within the new environment and forwarded to the application. The application reconfigures its structure with the new service primitives and starts operating as normal. The adaptation procedure is primarily controlled by the middleware.

Continuing the previous example, soon after leaving home Steve also leaves the range of his home WLAN connection. His handheld device automatically switches connection to a wide area GPRS network available in the new outdoor environment. The middleware immediately recognizes that changes have taken place in the context and determines that the video streaming application is no longer capable of operating in the new environment. It collects contexts from the new environment, analyzes them and determines which service primitives in the application's structure clash with the new requirements. For instance, the middleware may discover that a different protocol stack is used in the GPRS network. Hence, packets of different format should be received by the client. The middleware contacts a local repository containing service primitives and substitutes all the obsolete primitives within the application with the operational ones. This way the application is being ported to another environment without being terminated and re-launched. In the best of the cases Steve would not even notice the switch between environments. But, generally, some slight distraction period may be evident if the volume of context data is quite significant for fast processing. In the case of non-real-time or asynchronous services strict continuity of a service session is not as important. In these cases packets can be buffered by the middleware somewhere within the system for a later retrieval by the application. The end result of all this

is that, without taking any additional actions with his handheld, Steve is watching the TV broadcast while sitting on the bus on his way to the university.

It should be noted that a change of an environment does not necessarily imply that a user has handed over to a different network system. It may also indicate that he has switched over to a different terminal, and an application should immediately "teleport" to the user's new device to preserve the service session from being prematurely disrupted. Some significant changes in the environment, such as, for example, base station crash or resource saturation, will also lead to re-selection of service primitives in order to adapt the application's performance to these new operational conditions.

3. SERVICE ADAPTATION MECHANISMS

The core element of the service portability framework is the context-aware middleware. It is designed to perform actual porting of service applications. The middleware is in charge of service adjustment at application's load-time and of application adaptation at application's run-time. However, application-specific issues cannot be sensibly managed by the middleware. They require application-aware adaptation, in which the middleware assists the application. The types of supported adaptation procedures are described in the subsections below.

In the proposed service adaptation framework all the data which are used as a basis for adaptation are treated as *context* for the purposes of uniformity and simplicity. This would allow collecting and storing all the necessary information in a common repository and inferring adaptation decisions in an easier way.

There are numerous definitions of context in various research fields from artificial intelligence to distributed computing. Some of the most popular definitions can be found in the works of Schilit *et al.*[11], McCarthy[12], Chen and Kotz[8]. However, we believe that the most comprehensive and convenient for our needs definition is provided by Dey[13]: "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves". This definition serves well for the proposed architecture because it

considers any information related to different entities which are involved in the service provisioning process as context. This view resembles the vision we described in Section 2.

Various contexts are collected by numerous context providers, such as sensors, monitors, and such special sources as user and equipment profiles. Physical contexts, such as location, signal strength, levels of interference, etc., are measured by sensors. Diverse network contexts, e.g. resource levels, are collected by monitors. Service context is retrieved from metadata provided within the generic service. User context can be obtained from user profiles. All the collected contexts are examined by the middleware and stored in the database. The way of modeling context for this infrastructure is a complex issue, especially taking into account such a broad definition of the utilized contextual information. However, we feel that ontological representation of context information is the most flexible and well-developed. Therefore, we assume that context ontologies are used for context modeling. The benefits of this approach will be discussed further in Section 4.

It should be noted that in this article we only consider functional organization of the middleware infrastructure. Architectural layout of its distributed composition is not described yet in detail. However, we will briefly describe the principle of our distributed composition. The proposed middleware architecture consists of three main functional blocks:

- Context Acquisition Proxy
- Context Manager
- Context Reasoning Engine

The middleware manages three data repositories:

- System Contexts
- Context Model
- Service Primitives

Context Acquisition Proxy controls the population of context providers. It collects readings from sensors/monitors at their report rate and sends updates to the context manager. Acting as a proxy, this facility filters received readings. It discards all inessential changes in monitored contexts using certain implicitly given criteria for their estimation.

Context Manager is responsible for maintaining the context model in a consistent state by updating it promptly with essential context changes. It also controls context information exchange between

different architectural entities and manages context dissemination procedures.

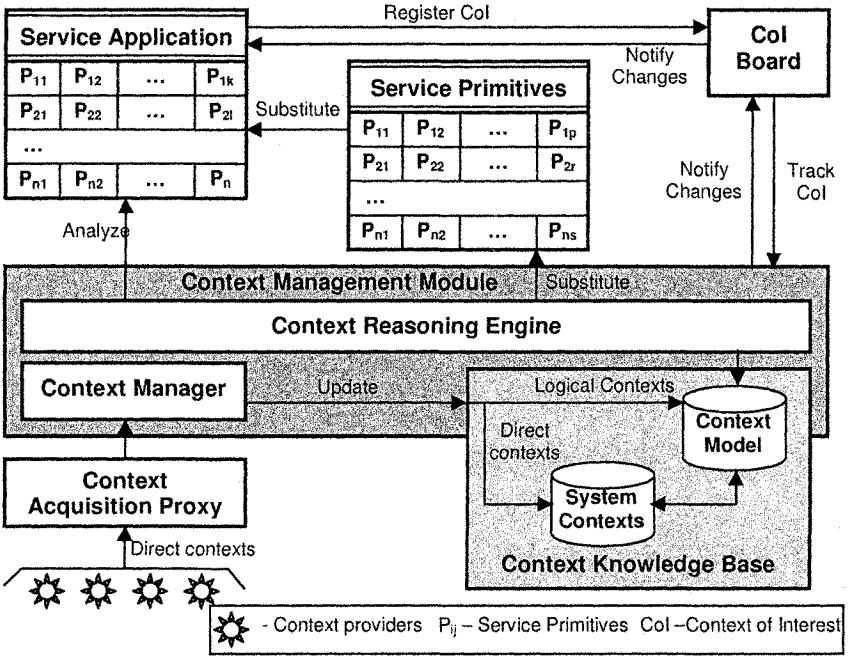

Figure 3. Service adaptation framework

Context Reasoning Engine is the brain of adaptation. Its apparent role is to perform reasoning on the context. The engine checks for changes in the context knowledge base looking for possible conflicts between the application and system contexts. By conflict it is referred to a certain discrepancy between the application's operational properties and the current environment parameters. This discrepancy would form a hindrance for the application to operate properly in the observed conditions and must, therefore, be resolved during the adaptation process. Whenever the context reasoning engine determines such a collision, it starts reasoning on the base of the context model trying to deduce a satisfactory solution for the detected conflict. The reasoning procedure may or may not cause a launch of an adaptation procedure. Adaptation, as an ultimate way of conflict resolution, usually results in a substitution of application's service primitives that clash with occurring contextual conditions. On basis of the obtained reasoning decision the engine selects new primitives

from the corresponding repository and instructs the application to substitute them (see Figure 3).

System Contexts Repository is a relational database that stores the whole variety of context variables monitored in the system and their current values.

Context Model Repository contains an ontological model of contexts present in the environment. It may express both physical attributes and logical properties of those contexts. The model is capable of formalizing complex relationships between different contexts. It is maintained in the actual state and gets updated as necessary by the context management facility.

Service Primitives' Repository stores an array of service primitives compatible with a current environment. The primitives are built in correspondence with the horizontal services that the environment provides. These primitives are basically pieces of executable program code. They are shaped as program modules and can be linked to each other through a unified interface. The primitives are constructed by special middleware services, which are out of the scope of this paper. If some substantial changes occur to the environment and influence its horizontal services, the corresponding primitives need to be removed from the repository and new primitives added. The service primitives' repository is used in the adaptation procedure, whenever the middleware decides that some primitives of a certain application are not fit to operate in a current environment. In this case, the middleware makes a decision about which primitives should be selected from the repository to substitute the application's obsolete ones.

From the architectural viewpoint, the middleware is composed in a distributed fashion. The main architectural entities are *Context Management Module* that comprise context manager and context reasoning engine facilities, and *Context Knowledge Base* that consists of context model and system contexts repositories. These two modules are centralized for a certain locality in an environment. Context acquisition proxies are allocated locally for groups of context providers present in certain parts of an environment. Data exchange between local proxies, central management modules and context-aware applications is based on a certain protocol. A survey or any justification of such protocols is beyond the scope of this article. The operation of the middleware architecture is synchronized by means of

event triggering system, which allows capturing unexpected situations in contexts.

The operation of the described middleware architecture depends on what type of adaptation procedure is being currently accomplished. The following subsections describe some possible adaptation procedures and specifics of the middleware operation for each of them.

3.1 Background adaptation

Background adaptation is an adaptation procedure that is entirely handled by the middleware without the application's assistance. The application under adaptation may not even be aware of the adaptation performed on it.

This is a default type of adaptation. The middleware always tries to perform it first, and only if the detected problem cannot be resolved by the middleware, it would proceed to other adaptation procedures.

Background adaptation is generally utilized when inappropriate service primitives of the application belong to the layers that correspond to the horizontal services provided by a local environment. For example, if a user has made a handover to another network system which uses a different protocol stack for transmitting packets, or different QoS framework are utilized, then certain service primitives on lower layers of the service reference model may appear inappropriate for a new environment and consequently turn the application inoperative. The middleware is capable of detecting and settling such a problem by its own strength, since transport and lower system layers are transparent to the application. So, the middleware reassigns the primitives of transport and network layers imperceptibly to the application.

The role of the context reasoning engine is to analyze the configuration of the application, and the service context, and to detect which service primitives of the application are obsolete. After that the engine assigns, to the application, new service primitives from the local service primitives' repository.

3.2 Application-aware adaptation

Application-aware adaptation is an adaptation procedure in which the application is aware of the adaptation being made and makes the final decision on how it is to be adapted. This type of adaptation is

carried out by the application with a possible assistance of the middleware.

Application-aware adaptation usually takes place when inappropriate service primitives of the application reside on the layers that are not related to a local environment but to the service itself. Metacontent, content and application layers[4] obviously belong to this category. For example, after Steve's handover to a GPRS environment on his way to the university campus it may appear that the system cannot accommodate the quality of video streaming service due to overall scarce bandwidth. The middleware has already failed to satisfy the requirements of the service by performing background adaptation. Therefore, the adaptation has to be made by the application to reduce the quality of the delivered service. In the above example, the application should follow the recommendation of the middleware and stop using the video content primitive, which is too resource-demanding, and keep the audio content primitive only in order to deliver the TV broadcast in an accurate and error-free manner.

This type of adaptation cannot be handled by the middleware on its own because the service primitives, which need to be substituted, are not transparent to the application and comprise its core functionality. Furthermore, new primitives for substitution cannot be retrieved from a local environment in this case because of their absence in the local service primitives' repository. They can only be provided to the application in advance at load-time (or retrieved from the service's origin), so that the application could adapt itself in critical situations. The role of the context reasoning engine here is to detect the conflict, and to attempt to find a solution on its own - after the failure of the background adaptation to analyze what kind of application adjustment is required for a successful solution - and finally to propose a deduced solution to the application for a subsequent accomplishment.

Besides that, the application may be given some adaptation strategies at the construction stage. It may be required to adapt its behavior during its operation with respect to certain changing factors. If the application can perform self-adaptation, it is usually called *adaptive*. The proposed middleware architecture facilitates the support for adaptive applications by providing them with a possibility to register their contexts of interest (CoI) within a special tracking facility called CoI board. When an application registers its CoI within the CoI board, necessary contexts start being tracked by the middleware (if it is technically possible) and the changes are reported

directly to the CoI board. The board then notifies the application about the changes in its CoI, and the application can adapt its behavior as necessary.

The benefit of such a framework is in that the application may register quite complex contexts, which cannot be measured directly, but only inferred on the base of the context model. Such complex contexts are deduced by the context reasoning engine, thus relieving the application from sophisticated computations and giving it the possibility to adapt itself almost effortlessly.

3.3 User-aware adaptation

User-aware adaptation is the most complicated type of adaptation. It implies that the final adaptation decision is made consciously by the user because neither the middleware nor the application succeeded to provide a satisfactory service quality level by their own strength.

Nevertheless, all the work to find an appropriate adaptation decision is done by the middleware and the application. In case they succeed to find any reasonable solution, they propose it to the user, who finally decides what to do. Such a solution, if it can be found, is called a *corrective action*[3]. By performing this corrective action the user ensures the required level of service quality.

Let us use the previous example to illustrate this idea. After Steve's handover to a GPRS network system it appears that the available bandwidth is too scarce for a satisfactory quality of video streaming service. The middleware fails to solve the problem due to a low physical capacity of wireless links. The application fails to reduce the quality of the delivered service, because the video component of the broadcast appears critical at the moment (Steve manually set in advance the application's options to indicate that video would be critical to display). However, having analyzed the system and user contexts, the middleware and the application find a corrective action soon after Steve gets out of the bus: if he moves to the nearest lobby, which is situated 30 meters away from his current location, Steve will get an acceptable quality level of video transmission, because the lobby is equipped with a WLAN hot-spot.

In principle, the middleware is capable of deriving corrective actions without any assistance of the application. However, only the application is able to communicate the corrective action that is found to the user.

4. APPLICABILITY TO INDUSTRIAL SEMANTIC WEB ENVIRONMENTS

The service portability framework is described in the previous sections in a rather broad way lacking any details about possible applications of the presented vision. The main intent for such broad description is to show that the framework does not in principle depend on specific environments and/or services.

We believe that the application of the described approach to industrial Semantic Web environments is currently the most suitable and justified way to go about. First of all, the Web Service Architecture[9] does itself provide flexible support for two-phase service provisioning. It can be easily seen that our view on service provisioning presented in section 2 resembles Web Service provisioning framework with minor amendments. These differences should not be applied to web services themselves, but can be realized within the middleware infrastructure.

Another technical motivation for this approach is the use of Semantic Web ontologies for modeling context. There are several major approaches to context modeling[14], and they have their own strengths and weaknesses. Among these approaches Ontological context modeling is currently the most reasonable method due to the following properties of ontologies:

- high flexibility and manageability
- possibility for distributed composition
- capturing incomplete and ambiguous information
- high level of formality
- applicability to existing environments

Furthermore, some of the context sources in a real enterprise environment may already provide contextual information in an ontological form (e.g. user and equipment profiles, service descriptions), which makes it even more logical to use ontological approach for modeling the rest of the context.

Finally, Semantic Web ontologies (in particular, those in the OWL format) provide means not only for context modeling but also for contextual reasoning, which significantly alleviates the implementation efforts for building context reasoning engines. Ontological approach to context modeling has already justified itself within several similar research works.[5,15]

From a practical viewpoint, the described infrastructure for service portability is easier and more reasonable to implement and deploy within a single enterprise network rather than on a wide scale for public communication systems.

A typical service provision scenario for an industrial Semantic Web environment is presented in Figure 4. Service Provider and Service Requestor are generic entities that represent the owner of a Web Service and its consumer respectively. These entities can express relationships of business-to-business type as well as business-to-customer type. The process of service discovery is out of the scope at the current moment.

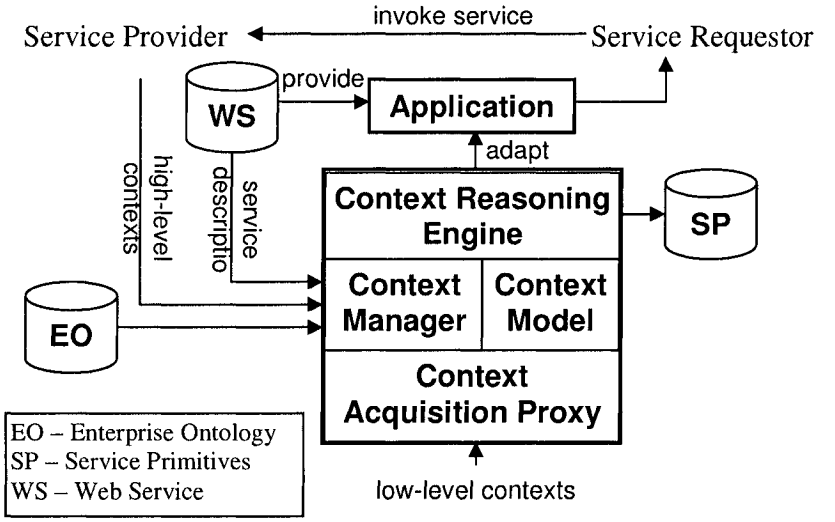

Figure 4. Service provisioning scenario in industrial Semantic Web environment

This scenario differs from the generic case illustrated earlier in the paper is in that high-level contexts are already represented in the ontological form within the enterprise system. The Web Service corresponds to the notion of Generic Service, and the service description, advertised by it, is a concrete instance of service context or meta-context. Low-level contexts, such as time, spatial coordinates, connection characteristics, etc. are received as usual from low-level context providers and transcribed to the ontological view by Context Management Module with respect to the available ontology structures in the system Enterprise Ontology. Enterprise Ontology describes the entire system in a comprehensive manner, annotating properties of its

elements and relationships between different entities within the environment.

Let us consider the following example. As soon as Steve arrives to the university campus he realizes that he does not know the place where the lecture is being held. He browses to the university's website and invokes the location-based campus guidance service. He submits a query for a lecture location to the service and gets an answer which indicates that the lecture will be held in the campus building owned by a special unit of an industrial company collaborating with the university. This unit has its own enterprise environment covering the building. As soon as Steve enters the building he is classified by the internal security system as a participant of the visiting lecture and authorized to use certain enterprise services available within the system. At this point the guidance application is adapted to operate on a possibly new technological base present in the enterprise network (new wireless access standards such as WLAN and Bluetooth, new protocol stack, QoS frameworks, etc.) and to use internal location-based Web service providing visitor guidance inside this particular building. To do this the middleware acquires the service description of the corresponding web service, analyzes the context model of the environment, and the local user profile, performs reasoning and finally extracts necessary service primitives from the environment to properly re-configure the application. Following the instructions on the screen of his handheld Steve takes an elevator to the third floor. The guidelines provided to Steve are the result of a context reasoning procedure performed by the new context-aware application and based on Steve's location inside the building. Steve's location is its specific "context of interest", which is tracked by the middleware on the application's demand. The path is constructed and locations are identified with respect to the enterprise ontology, which exists in the system and defines such relationships as, for example, "containment" to enable reasoning about location.

Bringing context on top of semantics in Web Services is an attractive feature of the framework that would allow Web Services to be even more flexible and intelligent. Notice also that the framework has potential to deal with the problem of Web Service composition by using context-awareness for customization of composite services.

5. CONCLUSIONS AND FUTURE WORK

In this article we described our vision on how to bring context-awareness into modern computing and communication environments. The most important thing about the framework presented is that it not only makes context-aware services available to current mobile users, but also proposes a featured context-aware approach to build up interoperability between today's communication systems.

The Service Portability framework is based on the idea of decoupling service applications from actual services. Such approach simplifies service design, since services no longer require to be endowed with application-specific details. This, in its turn, increases service reusability, allowing any service to be consumed through any type of environment without being modified or reissued. The portability of services is achieved by introducing adaptable service applications that can be reconfigured. These applications, instead of services, are adapted whenever environment undergoes any significant change. Such principle allows hiding any unnecessary details from service creators and managing run-time adaptation locally with the particular service application. Sophisticated efforts on remote service adaptation are not needed and service scalability is preserved. To manage the framework we specify a context-aware infrastructure which captures dynamic environmental changes in an efficient manner and provides reconfiguration and adaptation mechanisms for service applications. It utilizes horizontal services provided by environments to build service applications in a more pertinent way and as a result makes service concept more flexible and open.

Although the general principle of how the described infrastructure operates is expounded in the paper and clear motivation for it is given, there is a number of issues to be addressed yet. These include a programming model for reconfigurable applications with a modular structure, distributed composition of the architecture, details of context modeling approach, maintenance of the distributed context model, context dissemination between the parts of the system, etc. Before we can seriously speak about the architecture's feasibility and applicability to modern environments, these questions should be answered, and it is these questions that guide us in our further research work.

REFERENCES

1. M. Weiser, The Computer for the 21st Century. *Scientific American*, 1991
2. G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski, Challenges: An Application Model for Pervasive Computing, *Proceedings of 6th ACM Mobicom*, 2000.
3. M. Satyanarayanan, Pervasive Computing: Vision and Challenges, *IEEE Personal Communications*, pp.10-17, August 2001.
4. H. Chen, T. Finin, A. Joshi, L. Kagal, F. Perich and D. Chakraborty, Intelligent Agents Meet the Semantic Web in Smart Spaces, *IEEE Internet Computing*, October 2004, pp. 2-12.
5. T. Gu, H. K. Pung, and D. Q. Zhang, Toward an OSGi-Based Infrastracture for Context-Aware Applications, *IEEE Pervasive Computing*, October-December 2004, pp. 66-74.
6 T.Kindberg and J.Barton, A Web-based Nomadic Computing System, *Computer Networks*, Vol. 35, No. 4, pp. 443-456.
7 A.K. Dey, D. Salber, and G.D. Abowd, A Conceptual Framework and Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, *Human-Computer Interaction*, Vol. 16, No. 2-4, 2001, pp. 97-166.
8. G. Chen and D. Kotz, A Survey of Context-Aware Mobile Computing Research, Technical Report TR2000-381, November 2000.
9. Web Service Architecture, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, (last retrieved on 13.05.2005)
10. D. Zhovtobryukh and N. Kohvakko, Service Reference Model for Modern Communications, *In Proceedings of the Fourth International Network Conference*, edited by S.M.Furnell, and P.S.Dowland, 2004, pp. 221-228.
11. B. Schilit, N. Adams, and R. Want, Context-Aware Computing Applications, *In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pp. 85-90, 1994.
12. J. McCarthy, Notes on Formalizing Context, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp.555-560, 1994.
13. A.K. Dey, Understanding and Using Context, *Personal and Ubiquitous Computing*, Vol. 5, Springer-Verlag, 2001, pp. 4-7.
14. T. Strang and C. Linnhoff-Popien, A Context Modeling Survey, UbiComp 2004 Workshop, 2004
15. H. Chen, T. Finin, and A. Joshi, An Ontology for Context-Aware Pervasive Computing Environments, *ACM Knowledge Engineering Review*, pp. 197-207, Vol. 18, Issue 3, September 2003.
16. D. Zhovtobryukh, Integration Issues in Communication Environments, *In Proceedings of the 1st International Workshop on Ubiquitous Computing IWUC 2004*, 2004, pp. 28-37.
17. A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, The Anatomy of a Context-Aware Application, *Wireless Networks*, March/May 2002, V.8 No. 2/3, pp.187-197.