Context Provisioning for Future Service Environments

Dana Pavel, Dirk Trossen
Nokia Research
Helsinki, Finland

Abstract—Adapting services and applications to the context of their users bears the promise for increased user experience by taking into account location, time, activity and other situation-dependent information. Hence, context awareness has been recognized as crucial for future service deployment, in particular for future mobile services. However, many solutions in this space fall short when it comes to the actual deployment, either due to complexity reasons or the lack of widespread acceptance of the used technologies. The solution developed in our work aims for a platform that is highly extensible, achieves interoperability across devices and networks but is also grounded in available standards, making its widespread deployment possible almost immediately. The solution is rooted in two major design principles as well as in the usage of the event framework for the Session Initiation Protocol (SIP) to ensure this aim. In order to show the viability of our work, we developed and tested a proof-of-concept prototype within a mobile scenario.

Index Terms—context provisioning, SIP events, semantics

I. INTRODUCTION

Adaptability of services and applications is recognized as one of the most important characteristics for future applications, services and user interfaces. Context awareness, where the context is defined as general as in [13], is one of the main enablers for this adaptive character. Due to the increased importance of building adaptive services, there is need for an infrastructure that can support this type of services. The main purpose of any service infrastructure is to provide a clear separation between logically different layers, like data provisioning and usage. Further, the reuse of common functionalities would enable a certain economy of scale rather than implementing every feature from scratch. Through this commonly agreed functionality, interoperability is enabled between services based on the infrastructure.

When considering infrastructure support for adaptive services, the support for context provisioning is crucial since context serves as the input information into the adaptive decision algorithms. The challenge in this provisioning lies in the highly distributed character of many of the context sources, i.e., information could be acquired from sources found within local proximity as well as on remote locations. For instance, when determining a certain activity status for a user, calendar information would most probably be retrieved from a corporate server but it could be combined with context information collected from user's device and sensed information from her environment.

Also, context could include a vast range of information, depending on the intended scope, including network characteristics, billing, privacy and access control policies, and user-specific context. Given the large spectrum of information that could be involved as well as certain ownership and privacy considerations, it is not seen as realistic that context information can all be gathered and processed in a central location, like it happens nowadays in multiple context aware prototypes. Instead, the infrastructure has to support that context can be obtained in a distributed manner through polling context sources for information as well as through subscribing to context sources in order to obtain context changes and future availability information about certain context sources.

In addition, the actual application logic that uses the provisioned context information for performing a particular context-aware operation could be distributed as well, in particular when we consider aggregation of context information. For example, instead of pulling all the information necessary for determining if a meeting is going on in a certain location, this could be done by a Meeting service provider that, based on a certain model, will fuse information from various sources and then deliver its information to its consumers. This distribution of logic is even more important when considering a mobile device as the application host. Even though these devices will become more powerful in the future, there will remain resource constraints, such as memory, processing power and battery lifetime that will prevent the proper performance of certain heavyweight algorithms.

The possibly widespread distribution of context information is likely to lead to use cases that will operate in several network domains, are spread over different devices and involve several pieces of service logic. As a consequence, interoperability at various levels (communication and semantic) is a highly important issue that can determine if a certain solution is deployable or not. While the actual deployment depends on multiple factors (more or less technology related), it is desirable that any infrastructure solution is as close as possible to existing standards or common practice and that anticipates certain availability of related technologies.

In our work, we created a context provisioning architecture, together with a middleware specification and an initial implementation, which allows for providing context information between consumers and providers in the

{dana.pavel,dirk.trossen}@nokia.com

Internet. In order to ensure an interoperable and therefore extensible architecture, we tackled interoperability on protocol and semantics level. In order to enable protocol level interoperability, we decided to re-use main parts of the SIP and the SIP Events framework as our means for providing the actual information between the distributed parties. In order to ensure application-level semantic interoperability, we used concepts around OWL, OWL-S, together with SWRL, allowing for flexibility within application-specific domains while ensuring a common standard for semantic description of the information. With this, we enabled a high degree of automation regarding discovery, invocation and execution monitoring of our platform.

In this paper, we will describe some of the main design criteria for our solution, outline the architecture and middleware main characteristics, and ground the whole solution to specific examples and implementation description. For this, the remainder of the paper is organized as follows. We review some of the extensive related work in the area in Section II. Our proposed solution is presented in Section III, outlining the main design principles and the high level architecture. Section IV then outlines the initial implementation of our work, describing a particular use case, the middleware of our platform and the test bed of our solution. Section V concludes the paper and includes a description of future work.

II. RELATED WORK

Platforms for context-aware applications have been a focus of the research community for many years, indicated by the variety of different approaches presented for such platforms. For the sake of space however, we limited ourselves to presenting here only platforms we found as highly related to concepts or involving essential concepts for our work. For instance, work like [11] showed the feasibility of such adaptive applications even in small devices like mobile phones. The need arose rather quickly to enable a generic platform support for context awareness. For instance, one of the early works by Schilit et al. [10] outlined a platform for developing context-aware applications in a lab environment. Dey's Context Toolkit [13] used the *context widget* concept, enabling a plug-and-play style of environment for developing context-aware applications. Kanter's more recent work in [14] intends to place the environment in soon-to-come mobile environments such as UMTS (Universal Mobile Telecommunications System). The recent work by Khedr et al. [15] applies agent-based approaches for building a context-aware platform that even spans its reach for context down to the network level (e.g., incorporating Quality of Service parameters as context information). The Context Broker Architecture [16] uses OWL-S for its ontologies but implements a centralized broker model in its architecture. Although ontology concepts are considered in almost all these approaches, it is their inherent integration into the delivery platforms that restricts applications unnecessarily. Further, connectivity interoperability is largely not addressed by these platforms either.

Outside the world of research labs, the IETF (Internet Engineering Task Force) standardization work on SIP [2] and SIP events [3] greatly influenced our approach. The Session Initiation Protocol (SIP) enables a separation of user identifier (URI) from endpoint identifier (IP address), enabling therefore application layer mobility of devices. It implements a session model, in which the signaling of the session setup is separated from the actual data transfer. For the session setup, a federation of SIP proxies is used. SIP has been chosen by virtually all mobile standardization bodies for future Internet multimedia services. The most important extension to the basic SIP framework is the work around *SIP events*, which uses SIP for creating an event delivery framework for the Internet.

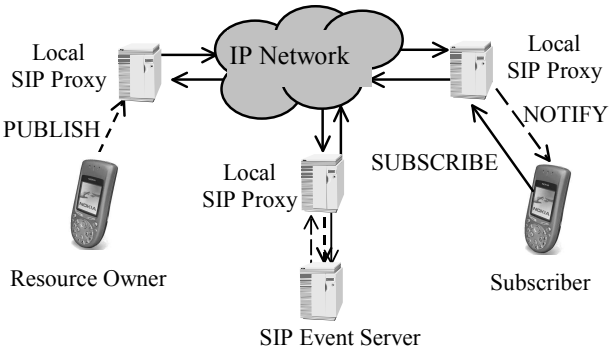

Fig. 1. SIP Events framework

Figure 1 summarizes the overall architecture of the SIP events framework, depicting a mobile subscriber, a resource owner, a SIP event hosted on a fixed SIP event server, and the messages involved. As defined in [3], SIP events generally represent state information for a particular resource. The specific semantic of SIP events is not specified in [3]. Instead, SIP events with specific semantics are supposed to be defined in separate standardization documents, following the template description given in [3], specifying for instance behavior of the network entities, the format of the state information and rate limitations for notifications. An example for such specific event description is the *presence event*, describing the current presence state of a user in the Internet. Other proprietary examples are described, e.g., in [4] for location events. In addition, the work in [9] provides a framework to enable access authorization for state information that is conveyed using SIP events. It therefore provides a very powerful tool to implement delivery of any event over the Internet. The SIP standard and its SIP event extensions form the major foundation for the next generation mobile systems (3G). It can therefore be expected that SIP technology will be widely available in future mobile phones. At the time of writing this paper, several models with full SIP stack have been announced in the marketplace.

Another related work that heavily influenced the design of our approach is the work around OWL-S (Semantic Markup for Web Services). As described in [7], OWL-S is a submission to W3C that specifies an OWL-based semantic description of a service. Using OWL-S enables the

automation of service discovery, service invocation, service execution monitoring, and service (de-)composition. OWL-S provides means for service ontologies definition, allowing for describing services on a semantic level. The OWL-S service ontology describes a service using three parts [7]:

- *Service profile*: Includes information about the provider, a functional description of the service (i.e., inputs required, outputs generated, preconditions required, expected effects resulted from performing the service), and a set of service properties, e.g., service category. The profile is usually used to advertise the service in discovery registries.
- *Service model*: Contains a detailed description of how the service is performed. The information can be used, e.g., by a discovery agent, for determining if the functionalities offered by the service meet its needs, for composing services as well as for monitoring service execution.
- *Service grounding*: Specifies how a client can access a service (e.g., information about protocol, message format, port, etc.) in order to enable automatic service invocation. An example of grounding OWL-S descriptions to WSDL is presented in [7].

It is worth mentioning that ontology-based design and context awareness are gaining more and more ground. Therefore the list of related work can grow arbitrarily long. However, we did not encounter work that combines SIP events with OWL-S. It is exactly this combination that makes our proposed solution unique.

III. DESCRIPTION OF PROPOSED SOLUTION

This section attempts to summarize the main design principles and architectural characteristics of our work.

A. Design Principles

There are two underlying principles for the design of our distributed architecture. The first one describes the separation of commonly agreed (even standardized) and application-specific functionality, called *ontology-based design principle*. The second one describes the principle of communication between the different components, namely the *event-based communication design principle*.
We first outline these design principles of our architecture in the following before presenting the resulting architecture on a component level.

1) Ontology-based Design Principle

It is desirable for a deployable distributed context provisioning system to provide dynamic information discovery, knowledge representation and sharing, context reasoning, and extendibility. This requires a high level of interoperability between a large variety of different devices which can feed or consume context in our infrastructure.
In order to ensure this interoperability, the separation of commonly agreed (even standardized) communication infrastructure and application-specific functionality is crucial. While one wants to keep the need for common agreement (and therefore for standardization in time-consuming standards bodies) as little as possible, one would like to avoid ambiguity of semantics and implementation details that would possibly lead to non-interoperable applications. This seemingly contradictory requirement is fulfilled in our platform with an *ontology-based design*.
In this, an *ontology* is viewed as a domain-specific representation of concepts and relationships between concepts that can be shared between people and application systems [7].
With this in mind, we designed our system in a way that it uses a commonly agreed event delivery mechanism (see our second design principle below), while it uses ontologies for describing the actual context, including hierarchies of contexts, but also the relations between internal architectural components. While some ontologies are specifically created for our system, we have tried and further intend to reuse as many existing ontologies as possible from other areas. Ontologies are integrated in the architecture by inherently enabling their usage in the event delivery, e.g., through using content indirection methods (in order to point to ontologies wherever necessary) and leaving the interpretation of the ontology information to application-specific logic.
The support of ontologies for semantic definition of context information allows for extension towards vertical application areas (e.g., health care) with their own particular semantics for certain ambiguous context. The semantic modeling of context provisioning hierarchies also enables a distributed acquisition and aggregation of context information from different sources based on a particular semantic meaning. Also, dynamic discovery is enabled through match-making mechanisms, using the ontology information of the inquired context information.
In our solution, we chose OWL-S as the underlying framework for the ontology-based design of our platform. We will explain throughout the remainder of the paper how the OWL-S constructs are used.

2) Event-based Communication Design Principle

Most of the interfaces within our architecture are based on a common event delivery paradigm with the following characteristics:

- The event delivery is based on individual *subscriptions* to *events*. These events usually represent state information associated with certain resources.
- It is possible to provide state information of event state itself, so-called *event templates*. Examples are information about subscription's state or information about subscribers within certain subscriptions, see also [3].
- In order to allow for proper identification of the particular subscription, allowing for several subscriptions at any given time, a *dialog identifier* is provided with each subscription confirmation.
- The semantic of each event is application-specific and based on a certain semantic description as defined by a given ontology. This agnostic behavior defines the

borderline between protocol and semantic level interoperability.
- The event delivery allows for arbitrary payload for each event subscription and notification.
- For each subscription, there exist a subscriber (client) and an event server. Both are addressed according to the particularly used transport mechanism, e.g., using IP addresses, URLs or SIP URIs [2].

With this in mind, the following five methods are defined for implementing the event delivery within our architecture:
- ***SUBSCRIBE*** allows for initiating a subscription. It contains: *event name, event-specific content* (such as query or filter descriptions), *lifetime* of the subscription (if 0 then it's a poll), *FROM* and *TO* parameters, identifying the subscriber and event server.
- ***PUBLISH*** allows for publishing information that is relevant for a particular subscription, e.g., location information that is used for a location event. Note that the sender of the PUBLISH method is usually not the subscriber to the event, while the receiver is the server that hosts the event. The method contains: *event name*, *event-specific content*, i.e., the information relating to the event, *FROM* and *TO* parameters, identifying the publisher and event server.
- ***CONFIRM*** confirms each publication to the publisher and each subscription to the subscriber. The confirmation can either be positive (subscription granted) or negative. In the latter case, a reason code is provided. In the positive case for a subscription, the *dialog identifier* is provided to the subscriber to identify the subscription dialog. For a publication, there is no dialog established.
- ***NOTIFY*** allows for sending notifications for a particular subscription. These notifications are triggered depending on the particular event semantic. Examples are state changes in the event or other reasons. There is always an initial NOTIFY, conveying the initial state of the subscription right after the subscription has been granted (note that this initial NOTIFY is not sent if the subscription has not been granted, indicated through a negative CONFIRM). If the lifetime of the subscription is non-zero, future NOTIFY messages are possible, depending on the semantics of the event subscription. The NOTIFY method includes *FROM*, *TO*, and *event name* fields from the original subscription. It further includes the *dialog identifier* to properly relate notification and subscription.
- ***BYE*** explicitly terminates a subscription. The dialog identifier has to be included. A BYE method can be sent from either the subscriber or the event server. In the latter case, a reason code is provided.

The above outlined event delivery is very similar to the SIP event framework [3]. However, the event delivery is kept generic enough to be implemented over a variety of different protocols, including but not limited to SIP events, HTTP, or directly over TCP/IP.

The event delivery of our platform is tied into the ontology-based design through specifying a particular service grounding in OWL-S, our choice of implementing the ontology-based design. For that, the provided operations of our event delivery, i.e., *request-response* and *subscription-notification,* are defined as the service grounding for all context provider services within the platform. Since the event delivery is assumed to be agnostic with regards to the payload, the OWL-S description is directly used within the delivery. Hence, format conversion is not necessary to be specified in the service grounding. However, when implementing the event delivery over different protocols, such as Web Services, it is the task of the particular mapping implementation of our event delivery to perform necessary format conversions. For instance, the mapping onto Web Services would require a mapping onto WSDL, e.g., as described in [7].

B. Component-Level Architecture

Based on design principles laid out in the previous section, Figure 2 shows the high-level view of our resulting component-level system architecture.

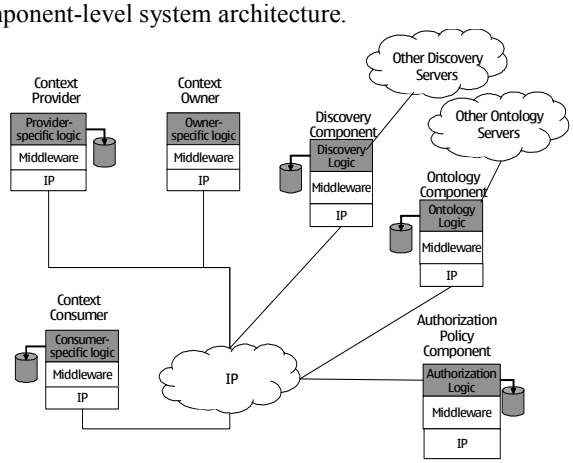

Fig. 2. Component-level system architecture

As can be seen, each distributed element of our architecture is based on a *middleware*, which provides a common platform for distributed context-aware applications. This middleware implements common functionality for discovery and provisioning of context information to the different entities with ontology and access authorization support.

Within each element in our architecture, *component-specific functionality* should be implemented in addition to the common middleware, shown as gray boxes in Figure 2. Note that this element-specific functionality is not within the scope of the platform itself and can implement proprietary and differentiating functionality, such as reasoning. This logic, however, will use the common middleware for implementing its functionality, wherever possible.

In the following, we briefly describe the functionality of these components.

1) Context Consumers/Providers/Owners

The main components of our architecture are the context owners, providers and consumers. While this is not a new concept, it is not quite usual for using a service oriented view to model a context-aware system. Also, context owners and providers are considered separate since they do not necessarily have to be the same in most deployments. A context provider can be simply an intermediary for context provisioning (e.g., a Location provider) or an aggregator, combining data from various sources (e.g., a Meeting provider). Note that the topmost context consumer in the architecture would be the application logic that makes use of the obtained context information for its particular use case.

For each piece of context information provided or received within our platform, it is likely that there is need for some context-specific *recognition* and *reasoning* to process the actual information. Such specific logic could serve as a differentiating element in a (context) service offering of particular providers, e.g., through the quality of the used reasoning method. Of course, one could also imagine that certain libraries would exist for certain basic reasoning functionalities, which is the case in certain AI areas, like constraint programming (e.g., [19]). However, our middleware does not provide these functionalities as commonly used ones rather than leaves it entirely to the provider and consumer to implement the *context-specific logic* of these components.

Part of the context-specific logic is also the realization of *aggregation* functionality. Aggregation is the process of collecting context information from various context providers, processing the information and offering some derived information further to other consumers. These hierarchies of context providers are built through an inter-play of discovery, aggregation, acquisition, and ontology functionalities provided by the middleware.

2) Authorization Policy Component

This component authorizes the transfer of data from context providers to context consumers. It allows owners of the context to have control over their information. The middleware for this component provides generic functionality to manage and retrieve access policies for certain context information. These access policies are used in the actual context provisioning to ensure proper access rights for each subscription before granting the subscription eventually. However, the access policy could specify that access is supposed to be granted at the time of subscription. For this, the component middleware provides additional functionality, e.g., through some HTTP-based web forms.

It is not within the scope of the component middleware to define a particular format of the access policy or even the scope of the policy (e.g., level of access information). This is seen as part of the component-specific logic. Also, the storage and particular means of maintenance (e.g., web-based forms) are seen as part of the local implementation.

3) Discovery Component

This component provides functionality to discover context sources within the system. It is important to note that while we describe it as a single component, there could actually be fully distributed federations of discovery servers working together in providing the required functionality. The mechanism in [17], for instance, describes a solution that integrates well with our architecture, using a SIP-based solution to this problem.

At the middleware level, this component should insure a uniform system-wide discovery of context sources. This is achieved through providing a subscription-based discovery mechanism, which allows for discovering but also subscribing to the future availability of particular context information. This discovery or availability request itself uses pointers to ontologies to allow for defining own context ontologies.

The discovery-specific logic should be the place to implement support for legacy technologies, e.g., to connect to a SLP (Service Location Protocol [8])-based service discovery system. This logic would also be concerned, for instance, with resolving and reformatting the discovery queries from the system query into the appropriate legacy query. Further, such logic would implement a middleware-compliant availability subscription in cases in which the particular legacy technology does not support such subscription (e.g., as in SLP).

4) Ontology Component

While most of the ontologies used in our solution could be directly addressed by URLs from their respective locations, certain ontologies would be local to the context-aware system. For example, as will also be discussed later, in the current implementation, own ontologies are created for representing the relationships between our middleware components. It can also be envisioned that certain other ontologies would be kept local within the particular deployment, e.g., within enterprises.

The middleware part of this component should insure proper access to existing ontologies, plus certain other operations with ontologies that a certain consumer might require. Please note that the ontology component, similar to the discovery component, does not have to be mapped onto a single entity but can instead use entire federations of ontology servers.

As part of the ontology logic, functionalities like ontology maintenance (storage, verification, merging, mapping, etc.), proper format adjustment of the ontology, and reasoning logic for selecting an appropriate ontology can be envisioned.

IV. PROOF OF CONCEPT

We implemented our proposed solution architecture in a proof-of-concept prototype, as described in this section. We start off with the initial scenario, before outlining the platform middleware implementation and the test bed for our prototype. The scenario is placed in an office environment, where some of the assumptions made in terms

of context sources held true (e.g., calendar-based activity determination). However, other context sources could be used for determining certain context states, such as sensorial data for meeting room occupancy.

A. Implemented scenario

Norman is in a project meeting, and his phone is switched to Meeting mode. His colleague, John, is running late and is trying to call and let the others know about it. Since he is supposed to be part of the meeting group, his call is allowed to go through to Norman. Later during the meeting, Julie, Norman's wife, is calling. Her call is not marked urgent therefore her call is transferred to the voicemail system. Norman is notified via short message that a voicemail is waiting for him. Later during the day, once his schedule clears up a bit, he receives a notification that Joe and David are on the basketball court, playing a game. He decides to go for a short game before leaving for home. In the meanwhile, his wife updates his calendar with a shopping list so that Norman will have to stop on his way home and pick up a cake and drinks for the dinner tonight. When he gets ready to leave, his device obtains current traffic information and by combining it with the shopping list and knowledge about where Norman usually shops, it can determine which would be the best route to take. Norman receives a notification on the required shopping stop together with the suggested route. While in the car, Norman is also checking his mother's wellness status. He can see that she has not been too active for the past two days. Fearing something might not be ok with her, he decides to call her and find out how she is doing. He finds out that she had a minor cold and that's why she did not go out, but she is doing much better now.

B. Our Platform Middleware

As the key element in our proof-of-concept, we implemented the platform middleware, shown in our architecture of Figure 2 as the main common entity. It provides commonly used functionality across all different components in our architecture. In the following, we summarize the important characteristics of this middleware.

Three major parts exist in the middleware, as shown in Figure 3, responsible for discovery of context providers, provisioning of context information and the general provisioning of information within the event delivery paradigm. Each of the parts has been fully specified with dedicated interfaces. Further, the information within our system is modeled, based on our ontology-based design principle.

1) Context Registration, Discovery, and Availability

This part provides functionality related to discovery of context information throughout our system architecture. Its main functionalities are to allow context consumers to *discover* particular context information and to *register* context information (by registering a particular context provider description).

For the *discovery* of context information, the context consumer's middleware communicates with the discovery component of the system by sending an appropriate discovery query for the desired context information. Some examples of possible queries would be "return all location providers", or "return a temperature provider that provides the information with an accuracy of 80%". It is also possible that a context consumer uses a specific ontology in the query in order to better describe what it means. In that case, extra steps are required in order to retrieve/parse the relevant concept/ontology, these steps being provided by our middleware.

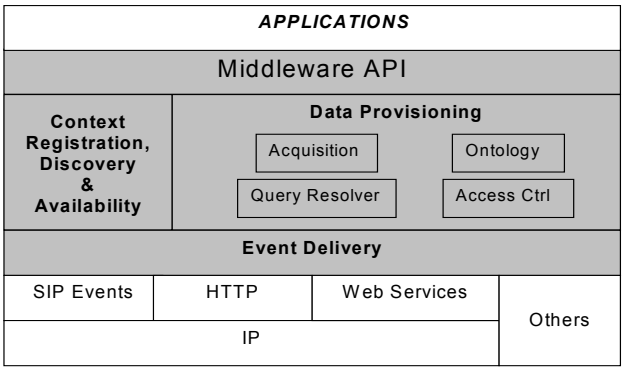

Fig. 3. Middleware

The discovery functionality includes also the possibility to subscribe to availability of future context information since such option is seen as important in context-aware environments due to the frequently changing availability of context sources. Hence, once a matching context provider will become available in the future, the context consumer will receive a notification with the particular context provider's service information.

The middleware of the discovery component determines (through its access control part) whether or not the requestor has proper access rights for the provided information of matching context providers. If required (e.g., there is no local policy available for the particular access), the local access control part of the middleware will consult the authorization policy component to retrieve the appropriate access control policy.

For the *registration* of context information, a context provider has to follow three main steps. First, it has to obtain the system-specific ontology (so that it uses the same concepts like the other system components) or register its own one (this registration happens with the ontology component of our architecture). Second, it has to define access control policies for the context information provided. It is assumed that the context provider has the proper rights to define the access policies. Another possibility is that the context owner would directly set the access policy, using the same interface as the context provider. It is the responsibility of the authorization policy component to perform authentication mechanisms to ensure proper access right settings. Third, the context provider has to register itself with the discovery component (specifying the provided context information as well as the required input

parameters), indicating its availability for potential context consumers. If there exists at the discovery component any subscription to the availability of a context provider that matches the newly registered one, an availability notification is sent to the appropriate context consumer(s) by the discovery component.
We achieve extendibility of our system with respect to new and complex context information through supporting hierarchies of context providers. This means that new context providers can be created through aggregating existing ones. The entire process needs support from various existing components of the middleware and is mainly based on a semantic description, based on ontologies, of the context information decomposition. For creating an aggregated context, the provider will become a consumer and will have to discover needed context providers to link to. If in this process some ontological differences between the context provider of aggregated information and lower-level context providers would appear, a certain ontology mapping intelligence would be required. This would be implemented in the ontology logic of the ontology component (Figure 2).
Once the proper hierarchical relations are created, the new context provider would have to register itself with the discovery component of our architecture, using the above outlined part of the middleware.

2) Data Provisioning

The actual context provisioning functionality of the middleware includes the following tasks:

- acquisition handling: enables polling or subscription-based delivery of context data and context metadata (i.e., data about how certain context is obtained by a provider) from provider to consumer
- query resolving: allows for more complex queries that include certain conditions that restrict the delivery of context information to the consumer
- handling of access rights: insures proper handling of privacy rights for a certain context information by consulting with the policy server
- support for ontologies

The process of acquiring context information consists of the context consumer requesting data from a context provider and the context provider fulfilling the particular request – be it sending context data back to the context consumer or denying the request. This process includes acquiring additional data from other context providers to fulfill the particular request. In other words, if the provided context information requires additional input from other sources, hierarchies of context providers are established, i.e., enabling the aggregation of context information. In this process, the original query is decomposed appropriately to formulate sub-queries accordingly. Furthermore, each context provider verifies the access rights for retrieving the particular context information before successfully granting the request.
The acquisition part of the middleware supports two kinds of requests, namely asking for changes in a particular context of an entity (entity being, e.g., persons, groups, places, etc.) and asking for a list of entities being in a particular context. In other words, following types of requests are possible: *"What is the context X of entities Y?"* and *"What entities are in a context X?"*.
For answering the second question, the context provider would need to perform certain data mining functionality over its existing context data and entities associated with this context data. This data mining functionality is not within the scope of the architecture and should be implemented in the logic of the context provider.
The acquisition functionality also supports the signaling of changes in semantics for a particular piece of context information. For this, the semantics are interpreted as a state of the context metadata, i.e., if the semantic changes, the state of the context metadata changes. We then use event templates (see Section III.A.2) to signal this information to the context consumer. With this, the platform allows for future extensions towards *semantic mediation*, i.e., mediating the change of semantics, for instance, through finding semantically close replacements.
In our system, we assume that each context provider would have an underlying ontology (i.e., semantic description) that describes the particular piece of context it provides and how it is obtained. For example, in the case of a 'meeting' context provider, its ontology can specify that 'meeting' is obtained by combining information about location, identity, proximity, microphone and calendar. We provide a method that allows a context consumer to obtain information about how a certain context is obtained (i.e., its ontology metadata), if the context provider is willing to share that information. This information could be valuable in deciding which provider to use when several providers are available for the same type of context information. For that, similar steps as in the acquisition of context information are implemented, resulting in the return of the semantic description rather than actual context information.

3) Event Delivery

The event delivery part of the middleware implements the event communication design we outlined in Section III.A.2 in order to provide a basis for all remote communication between components.
Incoming requests, such as SUBSCRIBE at the event server side or NOTIFY at the subscriber side, are dispatched to the appropriate local functionality, based on the event information given in the request. In this dispatching, the particular middleware part is also provided with the information included in the request, i.e., event name, event body, dialog identifier, and FROM/TO information.
Initially, we focused on implementing the event delivery using SIP and SIP Events. The discussion in [5] provides some of the reasoning behind using SIP events for this purpose, of which the availability in future mobile devices as well as the application layer mobility provided by SIP are the major ones.
However, in the process of realizing our platform, we decided to also consider mapping the architecture onto Web

Services in order to incorporate this service environment into our system.

4) Information Modeling

OWL-S was introduced in Section III.A.1 as our main choice to semantically describe context services in a distributed environment. Since we followed a service oriented model for our context provisioning middleware, we found that OWL-S would be a good candidate for describing context providers themselves. Therefore, in our platform, we use OWL to describe our system-specific ontologies, OWL-S for describing the context providers, OWL-QL as the query language in the discovery process and OWL combined with a rule language, like SWRL, for the provisioning part.

In both the discovery and the data provisioning, the query dialog has to be mapped onto our event delivery operations within the middleware. For the discovery, an initial query is mapped onto an initial subscription sent to the discovery server with the initial answer bundle delivered in the first NOTIFY of the subscription. Note that with the subscription model of our system, the discovery server might send other answer bundles in the future upon availability of particular context providers, if they match the original query. If the client desires to send a refined server continuation to the discovery server, this is mapped onto a re-subscription of the original subscription, i.e., a SUBSCRIBE is sent within the same dialog. Both sides can terminate the dialog with the BYE method (see Section III.A.2), i.e., subscriptions (which equal queries in the OWL-QL query dialog) can be terminated either from the server or from the client side.

For the provisioning part, we used the property of SIP events to accept any type of payload. As a consequence, the OWL queries for particular context information are transferred directly from consumers to providers through the middleware. The OWL query is constructed based on the OWL-S description the consumer obtained from the provider (regarding the required inputs and provided outputs). In this, note that the grounding of the OWL-S definitions is based on the design of our event delivery, as described in Section III.A.2, i.e., the operations and format are mapped accordingly. In case that a consumer wishes to convey to a provider more than the information defined in OWL (e.g., some conditions that have to be fulfilled in order for the provider to send the context information to the consumer), it is desirable to extend the information through some rule language, e.g., SWRL. In cases of complex rule languages though, only a subset of such rule language is likely to be required.

When defining the provider ontology, we divided context providers into *abstract* and *physical context providers*. The former refers to a context provider that determines its provided context through aggregating lower-level context or to certain context providers that provide information from the virtual world (e.g., application environment). The latter refers to providers of atomic information, e.g., based on physical sensors. This differentiation in the ontology is used to enable aggregated context and de-composition of context provisioning requests, i.e., the determination of lower-level context providers within an aggregating provider. Ultimately, it also enables semantic mediation within our platform, i.e., the dynamic replacement of pieces of context information within a hierarchy of provided context through "similar", i.e., semantically close, information.

Within each definition of an aggregated context provider, a property is defined called *Dependencies,* contains a list of context providers the service depends on. These dependencies are used to indicate the aggregated pieces of lower-level context information. For example, an Activity Context Server could depend on a Meeting server or a Presence server.

With this, the full range of ontology-based design advantages, as described in [7] is exploited. In other words, such ontology basis enables us to perform automated discovery based on this ontology by using the service profile for registration with the discovery system. It also allows for invoking the context provisioning service automatically based on the parameters that are given in the service profile. In the future, we also envision using our means for signaling semantic changes of context provider information (see Section IV.B.2) for semantic mediation in cases where certain context providers become unavailable during providing the particular context information.

C. Use Case Implementation

We implemented the proof-of-concept for our work on the basis of the use case presented above and, of course, on top of the middleware functionalities presented in the previous section. First, we outline the different functional components, i.e., the different providers and consumers that we needed to implement. We then walk through the use case again step-by-step, describing the information flow between these components. At last, we describe the actual test bed and the used tools and software.

The context-aware logic that implements the decision making is implemented in a context consumer (called *Digital Shadow* in the following). For the different sources of context information, we implemented the following context providers:

- *Calendar context provider*: delivers information on current entries in the calendar of a user. The information is pulled directly from a corporate Exchange server via HTTP.
- *Phone profile context provider*: delivers information about the currently selected phone profile of a user.
- *Login status context provider*: delivers information whether or not a user is logged into his laptop.
- *Activity context provider*: derives the current activity of the user by using the input from the three context provider above. One could imagine extending this provider with input of the user's location and other information.
- *Identity context provider*: delivers information of the identity of a particular user. This information is usually pulled on demand by the context consumer. The provider is performing essentially a mapping from one

identity information to another one using different directories, e.g., LDAP. The information returned by the provider includes name, organizational relationships, and other profile data.

- *Basketball court occupancy context provider*: delivers information on whether or not the local basketball court is occupied. Remote sensing with infrared curtains is used for this purpose. In addition, separate means would be required (e.g., RFID-based) for identifying the people present on the court.
- *Wellness context provider*: derives a wellness indication of close family members through remotely sensing health-related information from a particular user. We used galvanic skin response and pulse as simple indicators, obtained via the remote sensing system described in [18].

The providers described above and the context consumer, i.e., the Digital Shadow, allow us to implement the use case, as described in Section IV.A, as follows. Upon starting, the Digital Shadow subscribes to the context providers for activity, court occupancy and wellness. As described above, the Activity context provider in turn subscribes to the providers for calendar, phone profile and login status. Note that we did not use the discovery part in this scenario due to the rather static form of configuration. Usually however, appropriate discovery steps would need to be performed. Further, note that we will not elaborate on the rule creation and management that specifies the behavior of the context consumer since the focus of our prototype was mainly related to the context provisioning.

When John calls Norman to indicate that he's running late, the Digital Shadow is invoked by the call processing server (a SIP proxy in our case since we used Voice over IP) to make a call routing decision. The Digital Shadow pulls the identity and profile information, based on the caller's phone number, from the Identity context provider. The identity information includes the caller's group information. Together with the activity information, indicating Norman's meeting with his group and that John is supposed to be one of the required participants, the call routing decision is made to let the call through. This is signaled back to the call processing server.

When Norman's wife calls later, the local rule at the Digital Shadow indicates that the call shall be diverted to voicemail due to Norman's ongoing important meeting activity and the fact that the wife's call is not marked urgent.

After the activity changes from the meeting activity, the Digital Shadow (notified by this change through its subscription with the Activity context provider), generates a short message to Norman about the wife's phone call.

Later during the day, the Digital Shadow is notified of increased basketball court occupancy. Since Norman's activity indicates no urgency, a message appears on Norman's screen, suggesting a game break.

When Norman is preparing to leave work, the Login status context provider notifies the Digital Shadow when Norman is shutting down his computer. Due to the updated entry in Norman's calendar (i.e., the shopping list), requiring a stop at the supermarket, the Digital Shadow retrieves appropriate traffic information from a Web Service based provider in the Internet and combines it with the store information. Norman is then suggested a specific route that will manage to avoid the bad traffic conditions on the way home.

When in the car, Norman retrieves his mother's wellness context from the Digital Shadow. The information shows little activity within the last two days so that Norman decides to call his mother.

We realized the functional component implementation described above in a test bed, shown in Figure 4. The use case is running within a local Wireless LAN. All devices in the local WLAN are running our middleware implemented in C++ under Linux. We used the Nokia open source SIP stack for the SIP event delivery part. An additional SIP proxy provides the necessary SIP registration but also call processing functionality of the use case. The Digital Shadow, i.e., the context-aware logic, is implemented on laptop T20. Norman's iPaq device has a user interface to the Digital Shadow, showing indications of particular actions of the Digital Shadow.

The laptop T22 hosts all context providers, as outlined above, although as different SIP identities, i.e., every provider has a dedicated SIP URIs. With that, the scenario could be run on multiple machines. Calendar and LDAP information is pulled via the Intranet connection, while the Digital Shadow retrieves traffic information via the Internet through Web Services. The court occupancy and wellness context provider, implemented on laptop T22, obtain the remotely sensed input data via a phone-based remote sensing platform [18].

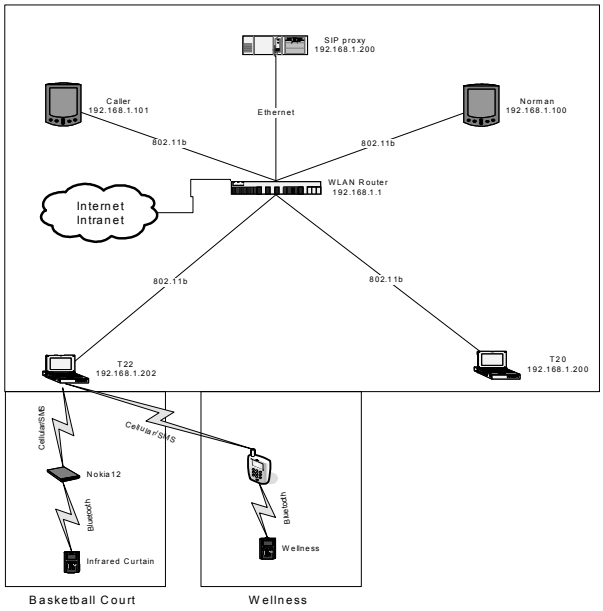

Fig. 4. Platform test bed

The different context providers are modeled in OWL-S based on our information model as described in Section IV.B.4 and made available to the system. As for the reasoning part, the Digital Shadow implements a simple

rule-based call decision that can easily be replaced with more complex decision making algorithms.

V. CONCLUSIONS AND FUTURE WORK

Context information can reside anywhere in the network and includes usually highly different semantic descriptions of the involved information. For this reason, the focus of our work was on ensuring interoperability on protocol and semantic level in order to achieve a high degree of extensibility. For that, we based our presented proposal on two major design principles, namely a consequent ontology-based design and an event-based delivery principle. This was coupled with the strict implementation requirement to re-use as many standards as possible in order to enable almost immediate widespread deployment.

This resulted in an architecture design that builds on a middleware for the commonly used functionalities to provide context information in a distributed environment. For this part, we based our prototype on the existing SIP events framework, which will become largely available in the newest generation of mobile phones, while it is available in many desktop systems for quite some time. With this, we enable almost immediate and wide-scale deployment of our system in network environments.

The application-specific part is enabled through the usage of ontologies throughout the system. Ontologies are used for describing internal components but in particular the context providers themselves. For this, the providers are interpreted as services in a distributed environment. This allows for using the concepts of OWL to enable automation on the level of discovery, invocation and execution monitoring. Even semantic mediation is enabled with a mechanism in our middleware that allows for signaling semantic changes to the context consumers. The creation and management of rules as well as the integration of machine learning algorithms that determine the context-aware behavior for particular use cases were not within the scope of our described solution although we recognize the necessity for such functionalities.

We implemented our proposed solution in a prototype, realized in a small scale test bed. For that, the required middleware was specified and implemented on a mix of mobile and fixed devices. We demonstrated a particular use case that allows for adapting call routing decisions and event suggestions to the user's context.

In our future work, we will focus on transferring the platform onto mobile phones. The used SIP events framework for our event delivery part should make this transfer feasible due to upcoming availability of this standard in newer mobile phones. Given that we built the middleware so that it would support both SIP events and Web Services events, we would like to extend our tests to incorporate both approaches. With this, we would like to analyze the applicability of our design for higher scale deployments. We will further study aspects around discovery and semantic mediation for more dynamic, in particular mobile, scenarios. In addition, we would like to extend our work in the ontology-based design and select proper tools for our particular mobile environment.

VI. REFERENCES

[1] M. Weiser, "The Computer for the 21st Century", Scientific American (1991)

[2] J. Rosenberg et al., "SIP: Session Initiation Protocol", The Internet Society, RFC 3261 (2002)

[3] A. Roach, "SIP-Specific Event Notification", The Internet Society, RFC 3265 (2002)

[4] D. Trossen, "Providing a Location Service based on the SIP Presence Extensions", Proceedings of IEEE Wireless (2002)

[5] D. Pavel, D. Trossen, "Context Provisioning and SIP Events", Workshop on Context Awareness at ACM SIGMOBILE conference on mobile systems and applications (MobiSys), (2004)

[6] R. Fikes, P. Hayes, I. Horrocks, "OWL-QL: A Language for Deductive Query Answering on the Semantic Web", Knowledge Systems Laboratory, Stanford University, available at http://ksl.stanford.edu/KSL_Abstracts/KSL-03-14.html, (2003)

[7] "OWL-S: Semantic Markup for Web Services" (November 2004) , http://www.w3.org/Submission/OWL-S/

[8] E. Guttman et al., "Service Location Protocol Version 2", The Internet Society, RFC 2165 (1999)

[9] J. Rosenberg, "The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)", Work In Progress, Internet Draft (2004)

[10] B. N. Schilit, N.I. Adams and R. Want, "Context-aware computing applications", Proceedings of the 1st International Workshop on Mobile Computing Systems and Applications, pp. 85-90, Santa Cruz, CA, IEEE (1994)

[11] A. Schmidt, T. Stuhr, H. Gellersen: Context-Phonebook, "Extending Mobile Applications with Context", Proceedings of 3rd International Workshop on Human Computer Interaction with Mobile Devices (2001)

[12] K. Arabshian, H. Schulzrinne, "GloServ: Global Service Discovery Architecture", Workshop on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous), (2004)

[13] A. Dey, "Understanding and Using Context", Personal and Ubiquitous Computing, Vol. 5, No. 1, (2001)

[14] T. Kanter, "Going Wireless: Enabling an Adaptive and Extensible Environment", Special issue on Personal Environment Mobility in Multi-Provider and Multi-Segment Networks of the ACM Journal on Mobile Networks and Applications (MONET), (2003)

[15] M. Khedr and A. Karmouch, "'ACAI: Agent-Based Context-aware Infrastructure for Spontaneous Applications", Journal of Network & Computer Applications, Vol. 28, Issue 1, pp. 19-44, (2005)

[16] H. Chen, T. Finin, A. Joshi, "Semantic Web in the Context Broker Architecture", PerCom 2004

[17] D. Trossen, D. Pavel, "Service Discovery and Availability Subscriptions Using the SIP Event Framework", Proceedings of IEEE International Conference on Communication (ICC), (2005)

[18] D. Trossen, D. Pavel, "Building a Ubiquitous Platform for Remote Sensing Using Smartphones", Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services, (2005)

[19] Marc Torrens, Rainer Weigel and Boi Faltings, "Java Constraint Library: Bringing Constraints Technology on the Internet Using the Java Language ", AAAI Workshop on Constraints and Agents, 1997