Discovery and Composition of Services for Context-Aware Systems

Cristian Hesselman[1], Andrew Tokmakoff[1], Pravin Pawar[2], and Sorin Iacob[1]

[1] Telematica Instituut, The Netherlands
[2] University of Twente, The Netherlands
{cristian.hesselman, andrew.tokmakoff, sorin.iacob}@telin.nl,
p.pawar@utwente.nl

Abstract. We consider the challenge of dynamically adapting services to context changes that occur in ubiquitous computing environments (e.g., changes in a user's activity) and propose the Context-Aware Service Enabling (CASE) platform for that purpose. The CASE platform combines context-aware service discovery with service composition, acting as an enabler for the development of adaptive context-aware applications. In this paper, we illustrate the need for context-aware service discovery and composition in pervasive 4G environments and present the architecture of the CASE platform. The CASE platform enables applications to easily adapt to changes in service availability, which may result from changes in client and/or service context. We also provide an overview of the platform's technical realization.

Keywords: Service Discovery, Context-awareness, Service Composition.

1 Introduction

The vision of 4G mobile networks is one of new high-speed radio access network technologies, an all-IP network, and a ubiquitous service platform [3]. Such a service platform has a pivotal role in the 4G vision [8] as it provides a common underlying set of functions that are enablers for the realisation of innovative new services. These services are expected to make use of advanced mobile terminals that have various radio technologies at their disposal and will be made available by providers that build upon the core services offered by the underlying service platform.

Two essential aspects of ubiquitous computing, as seen in the 4G vision, are those of service discovery and of context-awareness. The first is a service that allows applications and/or services to discover and bind to services that appear and disappear in highly dynamic mobile environments. The discovery of appropriate services can benefit from knowledge of both client and service context. Furthermore, it may be the case that no direct match for the requested client service can be obtained. In this case, service composition can be utilised to dynamically "construct" a composite service that matches the request of the client. This may also be subject to changing context since a composed service may need to be re-composed over time or may become "inappropriate" for a client as its context or that of the composed service changes.

P. Havinga et al. (Eds.): EUROSSC 2006, LNCS 4272, pp. 67 – 81, 2006.

In the following sections, we will further discuss some of these essential concepts and present them in relation to the Context-Aware Service Enabling (CASE) platform, which provides a suite of core functionalities for context-aware service discovery and composition that are needed as part of a broader '4G services platform'. We first introduce our main concepts (Section 2) and then present the architecture of the CASE platform (Section 3). Next, we provide an overview of the platform's technical realization (Section 4), which is currently under development. We conclude with a discussion of related work (Section 5) and a summary (Section 6).

2 Context-Driven Service Adaptation

One of the most critical issues in pervasive computing environments is that of ever-changing context. This applies equally well to mobile devices as to the (fixed) services these devices use: mobile devices are regularly subject to location, network, and power context changes, whilst services can for instance be subject to changes in the types of devices they have to serve. In pervasive computing environments, such changes should result in a service being dynamically adapted to the new context of a mobile device or of the service itself.

2.1 Motivating Example

In Fig. 1, we present an example of an adaptive service that streams the latest news (audio and video) to mobile users over the Internet. The service can dynamically add and/or remove the video stream from a multimedia news transmission in reaction to changes in the context of a user.

In Fig. 1, the service responds to changes in the activity of user Bob. Bob's initial activity is 'driving on highway 35' (Fig. 1a), which results in the service delivering the news transmission in audio-only mode (for safety reasons). Bob then stops at his destination and alights from his car. This has the effect of changing his activity to 'walking in downtown Amsterdam' (Fig. 1b). As a result, the service adjusts to also deliver the video part of the transmission to Bob.

This change in service configuration could also result from other context changes, such as a change in available networks. Similarly, it is also possible that the service's context may change (e.g., due to network outages or overloading) which would also result in an adapted service being delivered to Bob. In the remainder of this section, we provide a more detailed discussion of some important underlying concepts, including services, context sources, context agents, and service adaptation.

2.2 Services

We consider a *service* to be a unit of well-defined functional behaviour (in syntax and semantics) that is offered by a software entity for use by other software entities. The adaptive broadcasting service of Fig. 1 is an example of such a service.

A service can be a *composite service* in that it can consist of one or more *constituent services*. A constituent service provides part of the composite service or helps other constituent services to do so. In general, a constituent service can itself be

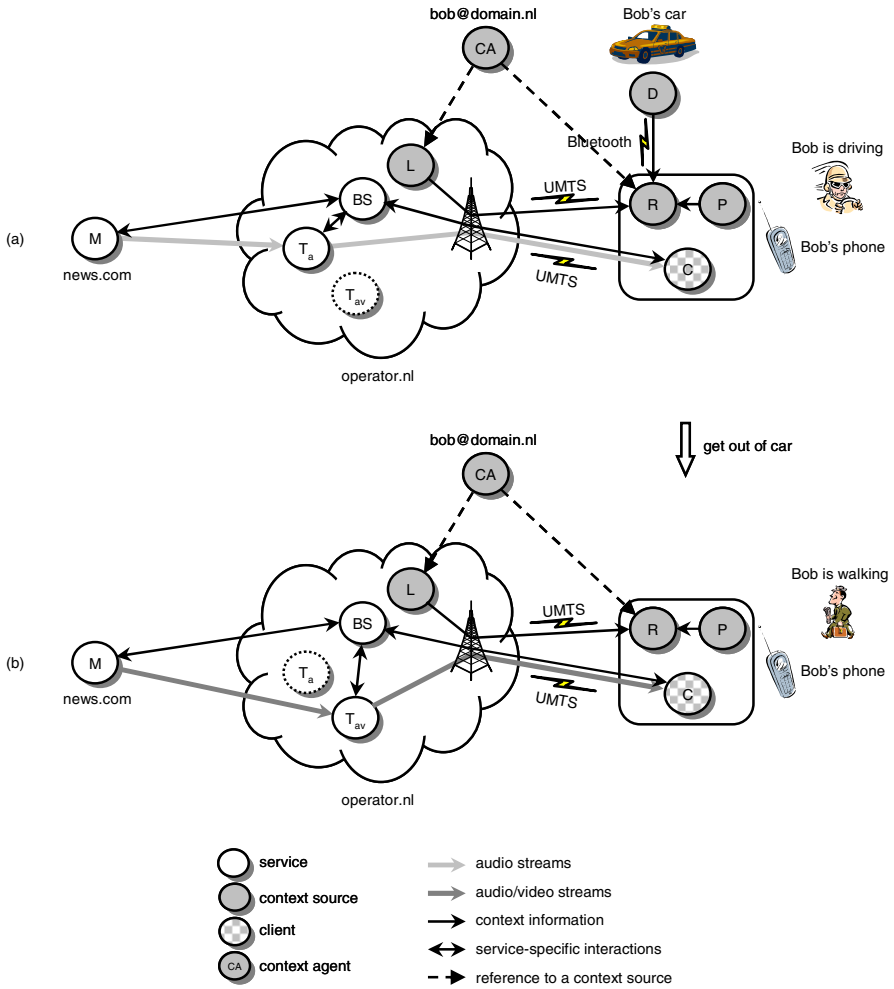

Fig. 1. An example of adaptive multimedia broadcasting

a composite service, which means that it can also be further decomposed into yet another set of constituent services. In the example of Fig. 1a, the broadcasting service (BS) is a composite service. Its constituent services are a multimedia streaming service (M) operated by a newscaster (news.com) and an audio transcoding service (T_a) operated by an UMTS operator (operator.nl). The streaming service transmits an audio stream, which the transcoding service receives and adjusts to match the capabilities of Bob's UMTS phone (e.g., by scaling the stream to a lower bit-rate).

Constituent services are arranged into a *service graph*, where the graph's edges indicate how the services relate to each other. Different constituent services of the same composite service could potentially operate in different execution domains. In Fig. 1a, the constituent services run in both the news.com domain (M) and also in the domain of the UMTS operator (T_a).

Clients are applications that interact with (potentially composite) services. These interactions are service-specific. The client shown in Fig. 1 (C) runs on Bob's mobile phone and is responsible for rendering the audio and video streams that it receives from the composite broadcasting service.

2.3 Context Sources and Agents

A *context source* is a service that provides access to context information, such as the location of a user or the activity a user is currently engaged in [1]. A context source provides an interface that enables clients to directly access context information via a request-response interaction or by subscribing to events that signal a change in context information (e.g., when a user moves from one room to another).

As with all services, a context source can be a composite context source that can aggregate context information and also determine higher-level context information from more elementary context information (cf. the interpreters of [16]). Fig. 1 depicts an example in which a composite context source (R) enables a client to determine Bob's current activity (e.g., 'Bob is driving on highway 35' or 'Bob is walking in downtown Amsterdam'). To be able to supply this sort of information, the composite context source consists of three different constituent context sources. They can provide:

- the location of a particular user in the operator's UMTS network (L),
- information regarding Bob's car (D) (e.g., who is driving it, its direction of travel and its velocity), and
- acceleration and orientation information about Bob's phone (P) (e.g., using a gyroscope and an accelerometer).

Using the context information provided by these constituent context sources, the composite context source can determine what activity Bob is currently engaged in.

Fig. 1 illustrates that composite and constituent context sources may operate in different execution domains. Context sources P and R reside on Bob's mobile phone, whereas L and D are located in the UMTS infrastructure and in Bob's car, respectively.

A *context agent* is a service that stores references to context sources. A context agent represents an *entity* whose context needs to become discoverable. These entities can be classified into people, places (e.g., a meeting room), and things (e.g., mobile devices or software components) [16]. The references that a context agent stores point to context sources that can currently provide context information about the entity represented by the context agent. For example, the context agent associated with the person 'Bob' could contain a reference to the composite context source that provides access to Bob's current activity (R) and a reference to the context source that provides lower-level information about Bob's current location (context source L).

A context agent acts as a single, persistent point of access for context information about a particular entity and should therefore be 'always on'. Each context agent has a unique identifier, for instance based on the type of URLs defined by the Session Initiation Protocol (SIP) [18]. In this case, the context agent of Bob (Fig. 1) would be identified by a SIP URL like sip:bob@domain.

After a client has resolved the identifier of a context agent to a network address (e.g., using SIP and DNS), it can access the context agent. A context agent provides a request-response interface, which enables clients to retrieve a subset of the references stored by the context agent. A request indicates the types of context sources the client is interested in (e.g., context sources that can provide information about the temperature at Bob's current location). The context agent's response consists of references to context sources that can provide this information. A context agent also provides a publish-subscribe interface, which enables clients to asynchronously receive updates in the context agent's list of context sources. After a client has obtained a set of references to context sources, it can use their interfaces to get the actual context information.

A context agent can be realized in various ways, for instance as a web server [17]. A context agent can furthermore be combined with a context source, in which case the context agent also implements a context source interface. This means that the context agent can also return actual context information instead of just references to context sources. In this case, a context agent is similar to the context aggregators discussed in [16].

Observe that a context agent does not need to be physically co-located with the entity it represents. For example, the context agent of a mobile device could be located somewhere in the network infrastructure rather than on the device itself.

2.4 Service Adaptation

In pervasive computing environments, a service may need to be adapted in response to a context change. These adaptations may need to occur *while the service is being used*. In the example of Fig. 1, the broadcasting service (BS) is adapted in response to a change in Bob's activity (from 'driving' to 'walking') while Bob is listening to/watching a news transmission. This dynamic adaptation is realized by a re-composition of the service's graph: the audio transcoding service (T_a) is removed and is replaced with a transcoder that can handle both audio and video streams (T_{av}).

Context sources provide the means to detect context changes, but may themselves need to be re-composed as a result of such a change. For example, when Bob gets out of his car (Fig. 1), the composite context source (R) changes since the car's context source (D) becomes unavailable. In the example, context source R can still function without D, but D may also need to be replaced with an equivalent context source.

3 Service Discovery and Composition

The main function of the CASE platform is to dynamically adapt services (including context sources) by changing their composition in response to context changes, possibly while these services are being used (cf. the example of Fig. 1). To accomplish this, the platform consists of a composition service and two types of discovery services: a context-aware discovery service and a basic discovery service. Fig. 2 illustrates this. The arrows in Fig. 2 represent interactions.

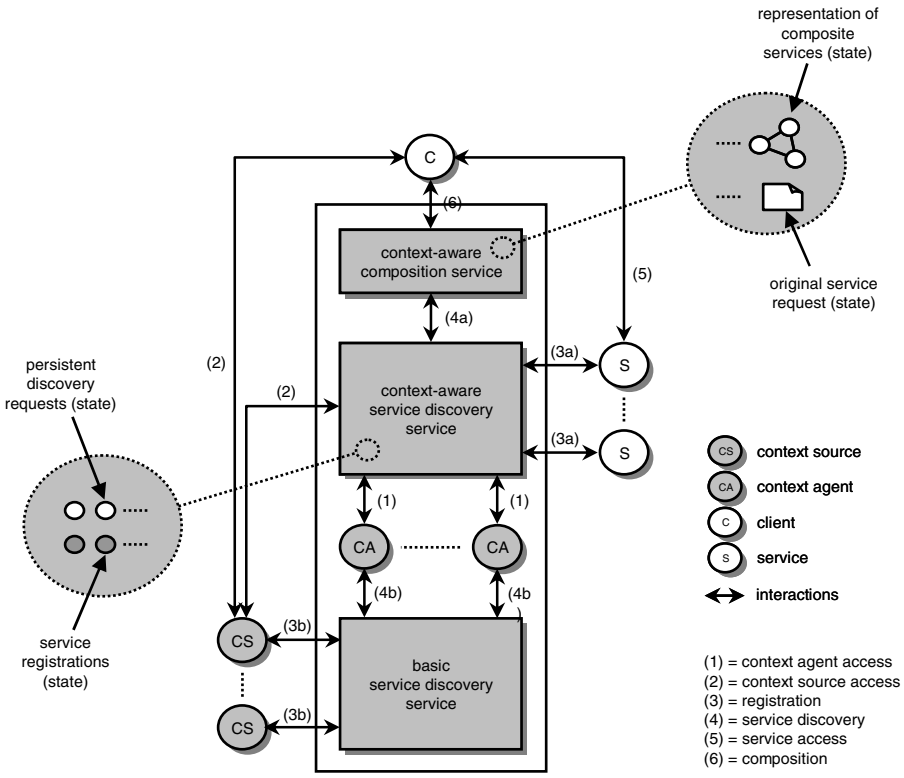

Fig. 2. Architecture of the CASE platform

The CASE composition service dynamically (re)composes services based on requests from clients and returns references to composite services to these clients (interaction 6 in Fig. 2). Clients can subsequently access these services (interaction 5).

The composition service uses the context-aware discovery service to locate the constituent services its needs for a particular composition (interaction 4a). Clients may however also bypass the composition service and directly interact with the context-aware discovery service.

The context-aware discovery service dynamically discovers services. It optimizes the discovery process by means of context information (e.g., by only considering near-by services [2]), which it obtains via context agents (interaction 1 in Fig. 2). Context agents provide references to context sources (see Section 2.3), which they find by accessing the basic service discovery service (interaction 4b in Fig. 2). This discovery service is context-unaware and can be implemented using well-known discovery protocols such as SLP or WS-Discovery. The context-aware discovery service access the context sources to actually get the context information it needs (interaction 2).

Observe that the distinction between a context-aware discovery service and a basic discovery service is a logical one. In an implementation, the two discovery services may partly overlap. Also note that in a pervasive computing environment the

composition service and the two discovery services will typically be realised in a distributed manner.

In this paper, we will concentrate on the context-aware service discovery service (Section 3.1) and in particular, on the interactions that occur at its interfaces. We will also briefly discuss the composition service (Section 3.2) and its interfaces and interactions.

3.1 Context-Aware Discovery Service

The context-aware discovery service is an extension of a traditional discovery service in that it uses context information during discovery. The discovery service obtains this information through context agents, which we introduced in Section 2.3. Fig. 3 shows an example in which the discovery service makes use of three context agents, one associated with Bob, one with the service S, and one associated with Bob's car. Each context agent stores references to context sources that can provide context information about the associated entity (Bob, S, and Bob's car in this example). The context-aware discovery service obtains context information in two steps: it first obtains references to relevant context sources through one or more context agents (interaction 1 in Fig. 2/Fig. 3) and then accesses those context sources to obtain the actual context information (interaction 2 in Fig. 2/Fig. 3).

In this paper, we assume that context agents can be found through their identity (e.g., a SIP URL) using an external discovery mechanism (e.g., SIP and DNS). We also assume that context agents are able to deal with changes in their set of context sources and are able to keep this set current.

As with established discovery services [7], the CASE context-aware discovery service provides three interfaces:

- A *registration* interface, which enables services to become discoverable by *registering* their descriptions with the discovery service;
- A *discovery* interface, which allows discovery clients (the composition service or the clients of the platform) to *find* services by matching their discovery requests with the descriptions of registered services. During discovery, a client obtains information about the existence of services, their applicable parameters, and their semantics (e.g., using ontologies [4]).; and
- A *bootstrapping* interface, which clients and services use to discover the service discovery service.

In Fig. 2 and Fig. 3, the interactions that occur at the registration and discovery interfaces are labelled 3 and 4, respectively. We will not consider the bootstrapping interface any further in this paper.

The interfaces of the CASE discovery service extend traditional registration, discovery, and bootstrapping interfaces. In this paper, we define these extended interfaces in terms of a set of device-local service primitives, their parameters, and the order in which the primitives are exchanged. To keep the discussion as general as possible, we assume that service primitives are exchanged asynchronously.

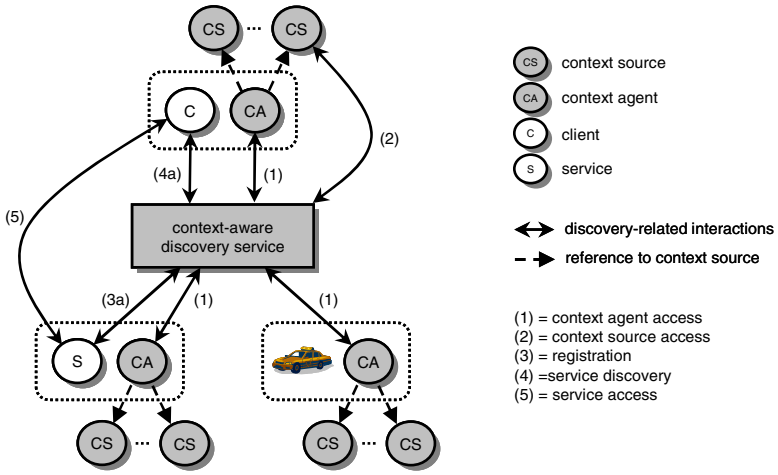

Fig. 3. Service discovery example

The discovery interface supports active and passive discovery. In *active discovery*, discovery clients actively request the discovery of certain services, whereas in *passive discovery* they wait for the discovery service to push such services to them (on a subscription basis). Passive discovery is particularly useful when a discovery client is constantly looking for 'better' services. Passive discovery might for instance be useful in the scenario of Fig. 1 if Bob's client is continuously looking for transcoding services that can deliver a certain new transmission at the highest possible quality in Bob's current context.

Active Discovery Interface. The active discovery part of the discovery interface consists of a discovery request primitive and a discovery response primitive (interaction 4a in Fig. 2/Fig. 3). As in traditional service discovery, clients use a discovery request to invoke discovery and subsequently receive a response that contains references to matching services. The request contains the usual parameters, which are a semantic specification of the services the client is trying to find (e.g., transcoding services), a set of constraints (e.g., transcoders that support MP3 audio), and a description of the scope in which the discovery service should look for matches (e.g., in terms of a geographical area or a number of network hops) [7].

The CASE-specific parameters in a discovery request are:

- A client context specification, which describes the context information of the discovery client. This parameter either consist of actual context information (which the client obtained via its context agent) or of a reference to the client's context agent. The client context specification is optional because some clients may not aware of having a context agent, and
- An (optional) set of additional constraints that describe the context that prospective services should to be in (e.g., printing services that must be located in a certain building).

Each of the references in a discovery response primitive comes with a description of the corresponding service, which enables the client to intelligently select the most appropriate service out of a number of alternative matches.

Passive Discovery Interface. The passive discovery part of the interface consists of three primitives: a persistent discovery request, a persistent discovery response, and a persistent discovery notification (also interaction 4a). A persistent discovery request is essentially an active discovery request that has a specified lifetime. Discovery clients use a persistent discovery request to instruct the CASE discovery service to generate a discovery notification when it discovers services that are 'better' than the ones it proposed in previous notifications. Before issuing such notifications, the discovery system first confirms the receipt of the discovery request by passing a persistent discovery response back to the discovery client.

With passive discovery, the scalability of the CASE discovery service is an important concern because it needs to maintain state for each outstanding persistent discovery request (see the enlargement in Fig. 2). The service discovery service therefore uses softstate persistent requests, which means that it removes the state associated with a persistent request unless that state is refreshed before a specified time (leasing).

The parameters of a persistent discovery request are similar to those of an active discovery request. The differences are that a persistent discovery request also contains:

- A reference to the client (e.g., in the form of a URL) so that the service discovery service can asynchronously deliver discovery callback notifications; and
- A specification of the types of discovery notifications the client wishes to receive (e.g., notifications that signal the appearance of a new matching services or events that indicate the disappearance of such as service).

A persistent discovery response indicates if the discovery service successfully handled the preceding request.

Registration Interface. The registration interface of the CASE context-aware discovery service is almost the same as for established service discovery services. The most important primitives are registration requests and registration responses (interaction 3a in Fig. 2/Fig. 3). A service uses a registration request to register with the discovery service, which then passes back a registration response.

The usual parameters of a registration request are a service description (augmented with semantic descriptions), a specification of the scope in which the service is available (e.g., a number of network hops or an administratively defined scope), and a self-reference so that discovery clients can actually contact the service.

The CASE-specific parameter of a registration request is the context of the service (optional), either in the form of the actual context information or as a reference to the service's context agent.

Operation. Fig. 4 describes the sequence of high-level operations used for processing persistent discovery requests. The diagram illustrates how context information

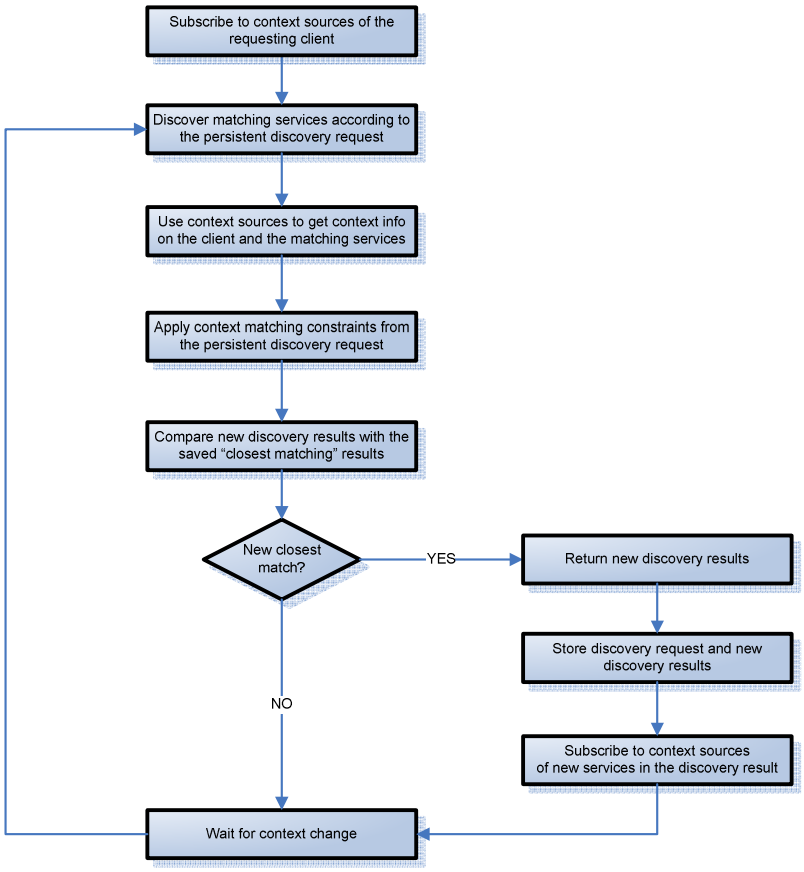

Fig. 4. High-level behavior of the context-aware discovery service

triggers the recalculation of a set of matching services. This may result in new services being returned to the client in discovery notifications.

3.2 Composition Service

The composition service is responsible for dynamically constructing composite services, based on client requests (see interaction 6 in Fig. 2/Fig. 3). The composition service can also re-compose the service to keep it "matched" with what the client initially requested if, for example, some of the service's constituent services suddenly become unavailable or when there is a context change (interaction 2, callback from a context source). In this case, the composition service proposes one or more recompositions of the service that the client is utilizing. The client can then decide which (if any) of the proposed composite services it would like to bind to.

In Fig. 1, the composition service could for instance propose a recomposition of the broadcasting service (T_a replaced by T_{av}) shortly after Bob gets out of his car. The client on Bob's mobile host can then decide if it wants to bind to the newly composed

broadcasting service. Alternatively, the client could also delegate such binding decisions to the composition service, but this would require the client to inform the composition service of its 'binding policy'.

A service request to the CASE composition service includes a semantic construct parameter that specifies the desired functionality (i.e., the composite service). The composition service returns a reference to the composite service that it has constructed.

To automatically compose services, discoverable constituent services must not only provide an explicit description of their interfaces and parameters, but also of their functionality (see the registration interface in Section 3.1). A commonly used approach for functionality description relies on the use of domain Ontologies [4]. Such semantic service descriptions enable the design of effective mechanisms for automatic composition [5].

Assuming that the functionality of a (composite) service and the query for that service (see interaction 6 in Fig. 2/Fig. 3) are expressed in a semantically consistent way (e.g., in terms of the same ontology), it is possible to estimate the "gap" between the functionality of the requested (composite) service and that offered by any of the available (constituent) services [6]. Abstractly speaking, the behaviour of the CASE composition service can then be defined as an iterative process by which new functions (i.e., constituent services) are added at runtime to an existing aggregation (i.e., composite service), until a certain acceptable error threshold between the desired and available functionality is reached.

After each iteration, the composition service evaluates the newly constructed composite service and calculates a *utility measure* (u) as a function of the semantic similarity between the required and achieved functionality, and some non-functional constraints (e.g., response time and cost). The context of the client can influence the set of constituent services that the composite service selects for a particular composition as well as the way in which they are arranged in the composite service's service graph. Once a service has been constructed, context changes such as those of Fig. 1 might require the composition service to select new constituent services (e.g., transcoders) and use them to recompose the composite service (e.g., BS in Fig. 1).

4 Technical Realization

Fig. 5 shows the technical realization of the CASE platform, which we are currently developing. We are concentrating our efforts on the implementation of the context-aware service discovery service and will add the composition service at a later stage. Our basic discovery service is the Jini lookup service [14], which we selected as a technology for initial investigations. At this stage, we also have omitted context agents from our implementation and let context sources directly register with the context-aware discovery service.

The Jini infrastructure enables Jini services to register with Reggie [10] (the Jini lookup service) using its discovery and join protocols. In our implementation, the context-aware discovery service, other services in the Jini network, and context sources are all Jini services. A reference to a context source consists of a serviceID, which is generated by the Jini lookup service when a context source registers with it.

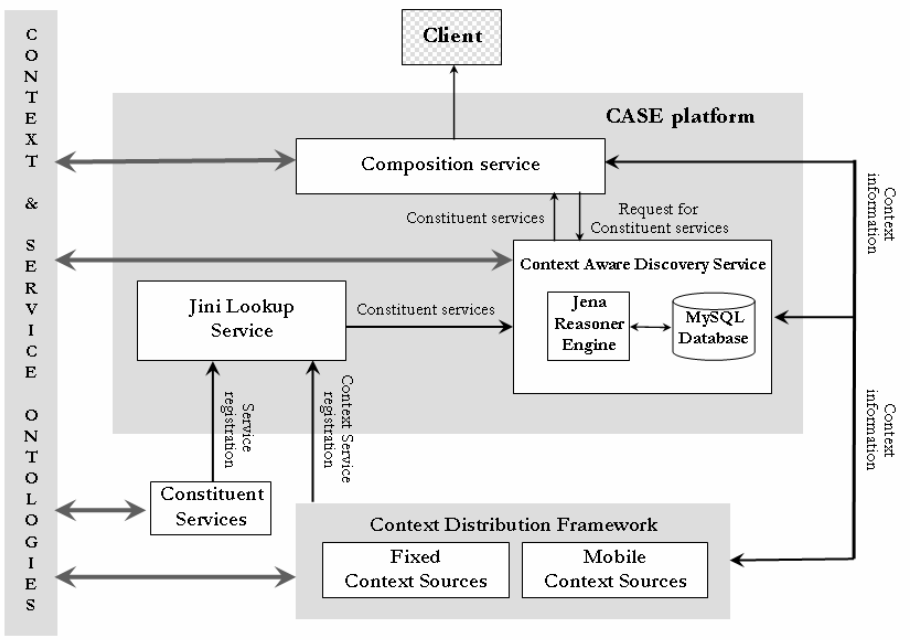

Fig. 5. Technical realization of the CASE platform

Services are registered using Jini's Service Entry functionality and provide references to their associated context sources. These context sources supply context information on the service according to the context ontology outlined in [19].

A context source uses the remote eventing mechanism provided by Jini to notify clients of changes in context information. A client (in our case the context-aware discovery service) interested in the context information implements a remote event listener interface to receive remote events. The context-aware discovery service also subscribes to the Jini lookup service so that it is notified when a new service registers.

As shown in Fig. 5, our implementation involves fixed context sources as well as mobile context sources. A mobile context source participates as a service in the fixed network using the Mobile Service Platform (MSP). The MSP design is based on the Jini Surrogate Architecture Specification [15], which enables devices that cannot directly participate in a Jini Network to join a Jini Network (with the aid of a third party). The MSP consists of an HTTPInterconnect protocol to meet the specifications of the Jini Surrogate Architecture and provides a custom set of APIs for building and running services on a mobile device.

A context source in the fixed network exports a service proxy to the Jini lookup service. The CASE context-aware discovery service uses this proxy to communicate with a context source. We use ontologies to describe context sources and services, thus facilitating a common semantics for context information and service descriptions (OWL-S). The Context Distribution Framework (CDF) provides the necessary APIs to implement context sources in the mobile and fixed network and to access context

information. It also provides support to update and distribute context information using ontological representation.

Our implementation utilises Jena [9] for service and context matchmaking. Jena is a framework for building semantic web applications. It includes a rule-based inference engine, support for ontologies, a querying mechanism, and persistent storage capability using a database. We supply Jena with the context information obtained from context sources and utilize its querying capabilities to interpret context information for the purpose of matching registered services. When a client issues a persistent service discovery request, Jena stores the service context information in a MySQL database.

5 Related Work

This section provides an overview of existing approaches that use context information to assist in the service discovery and composition processes.

The work reported in [11] combines service-oriented and context-aware computing in order to provide composite services to users. Their service descriptions consist of service context information like location, usage conditions, and a Context Of Interest Function (COIF). During service discovery, the value of the COIF is calculated at run-time to select a better service if the matching process returns more than one service. The service composition process uses the client's context information to search for the closest basic services which support the user device. The set of services selected during service discovery is further refined using context parameters that are relevant for the composition. Our approach is similar to that described in [11]. However, the major factor which distinguishes the CASE platform is its support for dynamic service re-composition whilst a (stateless) service is being utilized.

The architecture discussed in [13] builds context-aware applications as a dynamically-composed sequence of calls to fine granularity Web services based on context information. The user specifies a request for a composite service that consists of context data and a goal. This goal is converted into sub-goals using a BPEL4WS control flow template. A goal-oriented planning system SHOP2 transforms these sub-goals to corresponding plans. Later, each plan is mapped to the equivalent BPEL4WS plan describing the composite Web Service. In [13], the user provides context information manually, however in our work, we use context sources and context agents to automatically obtain context information as well as changes to the context of the service user. Thus, we provide more concrete support for the acquisition of context information.

Besides the use of context information for service composition, there has been work reported on composing higher-level context from lower-level context elements. [12] proposes to use compositions of basic context elements to build higher-level contexts. A context composition mechanism, upon receipt of a request for higher-level context, forms all equivalent context expressions from the lower-level context elements and determines which equivalent context expressions can be instantiated. The CASE platform is targeted to achieve composition of all the services and therefore, can also be used to compose higher-level context using the context information gathered from lower level context sources.

The CASE platform dynamically re-composes services by subscribing to and utilizing changes in context information of both clients and services. This mechanism promises to simplify the design of clients in pervasive environments as they need not explicitly search and compose the best services when their (or the currently composed services) context changes. The other main aspect of our work is that the context sources hosted on mobile devices are modelled as services which can participate in service discovery to offer context information (and changes to such information) in a standardized way. The use of ontologies for the representation of context information ensures that services, context sources, and the CASE discovery and composition services can meaningfully share context information. However, it is further possible to improve the matchmaking behaviour of the CASE composition service by using the COIF function of [11].

6 Summary

Future 4G service platforms will need to be able to dynamically (re)compose services to deal with the frequent context changes inherent to 4G systems and the pervasive computing paradigm in general. Service composition is also required since statically-composed services will often not directly match requests for specific non-trivial services.

Dynamic service composition relies heavily on service discovery. The use of context information during discovery reduces the number of candidate services that match the discovery request which is essential in pervasive environments, where there may be many discoverable services.

The combination of context-aware service discovery and dynamic composition can be considered to be the “next level” of intelligent service discovery/matchmaking. We expect that in future 4G service platforms there will be an increasing need for such additional intelligence to aid the development of genuinely adaptive end-user applications. CASE is an example of a set of services that will be part of these platforms and is a step in the direction of more intelligent pervasive computing service platforms.

Acknowledgments. The authors would like to thank colleagues in the IST Amigo, Freeband Awareness, and Freeband AMUSE projects who have contributed to the work described in this article.

References

1. “AWARENESS Service Infrastructure D2.10 - Architectural specification of the service infrastructure”, https://doc.telin.nl/dscgi/ds.py/ViewProps/File-47455
2. O. Ratsimor, V. Korolev, A. Joshi, and T. Finin, “Agents2Go: An Infrastructure for Location-Dependent Service Discovery in the Mobile Electronic Commerce Environment”, ACM Mobile Commerce Workshop, July 2001
3. M. Etoh, “Beyond 3G: From3G To Seamless Intertechnology Wireless Networks”, http://www.docomolabs-usa.com/pdf/PS2003-062.pdf

4. "OWL-S: Semantic Markup for Web Services", http://www.daml.org/services/owl-s/1.0/owl-s.html
5. K. Fujii and T. Suda, "Dynamic Service Composition Using Semantic Information", 2nd ACM International Conference on Service Oriented Computing (ICSOC '04), November 2004
6. V. Oleshchuk and A. Pedersen, "Ontology Based Semantic Similarity Comparison of Documents", 14th International Workshop on Database and Expert Systems Applications (DEXA'03)
7. F. Zhu, M. Mutka, and L. Ni, "Service Discovery in Pervasive Computing Environments", IEEE Pervasive Computing, October-December 2005
8. C. Noda, et al., "Distributed Middleware for User Centric System", 9th WWRF, Zurich, Switzerland, July 2003
9. HP Labs, "Jena – A Semantic Web Framework for Java", http://jena.sourceforge.net/, October 2005
10. Sun Microsystems, "Reggie: Sun Microsystems Jini Lookup service implementation", Jini Technology Starter Kit v2.1, http://starterkit.jini.org/downloads/index.html, October 2005
11. S. K. Mostefaoui, A. Tafat-Bouzid, and B. Hirsbrunner. "Using Context Information for Service Discovery and Composition." Proceedings of the Fifth International Conference on Information Integration and Web-based Applications and Services, iiWAS'03, Jakarta, Indonesia, 15 - 17 September 2003. pp. 129-138.
12. G. Thomson, S. Terzis, and P. Nixon, "Towards Dynamic Context Discovery and Composition", 1st UK-UbiNet Workshop, Imperial College, London, England, September 2003.
13. M. Vukovic and P. Robinson, "Adaptive, planning-based, Web service composition for context awareness", International Conference on Pervasive Computing, Vienna, April 2004.
14. Sun Microsystems, "The JINI Architecture Specification", http://www.sun.com/software/ JINI/ specs/ JINI1_2.pdf, December 2001.
15. Sun Microsystems, "JINI Technology Surrogate Architecture Specification", http://surrogate.JINI.org/sa.pdf, October 2003.
16. Dey, D. Salber, and G. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications", Special issue on context-aware computing in the Human-Computer Interaction (HCI) Journal, Volume 16 (2-4), 2001, pp. 97-166.
17. P. Debaty and D. Caswell, "Uniform Web presence architecture for people, places, and things", IEEE Personal Communications, Volume 8, Issue 4, Aug 2001, pp. 46-51
18. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002
19. J. Kalaoja, J. Kantorovitch, S. Carro, J. María Miranda, Á. Ramos and J. Parra, "The Vocabulary Ontology Engineering", 8th International Conference on Enterprise Information Systems, Paphos, Cyprus, May 2006