Conceptual Integration of Agents with WSDL and RESTful Web Services

Lars Braubach and Alexander Pokahr

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{braubach, pokahr}@informatik.uni-hamburg.de

Abstract. Agent communication has been standardized by FIPA in order to ensure interoperability of agent platforms. In practice only few deployed agent applications exist and agent technology remains a niche technology that runs its own isolated technology stack. In order to facilitate the integration of agents with well-established and used technologies the connection of agents with web services plays an important role. This problem has traditionally been tackled by creating translation elements that accept FIPA or web service requests as input and produce the opposite as output. In this paper we will show how a generic integration of web services can be achieved for agents that follow our active components approach. Active components allow encapsulating agent behavior in black box components that may act as service providers and consumers with explicit service interfaces. Thus, the integration approach will directly make use of these services. Concretely, the presented approach aims at answering two important questions. First, how can specific functionality of an existing agent system be made available to non-agent systems and users? Second, how can an agent system seamlessly integrate existing non agent functionality? The first aspect relates to the task of service publication while the latter refers to external service invocation. In this paper a generic conceptual approach for both aspects will be presented and it will be further shown how a specific integration with both WSDL and RESTful web services can be achieved. Example applications will be used to illustrate the approach in more details.

1 Introduction

One prime objective of the FIPA standards is ensuring interoperability between different agent platforms by defining e.g. the message format and communication protocols. As these standards have been made over 10 years ago they could not foresee that in practice agent technology would not be adopted to a high degree so that interoperability between agent systems is not a key concern nowadays. In practice, also the need for interoperability between different kinds of technological systems was present for a long time and with web services a set

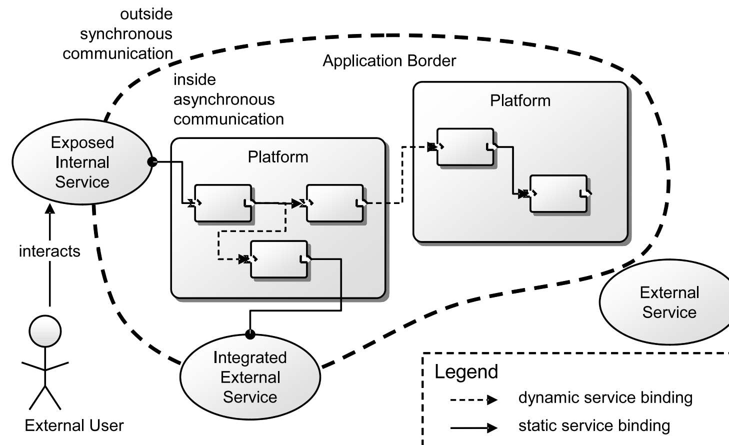

Fig. 1. Motivating scenario

of standards has emerged that is generally accepted and has already proven its usefulness within many industry projects.

From a developer perspective, standards-based interconnections are important for two main reasons (cf. Fig. 1). First, specific functionality that is needed by the software to be built could be available from another vendor as a service. Hence, it would be beneficial if it is possible to seamlessly integrate such existing functionality in the agent system and hence let it reuse this outbound knowledge. Second, it should be possible to expose functionality of a newly built application in a standardized way, such that it can be easily incorporated in other external applications. Both aspects, accessing external functionality and exposing functionality to external applications, call for *openness*, i.e. open and standardized interfaces for encapsulating the accessed or exposed functionality in programming language and middleware independent way.

This paper tackles the question, how existing web service standards and models can be integrated into agent platforms, or more specifically:

- How can (partial) functionality be exposed as web service to external system users on demand (cf. Figure 1, left)?
- How can existing web services be integrated as functionality inside of the system in an agent typical way (cf. Figure 1, bottom)?

The rest of this paper is structured as follows. Next, Section 2 gives a background of the active components programming model. The web service integration concept is described in Section 3. Illustrative examples in Section 4 show how the concept is put into practice. Related work is discussed in Section 5, before the paper closes with conclusions and an outlook in Section 7.

2 Active Components Fundamentals

The service component architecture (SCA) is a recent standard, proposed by several major industry vendors including IBM, Oracle and TIBCO, that aims at

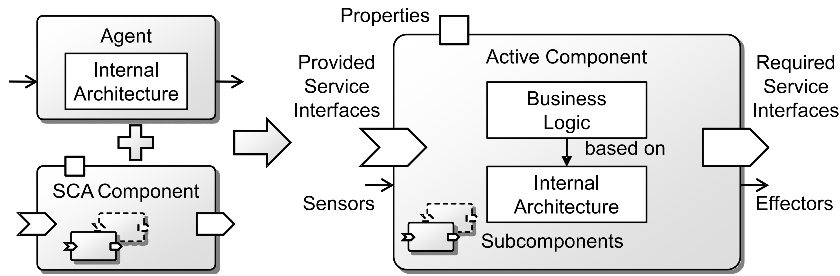

Fig. 2. Active component structure

providing a high-level design approach for distributed systems [9]. SCA fosters clearly structured and hierarchically decomposed systems by leveraging service orientation with component concepts. In an SCA design a distributed system is seen as set of interacting service providers and consumers that may reside on possibly different network nodes. Each component may act as service provider and consumer at the same time and defines its interfaces in terms of provided and required services in line with the traditional component definition of [15]. From a conceptual point of view SCA simplifies the construction of distributed systems but also has inherent limitations that delay widespread adoption.

Two aspects are especially crucial. First, even though a system is seen as set of interacting components, these components are rather statically connected with so called wires between required and provided services. The underlying assumption is that at deployment time all component instances are known and can be directly bound together. This assumption is not true for many systems with components or devices appearing or vanishing at runtime. Second, the interactions between components are supposed to be synchronous. This keeps the programming model simple but may lead to concurrency problems in case component services are accessed by multiple service requesters at the same time. Low level mechanisms like semaphores or monitors have to be employed to protect the component state from inconsistencies, but these mechanisms incur the danger of deadlocks if used not properly.

Active components build on SCA and remedy these limitations by introducing *multi-agent concepts* [3,12,13]. The general idea consists in using the actor model as fundamental world model for SCA components [8]. This means that components are independently behaving actors that do not share state and communicate only asynchronously. In this way concurrency management can be embedded into the infrastructure freeing developers from taking care of ensuring state consistency of components.[1] As it is fundamental property of the actor

[1] Asynchronous communication helps avoiding technical deadlocks. Of course, at the application layer circular waits can be created making the components wait on each other. In contrast, to a technical deadlock, in this case no system resources (threads) are bound to the waiting entities. Furthermore, the "application deadlocked" components remain responsive and can answer other incoming service requests or act proactively.

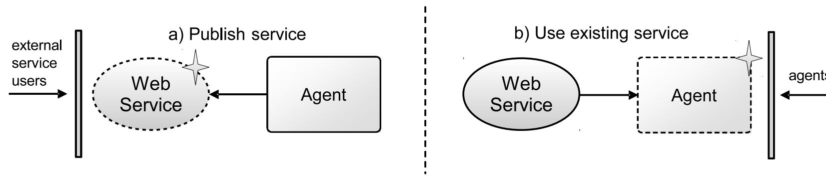

Fig. 3. Service publication and usage

model that actors come and go over time, active components do not encourage static binding of components but instead rely on dynamic service discovery and usage. In Fig. 2 an overview of the synthesis of SCA and agents to active components is shown. It can be seen that the outer structure of SCA components is kept the same and the main difference is the newly introduced internal architecture of components. The internal architecture of active components is used to encode the proactive behavior part of a component that can be specified additionally to provided service implementations. More details about the integration can be found in [3].

3 Web Service Integration Concept

In this section the overall integration concept as well as more detailed design choices will be presented. In Fig. 3 the ideas of publication and usage are sketched. The publication (cf. Fig. 3a) assumes that an agent is present that wants to make available some part of its functionality as web service for external users. The publication process thus has to ensure that a proper web service is generated and made available. In contrast, the usage approach (cf. Fig. 3b) aims at making accessible an existing web service to other agents from a multi-agent system. Here, the service functionality has to be wrapped in an agent conform manner, i.e. here a new agent is created for a service.

3.1 Making Functionality Accessible as Web Service

In order to make functionality of an application available to external system users, component services can be dynamically published at the runtime of the system. In general, it has to be determined when, where and how a service should be published. A common use case consists in performing the service publication according to the lifecycle state of the underlying component service. For this reason, per default, service publication and shutdown is automatically checked for when a component service is started or ended. The component inspects the provided service type descriptions for publish information and indirectly uses this information to publish/shutdown the service via delegation to the infrastructure. The publish information is composed of four aspects: *publish type*, *publish id*, *mapping* and *properties*. To support arbitrary kinds of service publications such as REST (Representational State Transfer) [5] and WSDL (Web Services Description Language) [17] the component uses the publish type from the service

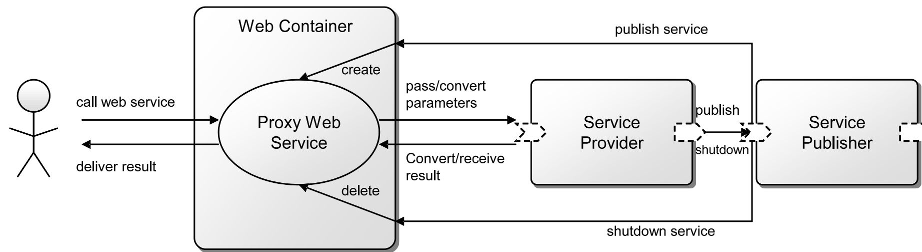

Fig. 4. Web Publishing

specification to dynamically look up a suitable service publisher (cf. Figure 4). This is done by iterating over all available service publishers until one is found that supports the requested publish type. In case a suitable publisher could be found it is instructed to publish or retreat the component service. Otherwise the publication has failed and an exception is raised within the service provider component. As an additional task the publisher may also support the advertisement of the newly deployed service within a service registry that can be accessed from external users. The service user can use the service information from the registry to locate the service and issue requests to it. The service itself acts as a proxy, which forwards the request the actual service provider and waits for a reply. The service provider executes the service domain logic and returns the service results to the proxy which in turn delivers it to the external user. It has to be noted that the incoming web service request is synchronous and therefore blocks until the internal asynchronous component service processing has been finished. In the following it will be shortly explained how WSDL and REST publishing work.

WSDL Publisher Conceptually, a direct correspondence between methods of the component service and the operations of the WSDL service is assumed, i.e. both services are syntactically and semantically equivalent with one minor exception. The exception is that WSDL services are mostly assumed to be synchronous whereas component services follow the actor model and are asynchronous. Therefore, publishing a component service requires the original asynchronous service interface being rewritten as synchronous version. Based on this interface the proxy web service component can be automatically generated using dynamic class creation using bytecode engineering or Java dynamic proxies.

Publishing a WSDL service is supported extensively in Java environments and also directly within the JDK. Most web containers like Axis2 and JDK internal lightweight container allow publishing annotated Java pojos (plain old java objects, i.e. simple objects). The container automatically reads the Java interface of the pojo and uses the additional annotation information to produce a WSDL description of the service. Java types are mapped using JAXB[2] databinding to corresponding or newly created XML schema types. In normal cases the message signatures of the Java interface are sufficient for creating the WSDL and

[2] `http://jaxb.java.net/`

only for edge cases further annotation metadata needs to be stated. For Jadex, therefore a default WSDL publisher is provided that creates an annotated Java pojo based on the supplied synchronous service interface and feds this into the web container to host the new service under the given URL.

REST Publisher REST service interfaces are potentially very different from object oriented service interfaces as they follow the resource oriented architecture style [5]. In REST the idea is that services work with resources on web servers and employ the existing HTTP communication protocol to address these resources via URIs. In addition, REST proposes special semantics to the different kinds of HTTP requests, e.g. a GET request should be used to retrieve a resource and PUT to create a new one. Taking this into account, a one-to-one mapping between method signatures of the object oriented service interface and REST methods is not directly possible or the ideal result.[3] Hence, the idea is to allow a very flexible mapping between both kinds of representations. In general, three different types of mappings are supported ranging from fully automatic, over semi automatic with additional mapping information to completely manual descriptions. Mapping information that needs to be generated encompasses the set of methods that should be published and for each method the following information:

- URL: i.e. the address that can be used to reach the service method. Typically, the URL of a service method is composed of two sections. The first section refers to the service itself and the second section refers to the method. This scheme treats methods as subresources of the service resource. In case multiple methods with the same name but different signatures exist it has to be ensured that different URLs are produced.
- Consumed and produced media types: REST services are intended to be usable from different clients such as browsers or other applications. These clients may produce and consume different media types such as plain text, XML or JSON. The REST service can be made accepting and producing different media types without changing the service logic by using data converters like JAXB for XML and Jackson[4] for JSON. The conversions from and to the transfer formats are done automatically via the REST container infrastructure respecting the given media types.
- Parameter types: i.e. the parameter types the rest service expects and the return value type it produces. In the simplest case these types directly correspond to the object oriented parameter types of the underlying service

[3] It has to be noted that characteristics like stateless interactions and cacheability, which are often associated with REST services, do not render REST useless for implementing multi-agent interactions. First, the web resources in REST are stateful being subject of creation, manipulation and deletion. Second, cacheability means that operations should be idempotent, which is achieved when e.g. mapping parts of an interaction protocol to the HTTP request types GET and HEAD. Given that for each stateful interaction a new REST resource is created both properties can be preserved.

[4] `http://jackson.codehaus.org/`

interface but often RESTful APIs intend to use basic string parameters in the URL encoded format of HTTP. If there is a mismatch between the object oriented and the RESTful interface, parameter mappers can be employed that transparently mask the conversion process. It has to be noted that the transformation of parameter values is n:m, meaning that n input values of the component service need to be mapped to m parameters of the REST service. Therefore, it has to be ensured that as well more than less parameters can be generated from the incoming value set. Parameter type generation is done in addition to conversions with regard to the consumed and produced media types.

- HTTP method type: REST defines specific meanings for HTTP method types like put, get, post, delete that roughly correspond to the CRUD (create, retrieve, update, delete) pattern. This means that different HTTP method types should be used depending on the action that should be executed on a web resource. Mapping these types from a method signature is hardly possible as the method semantics is not available to the mapper. Nevertheless, using other HTTP methods than originally intended is not prohibited per se. Possible negative effects that may arise concern efficiency as some of the method types are considered being idempotent so that existing HTTP caching can further be used.

The architecture of the REST publisher is more complex than the WSDL publisher. It partitions work into two phases. In the first phase the given component service interface is analyzed with respect to the methods that should be generated and how these should be represented in REST. For this purpose it is first checked if the developer provided custom mapping data via an annotated interface or a (possibly abstract) base class or if no mapping information has been given at all. If no mapping information is available, the publisher will use all interface methods and guess their REST interpretation. The same will be done for an annotated interface and all abstract methods if a base class was used. All given public non-abstract methods of a base class with REST annotations will be kept so that the user implementations are not be touched. The used heuristics for automatic method generation is very simple and tries to determine the different REST characteristics (according to the descriptions above) especially including the REST method type. To determine the REST method type the parameter and return value types are considered. As a default, the generator tries to interpret methods as GET and only assumes POST, if GET is not possible, e.g., due to complex data types, which could not be sent in GET requests. The generator is thus currently limited to using GET and POST methods and requires the developer to add specific mapping information if other types shall be used. Therefore, the generator has been developed as extension point for the REST publisher so that it can be easily exchanged with an enhanced version if necessary. Nonetheless, our experience with REST publication is that for more complicated services a dedicated mapping should be crafted manually by developer and the generator is mainly helpful for simple services and for rapid prototyping REST publications during manual development.

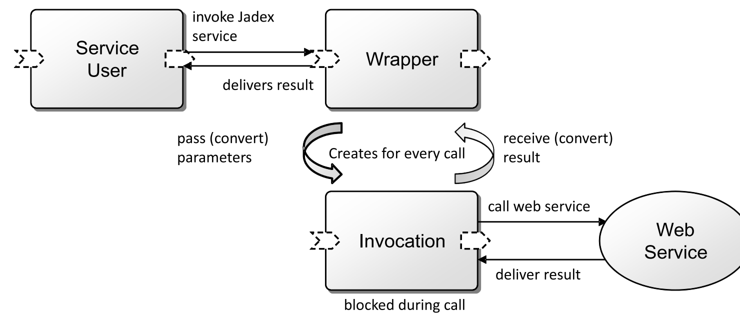

Fig. 5. Web Service Invocation

The result is a list of methods with exact descriptions how these methods should be created in REST. This list is passed on to the second stage in which a Java class is generated for the REST service via bytecode engineering. The generator first creates Java method signatures using the method name and parameter types produced in the first phase. Afterwards, it creates Java annotations for the REST specific mapping information according to the JAX-RS specification [5]. The publisher can directly pass this class to the REST container, which ensures that the service is made available.

3.2 Integrating Existing Web Services

Integrating web services aims at making usable existing functionalities as component services (cf. Fig. 5). In this way access to external functionalities can be masked and be used in the same way as other middleware services. Challenges in this integration are mainly limited to the question how an external service can be adequately mapped to the middleware and how it can be made accessible to service clients. The conceptual approach chosen is based on wrapper components, which act as internal service providers for the external functionality. A wrapper component offers the external functionality as provided service with an interface that on the one hand mimics the original service interface and on the other hand complies to the asynchronous requirements of the middleware, i.e. in the simplest case the internal interface is the asynchronous version of the external interface. The implementation of the provided service is represented by a specific forward mechanism that dispatches the call to the external web service. To resolve the synchronous/asynchronous mismatch a decoupled invocation component is used. For each service call such an invocation component is created, which is solely responsible to perform the synchronous operation. While the operation is pending the invocation component remains blocked, but as it has no other duties than performing the call this is not troublesome. This pattern keeps the wrapper component responsive and lets it accepting concurrent service invocations without having to wait until the previous call has returned.

[5] `http://jax-rs-spec.java.net/`

WSDL Wrapper The WSDL wrapper component heavily relies on the existing JAX-WS technology stack.[6] One core element of this stack is a tool called *wsimport* that is used to automatically generate Java data and service classes for a given WSDL URL. The generated code can directly be used to invoke the web service from Java. Based on this generated code the asynchronous service interface has to be manually defined relying on the generated data types for parameters. For this reason no further parameter mappings need to be defined. The wrapper component itself declares a provided service with this interface and uses a framework call to dynamically create the service implementation.

REST Wrapper The REST wrapper is based on JAX-RS technology but currently does not employ automatic code generation.[7] Instead, the asynchronous component service interface has to be created manually based on the REST service documentation. The interface definition should abstract away from the REST resource architecture and give it a normal object oriented view. The mapping of the component service towards the REST service is done with a mapping file represented as annotated interface. This annotated interface contains all methods of the original service interface and adds mapping information for the same types of information that already have been used for publishing, i.e. for each method the URL for the REST call, the consumed and produced media types, parameter and result mappings as well as the HTTP method type. The wrapper component definition is done analogously to the WSDL version with one difference. Instead of using a generated service implementation, the REST wrapper uses a dynamic proxy that uses the mapping interface to create suitable REST invocations.

4 Example Applications

In this section the publish and invocation web service integration concept will be further explained by using small example applications. The domain used to show how service publication can be achieved is a simple banking service, which offers operations for account management. For simplicity reasons it has been stripped down to one method called *getAccountStatement*, which is used to fetch an account statement viable for a specifiable date range. Integration of external services is shown using a WSDL geolocation service for IP addresses and the Google REST chart API.

---

[6] `http://jax-ws.java.net/`

[7] Automatic code generation can only be used for REST services that supply a web application description (WADL file) of themselves that represents the pendant to the WSDL file of an XML web service. Similar to wsimport, a tool called wadl2java is available that is able to create Java classes for data types and services of the REST service. A problem is that WADL has not reached W3C standard status and also is not in widespread use in practice.

```
01: public interface IBankingService {
02:   public IFuture<AccountStatement> getAccountStatement(Date begin, Date end);
03: }
04:
05: public interface IWSBankingService {
06:   public AccountStatement getAccountStatement(Date begin, Date end);
07: }
08:
09: @Agent
10: @ProvidedServices(@ProvidedService(type=IBankingService.class,
11:   implementation=@Implementation($component)
12:   publish=@Publish(publishtype="ws", publishid="http://localhost:8080/banking",
13:     mapping=IWSBankingService.class)
14: public class BankingAgent implements IBankingService {
15:   ...
16: }
```

Fig. 6. Java code for publishing a WSDL service

4.1 WSDL Publishing

Figure 6 shows how the service publication is specified in the Jadex active components framework. The existing component service interface *IBankingService* (lines 1-3), which uses asynchronous future return values[8] (see line 2) is augmented with a synchronous interface *IWSBankingService* (lines 5-7) providing the same methods. In the component definition (lines 9-16) the declaration of the provided service (lines 10-13) is extended with the publish information (lines 12, 13) specifying the target URL and the newly defined synchronous interface (line 12). In the publish information the publish type is set to WSDL web services (*ws*). In this example, the banking service is implemented by the component itself (line 14), which is stated in the provided service declaration using the predefined variable *$component* (similar to *this* in Java).

4.2 REST Publishing

As introduced earlier, REST publishing is supported in fully automatic, semi automatic and manual modes. In Figure 7 the fully automatic variant is shown, which is similar to the WSDL variant but doesn't require a synchronous interface to be manually derived. In contrast to the example above, the publish type is set to REST services (*rs*, line 8). The fully automatic mode uses internal heuristics to generate appropriate REST methods, which is difficult in many cases. Hence, additional mapping information can be supplied in both other modes. For this purpose an annotated Java interface or (abstract) class can be employed.

[8] A future [14] represents the result of an asynchronous computation, i.e. the future object is immediately returned to the caller will contain the real result value when it has been computed. The caller can use the future to check if the result already has been produced or use a listener to get a notification when this happens.

```
01: public interface IBankingService {
02:   public IFuture<AccountStatement> getAccountStatement(Date begin, Date end);
03: }
04:
05: @Agent
06: @ProvidedServices(@ProvidedService(type=IBankingService.class,
07:   implementation=@Implementation($component)
08:   publish=@Publish(publishtype="rs",publishid="http://localhost:8080/banking")))
09: public class BankingAgent implements IBankingService {
10:   ...
11: }
```

Fig. 7. Java code for publishing a REST service

```
01: public interface IRSBankingService {
02:   @GET
03:   @Path("getAS/")
04:   @Produces(MediaType.TEXT_HTML)
05:   @MethodMapper(value="getAccountStatement",params={Date.class, Date.class})
06:   @ParametersMapper(@Value(clazz=RequestMapper.class))
07:   @ResultMapper(@Value(clazz=BeanToHTMLMapper.class))
08:   public String getAccountStatement(Request request);
09: }
```

Fig. 8. REST publish mapping information

In case of an interface the method signatures are enhanced with REST annotations as shown in Figure 8. It can be seen that a method *getAccountStatement()* with one parameter of type *Request* (line 8) is delegated to a component service method with the same name but other parameter types. The method mapper annotation is used to specify the target method (line 5) and additional parameter and result mapper can be added to transform the corresponding values (lines 6 and 7). In this case a request mapper is used to extract two dates from a request and the result is generated as HTML using a simple bean property mapper. This example also shows the difference between parameter and media types. The Java return type in this example is string but the additional produces annotation (line 4) tells the client that it can expect HTML.

If even more flexibility is needed, instead of an interface a class can be used (not shown). In this class it is possible to add abstract methods and annotate them in the same way as in the interface. Additionally, other non abstract methods can be implemented with arbitrary domain logic to bring about service functionalities. If no generation is wanted, the wrapper class can also be implemented completely by the programmer.

In Fig. 9 a screenshot of the banking REST web interface is shown. This web site is produced by a banking agent with a publish annotation as shown above. This interface is automatically generated by the *getServiceInfo()* method and is per default linked to the root resource URL of the service (here localhost:8080/banking1/). It can be seen that the web site contains a new part for

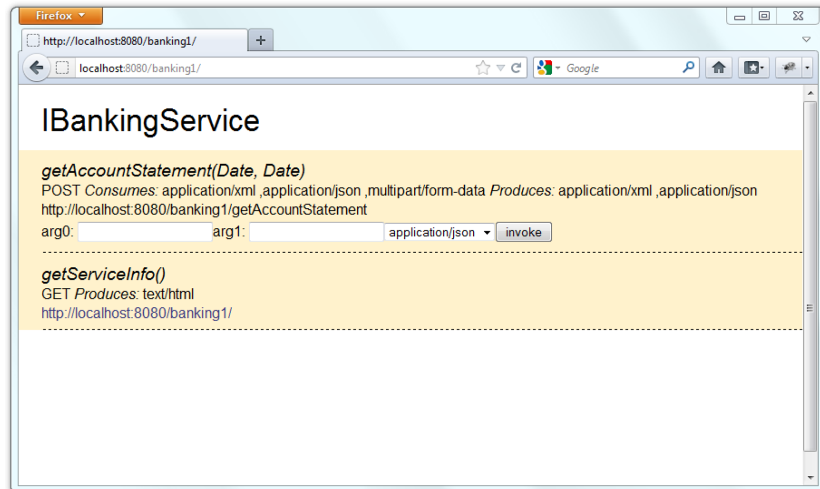

Fig. 9. Banking REST web service screenshot

```
01: public interface IGeoIPService {
02:   public IFuture<GeoIP> getGeoIP(String ip);
03: }
04:
05: @Agent
06: @ProvidedServices(@ProvidedService(type=IGeoIPService.class,
07:   implementation=@Implementation($component.createServiceImplementation(
08:   new Mapping(GeoIPService.class)))
09: public class GeoIPAgent {
10:   ...
11: }
```

Fig. 10. Java code for invoking a WSDL service

each service method with basic information about it, i.e. the method signature, REST call details, the URL and a form with input fields for all parameters. According to the media types the service method is able to consume a choice box is added to allow the user specifying in which format the input string shall be interpreted. This can be seen in the *getAccountStatement()* method, which accepts JSON and XML. Currently, the result value of a method call is produced in the same media type as the request but it is easily possible to add another control that allows to request the service to produce an alternative format.

4.3 WSDL Invocation

WSDL service invocation is illustrated using a geo IP service, which offers a method to determine the position of an IP address. After having generated the Java classes for data types and service using *wsimport*, based on the generated

```
01: public interface IChartService {
02:   public IFuture<byte[]> getBarChart(int width, int height, double[][] data,
03:     String[] labels, Color[] colors);
04: }
05:
06: public interface IRSChartService {
07:   @GET
08:   @Path("https://chart.googleapis.com/chart/")
09:   @Produces(MediaType.APPLICATION_OCTET_STREAM)
10:   @ParametersMapper(@Value(clazz=ChartParameterMapper.class))
11:   @ResultMapper(@Value(clazz=ChartResultMapper.class))
12:   public IFuture<byte[]> getBarChart(int width, int height, double[][] data,
13:     String[] labels, Color[] colors);
14:   ...
15: }
16:
17: @Agent
18: @ProvidedServices(@ProvidedService(type=IChartService.class,
19:   implementation=@Implementation($component.createServiceImplementation(
20:   IRSChartService.class))
21: public class ChartAgent {
22:   ...
23: }
```

Fig. 11. Java code for invoking a REST service

service interface an asynchronous version needs to be defined (cf. lines 1-3 in Figure 10). In the component declaration (lines 5-11) a provided service is specified using the asynchronous service interface (line 6) and an automatically generated implementation (lines 7-8). The framework method that is called to create the implementation takes as argument the wsimport generated service class.

4.4 REST Invocation

REST invocation is exemplified using the Google chart API, which can be used to create chart images of different types for a given data set. The implementation is shown in Figure 11. It consists of the asynchronous service interface (lines 1-4), the REST service mapping (lines 6-13) and the chart component definition (lines 17-23). For illustration purposes the component interface is reduced to one method that can be used to create a bar chart. The method expects the width and height of the image to produce, possibly multiple data series, label texts and series colors as input and produces an png image as output. The mapping is defined within an interface called *IRSChartService* (lines 6-15). It declares that the generated REST call uses HTTP GET on the google chart URL. In addition, parameter mappers for in- and output values need to be employed (lines 10-11, mapper code not shown) as the REST API expects a specific textual encodings for the data. The component implementation is very similar to the WSDL variant with exception of the mapping definition in terms of an interface.

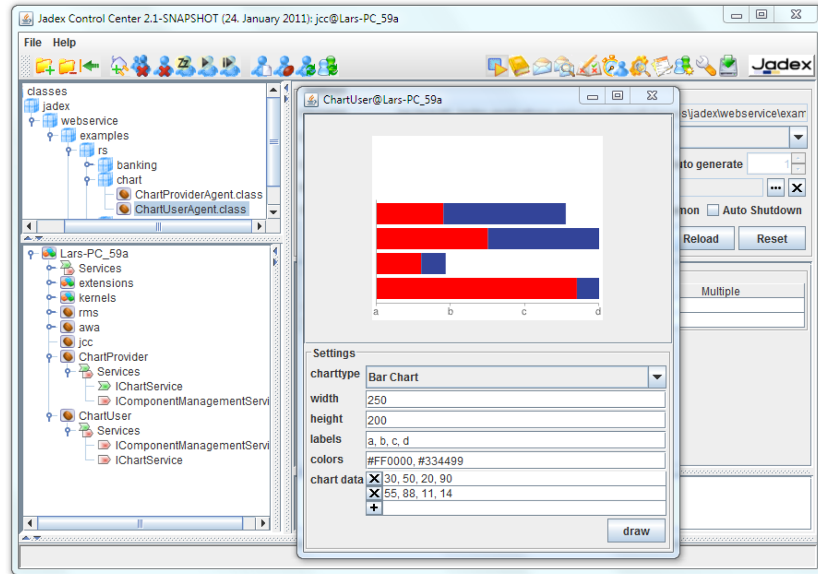

Fig. 12. Chart application screenshot

Fig. 12 shows a chart application screenshot. In the background the Jadex control center window of the platform is displayed while in the foreground the chart window is shown. The application consists of two agents. The *Chart-Provider* agent that takes over the wrapper role and offers an *IChartService* instance and on the other hand the *ChartUser* agent which own the graphical user interface for entering chart requests and displaying the resulting chart graphics. On the lower left hand side of the control center the running agent instances with required and provided services are depicted. It can be seen that the *Chart-Provider* offers an *IChartService* and the *ChartUser* requires an *IChartService*. The processing is done as follows. After a user has entered some configuration data in the chart window including e.g. width and height of the target image, series data, and colors, and issued a chart request via pressing the draw button, the chart user agent fetches its required chart service (which is dynamically searched on request) and calls the *getBarChart()* method. The service call is received by the user agent, which automatically transfers it to a REST call and hands it over to the external REST provider. The result is passed back to the user agent which displays the corresponding chart in the window for the user.

5 Related Work

In this section, the features of the approach proposed in this paper will be discussed with respect to the following areas of related work: 1) *programming level frameworks*, i.e. APIs and tools that ease the usage of web services from inside a general purpose programming language like Java, 2) *middleware extensions* that

aim at a conceptual integration between web services and agent middleware and 3) *SCA standards and implementations* that, although they don't focus on asynchronous programming, are an important conceptual inspiration of this work.

The approach presented in this paper is unique with respect to the simultaneous conceptual treatment of both directions of web service integration: publication and access. Treating both the same way has advantages e.g. with regard to developers only having to learn one API for both aspects. Programming level frameworks such as JAX-WS and Axis2[9] also follow this direction to the advantage of the programmer. E.g. in JAX-WS the developer can use the same techniques to generate Java classes and interfaces from an existing WSDL or vice versa, regardless if she wants to publish or access a web service. Interestingly, the conceptual integrations of *middleware extensions* focus usually on only one aspect. E.g. in the area of agent platforms, [6,11,2] are examples for dealing with exposing agent services as web services. On the contrary [16,10] discuss web service invocation from agents. The ProActive middleware [1] provides support both for web service invocation as well as web service publication. Yet, only the publication part provides a conceptual integration into the ProActive programming model by allowing to directly expose methods of ProActive objects as web services. The invocation part on the other hand is merely a set of utility classes comparable to other programming level frameworks. In [7] a transparent usage of web services as interoperability enabler for agents is fostered. They present an extension for JADE, which is based on the idea of having agents with a head and body, meaning that the cognitive agent part can be complemented with a web service body. This leads to an infrastructure, in which communication of agents is conceptually based on FIPA ACL, but can be technically transformed to web service invocations via SOAP. In addition, the approach also aims at bridging organizational borders by supporting content semantics via ontologies. Unlike the aforementioned approaches, the SCA standards treat service publication and invocation at the same conceptual level. Due to the prevalent synchronous programming model, SCA lacks an additional wrapper level for decoupling caller and callee during service invocation or execution.

Another important aspect of the approach presented here is the unified treatment of WSDL and RESTful web services. Most existing integration work is devoted to WSDL web services, e.g. [6,11,10,16] in the agent area and also implemented in ProActive. The main reason for this is probably the explicitly typed nature of the WSDL that lends itself to automatic code generation. REST on the other hand is much more free in the way a service is defined and used and thus requires more manual implementation or mapping specification. Publication of REST services is treated in [2], although they only support a simplistic mapping of only one operation per service. Similar to the conceptual middleware extensions, most programming level frameworks focus on one type of web service, with many standards (JAX-WS and JAX-RS) and non-standards based implementations being available for each type. One exception is Apache CXF[10],

[9] http://axis.apache.org/axis2/java/core/
[10] http://cxf.apache.org/

that incorporates APIs for RESTful as well as WSDL services. Yet, CXF does not aim at unification for the programmer, but at implementing the different available standards. The SCA standards only deal with WSDL web services and define a *ws* binding for provided and required SCA component services. Some available SCA implementation like Tuscany[11] and Frascati[12] additionally offer proprietary support for RESTful services. Yet, both require JAX-RS annotations in the service implementations that hinder a transparent usage of the same component functionality as WSDL and REST service.

In summary, the approach presented in this paper picks up earlier work on web services support for agent platforms, incorporates and extends existing ideas from SCA and combines a unified treatment of REST and WSDL with a conceptual model for an agent-style asynchronous provision and invocation of services.

6 Discussion and Current Limitations

In this paper a solution for the publication and invocation of web services has been presented. In constrast to other agent approaches it is not based on message conversions between an agent language (FIPA) and a web service language (SOAP or HTTP in case of REST). Instead, due to the more object oriented nature of active components, it becomes possible to directly use Java interfaces as service specifications, which closes the gap between services and agents to a large extent and is an improvement of the status-quo with respect to usability from a developer's perspective. The conceptual approach resembles the SCA proposal in this respect closely and extends it for agents. On basis of the service interfaces different annotations have been conceived making publication and invocation of web services a simple and rather descriptive task. Despite its advantages, the approach is not easily transferrable to other purely message based agent platforms like JADE. Although, the general ideas of publication and invocation could be kept, the explicit object oriented service representation is missing and an additional API layer would have to be introduced that is able to convert between service invocations and the agent's internal architecture. E.g. for JADE agents, the publication of a web service could mean that a corresponding agent behavior is instantiated for each service invocation. The behavior could then be implemented, like other JADE agent behaviors, as a simple Java class that encodes the agent's execution logic for reacting to the service invocation.

The approach in this paper also has some limitations that represent interesting topics of possible further work. With respect to service publication currently only service publishers exist that are able to publish a service on a web container that resides at the same host as the agent platform. In many business scenarios, it would be desirable having publishers that can deploy a service on a dedicated web container as in many cases web access is restricted from outbound computers to a specific web server instance. Another limitation consists in using only

[11] http://tuscany.apache.org/

[12] http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti

synchronous web services as support for asynchronous solutions is steadily increasing. Ultimate goal would be preserving the asynchronous service interfaces so that no substantial difference between Jadex and web service inferfaces exist any longer. Recent developements such as Ajax, Comet, Servet-API 3.0 etc. underline that in the web area stronger support for long lasting and asynchronous interactions gains traction. Concretely, regarding WSDL services, with JAX WS 2.0 it is already possible to automatically generate asynchronous service methods and also create asynchronous clients. Similarily, with the upcoming JAX RS 2.0 specification the same will be supported for for Java REST services.

7 Conclusions and Outlook

Web services are important for interoperability and extensibility as they allow integrating external functionality into applications as well as developed functionality being integrated in external applications. This paper focuses on web services support for agent platforms.

The proposed model provides a conceptual integration for both the publication of application functionality as web service as well as the invocation of external web services. To avoid dependencies between the implementation of application functionality and specific web services technology such as WSDL or REST, the model incorporates two important abstraction layers. First, the wrapper and invocation agents map a synchronous external web service interface to an asynchronous one and register the mapped service description transparently inside the middleware, such that external and internal services can be access in the same way. Second, publish services take care of exposing internal services as external web services and different publish services for REST and WSDL technology allow the same internal services to be transparently published using these different approaches.

The integration concept has been implemented as part of the open source active components platform Jadex[13]. Besides the simple examples presented in this paper, the web services integration is currently being put into practice in a commercial setting that deals with business intelligence processes and activities in heterogeneous company networks [4].

References

1. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.
2. T. Betz, L. Cabac, and M. Wester-Ebbinghaus. Gateway architecture for Web-based agent services. In *Multiagent System Technologies*, Lecture Notes in Computer Science, pages 165–172. Springer Berlin / Heidelberg, 2011.

[13] http://jadex.sourceforge.net/

3. L. Braubach and A. Pokahr. Addressing challenges of distributed systems using active components. In F. Brazier, K. Nieuwenhuis, G. Pavlin, M. Warnier, and C Badica, editors, *Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, pages 141–151. Springer, 2011.
4. L. Braubach and A. Pokahr. Developing Distributed Systems with Active Components and Jadex. *Scalable Computing: Practice and Experience*, 13(2):3–24, 2012.
5. R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.
6. D. Greenwood, P. Buhler, and A. Reitbauer. Web service discovery and composition using the web service integration gateway. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005)*, pages 789–790. IEEE Computer Society, 2005.
7. P. Karaenke, W. Schuele, A. Micsik, and A. Kipp. Inter-organizational interoperability through integration of multiagent, web service, and semantic web technologies. 98, 2012.
8. R. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM.
9. J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.
10. X. Nguyen and R. Kowalczyk. Ws2jade: integrating web service with jade agents. In *Proceedings of the 2007 AAMAS international workshop and SOCASE 2007 conference on Service-oriented computing: agents, semantics, and engineering*, AAMAS'07/SOCASE'07, pages 147–159, Berlin, Heidelberg, 2007. Springer-Verlag.
11. B. Overeinder, P. Verkaik, and F. Brazier. Web service access management for integration with agent systems. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 1854–1860, 2008.
12. A. Pokahr and L. Braubach. Active Components: A Software Paradigm for Distributed Systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*. IEEE Computer Society, 2011.
13. A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In C. Witteveen and J. Dix, editors, *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*. Springer, 2010.
14. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
15. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition edition, 2002.
16. L. Varga and Á. Hajnal. Engineering web service invocations from agent systems. In *Proceedings of the 3rd Central and Eastern European conference on Multi-agent systems*, CEEMAS'03, pages 626–635, Berlin, Heidelberg, 2003. Springer-Verlag.
17. World Wide Web Consortium (W3C). *Web Services Description Language (WSDL)*, June 2007.