Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective

Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, Schahram Dustdar
VitaLab, Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Wien, Austria
{anton,florian,platzer,treiber,dustdar}@infosys.tuwien.ac.at

ABSTRACT

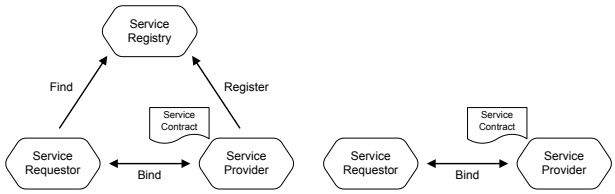
Service-oriented computing (SOC) receives a lot of attention from academia and industry as a means to develop flexible and dynamic software solutions. Facing the facts, service-oriented solutions are by far not as dynamic and adaptable as they claim to be. The initial idea of the SOA triangle to *publish-find-bind-execute* a service is often not implemented as envisioned due to a number of missing or wrongly-used concepts. In our ongoing VRESCO project, a service-oriented infrastructure is being developed which aims at solving a number of grand challenges currently evident in the SOC community. In this paper we present our initial work on providing a reasonable basis that addresses the issues of dynamic binding and invocation by leveraging a flexible solution based on notifications.

1. INTRODUCTION

During the last few years, Service-oriented Computing [9] has become an attracting research area to put forward a new paradigm for mastering the complexity of distributed applications by using loose coupling, platform independent interface descriptions and well-established standards. Service-oriented architecture (SOA) is a means for capturing these principles by providing an architectural model for developing service-oriented applications. A reasonable technology for implementing SOAs are Web services [1, 17]. In the last few years, Web services evolved from an RPC-centric model to a messaging-based communication model built on SOAP (Simple Object Access Protocol) [19]. There is still an ongoing debate within different communities whether to see Web services from an RPC or message-centric point of view. We believe that Web services should implement a messaging-based model to achieve its well-known benefits like loose coupling and a certain degree of version tolerance. RPC-centric Web services do not benefit from the SOA advantages in terms of architectural flexibility and result in Web services being just another distributed object technology [16] (compared to CORBA, Java RMI, etc).

IW-SOSWE '07, September 3, 2007, Dubrovnik, Croatia

The basic SOA model considers three main elements as shown in Figure 1(a). The *service provider* implements a given service and publishes the service description in a *service registry*. The *service consumer* queries the registry to find a certain service. If found, it retrieves the location of the service and binds to the service endpoint, where the consumer can finally invoke the operations of the service.



(a) SOA Theory (b) SOA Practice

Figure 1: Basic SOA Model – Theory vs. Practice

By implementing the SOA triangle, one could gain flexible solutions with respect to manageability and adaptivity of software systems. In practice however, software systems hardly ever implement the *publish-find-bind-execute* cycle as proposed by the SOA triangle. In Figure 1(b), we have depicted the current model as used in most of today's SOA applications. The current model solely consists of *service provider* and *service requestor*. This implies that the service requestor has to know the exact endpoint address of a service and has to generate a proxy to invoke the service. This is all done in a static way which does not conform to the basic principles of service-orientation. Building systems in such a way does not result in easily adaptable architectures and loosely-coupled systems. On the contrary, service providers and service requestors are tightly coupled. Changing the service endpoint address for instance, results in an unrecoverable application error.

In this paper, we identify a distinct number of reasons, why we believe that the basic SOA model cannot be implemented seamlessly with the currently available frameworks and tools. Due to space constraints we focus mainly on reasons related to the areas of dynamic binding and invocation as they represent two crucial core areas of service-orientation. We argue from a software engineering point of view that the implementation of the SOA triangle requires a service infrastructure and a set of programming interfaces that support application developers in building highly-scalable and dynamic SOA applications without the need to take care of the numerous Web service specifications.

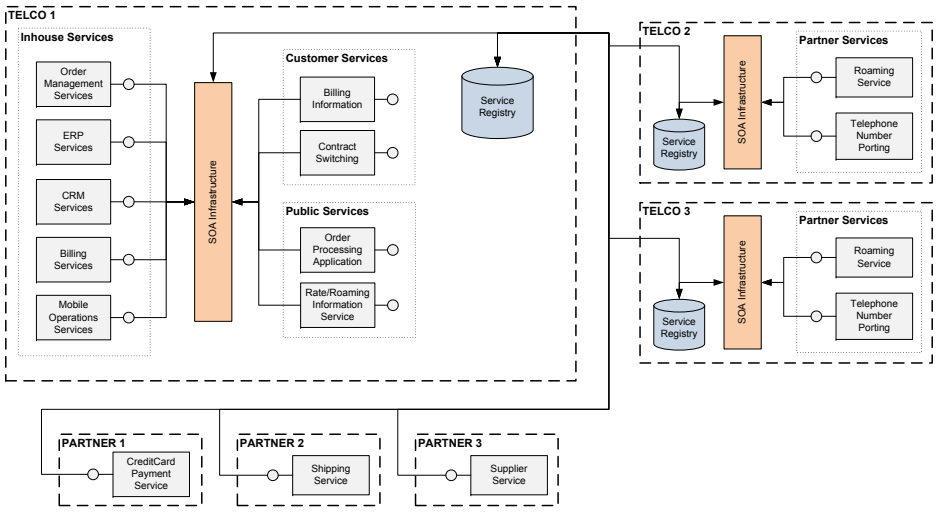


Figure 2: TELCO Case Study Architecture

We present the first results of our ongoing work in building such a service infrastructure.

The remainder of this paper is organized as follows: Section 2 presents an illustrating example where dynamic binding and invocation can be applied in a meaningful way within the SOA context. Section 3 discusses the state of the art concerning these issues. Section 4 introduces our design approach for dynamic binding and invocation in the VReSCO project. Section 5 positions our work among related approaches and Section 6 concludes the paper.

2. ILLUSTRATING EXAMPLE

In our illustrating example, we consider a telecommunication company (TELCO) that consists of multiple departments. This company provides different kinds of services to different kinds of service requestors: *Inhouse services* are shared among the different departments (e.g., CRM services, billing services, etc.). Furthermore, *customer services* are only used by customers (e.g., view billing information, send SMS messages, etc.) whereas *public services* may be accessed by everyone (e.g., order new mobile, query information about phone and roaming charges, etc.). Finally, *partner services* are used by partners and competitors (e.g., telephone number porting, roaming services, etc.).

The architecture of our case study is illustrated in Figure 2. TELCO 1 provides inhouse services, customer services, and public services via the SOA Infrastructure and the Service Registry. In addition, there are two competitors TELCO 2 and TELCO 3 that provide partner services for telephone number porting and roaming services. Moreover, several partners provide external services such as credit card payment services, shipping services, and supplier services.

To define a realistic scenario for our case study, we consider the ordering of a new mobile phone which is done using the *Order Processing Application*. After creating an order, the personal details of the user are added to the CRM database using the CRM services, and a new contract is created. In case of an existing customer, the contract details are updated. In the meanwhile, the provider verifies if the ordered mobile phone is currently in stock. If not, the phone is ordered using the supplier service of the partner. After successfully registering the new SIM card, the bill is created via the billing service. Finally the phone is shipped using the shipping service of the partner.

To be more concrete, we further consider the sub-process of telephone number porting which has been used as a case study to illustrate testing of SOAs in [2]. Mobile Number Portability is enforced by the European Union and enables customers to change the telecommunication service provider without losing their current phone number.

Figure 3 shows this sub-process in detail. First of all, TELCO 1 queries the CRM database to get the name of the customer that wants to port the phone number. To perform a successful number porting, the new service provider of the customer has to request the porting at each national competitor (which we assume are two). These requests are sent asynchronously to the competitors, which then notify the requestor if the porting was successful. If all competitors acknowledge this request, the changes can be executed. Clearly, these actions should be done within the scope of a transaction to guarantee the ACID properties. If the number porting was successful, TELCO 1 updates the CRM database by inserting the new phone number of the customer, as well as the billing information by adding a porting service fee. Finally, the customer receives a confirmation SMS that the new number is now active.

We hark back to this case study in the following sections, when describing current problems in SOA, as well as our approach to overcome these issues.

3. CURRENT PROBLEMS AND ISSUES

In this section, we address some of the current problems and issues we observed in the SOA model.

3.1 Web Service Registries

One significant problem with SOA relates to the shortcomings of current Web service registries, especially the two standardized efforts Universal Description Discovery and Integration (UDDI) [6] and ebXML [5]. Both approaches have gone through long running standardization processes and are broadly accepted as "the standards" for Web service registries. However, in practice, ebXML is used only in some

Figure 3: Telephone Number Porting

vertical industries, and UDDI is rarely used. This is highlighted by the fact that IBM, Microsoft, and SAP have shut down their public UDDI registries in 2005.

We presented a detailed analysis of Web service registries in [3]. Besides the missing support for dynamic binding, we identified additional shortcomings of current registry approaches that hamper the full utilization of the SOA triangle. Among others, support for notifications, compositions, dynamic searching and querying are included. However, in this paper we mainly focus on dynamic binding and invocation.

3.2 Dynamic Binding

In general, dynamic binding (or late binding) is the process of linking an abstract service to a concrete service instance at execution time. Ideally, this should be handled transparently for the application developer. Dynamic binding in service-centric systems is mostly used in combination with runtime service discovery. This means that the service consumer tries to find a service matching a given criteria. Based on the available services, the consumer selects the service that best fulfills her constraints – for instance, functionality or Quality of Service (QoS) – and binds to the service dynamically at runtime. Since, the QoS may change over time, it might be necessary to re-bind to another, functionally identical service (e.g., due to a better QoS).

Considering the number porting process of our example, there are two scenarios where dynamic binding plays an important role. Firstly, the TELCO should dynamically bind to the services of its competitors, for instance when selecting the number porting service of the competitor. We refer to such kind of binding as *content-based dynamic binding*. Due to the fact that the number porting interfaces will not match exactly among the competitors, a set of conversion rules is used to handle this heterogeneity.

Another important aspect is the use of QoS information for the selection and dynamic binding of certain Web service instances [20], which we refer to as *QoS-based dynamic binding*. This can be used for performance intensive operations, e.g., mobile operation services such as sending SMS and MMS. The latter are usually highly demanded services by mobile customers. In our case study, QoS-based dynamic binding is used when sending the confirmation SMS of the number porting. To achieve this kind of dynamic binding, a distinct number of service instances are running on one or more hosts. Furthermore, a QoS monitor (as developed in our previous work [14]) constantly monitors the QoS attributes of these services. The dynamic binding mechanism is then used to select a specific service instance which best fits according to the selection strategy. We present the details of both approaches in Section 4.

Considering the state of the art, existing Web service registry approaches claim to allow dynamic binding. In fact, dynamic binding is not natively supported. A service requestor can search the registry based on certain criteria and then retrieve the endpoint of the service. However, the main advantage of dynamic binding is *endpoint transparency*.

```
1  // Query the registry for a service with the
2  // given name and given QoS values
3  list = query("MessagingService", 500, 0.9);
4
5  // select one service
6  serviceToInvoke = selectBestService(list);
7
8  try {
9    invokeService(serviceToInvoke,
10                 "sendSMS", arguments);
11 } catch (ServiceException e}
12   // something went wrong: re-query UDDI
13   // and try to invoke another service
14 }
```

Listing 1: Dynamic Binding Example with UDDI

Listing 1 shows a pseudo-code example for *QoS-based dynamic binding* using UDDI. The `query()` method in line 4 queries the UDDI registry for a service with the given name. Furthermore, it takes two other parameters, the maximal response time of a service and the minimal availability required from the service provider. The QoS attributes of each service are stored in a `tModel` which is associated with the service as described in [13]. The method for querying is about 30 lines of code and cannot be simply reused, since QoS is not a first-class citizen in UDDI.

The passiveness of dynamic binding with UDDI becomes evident especially when a service invocation exception occurs (line 11) and dynamic rebinding becomes necessary.

The main concern is that the client is responsible for taking all required actions (e.g., re-querying the registry for new service bindings, polling the registry for possibly new services, etc.). Furthermore, selecting the best service (line 6) and dynamically invoking it (line 9–10) cannot be done as easy as shown here for demonstration purposes. It is up to the client to select and invoke the best service. The main drawback of this approach is the fact that it involves a lot of client-side code since current registries (and their APIs) do not provide support for implementing such dynamic binding. Although we only use UDDI for illustration purposes, the same problems related to dynamic binding and invocation are also inherent to ebXML.

3.3 Dynamic Invocation

Dynamic invocation represents an issue closely related to dynamic binding. Web services are typically invoked in a static manner, that means, stubs are generated from service descriptions in WSDL and then used to invoke service operations. The main drawback of this approach is the lack of dynamism. In some cases (e.g., when invoking a service returned from a service discovery query) it should be possible to invoke the service just by using its endpoint, the operation name and the required input message. Current Web service technologies do not provide an elegant way to use such a dynamic invocation.

Per definition, Web services exchange SOAP messages, either in a synchronous or asynchronous manner. Nevertheless, when looking at existing Web service implementations, RPC-style interactions with previously generated stubs are currently predominant. However, this was not the initial idea of SOA. In contrast, the current use of Web services leads to inflexible applications which are specified at design time and cannot be changed at runtime.

The Apache Web Service Invocation Framework (WSIF)[1] provides a way to perform dynamic invocation of Web services. However, the framework provides only limited support and is basically only applicable for simple services.

4. THE VReSCO APPROACH

In this section we describe the initial work on our approach for efficiently implementing service-based applications. Firstly, we present an overview of the VReSCO infrastructure and its architecture. The VReSCO project aims at addressing some of the current challenges in SOC [8]. This paper presents our initial steps since many of these infrastructure components are currently work in progress. Secondly, we focus on two general issues for developing service-oriented applications, namely *dynamic binding* and *invocation* and how we are able to solve it using VReSCO.

4.1 Overall Architecture

The main goal of VReSCO is to provide an infrastructure (through a client-side API) to application developers that enables efficient and flexible development of service-oriented applications without bothering the developer with all the details of numerous Web service specifications such as UDDI. Furthermore, it provides a flexible infrastructure that allows to cope with many deficiencies of existing frameworks, namely the lack of dynamism as required to implement SOAs [8]. The client-side library transparently handles

[1] `http://ws.apache.org/wsif`

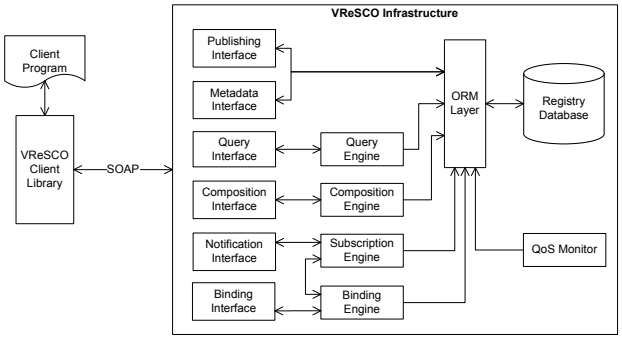


Figure 4: VReSCO Architecture

the SOAP communication with the VReSCO infrastructure and its services which include, among others, publishing, searching, querying and composing services. The overall architecture of VReSCO is depicted in Figure 4.

The most important VReSCO services provided to the application developer are:

- **Publishing and Metadata Service:** It is used for static and dynamic publishing of services with its interface description and associated metadata. Static publishing is done using a Web page as front-end for the service. Dynamic publishing is done at runtime and makes a service immediately available for others within the infrastructure. Metadata includes functional attributes (e.g., operations, messages, pre- and post-conditions) and non-functional attributes such as domain-specific information, performance and dependability (e.g., response time, availability, etc) as proposed in [14].

- **Searching and Querying Service:** It allows to query and search for available services published within the infrastructure. Querying enables to find exact matches of the query string in service descriptions and its non-functional attributes (e.g., QoS), whereas searching allows to find services using full text queries with fuzzy matches (see our previous work in [12]).

- **Binding and Invocation Service:** Issuing a query at runtime requires the ability to dynamically bind to the returned service. After the binding, an invocation can be performed. The provided service is well-suited to support the application developer to transparently handle these aspects. The *find-bind-execute* cycle of the basic SOA model leads to flexible applications by providing full transparency of the concrete services the application is bound to. We present more details on binding and invocation below.

- **Notification Service:** Notifications play a central role in SOAs, although there are not very explicit. This service allows the developer to register for receiving different events: i) when new services in specific categories are available, ii) when the QoS of a service changes below or above a given threshold, iii) when the service interface has changed. The use of notifications is transparent in the sense that the client library handles the creation of local endpoints to receive notifications from VReSCO if an event occurs.

- **Composition Service:** Composing services is a crucial area within the SOC community. The VReSCO infrastructure is coupled with an orchestration engine to achieve a composition of services by letting the user specify composition requests in a domain specific language (DSL) which is currently investigated. The information encoded in the DSL is then used and matched with the services available within VReSCO. This iterative and semi-automated composition development process allows a stepwise refinement of the composition until it is deployable. This refinement also includes the development of missing services or necessary data transformations.

The VReSCO services are provided as normal Web services with WSDL interfaces, and designed to interact with a client-side library. They are therefore not foreseen to be invoked by the developer himself, i.e. the latter merely uses a set of API classes. The main reason for providing a client-side library is the ease of use for the developer without the need to generate stubs for the infrastructure services, create endpoints for receiving notifications, etc. All these aspects are handled by the client library, which is available for two main platforms, Java and C#/.NET. In addition, it is also possible to use the VReSCO services directly, for example on other platforms such as Ruby, Python or any other language. Of course, this is not as comfortable as having a client-side API but it is not a limitation.

Besides the provided infrastructure services, a number of other components are needed for the implementation of the services. A central part is the *Registry Database* and the *ORM Layer* (Object Relational Mapping) for accessing the registry database. Based on the reasons provided in Section 3, we currently do neither use UDDI nor ebXML. Instead, we use our own data model for persisting the services and their metadata. The registry itself is only accessed using our infrastructure services and not directly from outside, which allows us to use an efficient and extensible data model with our database of choice.

Another central component is the *QoSMonitor* for measuring the performance and dependability related attributes. Our approach presented in [14] implements a client-side approach based on a low-level TCP analysis. This implies that no access to the service implementation is required for performing the measurements. Other components include a *Query Engine*, a *Composition Engine* and a *Subscription Engine* which are out of scope of this work and therefore not described in more detail.

The VReSCO system is currently being implemented on top of the Microsoft Windows Communication Foundation (WCF)[2], which is a set of APIs to build and host service-oriented applications. Moreover, we use the object relational mapping tool NHibernate[3] for persisting services and their metadata in the registry database.

4.2 Dynamic Binding and Invocation

Achieving dynamic binding requires a strong support from the VReSCO infrastructure. As described in Section 3, we distinguish two cases of dynamic binding within an application, *QoS-based* and *content-based dynamic binding*.

[2] `http://wcf.netfx3.com/`

[3] `http://www.nhibernate.org/`

QoS-based Dynamic Binding.

The main idea is the transparent dynamic binding to a concrete service according to some QoS criteria. The criteria are specified when issuing the query for a given service to the VReSCO runtime. Listing 2 shows the usage of dynamic binding with VReSCO using our case study introduced in Section 2. In line 1, a binding listener is registered at the VReSCO proxy (identified by the `vresco` instance) to receive notifications when new services match the query in line 2–5. As explained above, the query language allows to specify QoS parameters and a query model. In this case, `QueryMode.Relaxed` means that if the query cannot find a service with the appropriate QoS values, the registry should return the service that best fits the given criteria.

```
1  vresco.registerBindingListener(this);
2  service = vresco.query("MessagingService",
3            "QoS.ResponseTime <= 500ms
4                 and QoS.Availability > 0.9",
5            QueryMode.Relaxed);
6  vresco.setRebindingStrategy(
7      BindingStrategy.UpdateOnBetterMatch);
8  ServiceProxy proxy =
9          vresco.createServiceProxy(service);
10 proxy.sendMessage("sendSMS");
```

Listing 2: Dynamic Binding with VReSCO

Another remarkable point is the *rebinding strategy* which is defined in line 6–7. This specifies to rebind to another service with better QoS if one gets available that matches our query. This rebinding is transparent and triggered by a change in the perceived QoS. In line 8–9, the service proxy that is necessary to achieve this level of transparency is generated by the VReSCO infrastructure. Finally, the service is invoked using the service proxy in line 10.

Content-based Dynamic Binding.

To achieve content-based dynamic binding a mapping between some application logic and a distinct service has to be provided. For instance, considering the case study in Section 2, a number of external and partner services are available and published in the VReSCO registry. In order to achieve a transparent dynamic binding to the TELCO telephone number porting services, we need to establish a mapping between the service category and the service identifier used in the application code. An example query for generating a service proxy for the telephone number porting service for TELCO 2 could be:

```
vresco.query(PartnerService.PhoneNumberPorting,
             PortingProvider.TELCO2);
```

Prior to issuing this query, the VReSCO infrastructure needs to know the mapping from the `PartnerService` and `PortingProvider` parameters to the concrete services in the registry. This can be configured by adding metadata to each telephone number porting service, in this case the category (`PhoneNumberPorting`) and the service provider identification (`TELCO2`).

When a telephone number needs to be ported, each provider must be informed. Thus, it is often desired to hide the differences of the available service interfaces in the same category by letting the service proxy handle the heterogeneity transparently. This aspect is handled using a set of *conversion rules* that describe how the input and output messages of services in a specific category need to be transformed by

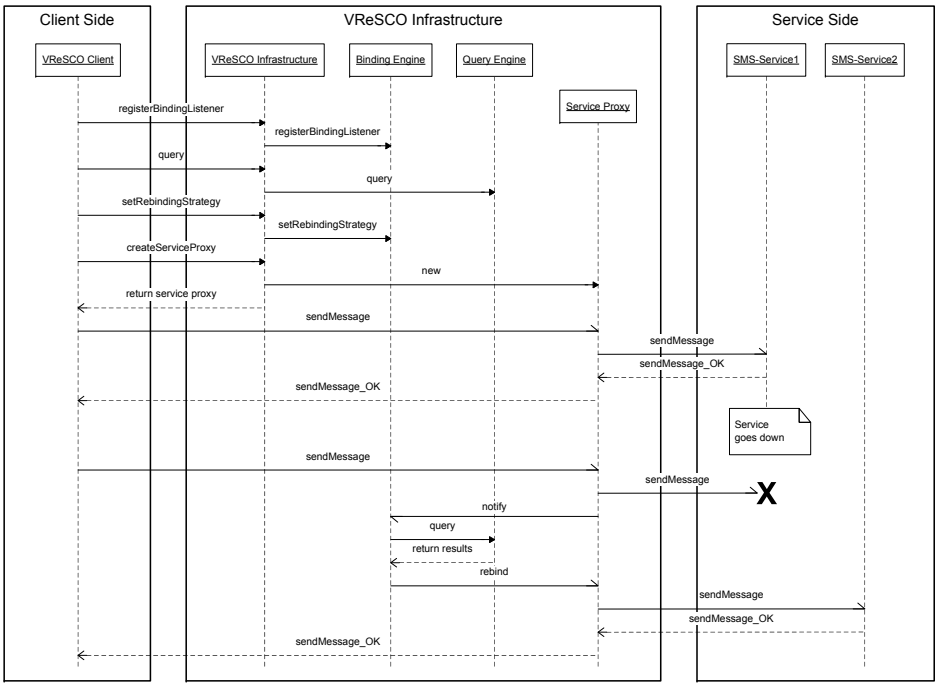


Figure 5: VReSCO Dynamic Binding

the service proxy. The service proxy can then be used uniformly for each service in the same category. The input and output message understandable by the service proxy have to be specified as additional metadata to the service category. For each service in the service category, conversion rules for converting messages to the messages understandable by the service proxy have to be added as metadata. This information is further used by the service proxy to dynamically invoke the service.

The conversion rules are specified using XSLT (or alternatively XQuery) in the first version of our implementation. On execution, the proxy receives the uniform XML message, also defined as metadata for each group of services, and based on the current service binding it executes the corresponding rule to transform the message and redirect it to the original service. To enhance the performance of this mediation step, the XLST stylesheets are compiled which keeps the overhead at the minimum. Due to space restrictions, the concrete syntax and execution of these conversion rules are not described in more detail.

Dynamic Binding in Detail.

The detailed dynamic binding process is illustrated in Figure 5 which shows the interaction of client- and service-side with the VReSCO infrastructure. VReSCO clients use the provided API to register a binding listener, query for services fulfilling some given constraints, and setting the rebinding strategy as explained above. The VReSCO infrastructure forwards these requests to the responsible components, which are the *Binding Engine* and *Query Engine*, respectively. Accordingly, the client creates a service proxy which is then used to invoke the concrete service instance. Now consider that *SMS-Service1* goes down and is not available any more. In this case, the service proxy can no longer reach the bounded service instance. Therefore, at the next service invocation, a notification is sent to the *Binding Engine* using the registered binding listener. On being notified, the *Binding Engine* uses the *Query Engine* to query the registry for matching services, and then rebinds to an alternative service according to the current rebinding strategy. Finally, the service proxy uses *SMS-Service2* for the service invocation.

5. RELATED WORK

Pautasso and Alonso [10] discuss several different binding models for (Web) services, as well as different points in time when the bindings are evaluated. The motivation of their work is the shortcoming of current composition languages such as WS-BPEL [7]. In WS-BPEL, dynamic binding is supported by re-assigning endpoints using the `partnerLink` construct. Endpoints represent specific ports of a service interface at runtime which are usually identified using WS-Addressing [18]. However, dynamic binding in WS-BPEL can only be achieved if the interfaces of the different services are identical, which limits the flexibility of this approach. The authors present a flexible binding model using their JOpera system. In this approach, binding is done using reflection and therefore does not require a specific language construct.

Di Penta et. al. [11] present the WS-Binder framework for enabling dynamic binding within BPEL processes. They distinguish between three different types of binding mechanisms: *Pre-execution workflow global binding* occurs prior to the execution of a composition and is done using genetic algorithms. *Run-time local binding* allows to select service bindings while the composition is already running. Finally, *run-time workflow slice re-binding* stops the execution of the composition in case of an error (e.g., service is not available or QoS values are not as desired), and determines the work-

flow slice still to be executed using global binding. This approach is built on top of WS-BPEL and uses proxies to separate the abstract services with the concrete services instances. Both approaches have in common that they rather focus on dynamic binding with respect to composition environments and do not focus on the binding at the core SOA level as addressed by our approach.

The latter work is part of a larger platform which was developed within the scope of the European project SeSCE[4]. Other activities in this project have been done in the area of service discovery [4, 15]. The authors distinguish between three types of service discovery. *Early service discovery* occurs in the requirements engineering phase and is driven by the requirements specification. *Architecture-driven service discovery* is done during the design phase and is driven by the specification of functionality, quality attributes and constraints. Finally, *runtime service discovery* deals with the discovery and replacement of services at runtime. In contrast to our work, the approach presented in [15] focuses on runtime monitoring the compliance of service-centric systems to requirements and discovering alternative services at runtime, whereas dynamic binding is not explicitly addressed.

6. CONCLUSIONS

In this paper we address one fundamental shortcoming of today's SOA implementations, namely, dynamic binding and invocation. We illustrate the set of today's challenges by utilizing an example based on which those shortcomings are analyzed henceforth. SOAs had foreseen the publish-find-bind cycle (SOA triangle), whereas as today, most SOA implementations use (for practical reasons) only the interaction between service requestor and service provider with service contracts. This, of course, limits the envisaged potential of SOA implementations considerably. In our research project VReSCO we provide a client-side API to allow for dynamic binding and invocation of services to solve many of today's problems related to dynamic binding and invocation and its relationship to registries. In this paper we discuss those implemented parts of our infrastructure which can be of help when building large-scale SOAs requiring dynamic binding and invocation.

7. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer Verlag, 2004.

[2] S. Dustdar and S. Haslinger. Testing of Service-Oriented Architectures - A Practical Approach. In *5th International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld (Net.ObjectDays 2004)*, pages 97–109, Sept. 2004.

[3] S. Dustdar and M. Treiber. A View Based Analysis on Web Service Registries. *Distrib. Parallel Databases*, 18(2):147–171, 2005.

[4] A. Kozlenkov, G. Spanoudakis, A. Zisman, V. Fasoulas, and F. Sanchez. Architecture-driven Service Discovery for Service Centric Systems. *International Journal of Web Service Research*, 4(2):82–113, 2007.

[4] `http://secse.eng.it/pls/secse/`

[5] OASIS International Standards Consortium. *ebXML Registry Services and Protocols v3.0*, Mar. 2005.

[6] OASIS International Standards Consortium. *Universal Description, Discovery and Integration v3.0 (UDDI)*, Feb. 2005.

[7] OASIS International Standards Consortium. *Web Service Business Process Execution Language v2.0 (WS-BPEL)*, 2006.

[8] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing Research Roadmap, 2006. `http://infolab.uvt.nl/pub/papazogloump-2006-96.pdf` (Last accessed: May 31, 2007).

[9] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Dezember 2003.

[10] C. Pautasso and G. Alonso. Flexible Binding for Reusable Composition of Web Services. In *Proceedings of the 4th International Workshop on Software Composition (SC'2005), Edinburgh, UK*, pages 151–166, 2005.

[11] M. D. Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. D. Nitto. WS Binder: a Framework to enable Dynamic Binding of Composite Web Services. In *Proceedings of the International Workshop on Service-oriented Software Engineering (SOSE'06)*, pages 74–80, 2006.

[12] C. Platzer and S. Dustdar. A Vector Space Search Engine for Web Services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*, 2005.

[13] S. Ran. A model for web services discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, 2003.

[14] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, Sept. 2006.

[15] G. Spanoudakis, A. Zisman, and A. Kozlenkov. A Service Discovery Framework for Service Centric Systems. In *Proceedings of the IEEE International Conference on Services Computing (SCC'05)*, pages 251–259, 2005.

[16] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):55–66, 2003.

[17] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[18] World Wide Web Consortium (W3C). *Web Service Addressing*, 2004.

[19] World Wide Web Consortium (W3C). *Simple Object Access Protocol v1.2 (SOAP)*, 2007.

[20] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6, 2007.