Towards a Diamond SOA Operational Model

Mohammed AbuJarour, Felix Naumann
Hasso-Plattner-Institut
University of Potsdam, Germany
{*firstname.lastname*}*@hpi.uni-potsdam.de*

***Abstract*—The triangular operational model with the three roles of service-registry, -provider, and -consumer has been the traditional operational model in Service-oriented Architectures (SOA). The central component in this model, the service registry, plays a passive role. This passive role is due to the lack of adequate information resources. For example, service descriptions provided by service providers are in general not rich enough for service selection. More elaborate service descriptions are vital in several aspects in Service-oriented Computing (SOC), such as service quality assessment, service discovery and selection, etc.**

To increase the usability of Web Services with poor descriptions, we propose a novel approach to extend the traditional SOA operational model by introducing a new role: the Service Invocation Proxy (SIP). The main goal of this role is to enable service brokers (registries) to play an active role, e.g., to enrich service descriptions with metadata about invocation statistics, service quality measures, and service usage contexts. A SIP plays its role during service invocation where it mediates between consumers and providers at runtime. We have implemented a proof-of-concept of this approach and show the details in the paper.

***Keywords*-Service Description, Web Service, Operational Model, Proxy**

I. The Triangular SOA Operational Model

The triangular SOA operational model, depicted in Fig. 1, has been the traditional operational model in Service-oriented Computing (SOC) [1]. Service providers publish their services to one or more service registries. Service consumers discover the required services that meet their (business) requirements among the already published services in the registry. A service consumer can invoke a particular service by issuing a request to the corresponding endpoint's URI.

In this traditional model, the service registry acts as a passive participant; it only reacts to requests, but does not behave actively. It reacts to publish requests from service providers by adding the intended service to its service collection. It reacts to discovery requests from service consumers by finding the best matching services among its service collection, with respect to the submitted query. Although its role is vital, we believe that a service registry can do more, especially in enterprise applications .

The limited role of service brokers (registries) is due to the limited information resources that service providers release about their services. Such service descriptions are not rich enough, because they are usually created by programmers who tend to focus on the functional (implementation) aspects. Moreover, several tools and frameworks that automate the process of generating Web Services from existing applications, e.g., the Web Tools Platform (WTP)[1], exacerbate the problem due to the limited descriptions they generate.

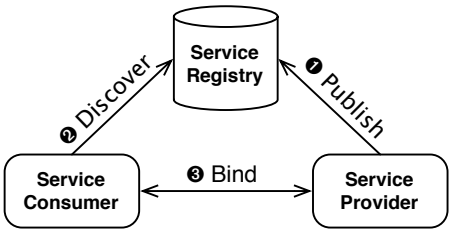

Figure 1: Traditional, triangular SOA operational model

Because of the increasing complexity of Web Services [2] and their driving business needs, we believe that the traditional SOA operational model is not sufficient for enterprise needs. We propose an extended SOA operational model, where a service broker can play an active role and help meet the increasing complexity of business needs in enterprise applications. The main challenges involved in this work include:

- **Dynamic SOA and business environments**: In today's market places, change has been the rule rather than the exception. IT systems should adapt to changes quickly. Nevertheless, such rapid IT responses require Business-IT alignment, which is difficult to achieve [3]. On the other hand, organizations would like to utilize each available IT technique to perform their business efficiently: Available IT solutions should be described in an understandable way for non-technical persons.
- **Increasing complexity**: Business needs are becoming more and more complex. This increasing complexity in business needs is reflected in increasing complexity in serving IT systems, as well as tools and frameworks used to implement such complex systems. Most of such tools focus on the functional aspects of the implemented systems and ignore service descriptions that can be used by service consumers to decide whether

[1]http://www.eclipse.org/webtools

and when to use a specific service.

- **Heterogeneity**: Different techniques and formats are used to describe and provide services. Different parties in SOA environments may have different notions of the same concept, e.g., reputation, document, message, etc. Furthermore, different scales could be used to describe the same notion, e.g., trust level.
- **Inadequate descriptions for service discovery and selection**: Full-text search is usually used by service registries as a means of service lookup, but the quality of the result list depends on the information used to lookup a service. Richer service descriptions lead to greater system automation and thus more business agility [4].
- **Obtaining values for non-functional properties**: In Service-oriented Computing, non-functional properties (NFP) of Web Services are the key to differentiate between Web Services with the same functionality, e.g., weather forecasting. NFP's are usually referred to as the Quality of Web Services, which represents an essential measure in SOC, but, obtaining values for these non-functional properties is not trivial.

The main contribution of this work is an extended SOA operational model, which adds a fourth role (Service Invocation Proxy) to the triangular operational model, i.e., service provider, registry, and consumer. A Service Invocation Proxy has the role of mediating service calls between service consumers and service providers. The main benefit of this added role is invocation analysis, which can be used to enrich service descriptions to enhance service discovery and selection, evaluating non-functional properties, etc.

The remainder of this paper is structured as follows. In Section II, we explore the related literature. In Section III, we introduce our proposed approach and a use-case. Further implementation details are introduced in Section IV. Alternative settings are introduced in Section V. Section VI concludes this paper and draws our future roadmap.

II. Related Work

The limitations of the current, traditional SOA operational model have been highlighted by several researchers in the community. In [5] the authors show that the triangular model is not used widely in practice because of the limited role of service registries. Removing the role of the service registry (breaking the triangle) violates the basic principles of SOC, namely loose coupling and dynamic binding. To restore this broken triangle, the authors proposed a software engineering approach that enables dynamic binding and invocations of Web Services. Our approach emphasizes the limitations in the triangular model and extends it to enable service registries to behave actively.

Another approach to achieve active Web Service registries was introduced in [6]. The authors use RSS feeds to announce any changes in the registered services to interested service consumers. The information provided by such feeds is generated by service providers, which tend to focus on the technical part of their services rather than providing documentation or descriptions.

The main reason behind the highlighted limitations of the triangular model in the aforementioned work is the lack of rich service descriptions [7]. Therefore, researchers have proposed several approaches to gather information about services to handle the problem of poor service descriptions:

In their early work Singh, and Maximilien proposed to extend the clients used by service consumers to gather required information to evaluate reputation in SOC [8]. However, extending running systems is not desirable for service consumers, because modifying a running system is expensive and error-prone. Another approach proposed in [9] is to ask service providers to provide the required or missing information. Evaluating quality attributes using this information needs trust; service consumers need to believe the information provided by service providers. An alternative option is to obtain this information from the community, i.e., from other service consumers. This approach is popular in multi-agent environments [10]. In this recent approach, trust is also an issue, because the community includes competitors. New services and services that have not been used by the community remain unexplored and unrated.

The Adaptive Service Grid project targeted similar problems in SOC [7]. To tackle the lack of rich service specifications in SOC, this project builds a registry of ontology descriptions of all Web Services. These ontologies are created manually by domain experts, and one cannot always assume that these ontologies exist in practice. An extension to UDDI registries was proposed in [11] by introducing the *Web Service QoS Certifier*. Its task is to verify that the claimed QoS values announced by service providers are correct before registering a service in the UDDI registry. That work is limited to UDDI registries and its target is also limited to QoS values.

Producing metadata about Web Services has been a popular approach especially to assess the quality of a service. In [9], the authors propose a formal approach to propagate reputation in composite services. They show that their proposed approach ensures fair distribution of reputation. However, much information is required to evaluate the proposed equations. They assume that they have access to this information, and if it is not available, then, past consumers or a community can be asked to provide it. But, again, this assumes a trust among the consumers.

Most existing service registries and repositories are based on UDDI, ebXML, or a mix of both: *Centrasite* is a UDDI service registry that is limited to the Web Services inside a single organization [12]. *FreebXML* is an ebXML service registry [13]. Sun's service registry is based on ebXML 3.0 with added support for UDDI 3.0 [14]. IBM's WebSphere Registry and repository mainly manages ser-

vices' metadata that is gathered from all available resources, such as UDDI registries [15]. Most of these registries have service providers as a single source of information about the considered services.

III. A Diamond SOA Operational Model

To expand the traditional, passive role of service brokers (registries), we extend the traditional SOA operational model (depicted in Fig. 1) by adding a fourth role: the *Service Invocation Proxy*, as shown in Fig. 2. Introducing this functionality as a separate role has the advantage of *low impact* on running systems. It is also appropriate to the situation where service consumers do not have the authority to change the registries they use to force them to add this functionality.

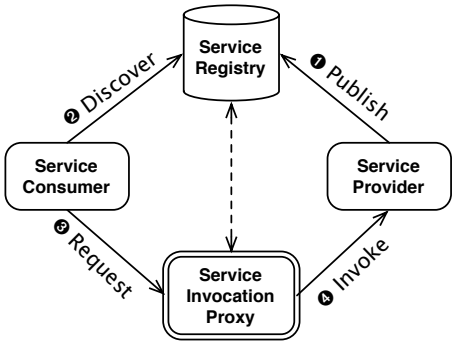

Figure 2: Extended Diamond SOA Operational Model

We call the extended operational model the *Diamond SOA Operational Model*. The main goal of the *Service Invocation Proxy (SIP)* in the diamond model is to enrich service descriptions, which are essential in several aspects in SOC, e.g., service discovery and selection, assessing service quality, etc. The SIP accepts SOAP requests from service consumers and passes them to the corresponding service providers. SOAP responses are then returned to the corresponding service consumers by the SIP. This model allows the SIP to analyze service invocations. Analyzing service invocations helps getting further information about each service, how it is being used, in which context and in conjunction with which other services it is being invoked.

Rich service descriptions can be used to enhance service discovery and selection as the involved parties know more about the considered services. With rich service descriptions, more non-functional properties (NFP) can be assessed. In our model, NFP values become more trustworthy, because they are neither claimed by services providers, nor supplied by service consumers, but are evaluated on real service invocations made by multiple service consumers.

We assume for now that the SIP has write-privileges on the considered service registry. If this not the case, an alternative architecture is introduced in Section V, where the generated and gathered metadata is stored in a separate *Invocation Registry*. In our approach, SIPs are implemented as "Universal Web Services" as we show in this section. Before introducing further details about SIPs, we introduce first a use-case where using a SIP reduces cost and resource consumption.

A. Use-case: Health Insurance Company

A health insurance company typically has several branch offices in different cities. A set of common daily tasks are identified as services and provided by its IT department in the form of Web Services. This set of Web Services is managed by a centralized internal service registry that is used by all branch offices. Furthermore, some tasks are usually out-sourced to external service providers, such as address validation and normalization, e.g., the `GlobalAddressVerfication` Web Service of StrikeIron http://ws.strikeiron.com/GlobalAddressVerification5?WSDL. An external service registry is used to find those external Web Services.

This setup is shown in Fig. 3. The left side shows the internal environment of the company with several branch offices, e.g., Berlin, Cottbus, etc., and one (or more) internal service registries *SR1*. The right side represents the external environment with several service providers *SP1 – SP3* and one (or more) service registries *SR2*.

Among the enormous number of daily tasks (business process instances) conducted in such a company – e.g., creating a new health insurance contract for a new customer – several sub-tasks (typically service calls) are repetitive and identical. A SIP can identify such repetitive service calls and return their cached results (from previous similar service calls) to the corresponding requesters (without invoking the services). This caching can help reduce internal network traffic and load on the internal servers and thus reduce resources consumption and increase response-time.

Out-sourcing some tasks to external service providers is not cheap. Controlling the number of external service calls is the key to choose the most suitable price model (payment strategy) for the company. Choosing subscription, usage-based, or a mixed payment strategy [16] depends on the total number of service calls that *includes* repetitive (identical) service calls. Identifying those repetitive service calls can impact the total price and payment strategy as only *unique* service calls count. Again, a cache at the SIP can help.

Health insurance companies try to keep their records up to date, especially, their records of diseases and recent methodologies to handle them. Such updates can be provided by several service providers. In the traditional SOA operational model, each branch office in the company issues separate requests to the corresponding service providers. In our example, this would amount to twelve service calls [4 (branch offices) $\times$ 3 (service providers)]. Using our diamond model reduces this number, because analyzing the first few

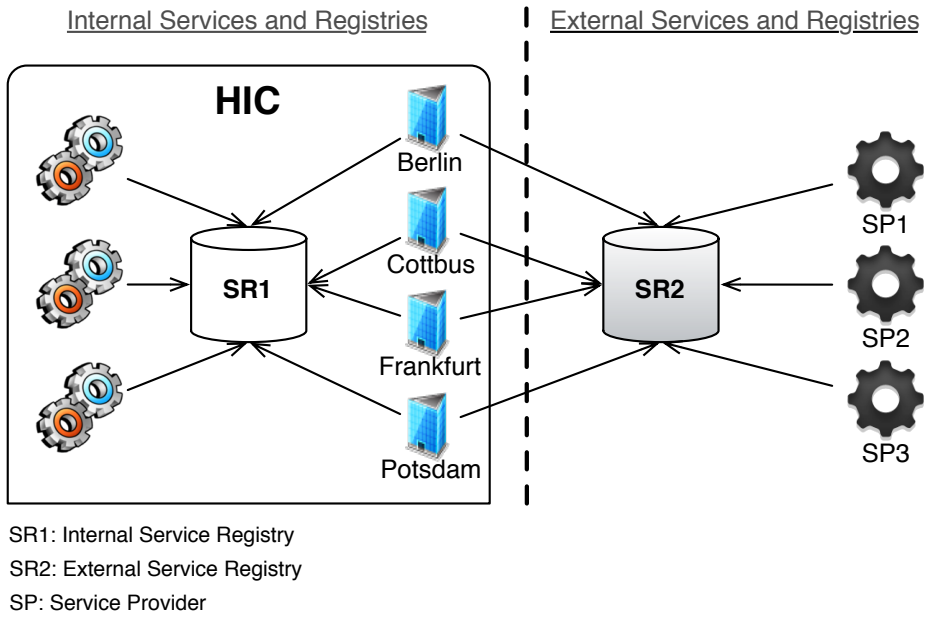

Figure 3: A health insurance company with four branch offices, an internal service registry (SR1), an external service registry (SR2), and providers (SP1 – SP3)

service calls can enable us to identify similar future requests. In our example, only three service calls are forwarded to their corresponding service providers and their cached results are provided directly to other service consumers for further similar requests.

In today's marketplaces, there are several Web Services provided by different providers to achieve the same functionality. In our example, S1 and S3 provide the same functionality. S1 can have a higher priority over S3 because of a business collaboration between S1 and HIC. However, if S1 takes much time to respond to HIC requests, then, such requests can be forwarded automatically to S3 that achieves the same task. Detecting this equivalence or similarity between the functionalities of these services can be achieved by SIPs. Moreover, analyzing the responses that a SIP gets from two Web Services for the same request can help it measure some non-functional properties of these services, such as completeness of results.

In this use-case *no privacy issues* are involved, because all branch offices belong to the same organization – the health insurance company.

B. Service Invocation Proxy (SIP)

A **Service Invocation Proxy** is a role in the extended SOA operational model and works as a mediator between service consumers and service providers. It can be implemented in different ways, e.g., as an agent, as a Web Service, etc. In this paper, we propose a Universal Web Service (UWS) to implement SIPs. A **Universal Web Service** is a Web Service that accepts *any* SOAP/HTTP request to invoke a Web Service, in turn invokes that Web Service, and returns the output to the original service consumer. A Universal Web Service acts as a proxy between service consumers and service providers.

SIPs play their basic role whenever a service consumer issues a request to invoke a Web Service. Traditionally, this request is passed directly to the corresponding service provider and the result is passed directly back to the service consumer. Using our diamond model, this request is sent to the SIP, which in turn calls the service and returns the result to the service consumer. Having both service call requests and responses, SIPs can analyze them and learn more about the *service*, e.g., the semantics of its inputs, the *service provider*, e.g., reputation, and the *service consumer*, e.g., peak periods.

This knowledge can be used by SIPs to predict future behavior of the involved parties. For instance, if a concrete business pattern has been identified that involves calling a set of services in a specific order, the SIP can pre-call some of these services as soon as the pattern is recognized. This functionality can help meet the increasing complexity challenge explained in Sec I.

Enrich service descriptions: One of the main challenges – highlighted in Sec. I – is the lack of adequate descriptions to enable service discovery and selection. To handle this obstacle, we propose using SIPs to enrich service descriptions by analyzing service invocations. Several things can be extracted from such analysis: identifying categories of service consumers, tagging Web Services, and resolving ambiguity in terms used to describe services.

Categorize service consumers: Another application is the classification of service consumers, who use a specific Web Service. For instance, a car rental Web Service that is used mainly by business people (quality first) vs. another one used mainly by families (cost first). Such categorization can be used to provide personalized ranking of result lists during service discovery based on the category of the service consumer. Common terms used during the communication with a service, e.g., inputs, outputs, etc., can be attached to that service as tags.

Disambiguate fuzzy terms: Resolving ambiguity in terms used to describe services requires context knowledge. However, typically, a brief (or no) description is released by service provider about each service. This limited context information cannot help resolve ambiguity in any ambiguous term used in a service description. For example, a term *bank* can refer to the financial institution or to river's side. SIPs can help resolve this ambiguity by analyzing service invocations, e.g., the term *bank* describing a Web Service that is used usually with another credit card service refers most probably to the financial institution not to the side of a river.

Measure the similarity between service calls: Analyzing service calls can also help SIPs identify similar service calls. Identifying similar service calls enables the caching of results and providing them in future similar cases without re-invoking the service. Caching reduces response-time and network traffic. A cached copy of the result is provided directly by SIPs without making an actual service call. Reducing the total number of service calls also reduces resource consumption and the total cost of calling these services, as shown in Section III-A.

C. SIP: Architectural View

We mentioned earlier that we implement the SIP through a Universal Web Service (UWS). Figure 4 illustrates the role of the UWS. A UWS accepts any SOAP request (to call a Web Service), and applies a set of *if-then-else* rules to handle the received request. The set of rules checks whether the request is new or is similar to a previous known and valid (according to its timestamp) SOAP request. New SOAP requests are forwarded quickly to their corresponding service providers. Otherwise a cached response might be returned to the service consumer. Figure 4 shows three requests to invoke Service *A*, and two requests to call Service *B*. Only two *A* requests were forwarded to the Service *A* because the remaining request is similar to a previous one. *B* requests are also similar, therefore, only one request is forwarded to Service *B*.

Figure 5 shows how SIPs fit into the SOA operational model. A service consumer issues a SOAP request (Req^1) to call a Web Service provided by a service consumer. Req^1 is passed through the UWS, which sends Req^2 to the intended service provider. Res^1 is then provided by

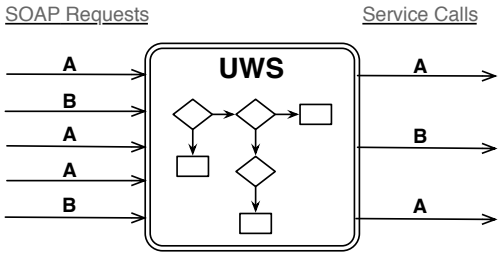

Figure 4: A Universal Web Service as a Service Invocation Proxy

the service provider to the UWS, which sends Res^2 to the service consumer. Alternatively, SOAP responses can be sent directly to the service consumer. The figure presents several architectural decisions:

- Should Req^1 and Req^2 be the same or should Req^1 be a nested version of Req^2?
- Should Res always be passed through the UWS or should this be optional, due to privacy and confidentiality issues, for example?

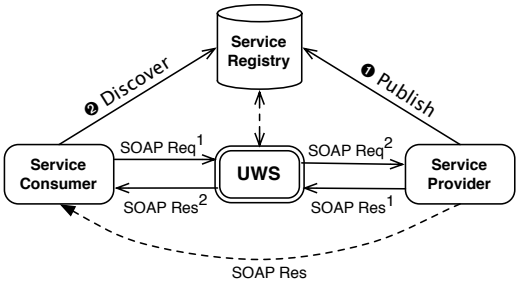

Figure 5: An architectural view of a SIP

In our implementation, we decided to let Req^1 and Req^2 be the same without nesting SOAP requests. By nesting SOAP requests, we mean submitting Req^2 as a parameter or an attachment in Req^1. Nesting SOAP requests results in much redundancy and increases network traffic. Passing SOAP responses through the SIP is optional to give service consumers more flexibility. That is, the SIP can be avoided if need be through configuration of the corresponding software infrastructure.

IV. Implementation Architecture

Java has been a popular framework to develop and run standard and enterprise applications. In this paper we assume a Java-based architecture. Enterprise applications run inside an application or web server, such as Apache Tomcat. Additional libraries or frameworks are required to run Web Service applications, such as Axis2 [17]. Axis2 has been a popular SOAP engine and framework for constructing SOAP processors, such as clients, servers, and gateways, etc. Furthermore, to smooth the process of creating, deploying,

and consuming Web Services, several tools and libraries have been developed, such as JAX-WS. JAX-WS 2.0 is a popular programming model that simplifies the development of Web Service applications and clients[18]. JAX-WS has become a core part of J2SE 6.

We have tested this implementation architecture for several public Web Services. Thorough evaluation of this approach are part of our next steps. Our implementation of a SIP comprises two libraries: JAX-WSD and Axis2D [2]. JAX-WSD is our extension of JAX-WS and is used on the consumer's side. On the other hand, UWS uses our extension of Apache Axis2, coined Axis2D. The next subsections give further details about both libraries.

A. From JAX-WS to JAX-WSD

The implementation of the consumer's part of SIPs should satisfy three basic design requirements:

- **Low impact**: Employing a SIP in a running system should not introduce changes in its implementation, i.e., there is no need to rewrite code.
- **Opt-out**: It should be easily possible to use the traditional triangular SOA operational model by disabling the SIP according to corporate policies.
- **Configurations**: Fine-grained configuration options should be provided, that enable global settings and application-specific settings.

To achieve our proposed approach, each SOAP request is marked as SIP-enabled and passed to the SIP instead of its intended service endpoint's URL. Service consumers are not required to re-write their existing applications to use a SIP; they can simply replace the JAX-WS API by our JAX-WSD, which provides the same functionality of JAX-WS and additionally supports the SIP architecture. To enable/disable SIP-usage and tune its features, the associated configuration files can be edited accordingly.

To employ our approach in a running system, only the JAX-WSD library is required. No further changes in the system are required. Service providers do not notice SIP interaction. A Service Invocation Proxy is viewed as a service consumer, from the provider's side. This satisfies the first design requirement: *low impact*. Enabling and disabling the SIP (triangular or diamond model) is easily configurable at three levels of configurations, satisfying the second design requirement: *opt-out*. Further settings can also be tuned at these levels of configurations, as we explain next. This satisfies the third design requirement: *configurations*.

B. JAX-WS or JAX-WSD

The deployment of our SIP is easily configurable using configuration files or through API means, where all required parameters are set. Service consumers can choose to turn the entire extension off and retain the traditional functionality of JAX-WS. Enabling the SIP feature requires setting further parameters, such as the UWS URL, the sensitivity of input and output parameters, the interaction mode (request-only, request-response), etc. At runtime, the application can decide which library to use depending on the settings provided in the configuration files or API, as shown in the control flow diagram in Fig. 6.

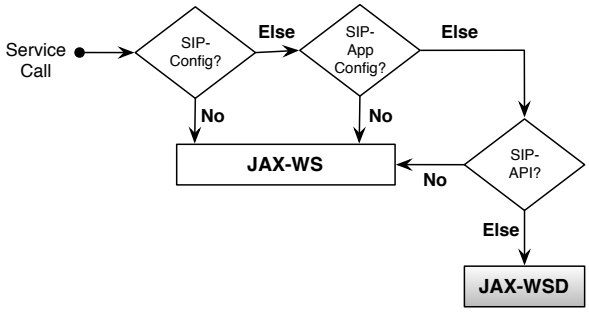

Figure 6: JAX-WS or JAX-WSD control flow

Three levels of configurations are provided: environment-wide, application-specific, and API-level. Disabling the SIP does not require changing the implementation (i.e., code); simply setting its corresponding parameter to false in the environment-wide configuration file (*SIP-config* in Fig. 6) switches to the traditional operational model.

If the configuration at the environment-wide level does not explicitly set required settings, e.g., disable the SIP, next levels of configurations are then used to determine these settings. The next level of configuration is the application-specific (or project-specific) configuration file (*SIP App Config*). This level of configuration gives the flexibility to enable the SIP on one project and disable it on others. The last configuration level is the API means (*SIP-API*), which gives application developers the flexibility to enable/disable the SIP for specific service calls.

C. From Axis2 to Axis2D

From the consumer's point of view, the SIP is a service provider. Therefore, a SOAP engine is required on the SIP side, such as Apache Axis2 [17]. Axis2 dispatches all incoming service requests and delivers each service request to its target Web Service managed by Axis2. This behavior does not satisfy our requirements, because the SIP offers a single Universal Web Service that is used by all consumers to call their various Web Services, e.g., weather forecast, news, stock quote Web Services. Hence, we developed extended version of Axis2, coined Axis2D.

Axis2D performs the traditional role of Axis2 and monitors SOAP requests to call the UWS, which are marked as SIP-enabled requests. Whenever such a SOAP request is received, it is forwarded directly to the UWS. Then, UWS applies a set of *if-then-else* rules to determine how to handle the received request. If the request is classified as *new*, then

[2] D refers to Diamond in both libraries.

its original Web Service is called and the result is cached and returned to its service consumer. Otherwise, i.e., the request is *duplicate*, then a cached response is returned directly to the service consumer. Cached copies of SOAP responses are time-stamped to control their validity. In addition, UWS can also capture and store any metadata pertinent to the request, such as requester, timestamp, etc.

V. An Advanced SIP Setting

The diamond model introduced in Section III assumes that the organization owns the service registry. Owning the service registry enables the SIP to write the metadata it generates back into the registry and therefore enrich service descriptions. In the health insurer use-case, this is true for the internal service registry (SR1) in Fig. 3. However, this is not the case with external service registries that are not owned by the organization. For example, the health insurance company cannot write the metadata generated by its SIP in the external service registry (SR2) in Fig. 3. To handle this case, an internal alternative registry is introduced to hold the generated metadata. This internal registry is called *Invocation Registry*. The role of the Invocation Registry is illustrated in Fig. 7.

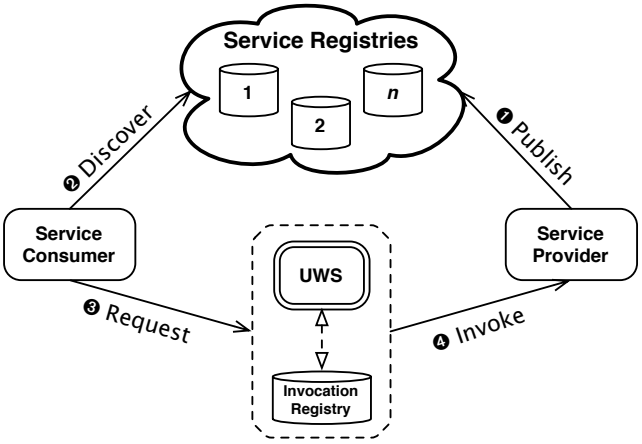

Figure 7: An advanced setting for using a SIP without owning the service registry

In large enterprises, the number of outgoing service calls is enormous. To handle this enormous number of service calls, several Universal Web Services (UWS) can play the role of Service Invocation Proxy instead of using only one UWS. This setting is also depicted in Fig. 7. Service requests can be partitioned by the geographic location of service consumers, service providers, or both. In the health insurer use-case, the SIP can use a UWS to handle all service requests from branch office Berlin, and another UWS to handle requests from branch office Frankfurt, etc. Alternatively, the SIP can use a UWS to handle all service requests to call services provided by German providers, and another UWS to handle other requests to call services of providers from USA, etc.

VI. Summary and Outlook

Several limitations have been identified in the traditional triangular SOA operational model. One of these limitations is the passive role of the service registry. To enable service registries to play an active role, further information about the considered Web Services is required. Several approaches have been proposed to gather this required information, e.g., extending service consumers' clients, asking service providers to give required information, or ask other service consumers to provide this information. However, these proposed approaches either require changing consumer's running systems or involve trust issues.

In this paper, we introduce an automatic way of gathering this information by extending the triangular SOA operational model with a fourth role, the Service Invocation Proxy (SIP). A SIP works as a mediator between service consumers and service providers. It invokes the required services by service consumers and returns the result. Using SIPs helps enrich service descriptions with metadata generated from analyzing services invocations, such as invocation statistics or service quality measures. Additional benefits of SIPs include caching, which is reflected in reduced response time and cost (especially when usage-based pricing strategies are used).

A SIP can be implemented in various ways, e.g., as an agent, a Web Service, etc. In our approach we implement the SIP through a Universal Web Service (UWS). A proof-of-concept of SIPs has been implemented which has two sets of libraries: JAX-WSD and Axis2D. JAX-WSD is used on the consumer's side and Axis2D is used by the UWS. This prototype has been tested on different types of Web Services; internal and external ones, parameterized and non-parameterized ones. Our future work includes further thorough evaluation of SIPs to measure the effect of caching, introduced delay, saved cost, etc.

A health insurance company use-case highlighted the potential usefulness of our approach, where all information is kept inside the boundaries of a single organization. Extending this approach to multiple organizations environments require concrete privacy and information security policies and mechanisms. Such policies are part of our future work as well. The case of multiple organizations environments raises several interesting research opportunities, such as identifying business patterns and learning which Web Services are used in conjunction with which others and when. Finally, we currently considered only XML-based Web Services (SOAP). One of our main extensions is to apply this approach to REST-ful Web Services.

REFERENCES

[1] L. Zhang, J. Zhang, and H. Cai, *Services Computing*. Springer, 2007.

[2] Sun Microsystems, Inc., "Effective SOA Deployment Using an SOA Registry Repository," http://www.sun.com/products/soa/registry/soa_registry_wp.pdf, 2005, white paper.

[3] A. Silvius, B. de Waal, and J. Smit, "Business and IT Alignment; Answers and Remaining Questions," in *PACIS 2009 Proceedings*, 2009.

[4] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth, "Enhancing Web Services Description and Discovery to Facilitate Composition," in *SWSWPC*, 2004, pp. 55–68.

[5] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards recovering the broken SOA triangle: a software engineering perspective," *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, 2007.

[6] M. Treiber and S. Dustdar, "Active Web Service Registries," *IEEE Internet Computing*, 2007.

[7] D. Kuropka, P. Tröger, and S. Staab, *Semantic Service Provisioning*. Springer, 2008.

[8] E. Maximilien and M. Singh, "Conceptual model of web service reputation," *SIGMOD Record*, vol. 31, no. 4, Dec 2002.

[9] S. Nepal, Z. Malik, and A. Bouguettaya, "Reputation Propagation in Composite Services," *Proceedings of the 2009 IEEE International Conference on Web Services*, 2009.

[10] A. A. F. Brandão, L. Vercouter, S. Casare, and J. Sichman, "Exchanging reputation values among heterogeneous agent reputation models: an experience on ART testbed," in *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2007, pp. 1–3.

[11] S. Ran, "A model for web services discovery with QoS," *SIGecom Exch.*, vol. 4, no. 1, pp. 1–10, 2003.

[12] "Centrasite Community," http://www.centrasite.org.

[13] "ebXML Registry-Repository," http://ebxmlrr.sourceforge.net.

[14] SUN, "SUN's Service Registry," http://www.sun.com/products/soa/registry.

[15] IBM, "WebSphere Service Registry and Repository," www.ibm.com/software/integration/wsrr.

[16] O. Guenther, G. Tamm, and F. Leymann, "Pricing Web Services," *International Journal of Business Process Integration and Management*, 2007.

[17] Apache, "Axis User's Guide," 2007.

[18] A. Gupta and D. Kohlert, "The Java API for XML-Based Web Services (JAX-WS) 2.1 ," 2007.