DYNAMIC PROVISIONING AND MONITORING OF STATEFUL SERVICES

Peter Tröger, Harald Meyer
Hasso-Plattner-Institute at University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

Ingo Melzer, Marcus Flehmig
DaimlerChrysler Research & Technology, P.O. Box 2360, 89013 Ulm, Germany

Keywords: Adaptive Services Grid, ASG, Service-Oriented Architectures, Service-Oriented Computing, Stateful Services, Web Service Resource Framework, WSRF, Web Services, J2EE.

Abstract: While tools for service-oriented architectures promise a seamless combination of stateless basic services to new applications, reality looks different. Real-world services are wrapping stateful behavior using application-specific concepts, the monitoring is only available through vendor-specific interfaces, and service installations are bound to particular execution hosts.
We present the *ASG Services Infrastructure* (SI) architecture as our practical solution to these real-world service integration issues. Our framework is based on established Web service standards and supports the dynamic hosting and monitoring of heterogeneous and stateful service implementations.

1 INTRODUCTION

Nowadays, real world service landscapes consist of heterogeneous services and legacy systems. Higher-level service functionality, like the automated and adaptive creation of service compositions, abstracts from these realities and assumes an idealized homogeneus Web service landscape. These approaches and tools expect the integration and monitoring of existing heterogeneous services in a unified manner, regardless of their technical varieties or proprietary state semantics. Services need to be observable according to their resource usage, performance metrics and service-specific quality characteristics.

The European research project *Adaptive Services Grid* (ASG) works on an open development platform for the automated and adaptive creation of complex service workflows, based on semantic service descriptions. In the context of this paper, we give an overview about our *Services Infrastructure* (SI) architecture, which was defined, implemented and operated as part of ASG during the last 18 months. SI provides a uniform and standards-based access to *stateful service instances* on dynamically allocated execution resources. This includes functionalities for deployment, instantiation, invocation, and monitoring of internal or proxy services. Our current SI implementation is based on a combination of *commercial-of-the-shelf* (COTS) middleware products with established standards from both the Web and the grid service community. It acts as foundation for semantic service composition in the ASG project, but works also in other service-driven research projects as unification layer for atomic services.

2 ASG FRAMEWORK BASICS

The ASG project, an integrated project of 21 European partners from seven countries, develops an architecture and reference implementation for automated and adaptive discovery, creation, composition, and enactment of services. ASG automatically creates service compositions, based on an user-provided semantic goal definition and requested quality parameters (i.e. response time, costs). It discovers all necessary services based on the semantic specification of their functionality, and combines them to a dynamically enactable service composition. More details are available in according publications (Harald Meyer and Mathias Weske, 2006) and the web[1].

[1] http://www.asg-platform.org/

Tröger P., Meyer H., Melzer I. and Flehmig M. (2007).
DYNAMIC PROVISIONING AND MONITORING OF STATEFUL SERVICES.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 434-438
DOI: 10.5220/0001286904340438

From the viewpoint of the service workflow component in ASG, all service-related functionality such as SLA negotiation, SLA monitoring and service invocation are seen as implementation-independent features. The representation of the service composition in WS-BPEL demands a unified SOAP access to integrated or newly created services, regardless of possible stateful behavior, interface semantics or implementation technology issues. ASG therefore introduced an abstraction layer between the external service landscape and the Adaptive Process Management component – the *Services Infrastructure* component.

2.1 Web Services In Reality

The initial motivation for a 'unification' layer in ASG arose from the project analysis of real-world candidate Web services for use case scenarios. Even though many Web services meanwhile rely on WS-I compliant interfaces, there are still interoperability problems in the daily work with them. Prominent examples are PayPal (proprietary SOAP header for credentials), Directi (enforced proprietary client library), Google (non WS-I compliant RPC-encoded SOAP) or Amazon (proprietary access / session key).

Demands for a language-specific client library are mostly reasoned by non-standardized SOAP header extensions for special purposes, such as session handling or authentication. Complex types in the WSDL structures can prevent the consumption of the Web service interface in particular programming languages. We also experienced several bugs and missing features in frameworks like Apache Axis, JBoss Axis, IBM WebSphere, or Sun WSDP in case of complex WSDL descriptions, which are wrapped by these libraries.

Web services are usually assumed to process incoming messages regardless of earlier calls, other activities in the system or the time of message arrival. This is commonly named as "stateless" service behavior. In contrast, existing real-world services (whether SOAP-based or not) often assume a functional understanding of server-side state on the client side. Server and client form a joint *session* in their interaction, where the servers response depends not only on the input arguments, but also on the current session state. One example is mandatory security authentication, which results in some proprietary user session handling such as with today's payment Web services. Another example is cart management in shopping services. In general, we found proprietary implementations of session identification in the operational interface, in additional SOAP-body data (being the wrong spot for this kind of meta-data), or in transport-protocol dependent methods such as HTTP cookies. The service consumer then has the burden of performing session-related activities, which are specific for each particular kind of service.

While this solution might be feasible for static client applications, dynamic service composition facilities such as ASG are not able to perform specialized state operations for each service. Service composition facilities instead demand a unified and interoperable way of interacting with all atomic services, regardless of their technical varieties or stateful interaction patterns. In addition, with the introduction of service contracting and monitoring capabilities, clients need to have an understanding of the 'contract partner instance' on the other side.

Based on this analysis, we propose the idea of a dynamic service hosting environment with an unified stateful service interface.

3 THE ASG SERVICES INFRASTRUCTURE

Our *Service Infrastructure (SI)* stack is a thin and scalable abstraction layer upon real-world service environments. We focused on the usage of existing and approved middleware products and standards, in order to keep the possibility of later reuse in industrial environments.

3.1 Service Instance Concept

Our approach extends the idea of stateless Web services with the concept of *service instances* on the infrastructure interface layer. Client applications (such as the workflow engine in ASG) are enforced to perform an explicit service instantiation through a factory operation. The resulting *endpoint reference document* describes all relevant data to perform a particular call to a *logical service instance*.

A logical service instance represents a stateful entity to the client, but does not necessary need to be realized by only one *physical service instance* on a particular execution host in the infrastructure. This slightly extends the idea of standard Web service frameworks, where services are referenced by an endpoint URI for a particular service instance on a particular machine. Instead, all clients communicate with an *coordination layer* (see Figure 1) that routes SOAP requests (specifically the SOAP body) to a matching execution host.

In our model, a logical service instance has queryable state and monitoring properties and a lifetime

model. Client applications such as the workflow engine are now enabled to consider stateful service state data in the service workflow decisions, since all according query operations become automatically part of the particular service interface.

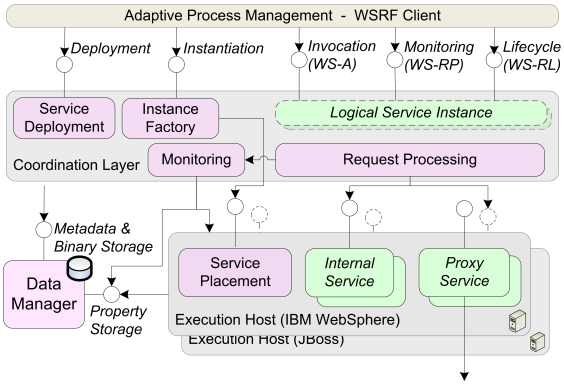

Figure 1: Service Infrastructure Architecture.

In order to realize the free mapping of virtual to physical service instances, the physical service implementations need to externalize their own state with the help of the surrounding infrastructure. For each *logical service instance*, the data storage of the infrastructure manages an own state data representation. The physical service instance on an execution host accesses the state storage with a specific client library, which passes the according logical instance ID from the request SOAP header transparently to the data layer. Therefore, every physical service instance automatically uses the matching state data of the logical instance it serves the request for. The request processing now can implement advanced mechanism like voting or load balancing for logical service instances. Scalability is achieved by using multiple coordination layer instances, where the factory operation balances instantiation requests between them.

In order to keep the consistency of the state data in case of multiple requests, it must be possible for a physical service implementation to demand serialization of incoming requests for the same logical service instance. Otherwise, it could happen that different physical instances concurrently access and modify the state data for the same logical service instance in an illegal way. In our framework, we provide this configuration option as part of the service deployment descriptor.

The classical scenario of purely stateless service implementations can be seen as special case of this idea, where the implementation does not store any state information in the infrastructure. However, since we also relate monitoring information to logical service instances, it is still necessary to have the explicit instance creation step with the factory operation.

A service instance concept automatically leads to the need for lifetime management. Logical service instances must be destroyed at some point in time, either implicitly or explicitly. Another aspect is the handling of fault situations, where the status of the stateful instance must be well-known to the client for an appropriate reaction.

With a classical stateless execution model, the profiling of multiple service operation calls over a time period cannot distinguish if a measured behavior arises from the service implementation itself, from some delay in the setup procedures, or from the execution environment it runs on.

With the logical instance approach, the client (such as the dynamic service profiling and negotiation components in ASG) can now differentiate between behavioral aspects observed for a service instance on a particular resource, the logical service instance in general, or the service implementation. In order to classify profiling results reasoned by some change in the resource binding, our monitoring data model supports the abstract identification of the resource which served the current / last request.

As another aspect, the service instance factory approach allows the easy integration of service negotiation capabilities, which are researched by different partners in the ASG project. With the concept of explicit service instantiation, each client negotiates service aspects with their own logical service instance. Due to the fact that in our concept each operation call anyway must refer to its service instance, the negotiated behavioral aspects are referenced automatically by each service call – and remain on the server side.

3.2 Service Properties Concept

A unified provisioning of state and monitoring data enables service composition facilities, like in the ASG architecture, to integrate all information seamlessly in their dynamic decision procedures during workflow enactment. One example for these *service properties* are resource-bounded monitoring values, such as response time, throughput or jitter. Even though many toolkits, information models and analysis environments already provide such data in different ways, there is still some conceptional flaw in the binding of monitoring properties to service interactions. By using the concept of service instance properties, it is now possible to relate monitoring data directly to a logical service instance endpoint.

Beside *resource-related* information, a service property can also provide *implementation-specific*

properties such as state information, or q*uality-related* information for dynamic SLA negotiation facilities. The validity scope of the property declares another categorization scheme. A service property can be valid either in the *service call scope*, *service instance scope*, or *service implementation scope*.

Properties from the *service call scope* are only valid during and / or after a service operation call, until the next service operation call happens[2].

Properties from the *service instance scope* are valid during the whole lifetime of a logical service instance. They usually represent average values of call scope values, as well as average values for continuous resource-driven measurements. Implementations can provide cross-operation information, like details about the interconnection to a wrapped external service.

Properties from the *service implementation scope* are valid as long as the service implementation remains deployed in the SI stack. This allows service implementations to persist and offer relevant information across different service instantiations. This includes configuration data, which is passed during the deployment process.

Our unified ASG information model for monitoring data relates such resource, service and state information in one type system. It is derived from existing standards in this area, like the DMTF Common Information Model (CIM) or the OASIS Web Service Quality Model (WSQM). The model relies on a combination of approved standards with standardized metrics from the JVM and application server monitoring interfaces. Details of this information model are out-of-scope for this paper.

3.3 Dynamic Service Placement

Many existing service infrastructures rely on some entry router hardware to provide load balancing and fail-over capabilities. With such an approach, the number of service containers offering the requested service limits the available throughput rate for a service type.

Reasoned by the fact that higher layers in ASG only work with the virtual representation of service instances, physical execution host crashes and overload situations need to be transparently handled by the SI layer. During normal operations, the coordination layer forwards an incoming request to one of the matching physical service instances. This physical instance is ensured to have enough resources available, in order to fulfill the demanded performance goal.

In case of a node failure, static instance bindings must be re-established on another empty execution node. In case of a non-contracted service binding, the load of the failed node is evenly distributed to the remaining execution resources, until no more machines with this service type are available. A scheduling algorithm in the coordination layer, based on the work from (A. Karve and T. Kimbrel and G. Pacifici and M. Spreitzer and M. Steinder and M. Sviridenko and A. Tantawi, 2006), manages the placement with respect to the current request load and earlier placement decisions. It relies on a specific placement Web service, which supports the upload and installation of the service binary. For each supported service implementation format (currently J2EE, servlet, and .NET), the SI provides an implementation of the same remote placement interface.

Since the factory service gives the coordination layer endpoint information to the client, it is possible to operate multiple coordination layer instances in parallel. It is also possible to operate multiple factory instances, in combination with some low-level balancing mechanism (such as round-robin DNS). Only the central state and data storage needs to have a unique view on the available data. Our current implementation relies on reliable standard database technology here, so that in sum we are able to provide a crash-fault tolerant hosting infrastructure.

4 IMPLEMENTATION DETAILS

Facing the problem of unified state representation with Web services, two major standards are available: The *Web Service Resource Framework* (WSRF) (Tim Banks, 2005) and the *WS-Context* (Mark Little and Eric Newcomer and Greg Pavlik, 2006) specification. Both specifications support the standardized representation of stateful entities in a Web service interface, and meanwhile both are approved OASIS specifications.

Since ASG manages cross-service state anyway in the composition engine, our SI framework concentrates on state issues of single service instances, where standardized state querying and lifecycle management becomes a major issue. With respect to this problem domain, we identified the WSRF as optimal approach.

The family of WSRF specifications is supported by companies such as IBM and Microsoft, the Globus

[2]We assume here that service operation calls for one physical instance are happening in a serialized manner. This is reasonable since reentrant service implementations usually also perform an internal synchronization of parallel calls, in order to protect their access to underlying data sources.

Alliance, HP, Fujitsu and CA. In our SI architecture, these standards form the coordination layer interface available for client applications.

The coordination layer is implemented as one J2EE application, containing multiple servlets for the different functional parts. The WSRF interfaces are realized as JAX-RPC handlers, based on the Apache WSRF implementation. The underlying service execution hosts can be different middleware technologies (such as J2EE, .NET or OSGI).

So far, our prototype integrates JBoss 4, IBM WebSphere 6, Apache Geronimo 1.0 and .NET installations on heterogeneous distributed machines in the testbed. All SI parts where developed with the JBoss 4.0 application server.

5 CONCLUSION

We presented a short overview of the ASG Services Infrastructure, which tries to fulfill the 'SOA promise' of seamless legacy integration and on-demand resource usage. Based on practical demands from the ASG project, we implemented the consistent representation of service state, lifecycle and monitoring data for higher layers. Our approach is based on the WSRF Web service specifications and standard middleware products. The framework can act as integration platform for existing stateful legacy services, as well as hosting platform for newly created service implementations.

Our future research efforts will concentrate on *service-level agreement* (SLA) fulfillment strategies, based on resource partitioning technologies and basic concepts from service capacity planning research (Daniel A. Menasce and Virgilio A.F. Almeida, 2002). We started to extend the dynamic service installation capabilities with the usage of grid computing resources, based on standardized Grid API's and environments. In addition, we are observing existing solutions for migration of stateful entities in both J2EE and .NET (Andreas Rasche and Andreas Polze, 2003), in order to relax the demand of state externalization in the service implementations.

REFERENCES

A. Karve and T. Kimbrel and G. Pacifici and M. Spreitzer and M. Steinder and M. Sviridenko and A. Tantawi (2006). Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 595–604, New York, NY, USA. ACM Press.

Andreas Rasche and Andreas Polze (2003). Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 164–171, Hakodate, Japan.

Daniel A. Menasce and Virgilio A.F. Almeida (2002). *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall.

Harald Meyer and Mathias Weske (2006). Automated Service Composition using Heuristic Search. In *Proceedings of the Fourth International Conference on Business Process Management*.

Mark Little and Eric Newcomer and Greg Pavlik (2006). Web Services Context Specification 1.0 (WS-Context).

Tim Banks (2005). Web Services Resource Framework (WSRF) - Primer, Committee Draft 01. http://docs.oasis-open.org/wsrf/.