ServiceGlobe: Flexible and Reliable Web Services on the Internet*

Markus Keidl Stefan Seltzsam Alfons Kemper

Universität Passau, 94030 Passau, Germany
<lastname>@db.fmi.uni-passau.de

1. INTRODUCTION

Web services are a new technology for the development of distributed applications on the Internet. By a Web service (also called service), we understand an autonomous software component that is uniquely identified by a URI and that can be accessed by using standard Internet standards like XML, SOAP, or HTTP [4].

Our objective in this work is to present new techniques for Web service execution and deployment which can be integrated into existing platforms: Dynamic service selection offers Web services the possibility to select and invoke services at runtime based on a technical specification of the desired service, thereby providing a layer of abstraction from the actual services. A generic, modular dispatcher service addresses load balancing and high availability of services. This dispatcher implements automatic service replication to be able to install new service instances on idle hosts.

We present these techniques within the scope of the ServiceGlobe system [1, 3], an open Web Service platform. ServiceGlobe supports mobile code, i.e., services can be distributed and instantiated during runtime on demand at arbitrary Internet servers participating in the ServiceGlobe federation. Also, it offers all standard functionality of a service platform like SOAP/XML communication, a transaction system, and a security system [5]. These areas are already well covered by existing technologies and are not the focus of this work.

Due to its potential of changing the Internet to a platform of application collaboration and integration, Web service technology gains more and more attention in research and industry; initiatives like HP Web Services Platform, Microsoft .NET, and Sun ONE show this development. All these frameworks share the opinion that services are important for easy application collaboration and integration and they try to provide appropriate tools and a complete infrastructure for implementing and executing Web services.

2. ARCHITECTURE OF SERVICEGLOBE

The ServiceGlobe system is a distributed and extensible service platform. It is completely implemented in Java and based on standards like XML, SOAP, UDDI, and WSDL. In this section, we present the basic components of the ServiceGlobe system. First of all, we distinguish between external and internal services.

External services are existing, stationary services, currently deployed on the Internet. Such services have arbitrary interfaces for their invocation. To integrate these services independent of their actual invocation interface, e.g., SOAP or RPC, we use *adaptors* to transpose internal requests to the external interface and vice versa. Thus, external services can be used like internal services.

Internal services are native ServiceGlobe services which are implemented in Java using the API provided by ServiceGlobe. ServiceGlobe services use SOAP to communicate with other services; they receive a single XML document as input and generate a single XML document as output. There are two kinds of internal services, namely *dynamic* services and *static* services. Static services are location-*dependent*, i.e., they cannot be executed dynamically on arbitrary ServiceGlobe servers, because they, e.g., require access to certain local resources like a DBMS or the file system. In contrast, dynamic services are location-*independent*. They are stateless, i.e., their internal state is discarded after a request was processed, and do not require special resources or permissions. Therefore, they can be executed on arbitrary ServiceGlobe servers.

Internal services are executed on *service hosts* which are standard Internet servers additionally running the ServiceGlobe runtime engine. If internal services have the appropriate permissions, they can also use resources of service hosts, e.g., databases. ServiceGlobe's internal services are mobile code, therefore their executables are loaded on demand from *code repositories* onto service hosts or, more precisely, into the service hosts' runtime engines. A UDDI server is used to find an appropriate code repository storing a certain service.

3. DYNAMIC SERVICE SELECTION

In general, Web services invoke other services by passing the service's URL or access point to the service platform. In contrast, *dynamic service selection* [2] enables Web services to state a technical specification of the services that should be invoked. It is the service platform's task to select suitable Web services utilizing UDDI. In UDDI, every Web service is assigned to a tModel. As a semantic classification, a tModel provides a classification of a service's functionality and a formal description of its interfaces. So, instead of explicitly calling an actual Web service, it is possible to "call" a tModel. Thus, one defines the functionality of the services to invoke rather than actual implementations. Additionally to the tModel, different kinds of constraints can be passed over to the service platform to influence dynamic service selection.

3.1 Constraints

Constraints enable Web services to control the *selection* of Web services based on the their metadata, the *invocation* of the selected Web services, and the *processing of replies* received from the invoked Web services. There are five different types of constraints: *Metadata constraints* are basically XPath queries. They are applied as a filter to the metadata of all services that are returned by UDDI when the service platform requests services assigned to a given tModel. *Location constraints* enable Web services to select services by the location of their execution host. Additionally, dynamic services can be restricted to be instantiated only on certain hosts. *Mode constraints* determine the number of Web services that should be invoked when calling a tModel. The number can be specified as an absolute value or as a percentage. *Reply constraints* are applied to the replies of Web services. Every reply that does not satisfy all reply constraints is discarded. There are two kinds of reply constraints. Property constraints verify special properties of a reply provided either by the service platform or included into the reply by the invoked Web service. Selection constraints are XPath queries which are applied to a service's reply, including its SOAP parts. *Result constraints* refer to the set of all received replies. There are

*This research is done in cooperation with the Advanced Infrastructure Program (AIP) group of SAP.

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM xxx.

two kinds of result constraints. Timeout constraints allow setting a maximal waiting time for replies. First-n constraints allow setting the number of replies after which a tModel call can be ended.

To influence dynamic service selection in a complex way constraints can be combined. For this purpose, the operators AND and OR are provided for combining constraints subjunctively and disjunctively, respectively (NOT is currently being implemented).

3.2 Evaluation of Constraints

Due to space limitations, we can only present a short summary of how constraints are evaluated when processing a tModel call.

Firstly, constraints from different sources are combined conjunctively into one combined constraint. Possible sources are, e.g., the Web service itself (the constraints have been compiled into its code, then) or the Web service's context. Then, this combined constraint is transformed into its disjunctive normal form and conflicts—which can arise when combining constraints—are resolved. Next, UDDI is queried for all information about services assigned to the given tModel and metadata as well as location constraints are applied to the metadata. After this, Web services are invoked in parallel as specified by mode constraints. Whenever a reply is received, all relevant reply constraints are applied to it. Also, result constraints must be checked to determine if the invocation phase must be ended. If so, all received replies are returned to the calling Web service and all outstanding requests are aborted. After that, dynamic service selection for a given tModel is finished.

4. MODULAR DISPATCHER SERVICE

For large-scale, mission-critical applications, such as an enterprise resource planning system like SAP, a single service host is not sufficient to provide low response times and high availability. Downtime can generate high costs. Therefore, it is necessary to run several instances of a service on multiple service hosts for fault tolerance reasons and a load balancing component to avoid load skew.

We propose a generic solution to this problem: a modular *dispatcher service* which can act as a proxy for arbitrary services. Using this service, it is possible to enhance (even many already existing) services with load balancing and high availability features, as long as concurrency control mechanisms are used, e.g., by using a database as back-end (as many real-world services do). An additional feature of our dispatcher is called *automatic service replication* and enables the dispatcher to install new instances of static services on demand.

4.1 Architecture of the Dispatcher

Our dispatcher is a software-based layer-7 switch, performing load balancing using a dispatching strategy which can access load information about all relevant resources. In contrast to existing layer-7 switches it is realized as a regular service. Thus, our dispatcher is more flexible, extensible, and seamlessly integrated into the service platform. There are three types of modules available to customize the dispatcher: The *dispatch module* implements the actual dispatching strategy. It can access the load situation of service hosts and other resources for the assignment of requests to service instances. *Advisor Modules* are used to collect data for the dispatcher's view of the load situation of all relevant resources. *Config Modules* are used to generate the configuration for new service instances. The modules can access the load situation history to find, e.g., the database host which was least loaded in the last few days.

To turn an existing service into a highly available and load balanced service, a properly configured dispatcher service must be started. Additionally, some new UDDI data has to be registered and some existing data has to be modified so that all service instances and all service hosts can be found by the dispatcher. A cluster of service hosts can be easily supplemented with new service hosts by registering them at the UDDI repository.

4.2 Automatic Service Replication

If all available service instances of a service are running on heavily loaded service hosts and there are service hosts available having a low workload, the dispatcher can decide to generate a new service instance using automatic service replication. Imagine service hosts A and B are heavily loaded and host C currently has no instance of Service S running. Thus, the dispatcher sends a message to service host C to create a new instance of Service S. The configuration of the new Service S is generated using the appropriate configuration module. If no service hosts having low workload are available, the dispatcher can buffer incoming messages (until the buffer is full) or reject them depending on the configuration of the dispatcher instance and the modules.

4.3 High Availability

Using several instances of a service greatly increases its availability and decreases the average response time. Assuming that the server running the dispatcher itself is highly available, the availability of the entire system depends only on the availability of the service hosts. Even assuming very unreliable service hosts with MTBF $=$ 48h (mean time between failure) and MTTR $=$ 12h (mean time to repair) a pool with 8 members will only be unavailable about 1.5 minutes a year. There are many possible adaptable solutions from the fields of database systems and Web servers including solutions based on redundant hardware, but this is out of the scope of this paper.

5. CONCLUSION

The implementation of ServiceGlobe and the two presented techniques, dynamic service selection and the generic dispatcher service, is finished. A demo of ServiceGlobe and an e-procurement scenario was given at VLDB'02 [3]. Currently, the ServiceGlobe system is installed on a blade server with 160 processors overall (with 2 and 4 processors per server blade, respectively) operated by the Advanced Infrastructure Program group of SAP. Right now, performance evaluations with high-volume business applications are conducted using this blade server and the presented techniques. For the future, we plan to work on caching of SOAP messages and to further investigate context for Web services.

6. REFERENCES

[1] M. Keidl, S. Seltzsam, and A. Kemper. Flexible and Reliable Web Service Execution. In *Proc. of the 1st Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologie*, pages 17–30, 2002.

[2] M. Keidl, S. Seltzsam, C. König, and A. Kemper. Kontext-basierte Personalisierung von Web Services. In *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, 2003. Accepted for Publication.

[3] M. Keidl, S. Seltzsam, K. Stocker, and A. Kemper. ServiceGlobe: Distributing E-Services across the Internet (Demonstration). In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 1047–1050, 2002.

[4] E. Rahm and G. Vossen, editors. *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*. dpunkt-Verlag, 2002.

[5] S. Seltzsam, S. Börzsönyi, and A. Kemper. Security for Distributed E-Service Composition. In *Proc. of the 2nd Intl. Workshop on Technologies for E-Services*, volume 2193 of *Lecture Notes in Computer Science*, pages 147–162, 2001.