ServiceGlobe: Distributing E-Services Across the Internet

Markus Keidl Stefan Seltzsam Konrad Stocker Alfons Kemper

Universität Passau
Fakultät für Mathematik und Informatik
94030 Passau, Germany
⟨*last name*⟩@db.fmi.uni-passau.de

1 Introduction

The next generation of Internet applications is emerging: *e-services*. By an e-service, we understand an autonomous software component that is uniquely identified by an URI and that can be accessed by using standard Internet protocols like XML, SOAP, or HTTP. An e-service may combine several applications that a user needs, such as the different pieces of a supply-chain architecture. For the end-user, however, the entire infrastructure will appear as a single application.

The ServiceGlobe system provides a platform on which e-services (also called *services* or *Web services*) can be implemented, stored, published, discovered, deployed, and dynamically invoked at arbitrary Internet servers participating in the ServiceGlobe federation. While current approaches mainly try to integrate existing services which are already running on dedicated servers, we provide the functionality to specify new, composite services which can be deployed dynamically on arbitrary ServiceGlobe enabled servers/devices. Besides the support of standard functionality of a service platform like SOAP/XML communication, a transaction system, or a security system, the ServiceGlobe platform addresses also various optimization issues like load balancing or network oriented deployment during service execution.

Due to its potential of changing the Internet to a platform of application collaboration and integration, e-service technology gains more and more attention in research and industry; initiatives like HP Web Services Platform [WSP], Sun ONE [Sun], or Microsoft .NET [NET] show this development. Although all of these frameworks share the opinion that services are important for easy application collaboration and integration, they handle the subject from a different point of view. Their approaches are server-centric and their focus is on providing a complete infrastructure to implement these services. Our approach is network-centric, focusing on the distributed execution of services and an elaborate selection and distribution of the services they deploy.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

2 Architecture of ServiceGlobe

The ServiceGlobe system provides a lightweight infrastructure for a distributed, extensible e-service platform. It is completely implemented in Java Release 2. In this section, we present the basic components of this infrastructure. Basically we distinguish two different types of services: external and internal e-services.

External services are existing, stationary services, currently deployed on the Internet, which are not provided by ServiceGlobe itself. Such services may be realized on arbitrary systems on the Internet having arbitrary interfaces for their invocation. Since we want to be able 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. This way, we are also able to access arbitrary applications, e.g., ERP applications. Thus external services can be used like internal services and, from now on, we consider only internal services.

Internal services are native ServiceGlobe e-services. They are implemented in Java using the e-service API provided by the ServiceGlobe system. ServiceGlobe services use SOAP to communicate with other services. Services receive a single XML document as input and generate a single XML document as output. There are two kinds of internal services, namely *dynamic* e-services and *static* e-services. Static services are location-*dependent*, i.e., they cannot be executed dynamically on arbitrary ServiceGlobe servers. Such

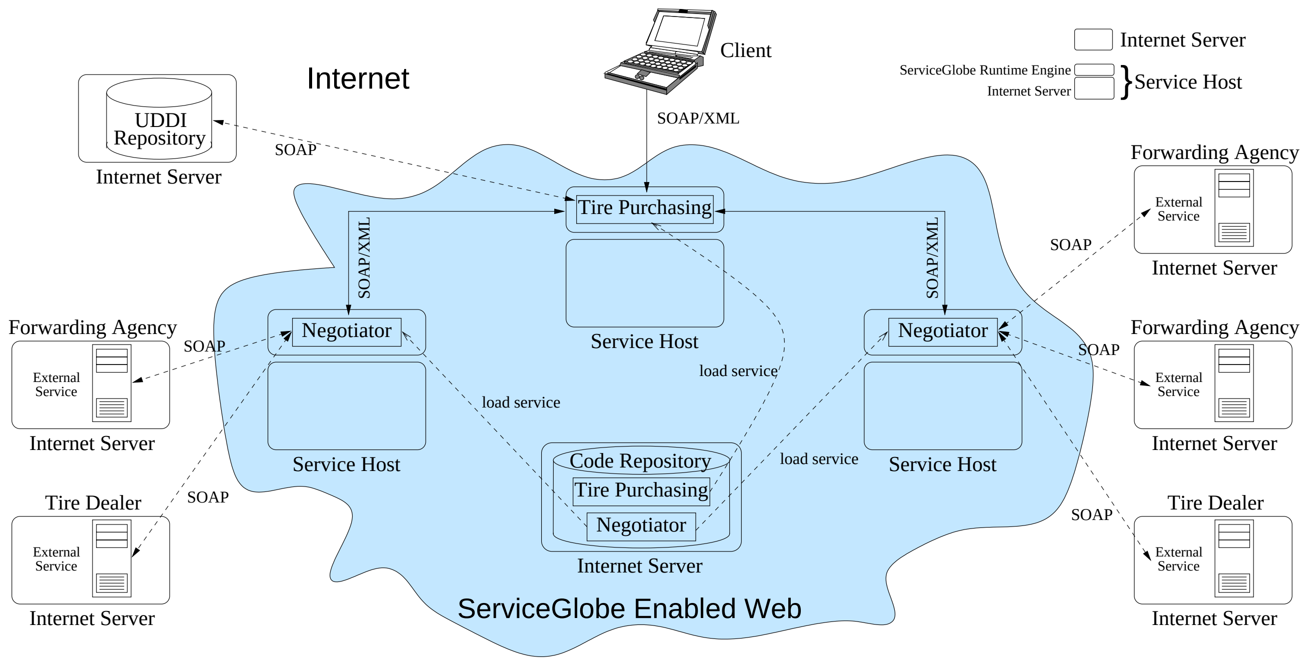

Figure 1: Architecture of a ServiceGlobe Enabled Web

services may require access to certain local resources, e.g., a local DBMS to store data, or require certain permissions, e.g., access to the file system, that are only available on dedicated servers. These restrictions prevent the execution of static services on arbitrary ServiceGlobe servers. In contrast, dynamic services are location-*independent*. They are state-less, i.e., the internal state of such a service is discarded after a request was processed, and do not require special resources or permissions. Therefore, they can be executed on arbitrary ServiceGlobe servers.

There is an orthogonal categorization for internal services: *adaptors*, *simple services*, and *composite services*. We have already defined adaptors. Simple services are internal services not using any other service. Composite services are higher-value services assembled from other internal services. These services are, in this context, called *basis services*, because the composite service is based on them. Note, that a composite service can also be used as a basis service for another 'higher-value' composite service.

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 [UDD00] server is used to find an appropriate code repository storing a certain service.

Figure 1 gives an overview of the basic components of the ServiceGlobe system and their mutual interaction (based on the e-procurement scenario of Section 3): The negotiator services use external services (adaptors are omitted in the figure) and the tire purchasing service uses two dynamic services. At first, a client sends a request (in SOAP) to execute the tire purchasing service to a service host. This service is loaded from a code repository (if not already cached) and instantiated on the service host. The tire purchasing service deploys several basis services (here, two negotiator services) during execution. Therefore, suitable service hosts are located by UDDI requests and the negotiator services are loaded and executed on them on behalf of the tire purchasing service. The negotiator services deploy external tire dealer and forwarding agency services to calculate their results. In the following, we describe the complete process of service composition and execution which is composed of four major steps:

Service Specification: This is the process of coding e-services. The basic method is to use Java and the e-service API provided by ServiceGlobe. A more comfortable way is to use a specialized programming language, e.g., XL [FK01], or a graphical tool to draw a representation (similar to a workflow graph) of the e-service. Both, programs and graphs must be compiled into Java classes using the ServiceGlobe API to obtain services executable on the ServiceGlobe system.

Dynamic Service Selection: In the UDDI information model, every service is assigned to a *tModel*, which is basically a template defining the semantics and interfaces of services implementing this template. Thus, a service is an *implementation* or *instance* of its

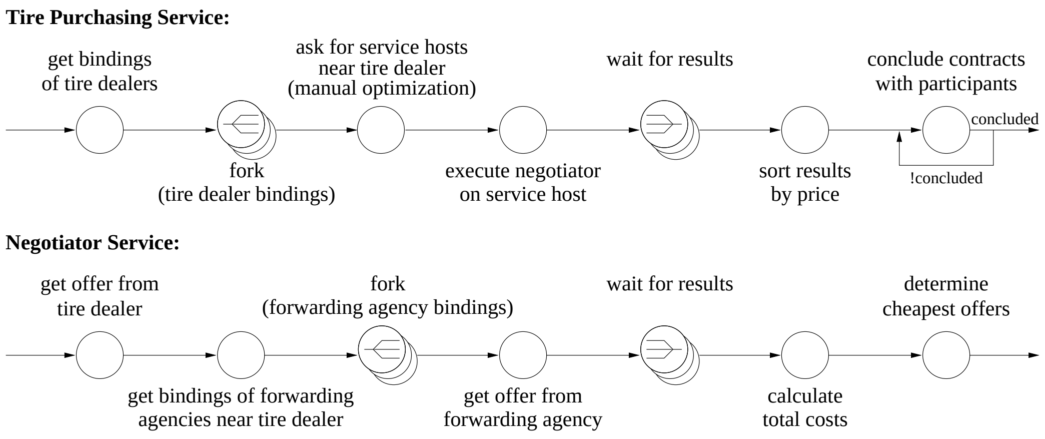
Tire Purchasing Service:
get bindings of tire dealers
fork (tire dealer bindings)
ask for service hosts near tire dealer (manual optimization)
execute negotiator on service host
wait for results
sort results by price
conclude contracts with participants
concluded
!concluded
Negotiator Service:
get offer from tire dealer
get bindings of forwarding agencies near tire dealer
fork (forwarding agency bindings)
get offer from forwarding agency
wait for results
calculate total costs
determine cheapest offers

Figure 2: Graphical Representations of the Services

tModel. In ServiceGlobe, a composite e-service need not call a concrete service, but it is sufficient to specify or "call" a tModel. The implementation of the tModel is chosen later on, during compilation or execution. This process of selecting the implementation for a tModel is called *dynamic service selection*.

Dynamic service selection is not limited to select only one instance for a tModel, it is also possible to select several instances. Since UDDI returns all services, i.e., their bindings that are assigned to a tModel, more than one can be called. Thus, a call to a tModel is substituted by one or more service calls. We distinguish the following modes of calling a tModel:

one: Only one instance out of all tModel instances returned by UDDI is called. The call is (multiply) retried in case of failures, e.g., temporary unavailability of the service. If the failures persist, an alternative e-service is tried.

some: A subset of all services returned by UDDI is called in parallel. The number of services to be called is specified as a parameter. Services, which continue to fail, are replaced with alternative services until the demanded amount of e-services responded successfully or no more services are available.

all: In this case, all tModel instances returned by UDDI are called in parallel. If faults occur, no alternative services can be called, simply because there are no remaining ones.

Quality-of-Service (QoS) constraints can be applied to further refine the set of services that are called, e.g., 'return only the results of the first ten services that reply'.

Service Distribution: The main advantage of dynamic services is their location independence. At runtime, this allows dynamic distribution of such services to arbitrary service hosts, opening a great optimization potential to ServiceGlobe: Several instances of a dynamic service can be executed on different hosts for load balancing and parallelization purposes. Dynamic services can be instantiated on service hosts having the optimal execution environment, e.g., a fast processor, huge memory, or a high-speed network connection to other e-services. Together with runtime service loading this provides a large flexibility in order to consider load balancing or optimization issues. After a service host is registered in the UDDI repository this new service host is instantly incorporated into service execution.

Runtime Service Loading: After service distribution, dynamic services are loaded from code repositories and executed on the chosen service hosts. Thus, the set of available services is not fixed and can be extended at runtime by everyone participating in the ServiceGlobe federation. We implemented a comprehensive runtime security system based on security systems presented in [BKK+01, SBK01] to deal with the security issues of mobile code introduced by runtime service loading. Thus, service hosts are protected against malicious services.

3 Description of the Demo

In this demo, we present the ServiceGlobe system and several of its key concepts using an automobile industry e-procurement scenario. For simplicity, we do not describe a complete e-procurement solution comprising of a marketplace service and automobile industry supplier services. Instead, we concentrate on the example of tire purchasing.

Tire Purchasing Scenario

In this scenario a car manufacturer requires an e-service for the task of purchasing tires and employing a forwarding agency to deliver these tires. In detail, this e-service has to perform the following tasks: First, it has to invite offers from available tire dealers for a given type and quantity of tires. Second, it must invite offers for the delivery of these tires to the car manufacturer. Third, it must calculate the cheapest combined offer for the purchase of tires from a tire dealer and the delivery of these tires by a forwarding agency. At last, the e-service must place purchase orders, based on the cheapest combined offer, at both, the tire dealer and the forwarding agency. If placing of a purchase order fails, the second cheapest combined offer should be tried (and so on).

The new e-service is split into two separate, dynamic services: a tire purchasing service and a negotiator service. This allows an optimized service execution by pushing negotiator services to service hosts close to tire dealers. The negotiator service instances are executed on several service hosts in parallel and thus it is assured that communication with the forwarding agency services is cheap. Using a graphical tool, the resulting services look like the graphs shown in Figure 2.

The *tire purchasing service* is the main service which is called directly by the car manufacturer. First, the service queries UDDI for tire dealer services (using a UDDI tModel). Using this technique, the service is independent of the available tire dealer services at a particular time and the implementation need not be changed to remove outdated or add new tire dealers. For every tire dealer service, we look for a service host close to the location of the tire dealer to minimize communication costs to the tire dealer service as well as to the forwarding agency services later on. Next, we execute a negotiator service on each of these service hosts. For performance reasons, all negotiator services are executed in parallel. Finally, we wait for the results of all negotiators. A timeout handles services that do not respond in time. After collecting the results, the combined offers (tires and delivery) are sorted by price and the tire purchasing service tries to contract a tire dealer and a forwarding agency starting with the cheapest offer. If one of the two participants fails for any reason, the contract conclusion is aborted and the next (more expensive) offer is used.

A *negotiator service* is called with information about which tire dealer it should contact. Its first action is to invite an offer from the given tire dealer. After that, it calls forwarding agency services using an all-mode call. Additionally, a QoS constraint filtering all forwarding agencies geographically close to the tire dealer is used. In our example, it is assumed that forwarding agencies in the neighborhood of tire dealers have the cheapest offers due to short routes. The negotiator invites offers from all selected forwarding agency services in parallel and waits for the results, calculates the overall costs (tires and delivery), and determines the five cheapest offers. These offers are sent back as result.

What Will Be Shown

The demo consists of two applications. First, a front-end to the tire purchasing service allows entering data into a form and executing the service. The front-end also allows viewing all XML documents exchanged between the various services in the execution. Second, a map application depicts all services and their location graphically. It allows to add and remove tire dealer and forwarding agency services at runtime, to view all interactions between the services, and to influence the execution by setting individual timeouts, simulate failures, and so on.

Using these applications we demonstrate some of the major concepts of ServiceGlobe. First, we show where and how dynamic service selection takes place in our e-procurement scenario: Negotiator services contact several forwarding agencies, using dynamic service selection and an additional QoS constraint filtering all forwarding agencies geographically close to the corresponding tire dealer. Second, we demonstrate runtime service loading and service distribution: The tire purchasing service can be instantiated on every service host on the Internet (which we simulate, of course). The negotiator services are pushed close to the location-dependent tire dealer services such that communication overhead is reduced. This leads to load balancing (different service hosts for negotiators), to parallelization (negotiators work in parallel), and to profit from cheaper communication costs (execution close to tire dealers).

References

[BKK$^+$01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal: Special Issue on E-Services*, 10(3):48–71, August 2001.

[FK01] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. *IEEE Data Engineering Bulletin*, 24(2):48–56, 2001.

[NET] Microsoft .NET. `http://www.microsoft.com/net`.

[SBK01] 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 (TES)*, pages 147–162, Rome, Italy, 2001. Springer.

[Sun] Sun Open Net Environment (Sun ONE). `http://www.sun.com/sunone`.

[UDD00] Universal Description, Discovery and Integration (UDDI) Technical White Paper. White Paper, Ariba Inc., IBM Corp., and Microsoft Corp., September 2000. `http://www.uddi.org`.

[WSP] HP Web Services Platform. `http://www.hp.com/go/webservices`.