Flexible and Reliable Web Service Execution

Markus Keidl Stefan Seltzsam Alfons Kemper

Universität Passau, Lehrstuhl für Dialogorientierte Systeme, Fakultät für Mathematik und Informatik, Innstr. 30, 94030 Passau, Germany, E-Mail: ⟨last name⟩@db.fmi.uni-passau.de, URL: http://www.db.fmi.uni-passau.de

Abstract. The next generation of Internet applications is emerging: *Web services*. In this work, we present novel techniques for a flexible and reliable execution and deployment of Web services which can be integrated into current service platforms. Dynamic service selection allows selecting Web service instances during runtime by means of semantic classifications. The selection can be influenced by specifying different constraints. We present a generic approach for load balanced and high available services using automatic service replication. Both approaches, dynamic service selection and automatic service replication, support the development of reliable services. Additionally, our system supports mobile code, i.e., Web services can be distributed and instantiated during runtime on demand. This feature opens a great optimization potential and additionally contributes to reliable service execution because unavailable hosts can be replaced dynamically by available hosts. We present these techniques within the scope of the ServiceGlobe system, an open platform for Web service specification, execution, and deployment.

Keywords: Web service; service platform; mobile code; dynamic service selection; load-balancing; high availability

1 Introduction

The next generation of Internet applications is emerging: *Web services*. By a Web service (also called *service* or *e-service*), we understand an autonomous software component that is uniquely identified by a URI and that can be accessed by using standard Internet protocols like XML, SOAP, or HTTP. A service may combine several applications that a user needs, such as the different pieces of a supply-chain architecture. For a client, however, the entire infrastructure will appear as a single application. 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 [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 have a different focus. They intend to provide a complete infrastructure for implementing services and executing them on dedicated hosts.

Our objective in this work is to present new functionality for Web service execution and deployment which can be integrated into existing service platforms. We propose techniques like dynamic service selection which allows selecting service instances during runtime by means of semantic classifications. This is achieved by providing the ability to call groups of interchangeable services using different modes; e.g., only one service of a group may be called; another one is only called if the first service fails. The selection can be influenced by specifying different kinds of constraints. We also address load balancing and high availability by providing a generic and transparent approach for automatic service replication. Both of these approaches, dynamic service selection and automatic service replication, support the development of flexible and reliable services.

We present these techniques within the scope of the ServiceGlobe system. ServiceGlobe provides a platform on which services can be implemented, stored, published, discovered, and deployed. Additionally, the system supports mobile code, i.e., services can be distributed and instantiated during runtime on demand at arbitrary Internet servers participating in the ServiceGlobe federation. Of course, ServiceGlobe offers all the standard functionality of a service platform like SOAP/XML communication, a transaction system, and a security system [SBK01]. These areas are well covered by existing technologies and are, therefore, not the focus of this work. Also, we assume that appropriate standards will be developed and incorporated into service platforms.

The remainder of this paper is structured as follows: Section 2 presents an e-procurement scenario which will be used as an example throughout this paper. Next, the ServiceGlobe system is introduced in Section 3. Sections 4 and 5 present dynamic service selection and automatic service replication, respectively. Finally, Section 6 gives some related work and Section 7 concludes this paper.

2 Example Scenario

In this paper, we are going to use an e-procurement scenario to explain our novel techniques within the scope of the ServiceGlobe system. For simplicity reasons, we do not describe a general and complete e-procurement solution consisting of a marketplace service and general supplier services. Instead, we concentrate on the example of tire purchasing [KSSK02]. We assume that there are Web services available providing the functionality to buy tires and others that offer services of forwarding agencies. In this scenario a car manufacturer requires an service for the task of purchasing tires and employing a forwarding agency to deliver these tires. In detail, this service has to perform the following tasks: First, it has to invite offers from available tire dealer services 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 service must place purchase orders, based on the cheapest combined offer, at both, the tire dealer and the forwarding agency. If placing a purchase order fails, the second cheapest combined offer should be tried (and so on).

This service is split into two separate services: a tire purchasing service and a negotiator service. A graphical representation of the resulting services is shown in Figure 1.

The *tire purchasing service* is the main service which is called directly by the car manufacturer. First, the service queries a UDDI server [UDD00] for all bindings, i.e., URLs, of tire dealer services. 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, the service looks 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. Service hosts are hosts connected to the Internet which are running the ServiceGlobe runtime

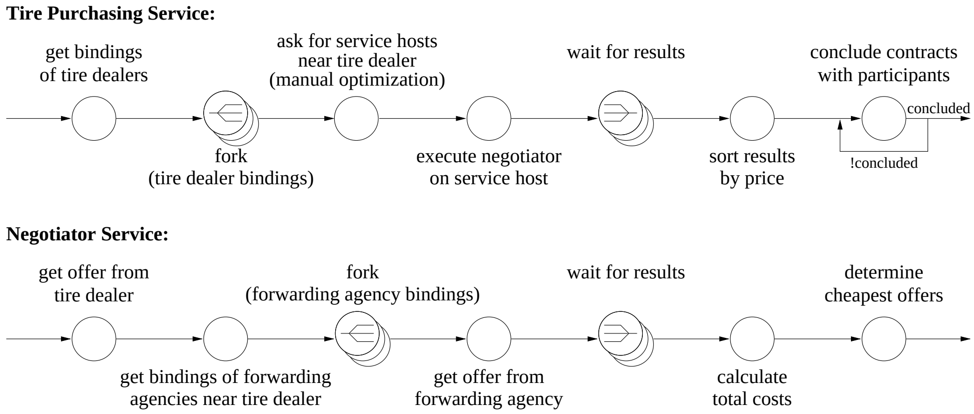

Figure 1: Graphical Representations of the Services

engine, thus being able to execute mobile code. Next, a negotiator service is executed on each of these service hosts. For performance reasons, all negotiator services are executed in parallel. Finally, the tire purchasing service waits 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 all forwarding agencies geographically close to that tire dealer. 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, waits for the results, calculates the overall costs (tires and delivery), and determines the five cheapest offers. These offers are sent back as result.

3 Architecture of ServiceGlobe

The ServiceGlobe system provides a lightweight infrastructure for a distributed, extensible service platform. It is completely implemented in Java Release 2 and based on standards like XML [BPSMM00], SOAP [BEK+00], UDDI [UDD00], and WSDL [CCMW01]. In this section, we present the basic components of the ServiceGlobe infrastructure. First of all, we distinguish two different types of services: external and internal services (see Figure 2).

External services are services currently deployed on the Internet, which are not provided by ServiceGlobe itself. Such services normally are stationary, i.e., running only on a dedicated host, may be realized on arbitrary systems on the Internet, and may have arbitrary interfaces for their invocation. Since we want to be able to integrate these services independent of their actual invocation interface, e.g., 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.

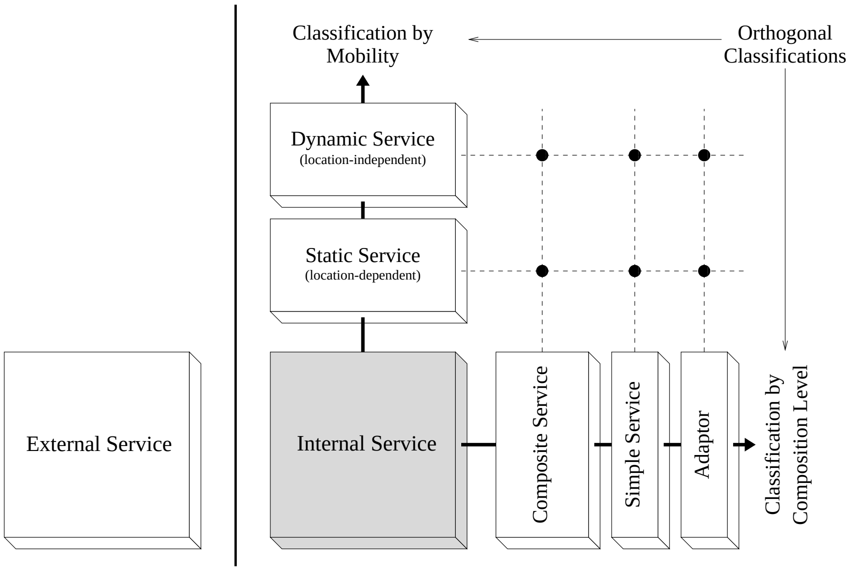

Figure 2: Classification of Services

Internal services are native ServiceGlobe services. They are implemented in Java using the service API provided by the ServiceGlobe system. Of course, it is feasible to use a specialized programming language, e.g., XL [FK01], or a graphical tool to draw a representation (similar to a workflow graph) of a service, but that is not the focus of our work. ServiceGlobe services use SOAP to communicate with other services. Services receive a single XML document as input parameter and generate a single XML document as result. 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. Such 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*, i.e., hosts connected to the Internet which are running the ServiceGlobe runtime engine. 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. Service hosts offer a SOAP service (called *runtime service loading*) to execute dynamic services. Thus, the set of available services is not fixed and can be extended at runtime by everyone participating in the ServiceGlobe federation. If internal services have the appropriate permissions, they

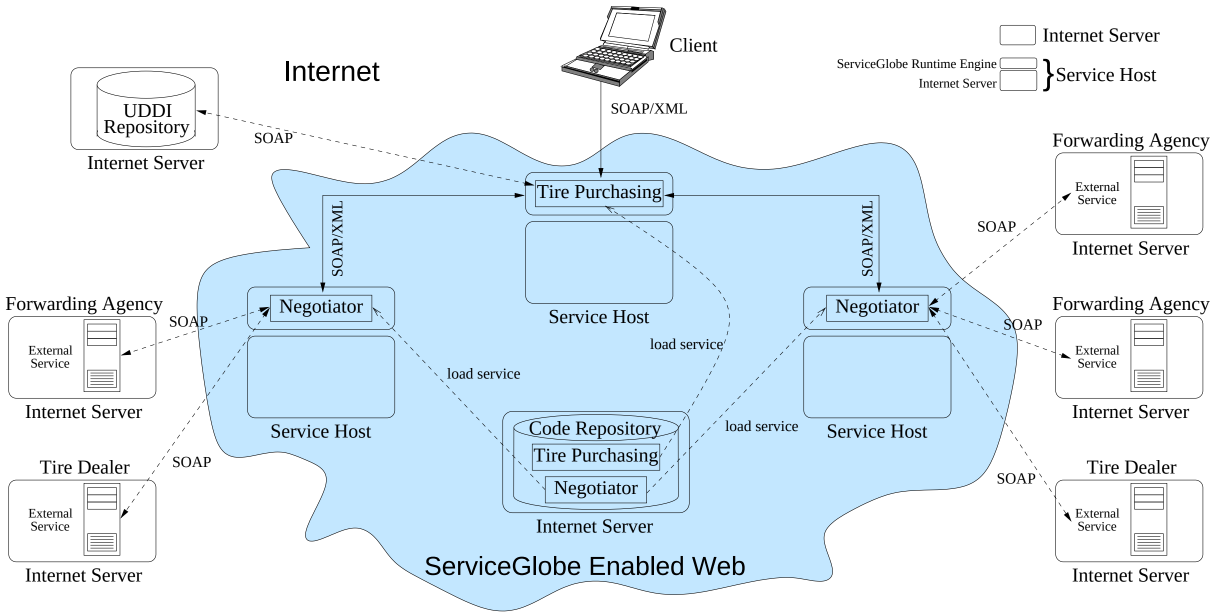

Figure 3: Architecture of a ServiceGlobe Enabled Web

can also use resources of service hosts, e.g., databases. These permissions are part of the security system of ServiceGlobe which is based on [SBK01] and they are managed autonomously by the administrators of the service hosts. This security system also deals with the security issues of mobile code introduced by runtime service loading. Thus, service hosts are protected against malicious services.

Runtime service loading allows *service distribution* of dynamic 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 services. Of course, this feature also contributes to reliable service execution because unavailable service hosts can be replaced dynamically by available service hosts. Together with runtime service loading this provides a large flexibility in order to consider load balancing or optimization issues.

Figure 3 gives an overview of the basic components of the ServiceGlobe system and their mutual interaction (based on the e-procurement scenario of Section 2): 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 SOAP request 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.

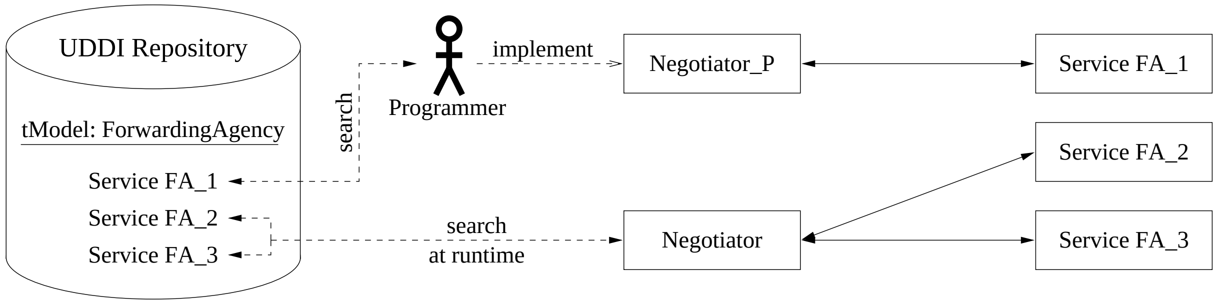

Figure 4: An Example of Dynamic Service Selection

4 Dynamic Service Selection

In general, composite Web services invoke other, remote Web services by passing the service platform the Web service's URL or access point. In contrast, *dynamic service selection* enables Web services to state a technical specification of the services that should be invoked. Its the service platform's task to select suitable Web services, possibly utilizing UDDI for this. Additionally to the technical specification, different kinds of constraints can be passed over to influence the dynamic service selection. The approach of dynamic service selection offers three main advantages:

- An important goal in distributed systems is a high reliability rate and fault tolerance. Using composite services, it can occur that some basis services may not be reachable, e.g., due to network partitioning, unavailable service hosts, or the unexpected crash of a basis service.[1] For this reason, hard-coded access points within a composite service are not desirable. Dynamic service selection provides a solution for this problem.

- Using constraints to influence dynamic service selection allows developing generic Web services. As will be show, it is not necessary to code special properties of the invoked Web services into the composite service itself. Instead, these properties are specified as (declarative) constraints and passed to the Web service at runtime. As new kinds of constraints become available, they can be used without modifying or re-compiling the Web service. Constraints can, for example, specify that only free Web services are used or that services of a given company should be preferred, if possible.

- Dynamic services are instantiated at runtime. Combining this with dynamic service selection offers potential for various optimizations. For example, constraints allow specifying that all dynamic services must be instantiated within a local LAN or on a given set of hosts.

In the following, a more detailed description of dynamic service selection will be given. In the description, UDDI is used as Web service repository. Basically because it is the de-facto standard for such a kind of repository and because it provides the necessary functionality to use it in conjunction with dynamic service selection.

[1]Related problems, although in the context of Web scripting languages, have been studied in [CD99a].

In UDDI, every service is assigned to a tModel providing a classification of its functionality and a formal description of its interfaces. Therefore, a service can be called an *implementation* or an *instance* of its tModel. With this, instead of explicitly stating an actual access point in a composite service, it is also possible to reference or "call" a tModel. Thus, one defines the functionality of the service that should be called rather than its actual implementation. As an example, see Figure 4. The tModel ForwardingAgency has three services assigned to it, namely Service FA_1, Service FA_2 and Service FA_3. Now, assume that a programmer tries to implement the new service Negotiator_P which should invoke another Web service which has to be assigned to the tModel ForwardingAgency. Normally, the programmer will search a UDDI repository for an appropriate service, e.g., Service FA_1 and use its access point in the new Web service. At runtime, Negotiator_P will invoke the basis service given by the hard-coded access point. With dynamic service selection, the programmer will develop the different Web service Negotiator. Negotiator will not contain any hard-coded access point, instead it will know the tModel of the service it should invoke. At runtime, the service Negotiator will query the UDDI repository for an appropriate Web service and invoke it (or even a set of appropriate services, as depicted in Figure 4).

4.1 Modes

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 (The different modes are similar to unicast, multicast, and broadcast communication on networks):

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. An example is shown in Figure 5 (rectangle tModel).

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. The circle tModel in Figure 5 shows a some call.

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. For an example, see the triangle tModel in Figure 5.

Basically, these three modes could also be implemented as constraints (see the following section). But they are a very fundamental part of the communication infrastructure, so the separation of these communication patterns is justified.

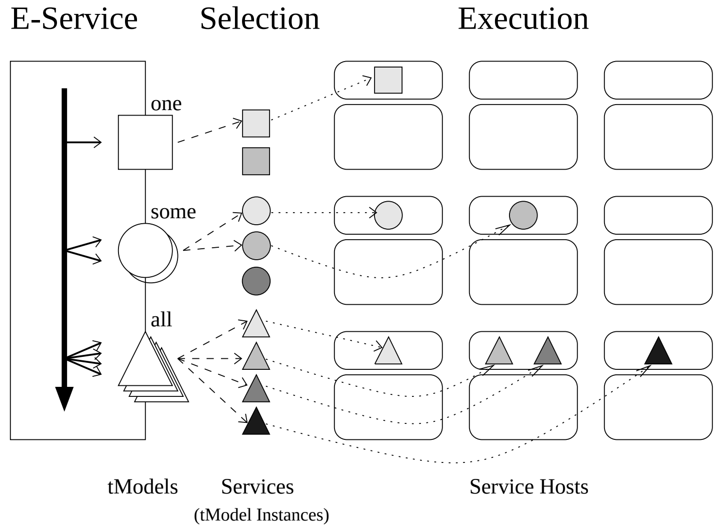

Figure 5: Dynamic Service Selection

4.2 Constraints

The services actually selected for a tModel call depend on several different kinds of constraints. Right now, these constraints must be specified by a Web service when invoking a basis service. There are two types of constraints: hard constraints or *conditions* and soft constraints or *preferences*. The difference is that conditions must be fulfilled, whereas preferences should be fulfilled. If no service is found at all or only an insufficient number of services is found which fulfill the preferences, alternative services are selected which need not fulfill the preferences.

The first kind of constraints are *metadata constraints*. They work as a filter for all tModel instances returned by UDDI. Metadata about a single service consists of its UDDI data (bindingTemplate, businessService, and businessEntity) and additional metadata stored in an (XML) metadata repository (see Figure 6 for an example). Metadata constraints are basically XPath [CD99b] queries. The ServiceGlobe system will only select services that comply to the constraints. For example, the following preference (a soft constraint) selects only services that are assigned to a businessEntity with the name `"Forwarding Agency"` in the first place, other services are only selected afterwards, if necessary:

```
<metadataPreference>
  /businessEntity/name="Forwarding Agency"
</metadataPreference>
```

A metadata constraint selecting all free services (and no services that charge requests) looks like the following:

```
<metadataConstraint>
  /serviceMetadata/costsPerCall="0"
</metadataConstraint>
```

Only services complying to these constraints are available to the mode-dependent selection of services as explained above.

```
<businessEntity businessKey='35C43D08-FBBF-2C59-AA29-CF032D051111'>
  <name>Forwarding Agency</name>
  <description>An example businessEntity for a forwarding agency</description>
  <businessServices>
    <businessService serviceKey='362C66B3-03C4-F54B-AC4F-CDC8E2871111'
                     businessKey='35C43D08-FBBF-2C59-AA29-CF032D051111'>
      <name>Forwarding Agency</name>
      <bindingTemplates>
        <bindingTemplate bindingKey='4FB831F6-327E-7815-3331-52DB9BF91111'
                         serviceKey='362C66B3-03C4-F54B-AC4F-CDC8E2871111'>
          <accessPoint URLType='http'>
            http://www.tempuri.org/services/forwarding-agencies
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo
              tModelKey='uuid:B39997F5-DF7F-9F62-0EEE-6F43345DE1A3'/>
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
<serviceMetadata>
  <costsPerCall>0</costsPerCall>
  <!-- other metadata may follow -->
<serviceMetadata>
```

Figure 6: Example of Service Metadata

The second kind of constraints are *reply constraints*. Reply constraints are applied immediately after a reply is received from a service. If the reply does not fulfill all given reply constraints it is discarded.

Finally, there are *call constraints*. They specify conditions on the set of all replies received from services (which must comply to all reply constraints), e.g., 'return only the results of the first ten services that reply'. An all-mode call with this constraint would invoke all services returned by UDDI, but only the first ten replies will be considered and returned to the calling service.

The process of dynamic service selection is sketched in Algorithm 1. Where suitable, procedures with descriptive names were used to subsume not so interesting details, to allow a brief and concise presentation of the algorithm. The procedure uddi-fetch-services (line 2) returns all services that are instances of the given tModel t. Then, filter-by-metadata-constraints removes all services from S that do not meet all metadata constraints in C (line 3). Next, services are called depending on the mode m (line 4). The while loop (line 5) is executed as long as the received replies do not comply to all call constraints, i.e., as long as the procedure check-call-constraints returns false. In the loop, if a service does not reply (Timeout) or it returns an error, an alternative service is called using the procedure call-alternative-service (line 8). If a valid reply is received and if it meets all reply constraints (procedure check-reply-constraints returns true), it is inserted into the set of received replies (line 11). Otherwise an alternative service is called. Finally, all pending services, i.e., services which did not reply, are aborted.

In our tire dealer example there is one place where dynamic service selection takes place. When a negotiator service contacts forwarding agencies, it calls the forwarding

Algorithm 1

```
Require: t : tModel, d : Document, m : mode, C : set of constraints
Ensure: R : set of replies
R ← ∅
S ← uddi-fetch-services(t)
filter-by-metadata-constraints(S, C)
call-services(S, d, m)
while ¬ check-call-constraints(R, C) do
  reply ← wait-for-reply()
  if reply = Timeout ∨ reply = Error then
    call-alternative-service(S, d, m)
  else
    if check-reply-constraints(reply, C) then
      R ← R ∪ {reply}
    else
      call-alternative-service(S, d, m)
    end if
  end if
end while
abort-pending-services(S)
```

agency tModel using the all-mode. Additionally, a constraint filtering all forwarding agencies geographically close to the tire dealer is used.

5 Automatic Service Replication

The flexible architecture of ServiceGlobe provides a feature called *automatic service replication*. Using this feature, it is possible to enhance existing services or develop new services with load balancing and high availability features, without having to consider these features during their development. We are using a technique similar to Layer 7 Switching which is widely used in the context of Web servers. Such switches do load balancing using several servers on the back end with identically mirrored content using a round robin strategy or load information about these servers. In contrast to these switches, our solution is a pure software solution and is realized as a regular service. Thus, it is more flexible and extensible and, additionally, it is seamlessly integrated into the service platform. Up to now, only the basic functionality is integrated into ServiceGlobe, but we will improve the functionality shortly, see Section 7.

We developed a general dispatcher service which is configurable to act as proxy for arbitrary services. This dispatcher is able to monitor a set of service hosts. Figure 7 shows a dispatcher for Service S monitoring three service hosts running two instances of Service S (both connected to the same DBMS). Using the monitoring services available at every service host, the dispatcher avoids overload situations. Upon receiving a message, the dispatcher looks for an appropriate service instance having the lowest load on its service host and forwards the message to the selected service instance.

If all existing service instances are running on heavily loaded service hosts, the dis-

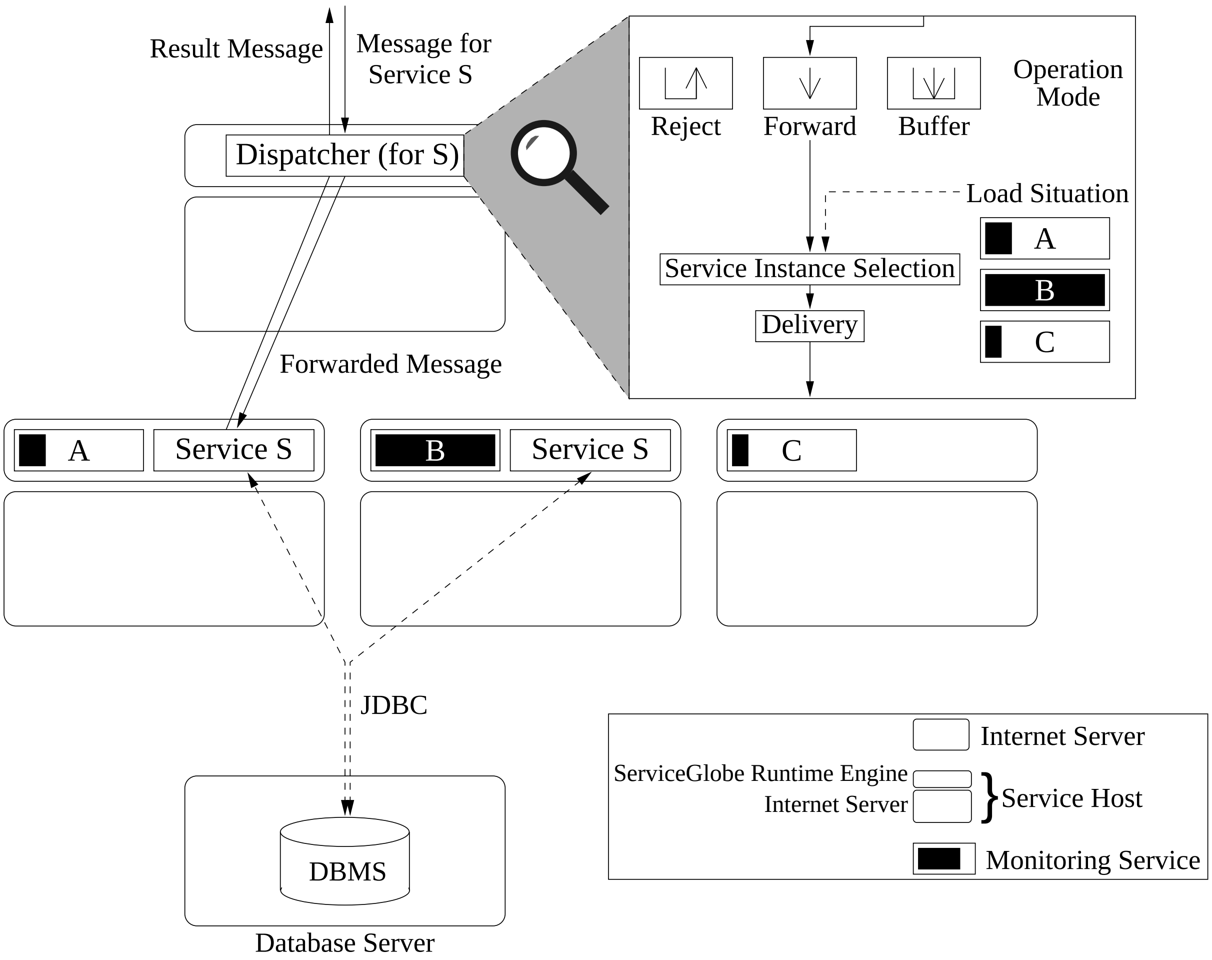

Figure 7: Survey of Automatic Service Replication

patcher generates a new instance on a service host having a low workload. If no such service host is available, the dispatcher can either buffer incoming messages or reject them (sending a “temporary unavailable” message back to the service caller) depending on the configuration of the dispatcher instance.

Using several instances of a service greatly increases its availability. Assuming that the server running the dispatcher itself and the database server are highly available, the availability of the entire system depends only on the availability of the service hosts. The availability of a pool of service hosts can then be calculated as follows:

$$\alpha = \alpha_{ServiceHost} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \tag{1}$$

$$\alpha_{pool} = \sum_{i=1}^{N} \alpha^i (1-\alpha)^{(N-i)} = 1 - (1-\alpha)^N \tag{2}$$

Equation 1 calculates the availability of a single service host based on its MTBF (mean time between failures) and MTTR (mean time to repair). The availability of a pool of N service hosts can be calculated using Equation 2. Even assuming very unreliable service hosts with MTBF $= 48h$ and MTTR $= 12h$ a pool with 8 members will only be unavailable about 1.5 minutes a year.

As we have shown, automatic service replication has several advantages: First, the development of services without having to consider high availability and/or load balancing features is much easier. Second, every existing service can be turned into a high available and load balanced service without a lot of work, as long as there is no data sharing between different instances of this service or it has some kind of concurrency control, e.g., by using a database as back end. This service enhancement is done by starting a dispatcher service parameterized with an appropriate configuration document and modifying some data at the UDDI repository. After that, the service instances are no more contacted directly, but via the dispatcher service controlling the forwarding of the messages. Third, a cluster of service hosts can be easily supplemented with new service hosts. The administrators of these service hosts only have to install the ServiceGlobe system and register them at the UDDI repository using the appropriate tModel, e.g., ServiceHostClusterS, indicating that these service hosts are members of the cluster S. The dispatcher will automatically execute services at these hosts as soon as the load at other hosts exceeds a limit.

6 Related Work

There are a number of frameworks for service composition and execution. Sun propagates its Sun ONE framework [Sun] which is based on J2EE [J2E] and the well-known Java language. Microsoft .NET [NET] is a bundle of well-known and new applications targeted at Web service development and deployment. HP is currently developing its Web Services Platform [WSP]. 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. They intend to provide a complete infrastructure for implementing services and executing them on dedicated hosts. Our objective in this work is to present new functionality for e-service execution and deployment which can be integrated into existing service platforms.

Although service composition languages are not the focus of our work, we are aware of work in this area. IBM's WSFL (Web Services Flow Language) [Ley01], Microsoft's XLang [Tha01], and HP's WSCL (Web Services Conversation Language) [BBB+02] are languages to describe how to compose existing services, i.e., to describe some kind of conversation. Compaq's Web Language [Web] (formerly WebL) specializes in fetching, parsing, and generating HTML and XML content. Besides service composition, the XL language [FK01] additionally offers very powerful statements for easy and efficient programming of services. HP's eFlow [CIJ+00] is similar, but more workflow oriented and based on a graphical notation. Regarding services and their composition there is also the ebXML [ebX] standardization effort which defines a standard for global electronic business. Of course, ServiceGlobe as well as most other service frameworks are based on the W3C Internet standards, most prominently XML, UDDI, WSDL, and SOAP.

7 Conclusion

In this work, we presented novel techniques for flexible and reliable Web service execution and deployment which can be integrated into existing service platforms. We introduced dynamic service selection which allows selecting service instances during runtime by means of semantic classifications. Different kinds of constraints enable services to influence the selection of services. We also addressed load balancing and high availability issues by providing a generic and transparent approach for automatic service replication. Using this feature, it is possible to enhance existing services with load balancing and high availability features, without having to consider these features during their development. The main idea is to use a general dispatcher service which distributes messages depending on the actual load situation.

For the future, we plan to further explore and extend the presented techniques. First, we enhance the automatic service replication feature, i.e., the dispatcher, to allow the configuration of newly instantiated services in complex environments automatically. For example, if the new service connects to a database management system running on several hosts using replicated data, it can be advised to connect to the instance of the DBMS having the lowest average load. Therefore, we require an autonomous monitoring service which is able to run outside the ServiceGlobe system. Additionally, the dispatcher should be enhanced to handle overload situations of database instances or other resources used by services. Thus, the dispatcher should, e.g., avoid all services connected to an overloaded database instance, instantiate caches for service or database results where suitable to relieve the load on the overloaded database instances, or contact the administrator that it would be ingenious to add an additional database server. Another challenge we have to face is the single-point-of-failure characteristic of the dispatcher service. There are some more features we might integrate into the dispatcher: an accounting module, a quality of service module, an access control module, and a module to detect respectively prevent denial of service attacks.

Second, it would be beneficial for dynamic service selection to allow the combination of different constraints using Boolean operators. As a consequence, conflicts may arise because of contradictory constraints and, therefore, appropriate strategies for resolving these conflicts are necessary. Furthermore, we will integrate constraints into the context of Web services. The term context refers to information about a consumer of a Web service that is used by the service to customize its execution and output. Consequently, constraints should be used to define which types of services a consumer is interested in and wants to use. In the ServiceGlobe system, context is transmitted as part of the SOAP messages (in the SOAP header) that services send and receive. Integration of constraints (in their XML representation) into this context information will not only enable an invoked service to take advantage of these constraints, but also further services invoked by this service, as context information of a service is (automatically) included into SOAP messages sent by it.

References

[BBB+02] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. Web Services Conversation Language (WSCL) 1.0, 2002. `http://www.w3.org/TR/wscl10/`, Retrieval on 2002-07-11.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, May 2000. `http://www.w3.org/TR/SOAP`, Retrieval on 2002-07-11.

[BPSMM00] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. http://www.w3.org/TR/REC-xml, Retrieval on 2002-07-11.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl, Retrieval on 2002-07-11.

[CD99a] L. Cardelli and R. Davies. Service combinators for Web computing. *IEEE Trans. Software Eng.*, 25(3):309–316, May 1999.

[CD99b] J. Clark and S. DeRose. XML Path Language (XPath) 1.0, 1999. http://www.w3.org/TR/xpath, Retrieval on 2002-07-11.

[CIJ+00] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. eFlow: a Platform for Developing and Managing Composite e-Services, 2000. http://www.hpl.hp.com/techreports/2000/HPL-2000-36.pdf, Retrieval on 2002-07-11.

[ebX] Electronic Business XML Initiative (ebXML). http://www.ebxml.org/, Retrieval on 2002-07-11.

[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.

[J2E] Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/, Retrieval on 2002-07-11.

[KSSK02] 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)*, HongKong, China, August 2002.

[Ley01] F. Leymann. Web Services Flow Language (WSFL 1.0), 2001. http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, Retrieval on 2002-07-11.

[NET] Microsoft .NET. http://www.microsoft.com/net/, Retrieval on 2002-07-11.

[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/, Retrieval on 2002-07-11.

[Tha01] S. Thatte. XLANG: Web Services for Business Process Design, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, Retrieval on 2002-07-11.

[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/, Retrieval on 2002-07-11.

[Web] Compaq's web language. http://www.research.compaq.com/SRC/WebL/, Retrieval on 2002-07-11.

[WSP] HP Web Services Platform. http://www.hp.com/go/webservices/, Retrieval on 2002-07-11.