Reliable Web Service Execution and Deployment in Dynamic Environments*

Markus Keidl, Stefan Seltzsam, and Alfons Kemper

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

Abstract. In this work, we present novel techniques for flexible and reliable execution and deployment of Web services which can be integrated into existing service platforms. The first technique, dynamic service selection, provides a layer of abstraction for service invocation offering Web services the possibility of selecting and invoking Web services at runtime based on a technical specification of the desired service. The selection can be influenced by using different types of constraints. The second technique, a generic dispatcher service capable of automatic service replication, augments Web services with load balancing and high availability features, without having to consider these features at the services' development. We implemented these techniques within the ServiceGlobe system, an open Web service platform.

1 Introduction

Web services are a new technology for the development of distributed applications on the Internet. 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 [18]. 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, or 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.

Our objective in this work is to present new techniques for Web service execution and deployment in dynamic environments. The first technique we present is dynamic service selection. It offers the possibility of selecting and invoking services at runtime based on a technical specification of the desired service. Therewith, it provides a layer of abstraction from the actual services. Constraints enable Web services to influence dynamic service selection, e.g., services can be selected based on the metadata available about them.

We address load balancing and high availability by providing a generic, modular dispatcher service for augmenting services with these features, without having to consider them during the services' development. The dispatcher is a software-based layer-7 switch with the known advantages: it forwards requests to different service instances and therefore reduces the risk of a service being unavailable and speeds up request processing because of load balancing respectively load sharing. Our dispatcher implements

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

a new feature called automatic service replication. Using this feature, new (individually configured) services can be installed on idle hosts on behalf of the dispatcher to leverage available computing power. Additional advantages are that the dispatcher is integrated into the service platform and is completely transparent to the callers of a service.

We implemented both techniques, dynamic service selection and the dispatcher, within the ServiceGlobe system [11] which is described in the next section. The techniques can also be integrated into existing service platforms without major modifications. A demo of ServiceGlobe was given at VLDB'02 [12]. As part of SAP's adaptive computing infrastructure, the ServiceGlobe system is installed on a blade server with 160 processors overall (with 2 and 4 processors per server blade, respectively), which is operated by the Advanced Infrastructure Program group of SAP. Several performance evaluations with high-volume business applications are conducted using this system. The presented technologies are currently integrated into the SAP NetWeaver platform to supplement its service virtualization capabilities. A demonstration was shown at the Sapphire 2003 [1], SAP's user conference.

The remainder of this paper is structured as follows: Section 2 introduces the ServiceGlobe system. Sections 3 and 4 present dynamic service selection and the generic dispatcher capable of automatic service replication, respectively. Finally, Section 5 gives some related work and Section 6 concludes this paper.

2 Architecture of ServiceGlobe

The ServiceGlobe system is a lightweight, distributed, and extensible service platform. It is fully implemented in Java Release 2 and based on standards like XML, SOAP, UDDI, and WSDL. 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 a transaction system and a security system [19]. These areas are well covered by existing technologies and are, therefore, not the focus of this work. We will now explain the ServiceGlobe infrastructure. First of all, we distinguish between external and internal services.

External services are services currently deployed on the Internet, which are not provided by ServiceGlobe itself. Such services are stationary, i.e., running only on a dedicated host, are realized on arbitrary systems on the Internet, and have arbitrary interfaces for their invocation. If they do not provide an appropriate SOAP interface, we use *adaptors* to transpose internal requests to the external interface and vice versa, to be able to integrate these services independent of their actual invocation interface, e.g., RPC. This way, we are also able to access arbitrary applications, e.g., ERP applications. Thus external services can be used like internal services.

Internal services are native ServiceGlobe services implemented in Java using the service API provided by the ServiceGlobe system. ServiceGlobe services use SOAP to communicate with other services. 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, e.g., they require access to certain local resources like a DBMS. In contrast, dynamic services are location-*independent*. They are state-less, i.e., the internal state of such a service is

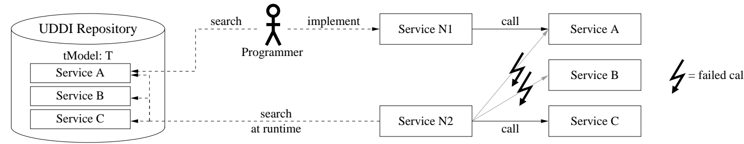

Fig. 1. An Example of Dynamic Service Selection

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*, i.e., hosts connected to the Internet which are running the ServiceGlobe runtime engine. ServiceGlobe's internal services are mobile code, therefore their executables can be loaded on demand from *code repositories* into a service host's runtime engine (this feature is called *runtime service loading*). A UDDI server is used to find an appropriate code repository storing a certain service. 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 can also use resources of service hosts, e.g., databases. These permissions are part of the security system of ServiceGlobe which is based on [19] 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 optimization potential: 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, large 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 the flexibility needed for load balancing or optimization issues.

3 Dynamic Service Selection

In UDDI, every service is assigned to a tModel[1] which provides a semantic classification of a service's functionality and a formal description of its interfaces. So, a service can be called an *implementation* or an *instance* of its tModel. With dynamic service selection (DSS), instead of explicitly stating an actual access point in a 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. Without DSS, the selection of services from UDDI based on a search criteria like a tModel has to be done manually by a programmer when implementing a Web service. Furthermore, the search criteria available in UDDI are less general and there are no criteria for influencing service invocation or for filtering service replies.

[1] In fact, a service in UDDI can be assigned to several tModels. DSS could be adjusted to allow calling services which implement several tModels. As there is no essential difference to calling a single tModel, this will not be considered in the following.

As example for DSS, see Figure 1: Three services are assigned to tModel T: Service A, B, and C. Assume, that a programmer wants to implement a new Web service which should invoke a service assigned to tModel T. Without DSS, the programmer would search UDDI for an appropriate service, e.g., Service A, and use its access point in the new Service N1. With DSS, the programmer will instead develop Service N2. This service does not contain any hard-coded access point, instead it contains a call to the tModel T. At runtime, the service will query UDDI for an appropriate Web service and invoke it. If an invocation fails, alternative services are tried until an invocation succeeds (as depicted in Figure 1) or no more alternative services are available.

As already mentioned, DSS is implemented within ServiceGlobe. The ServiceGlobe API provides methods for Web services to invoke tModels and to optionally specify constraints and/or use constraints contained in the service's context.

3.1 Constraints

Constraints are used to influence DSS. They can be passed to a service platform within a service's context or by specifying them directly when calling a tModel. The term context refers to information about the consumer of a Web service which is used by the service to adjust its execution and output to provide a customized and personalized version of itself. In the ServiceGlobe system, context is transmitted in the header of the SOAP messages that services send and receive. The integration of constraints into context information enables not only the invoked services to take advantage of them, but also further services invoked by these services, as the context information of a service is (automatically) included into SOAP messages sent by it.

Constraints can be differentiated into *preferences* and *conditions*.[2] Conditions must be fulfilled, whereas preferences should be fulfilled. When considering preferences in DSS, a service platform at first invokes services that fulfill these preferences. If there is an insufficient number of such services, additional services are invoked which do not fulfill all preferences (but, of course, they must fulfill all conditions). Orthogonally, there are five different types of constraints: metadata, location, mode, reply, and result constraints. For each type, there are preferences and conditions; though, for mode and result constraints, preferences are useless.

Metadata Constraints: Prior to the invocation of services, when the service platform requests all services assigned to a tModel, metadata constraints are applied as filter on all services returned by UDDI. Metadata constraints are basically XPath [6] queries that are applied to the metadata of a service. Metadata about a service includes primarily its UDDI data. Also, additional metadata which is stored in other metadata repositories [10] and that cannot be found in UDDI may be contained. The following example shows a metadata preference that favors services assigned to a businessEntity with name `Company`:

```
<metadataPreference>
  /businessEntity/name="Company"
</metadataPreference>
```

Location Constraints: Location constraints are used to specify the place of execution of a Web service, i.e., the service host. For static services, this allows their selection

[2] A similar classification of conditions of SQL statements in hard and soft constraints is described in [13].

based on their location. For dynamic services, this ensures that they are instantiated and executed preferably (preference) or strictly (condition) at the given location. The information about the location of services and service hosts is retrieved from the UDDI repository. The location can be specified by, e. g., a host's network address or geographically based on GPS coordinates or ISO 3166 codes.

Mode Constraints: DSS is not limited to invoke only one instance of a given tModel; it is also possible to invoke several instances. With a mode constraint the number of services that should be invoked can be specified. There are three modes available:[3] Using the *one mode*, only one instance out of all tModel instances is called. In case of a failure, e.g., unavailability of a service, an alternative service is tried. Using the *some mode*, a subset of all services returned by UDDI is called in parallel.[4] The number of services is specified as an absolute value or as a percentage. Services which fail are replaced with alternative services. Using the *all mode*, all returned tModel instances are called. Obviously, no alternative services can be called if failures occur. The following example shows a mode constraint that specifies that five percent of the available services should be invoked:

```
<modeCondition modeType="Some" number="5%" />
```

Reply Constraints: Reply constraints are evaluated after a reply of an invoked service was received. Every reply not fulfilling all relevant reply constraints is discarded. There are two kinds of reply constraints. *Selection constraints* are XPath queries which are applied to the reply of a service, including its SOAP parts. With *property constraints*, replies can be selected based on a set of properties of the reply. Properties must be provided either by the service platform or by the invoked service. A service accomplishes this by including corresponding XML elements in its reply. ServiceGlobe itself supports properties for encryption, signature, and age of data. Using the first two properties, it is possible to verify if a reply is encrypted or signed, respectively, and by whom it is signed. The third property can be used to check the age of the returned data.

Result Constraints: Result constraints refer to all replies received so far. There are two kinds of result constraints. With a *timeout constraint*, a maximal waiting time for replies of invoked services can be set. After its expiry, all pending services are aborted and all replies received so far are returned to the calling service. The following constraint is an example of a timeout constraint:

```
<timeoutCondition value="100" valueUnit="Seconds"/>
```

With *first-n constraints*, the call to a tModel can be ended after a predetermined number of replies was received. The calling service gets only these replies as result of its call. Services that have not responded until this moment are aborted.

3.2 Combination of Constraints

Constraints can be combined using the operators AND and OR. By the combination of constraints, conflicts can be created which may prevent fulfilling all given constraints. As a consequence, only a subset of the given constraints may be fulfillable, as the example in Figure 2 shows (`orGroup` represents the OR operator).

[3] These modes are similar to unicast, multicast, and broadcast communication on networks.
[4] It should be noted that one and all mode are obviously special cases of the some mode.

```
<orGroup>
  <metadataCondition>
    /businessEntity/name="Company"
  </metadataCondition>
  <timeoutCondition value="100" valueUnit="Seconds"/>
</orGroup>
```

Fig. 2. Combination of Constraints

Initially, the service platform has two choices: On the one hand, it can invoke *only services of the company* `Company` and wait for their replies (therewith fulfilling only the first constraint). On the other hand, it can invoke *all services* assigned to the tModel. But if a timeout occurs, the service platform faces the situation that it either must return all replies received so far immediately (therewith fulfilling only the second constraint) or that it must ignore the timeout and wait at least for all replies (therewith fulfilling only the first constraint). In the latter case, though, it invoked too many services initially. So, in general, the service platform is unable to fulfill both constraints at the same time.

3.3 Evaluation of Constraints

This section explains how a tModel call is actually executed and how constraints are evaluated in this process. At first, constraints from all different sources are combined conjunctively into one single combined constraint, called *main constraint*, using the AND operator. This constraint is passed as input to the tModel call. Its evaluation consists of two phases: First, it is transformed into disjunctive normal form (DNF) and conflicts are resolved. Second, UDDI is queried for services assigned to the given tModel and the services are invoked considering the main constraint.

Preprocessing of Constraints: First, the main constraint is transformed into DNF. Note, that the same constraint can now be present multiple times in the transformed constraint. Afterwards, all constraints of an *AND term*, i.e., a term only containing AND operators, are sorted according to their time of evaluation. The order is: metadata, location, mode, reply, and result constraints.

Then, the main constraint is checked for conflicts. Only conflicts within a single AND term are resolved in this phase, conflicts between different AND terms are resolved later, during the invocation phase. Within an AND term, a conflict occurs if it either contains more than one mode constraint or more than one result constraint. For mode constraints, this is obvious. For result constraints, there are some rare situations where several result constraints would make sense. But, as we see no real benefit, two or more result constraints per AND term are prohibited.[5] Of course, conflicts between metadata, location, or reply constraints are possible in principle, e.g., an AND term that contains metadata constraints with contradictory XPath queries. Detecting this type of conflict would require a detailed investigation of the XPath queries.

Conflicts are resolved by keeping only the constraint with the maximum priority and removing all other conflicting constraints. Priorities range from 0 (minimum) to ∞ (maximum) and they can be assigned to a term by its creator, e.g., the consumer or a Web service. An additional, implicit prioritization is given by the sequence of the terms

[5] The implementation would be straightforward, although requiring many, even though simple case discriminations.

in their XML representation. The later a term is defined there the less its priority is. If two terms have the same explicit priority, their implicit priority decides which one has the higher priority.

At last, identical mode and result constraints which are contained in several AND terms because of the transformation into DNF are merged.[6] The resulting terms are called *merged AND terms*. Without merging, a service platform would evaluate identical mode and result constraints multiple times which would result in a different result. Only mode and result constraints are considered for merging because, unlike the other constraint types, they are restrictions on sets of services respectively replies, not on single services or replies. Therefore, the result of the main constraint is only modified by duplicating them when transforming the main constraint into DNF.

Invocation of Web Services: After the main constraint has been preprocessed, UDDI is queried for all information about services assigned to the given tModel. These services as well as their metadata are stored in a *services list*. Initially, there is one such services list for every merged AND term. In the following, services which do not fulfill a condition are removed from a services list. Preferences are used to sort this list.

Now, metadata constraints are applied to the services list of their merged AND term, followed by location constraints. For the evaluation of location constraints for dynamic services, all available service hosts are retrieved from UDDI first. Then, the location constraints are used to filter and sort this list of service hosts (similar to services lists). For each merged AND term, the corresponding service hosts list is assigned to all dynamic services of this term.

Next, all mode constraints of the main constraint are evaluated in parallel, i.e., Web services are invoked as specified by the mode constraints considering all relevant services lists. As a consequence of the merging of identical mode constraints, services lists from more than one merged AND term may have to be considered. For each invocation of Web services based on a single mode constraint, the corresponding services list is processed sequentially, starting with the service at the top (which has the highest priority). Thereby static services are invoked only once, dynamic services can be invoked as often as there are service hosts in their service hosts list (service host are chosen according to their priority).

Every time the reply of a Web service is received, all relevant reply constraints are applied to it. Note, that the Web service may be contained in several services lists, so there can be more than one merged AND term with relevant reply constraints. The service platform must also check whether the invocation phase must be ended. This is the case if the result constraint with the highest priority is fulfilled. After the invocation phase ended, all outstanding requests are aborted and all replies are returned to the calling Web service.

4 Load Balancing and Service Replication

For large-scale, mission-critical applications, such as an enterprise resource planning system like SAP with thousands of users working concurrently, a single service host is not sufficient to provide low response times. Even worse, if there are any problems

[6] Basically, merging means factoring out identical mode and result constraints.

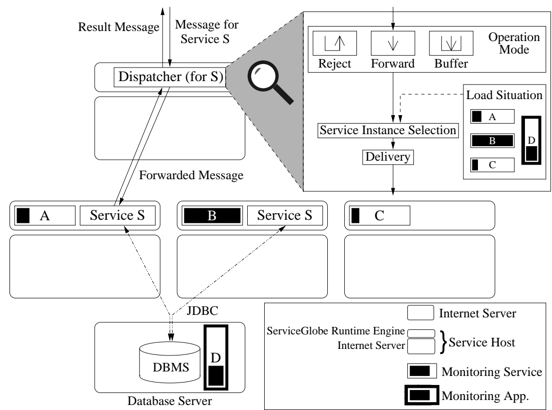

Fig. 3. Survey of the Load Balancing System

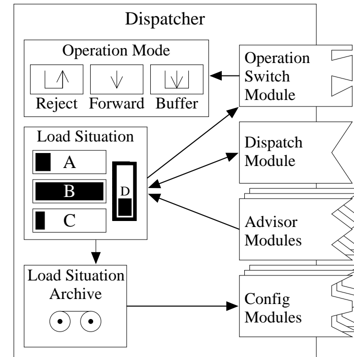

Fig. 4. Dispatcher's Architecture

with the service or the service host, the service will be completely unavailable. Such downtime can generate high costs, even if a service host is only down for some minutes. 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. A server blade architecture is very beneficial for this purpose, because scale-out of computing power can be done on demand by adding additional server blades. Of course, a traditional cluster of service hosts connected by a LAN can be used as well but with higher total cost of ownership and normally slower network connections.

Since it is very expensive and error-prone to integrate the functionality for the cooperation of the service instances directly into every new service, we propose a generic solution to this problem: a modular *dispatcher service* which can act as a proxy for arbitrary services. Using this dispatcher service, it is possible to enhance many existing services or develop new services with load balancing and high availability features without having to consider these features during their development. All kinds of services are supported as long as concurrency control mechanisms are used, e.g., by using a database as back-end (as many real-world services do). The concurrency control mechanisms ensure a consistent view and consistent modifications of the data shared between all service instances. Of course, if there is no data sharing between different instances of a service, the dispatcher can be used as well. 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[7]. Such switches perform load balancing (or load sharing) using several servers on the back-end with identically mirrored content and a dispatching strategy like round robin or more complex strategies using load information about the back-end servers. Our solution is a pure software solution and—in contrast to existing layer-7 switches—is realized as a regular service. Thus, our dispatcher is more flexible, extensible, and seamlessly integrated into the platform.

[7] This kind of switch is also used in the context of Web servers [4].

Figure 3 shows our dispatcher monitoring three service hosts running two instances of Service S (both connected to the same DBMS). The database server is monitored as well using a stand-alone monitoring application. Using information from monitoring services and monitoring applications, the dispatcher generates the dispatcher's local view of the load situation of the service hosts. Upon receiving a message (in this case for Service S), the dispatcher looks for the service instance running on the least loaded service host and forwards the message to it. As already mentioned, our dispatcher is modular, as shown in Figure 4. There are four types of modules:

Operation Switch Module: This module controls the operation mode of the dispatcher on a per-service level. In our implementation, the standard operation mode is *forward*, other modes are *buffer* or *reject*. The latter two modes are set to prevent the more expensive execution of the dispatch module when there are no suitable service hosts.

Dispatch Module: This module implements the actual dispatching strategy. It can access the load situation of service hosts and of other resources for the assignment of requests to service instances. Possible results of a dispatch strategy are an assignment of a request to a service instance, a command to initiate a service replication (see below), a reject command, or a buffer command. We implemented a strategy which assigns requests to the service instance on the least loaded service host based on the CPU load. We additionally implemented a more sophisticated strategy which handles the load of CPU and main memory on different types of resources (e.g., service hosts and database management systems) needed for the execution of a service. This strategy prevents overload situations not only on service hosts but also on other resources like DBMSs. Currently, we are working on performance experiments for these strategies.

Advisor Modules: Advisor modules are used to collect data for the dispatcher's view of the load situation of all relevant resources. We implemented advisor modules to measure the average CPU and memory load on service hosts (using the monitoring services) and on hosts running database management systems (using the monitoring applications). There are lots of reasonable different advisor modules. The simplest kind of advisor module only knows two conditions of a resource: available or unavailable. For service hosts, this could be done by a simple *ping* on the host running the ServiceGlobe system. More complex advisors can provide more detailed information like CPU or main memory load of a service host, or the load of a database management system depending on CPU, memory, disc I/O, and others.

Config Modules: The configuration modules are used to generate the configuration for new service instances. The modules can access the load situation archive which stores aggregated load information to find, e.g., the database host which was least loaded in the last few days. This is very beneficial if there are, e.g., several instances of a database system working on replicated data. Using historic load information, a new service instance can be advised to connect to the instance of the DBMS which had the lowest average load in the past.

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. After that, the service

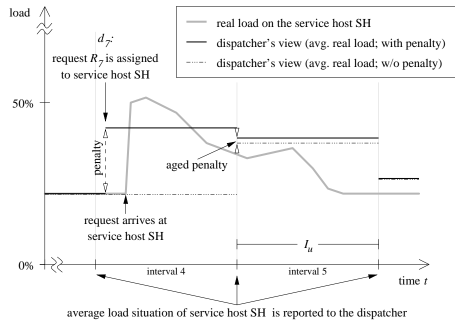

Fig. 5. Different Views of the Load Situation

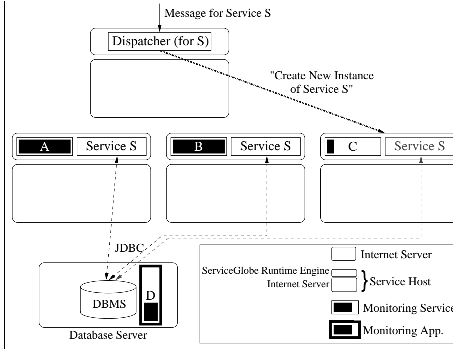

Fig. 6. Automatic Replication of Service S

instances are no longer contacted directly, but via the dispatcher service controlling the forwarding of the messages. 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., ServiceHostClusterZ, indicating that these service hosts are members of cluster Z. The dispatcher will automatically use these service hosts as soon as it notices the changes to the UDDI repository.

4.2 Load Measurement

The dispatcher's view of the load situation is updated at intervals of several seconds to prevent overloading the network. Thus, this view is constant between two updates. Therefore, a service host SH will still be considered having low load, even if several requests have been assigned to it after the last load update. Without precautions, the dispatcher might overload SH for this reason. To avoid these overload situations, the dispatcher adds "penalties" to its view of the load once a request is assigned. Figure 5 illustrates the load of SH, the load reported to the dispatcher (load without penalties), and the load with penalties.

The grey, thick line represents the load $L_{\mathrm{SH}}(t)$ of the service host SH. The dashed line represents the dispatcher's view $D'_{\mathrm{SH}}(t)$ of the load of SH which is the average load of SH over the last update interval of length I_u. This average load is calculated by SH and sent to the dispatcher at regular intervals. The function $int(t)$ calculates the number of the interval containing a given time t:

$$int(t) \;:=\; \lfloor t \,/ I_u \rfloor$$

The dispatcher's view can now be written as follows:

$$D'_{\mathrm{SH}}(t) \;:=\; \mathrm{avg}\left\{ L_{\mathrm{SH}}(t') \,\middle|\, int(t') = int(t) - 1 \right\}$$

The black, solid line shown in Figure 5 represents the dispatcher's view including penalties $D_{\mathrm{SH}}(t)$. The initial (maximum) value of a penalty (represented by $P^{m}_{\mathrm{SH,S}}$ in the equations) depends on the service S and the performance of the service host SH and is configurable. This way, every assignment of a request R_i, i.e., every dispatch operation (represented by $d_i, i \in \mathbb{N}$; d_7 in the figure), has an effect on the dispatcher's view of the load situation, immediately. If there is a load update from SH shortly after an assignment of a request R_i, but before SH started to process R_i, the associated penalty would

be lost if the dispatcher would replace its view with the reported load, because this load would not include load caused by R_i. Thus, the load reported by the load monitors and the dispatcher's view of the load situation are remerged using aging penalties: the penalties are decreasing over time and added to further load values reported by the service host until the penalties are zero. The time I_p until a penalty is zero is configurable and normally shorter than shown in the picture, e.g., twice the time a request R_i needs to arrive at SH plus the time SH needs to start processing R_i. After I_p, we assume that a request R_i arrived at SH and that the load caused by R_i is already included in the reported load, so that the dispatcher needs not to further add any penalties for R_i. Using our notation and defining $time(d_i)$ to indicate the time of the assignment d_i, $host(d_i)$ to indicate the destination host of the assignment d_i, and $service(d_i)$ to indicate the destination service of the assignment d_i, the view with penalties $D_{\mathrm{SH}}(t)$ can be calculated as follows: The penalty P_{d_i} for the assignment d_i is zero before the assignment. After I_p, it is zero again. In between this interval the penalty is calculated using a linear function $f_{d_i}(t)$ with the following constraints: $f_{d_i}(0) = P^m_{host(d_i),service(d_i)}$ and $f_{d_i}(I_p) = 0$.

$$P_{d_i}(t) := \begin{cases} 0 & \text{if} \quad t < time(d_i) \vee t > time(d_i) + I_p \\ f_{d_i}(t - \mathrm{time}(d_i)) & \text{else} \end{cases}$$

When receiving load updates from the service host SH, i.e., $t = x * I_u$ for $x \in \mathbb{N}$, the load including penalties is calculated by adding all aged penalties of assignments to this SH to the reported value:

$$Ass_{\mathrm{SH}} := \left\{ a \in \mathbb{N} \,\middle|\, host(d_a) = \mathrm{SH} \right\}$$

$$D_{\mathrm{SH}}(t) := D'_{\mathrm{SH}}(t) + \sum_{i \in Ass_{\mathrm{SH}}} P_{d_i}(t) \quad \text{if} \quad \exists x \in \mathbb{N} : t = x * I_u$$

Within an update interval, penalties of new assignments to SH, i.e., assignments done within the current update interval, are added to this load as soon as they occur:

$$NewAss_{\mathrm{SH}}(t) := \left\{ a \in Ass_{\mathrm{SH}} \,\middle|\, int(\mathrm{time}(d_a)) = int(t) \wedge t > \mathrm{time}(d_a) \right\}$$

$$D_{\mathrm{SH}}(t) := D_{\mathrm{SH}}\left(int(t) * I_u\right) + \sum_{i \in NewAss_{\mathrm{SH}}(t)} P^m_{\mathrm{SH},service(d_i)} \quad \text{if} \quad \forall x \in \mathbb{N} : t \neq x * I_u$$

4.3 Automatic Service Replication

If all available service instances of a static service[8] 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 a feature called automatic service replication. Figure 6 demonstrates this feature: 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.

[8] Dynamic services can be executed on arbitrary service hosts and need not be installed anyway.

4.4 High Availability / Single Point of Failure

Using several instances of a service greatly increases its availability and decreases the average response time. Just to get an impression about the high level of availability, we want to sketch this very simple analytical investigation. Assuming that the server running the dispatcher itself and the database server (in our example the database server is needed for service S) are highly available, the availability of the entire system depends only on the availability $\alpha_{ServiceHost} = \alpha$ of the service hosts. The availability of a pool of service hosts can be calculated as follows:

$$\alpha = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \quad (1) \qquad \alpha_{pool} = \sum_{i=1}^{N} \alpha^i (1-\alpha)^{(N-i)} = 1 - (1-\alpha)^N \quad (2)$$

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 $= 48$h and MTTR $= 12$h a pool with 8 members will only be unavailable about 1.5 minutes a year.

Because database management systems are very often mission critical for companies, there are different approved solutions for highly available database management systems [3, 9]. Thus, the remaining single point of failure is the dispatcher service. There are several possibilities to reduce the risk of a failure of the dispatcher. A pure software solution is to run two identical dispatcher services on two different hosts. Only one of these dispatchers is registered at the UDDI server. The second dispatcher is the spare dispatcher and it monitors the other one ("watchdog mechanism"). If the first dispatcher fails, the spare dispatcher modifies the UDDI repository to point to the spare dispatcher. If the clients of the dispatcher call services according to the UDDI service invocation pattern, any failed service invocation will lead to a check for service relocation. Thus, failures of the first dispatcher will lead to an additional UDDI query and an additional SOAP message to the second dispatcher. Of course, there are many other possible solutions which are adaptable for a highly available dispatcher service known from the fields of database systems [3, 9] and Web servers [4] including solutions based on redundant hardware, but this is out of the scope of this paper.

5 Related Work

The success of Web services results in a large number of commercial service platforms and products, e.g., the Sun ONE framework which is based on J2EE, Microsoft .NET, and HP Web Services Platform. Furthermore, there are research platforms like ServiceGlobe [12, 11] and SELV-SERV [2] which focus on certain aspects in the Web service area. In SELV-SERV, services with equal interfaces are grouped together into service communities, but no strategies for selecting services out of these service communities are presented. The project focuses rather on composing Web services using state charts. The eFlow system [5] models composite services as business processes, specified in eFlow's own composition language, and provides techniques similar to DSS. With dynamic service discovery, a composite service searches for services based on available metadata, its own internal state, and a rating function. Multiservice nodes allow to invoke several services in parallel, similar to DSS mode and result constraints, though

with different termination criteria. In contrast to eFlow, DSS allows the combination of all these different constraints in a flexible way. In addition, eFlow does not utilize standards like UDDI or WSDL for its adaptive techniques.

In [15] and [17], agent-based architectures are presented which provide service selection based on a rating system for Web services. In Jini [20], clients utilize a lookup service to discover services based on the Java interfaces they implement and service attributes. The lookup service's attribute search is limited to searching only for exact matches [16]. Extensions have been proposed to support, e.g., attributes providing context information about services [14] or more sophisticated match types [16]. Based on WSDL, WSIF [21] allows a Web service to select a specific port of a service it wants to invoke, i.e., its actual access point and the communication protocol and message format to use, at runtime. The selection is limited to the information provided by WSDL documents, as no service repositories like UDDI are consided.

A lot of work has been done in the area of load balancing, e.g., load balancing for Web servers [4] and load balancing in the context of Grid computing [8]. Grid computing is focused on distributed computing in wide area networks involving large amounts of data and/or computing power, using computers managed by multiple organizations. Our dispatcher is focused on distributing load between hosts inside a LAN. In contrast to dispatchers for Web servers [4], dispatchers for service platforms cannot assume that all requests to services produce the same amount of load, because the computational demands of different services might be very different. There are also commercial products available, e.g., DataSynapse [7] which offers a self-managing distributed computing solution. One of the key differences of this system is, that it works pull-based, i.e., hosts are requesting work, instead of using a dispatcher pushing work to the hosts. Additionally, DataSynapse requires an individual integration of every application, which is not necessarily an easy task for arbitrary applications.

6 Conclusion

In this work, we presented novel techniques for flexible and reliable Web service execution and deployment in dynamic environments. We introduced dynamic service selection which offers Web services the possibility to select and invoke services at runtime based on a technical specification of the desired service. We showed how constraints can be used to influence dynamic service selection. We also addressed load balancing and high availability issues by providing a generic, modular, and transparent dispatcher for load balancing including automatic service replication. We implemented these techniques within ServiceGlobe, an open Web service platform. For the future, we plan to work on caching of SOAP messages and to investigate context for Web services.

Acknowledgments

We would like to thank Wolfgang Becker und Thorsten Dräger of SAP's Advanced Infrastructure Program group for their cooperation.

References

1. SAP Keynote: Turning Vision into Reality: Customer Roadmaps to Lower TCO. `http://www.sap.com/community/events/2003_orlando/keynotes.asp`, 2003. SAPPHIRE '03.

2. B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proc. of the 18th Intl. Conference on Data Engineering (ICDE)*, pages 297–308, 2002.
3. R. Breton. Replication Strategies for High Availability and Disaster Recovery. *Data Engineering Bulletin*, 21(4):38–43, 1998.
4. V. Cardellini and E. Casalicchio. The State of the Art in Locally Distributed Web-Server Systems. *ACM Computing Surveys*, 34(2):263–311, 2002.
5. F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.
6. J. Clark and S. DeRose. XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999. W3C Recommendation.
7. DataSynapse Homepage. `http://www.datasynapse.com/`, 2003. DataSynapse, Inc.
8. Globus Project Homepage. `http://www.globus.org/`, 2003. Globus Project.
9. H. Hsiao and D. J. DeWitt. A Performance Study of Three High Availability Data Replication Strategies. In *Proc. of the Intl. Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 18–28, 1991.
10. M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proc. of the 18th Intl. Conference on Data Engineering (ICDE)*, pages 309–320, 2002.
11. 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.
12. 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.
13. W. Kieling. Foundations of Preferences in Database Systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 311–322, 2002.
14. C. Lee and S. Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In *2003 Symposium on Applications and the Internet (SAINT)*, pages 22–30. IEEE Computer Society, 2003.
15. E. M. Maximilien and M. P. Singh. Agent-based Architecture for Autonomic Web Service Selection. In *Workshop on Web-services and Agent-based Engineering (WSABE)*, 2003.
16. M. B. Møller and B. N. Jørgensen. Enhancing Jini's Lookup Service Using XML-Based Service Templates. In *13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 19–31. IEEE Computer Society, 2003.
17. S. K. Mostéfaoui and G. K. Mostéfaoui. Towards A Contextualisation of Service Discovery and Composition for Pervasive Environments. In *Workshop on Web-services and Agent-based Engineering (WSABE)*, 2003.
18. E. Rahm and G. Vossen, editors. *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*. dpunkt-Verlag, 2002.
19. 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)*, volume 2193 of *Lecture Notes in Computer Science (LNCS)*, pages 147–162, 2001.
20. J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.
21. Web Services Invocation Framework (WSIF). `http://ws.apache.org/wsif/`, 2003. The Apache Software Foundation.