A Method to Implement Location Transparency in a Web Service Environment

Xiaoshan Pan, PhD.
xpan@cadrc.calpoly.edu
Collaborative Agent Design Research Center, Cal Poly, San Luis Obispo, CA
CDM Technologies, Inc., San Luis Obispo, CA

Abstract

Location transparency offers some significant benefits in the areas of middleware, Service-Oriented Architecture (SOA) and Cloud Computing. However, methods for achieving location transparency in a Web service environment are scarcely presented in the literature. This paper introduces such a method by describing a design and HTTP protocol-based implementation of location transparency. A number of benefits, including support for the creation of a virtual platform and increased mobility, availability and scalability of services, are elaborated. Two significant capabilities - performance-based load balancing and failover - are demonstrated as part of the experimental results.

Key Words

Location Transparency; Web service; Service-Oriented Architecture; Cloud Computing; Virtual Platform; Intelligent Routing; Load Balancing; Failover

1. Introduction

In a Service-Oriented Architecture (SOA) environment, location transparency offers some significant benefits to service consumers, service providers and developers. When SOA is implemented using Web service technology, location transparency can be achieved through the construction of a SOA infrastructure[1] where Web services execute and interact with each other. Location transparency is an ability of a SOA infrastructure that enables service consumers and service providers to operate independently of their locations — a service consumer can consume a service without knowing where the provider is located, because the discovery of the location takes place at run-time.

From the perspective of a service consumer, location transparency creates the impression of a *virtual platform*, in which all services seem to reside within the same machine or programming space, while in reality the services may be widely distributed over a network (e.g., Internet). This also leads to the sense of a *Cloud* – "I send a request into the Cloud, and somehow it gets processed and a useful response comes back to me!" Therefore one practical usage of *virtual platform* is to enable a consumer to access remote services as though they were local (i.e., transparent access).

From the perspective of a service provider, location transparency offers advantages such as increased mobility, availability, and scalability. Location transparency enables a service consumer to break any dependency that it may have on a fixed location of a service provider.

[1] A SOA infrastructure refers to a service run-time environment that provides capabilities such as routing, location transparency, security, service mediation, and service orchestration to SOA based systems.

The service provider can be freely relocated, bringing the advantage of mobility. In turn, this allows a service provider to perform maintenance without causing service interruption by *switching on* a backup instance of the service at a different location while the service in production is taken off-line for maintenance. Additionally, when multiple providers of the same service contract exist, location transparency offers opportunities for the SOA infrastructure to perform load balancing and fault tolerance, which leads to increased service scalability. For example, when demand for a service increases, more instances can be created (e.g., through virtualization) and registered to the SOA infrastructure. When demand decreases, some service instances are taken down to free up resources for other usage.

Another benefit of location transparency is that the service location is eliminated as a concern for service consumer developers. Traditionally, the developers need the location and access details of a service which usually are specific to a service provider hosted at a physical location. With location transparency, a service provider is an abstract service contract (that can be implemented by multiple providers), and the developers are free to focus on solving business domain problems instead of making efforts to interface with (and later on be coupled with) a particular provider.

To achieve location transparency, binding the consumer with a provider must occur at run-time (instead of at design-time). More importantly, the binding needs to be dynamic—the binding should be changeable based on criteria such as the availability, performance, and service policy of service providers at any particular point in time.

2. Location Transparency in the literature

The concept of location transparency is not new. It has been explored in the area of middleware[2] research. Stal (2002) described using a proxy design pattern to achieve location transparency in a middleware:

> The basic idea behind this pattern is to introduce a proxy component as an intermediate layer between the client and the servant. The proxy resides within the address space of the client and implements exactly the same interface(s) as the servant… Using this approach, a client can remain oblivious to any details related to distribution, such as the servant location or communication protocol uses (p.72).

Fiege et al (2003) proposed to utilize publish/subscription mechanisms to achieve location transparency, which is "necessary to make existing applications mobile," and mobility is essential to the success of mobile computing, such as mobile services and devices. Belle et al (1999) described a naming and routing algorithm that could interconnect mobile entities and route messages between them, while the locations of the involved entities are transparent to each other.

The significance of location transparency also is emphasized by researchers from the SOA community. Channabasavaiah et al (2004) claimed that "SOA is an architecture with special properties, comprising components and interconnections that stress interoperability and location transparency" (p.21). Berbner et al (2005) described location transparency as "services should have their definitions and location information stored in a repository and be accessible by a

[2] Middleware is a piece of computer software that sits in-the-middle between application software, connecting software components or applications. Middleware aims to provide interoperability in support of a coherent distributed architecture and simplify complex distributed applications.

variety of clients that could locate and invoke the services irrespective of their location" (p.211). Srinivasan and Treadwell (2005) regarded location transparency as a means of conforming to one of the SOA principles – loose coupling, because it limits the coupling between services to interface agreement solely, not to some specific service implementations. Keen et al (2004) proposed an approach to use an Enterprise Service Bus (ESB) as an intermediary to "achieve location transparency by decoupling the client and service invocation" (p. 248). Brown (2008) mentioned a number of approaches to implementing location transparency in SOA, including:

- Proxy-based approach. Using this approach, "to the service user, the proxy presents what appears to be the service's interface…The proxy forwards all incoming requests to the real service interface and forwards replies from the service interface back to the service user through the proxy interface"(p.76).
- Message-based approach. This approach relies on an intermediary party – a message service broker – to facilitate communications between service consumers and service providers. "The message service interface is no longer tied to a specific destination. Instead, the message service provides a generic interface for sending and receiving messages regardless of the destination" (p.71). A service request waits in a message queue until a service provider picks it up and processes it. In so doing, the location of the service provider that processes the message is entirely transparent to the service consumer.
- Content-based approach. This approach also utilizes an intermediary party – a mediation service – to receive a service request and then forward the request to a chosen service provider. In this case, the mediation service selects a service provider for handling a request by examining the content of the request and matching it with a provider.

In the Cloud Computing paradigm, location transparency is one of the obvious features that a cloud provides. Mei et al (2008) talked about a "cloud user should not be aware of the distributed storage of data… and it is the cloud's responsibility to retrieve them for the user through location transparency" (p.468). This claim is also true when applying to the other types of resources that a cloud can provide, such as applications, platforms, and Web services. Vaquero et al (2009) listed "access transparency for the end user" as one of the primary Cloud characteristics.

However, regardless of the significance of location transparency to the areas of middleware, SOA, and Cloud Computing, how to implement location transparency in a Web service environment is scarcely presented in the literature. To date, the closest publicly-available documents on the subject are two patents, one by Loupia (2009) and the other by Chen (2009), both of which have obscured technical descriptions.

This paper presents a method for implementing location transparency as part of the capabilities of a SOA infrastructure in a Web service environment. To remain focused, the other aspects of the SOA infrastructure, such as service mediation, service security, and service orchestration, are not discussed. The rest of the paper is organized as follows: section 3 describes the design of a mechanism to achieve location transparency utilizing a Service Registry and an Intelligent Router; section 4 describes an HTTP protocol based implementation of location transparency; section 5 presents some of the experimental results; and, section 6 provides conclusions.

3. A design for location transparency

In this design, one primary component facilitating location transparency is a service registry. As far as its implementation is concerned, a service registry can be a database, a directory service, an XML file, or a UDDI[3] registry. A service registry provides a registration mechanism to service providers, enabling service consumers to discover a service provider in the registry and subsequently invoke the service provider. Figure 1 illustrates the basic idea of utilizing a service registry to facilitate location transparency. Three steps are involved: 1) a service provider is registered with a service registry; 2) a service consumer searches the service registry and discovers the service provider; and 3) the service consumer invokes the service provider. A key concept illustrated by this mechanism is that the binding between a service consumer and service provider can take place at run-time.

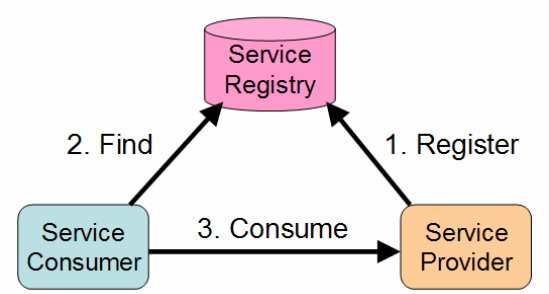

Figure 1: A service registry facilitates location transparency

Figure 1 suggests that a service consumer must perform the following steps to achieve location transparency at run-time:

1. Search a service registry for potential service providers;
2. Select a service provider if more than one is found (i.e., making routing decision); and
3. Send a request to the selected service provider and receive a response.

Assuming that steps 1 and 2 are performed by two software components, a Service Locator and a Router, respectively, we have Figure 2 below.

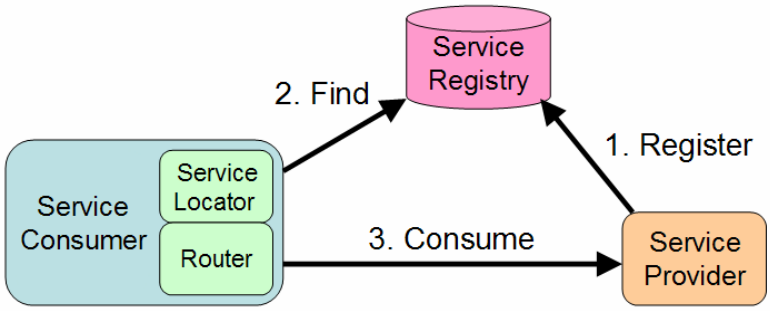

Figure 2: Service consumer embedded with a Service Locator and a Router.

Figure 2 implies that the Service Locator and the Router are part of a service consumer's internal logic, which may seem legitimate from the point of view of a single service consumer. However,

[3] UDDI refers to Universal Description, Discovery and Integration, a platform-independent, XML based registry for services to list themselves on the Internet. It enables businesses to publish service listings and discover each other and define how the services or software applications interact.

embedding these components inside a service consumer becomes problematic when multiple service consumers are involved. As illustrated in Figure 3, the Service Locator and the Router are implemented twice (i.e., Service Consumer A contains one implementation and the Service Consumer B contains the other), while the two implementations are technically identical. This introduces an implementation-redundancy issue, which is not only inefficient but also can quickly turn into a maintenance problem – just imagine hundreds of service consumers having to implement the Service Locator and the Router individually. Furthermore, from a design perspective, the focus of a service consumer is to work with business functions offered by a service provider, not finding service providers and making routing decisions.

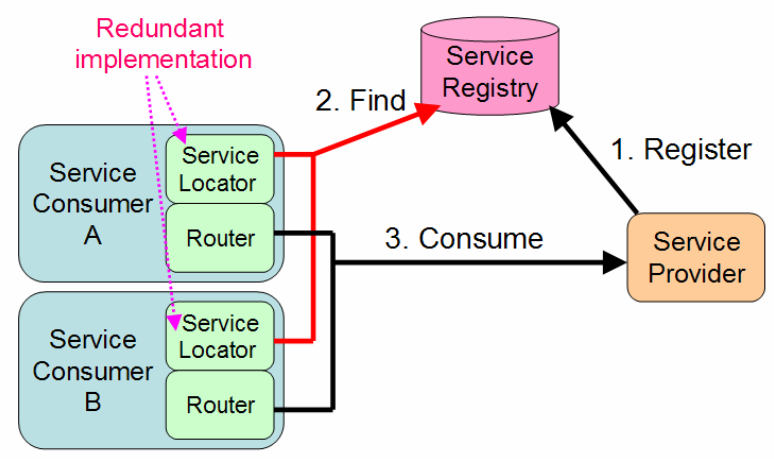

Figure 3: Redundancy implementation problem.

The SOA design disciplines advocate modularization of concerns in support of service reusability (Erl, 2008). Therefore a natural solution to the implementation redundancy problem, highlighted in Figure 3, is to make the Service Locator and the Router into separate modules that can be reused by any service consumer that would like to take advantage of location transparency. Let us call this reusable module an Intelligent Router (see Figure 4). This Router is considered intelligent because it *knows* how to locate a service provider dynamically, given a service request as its input.

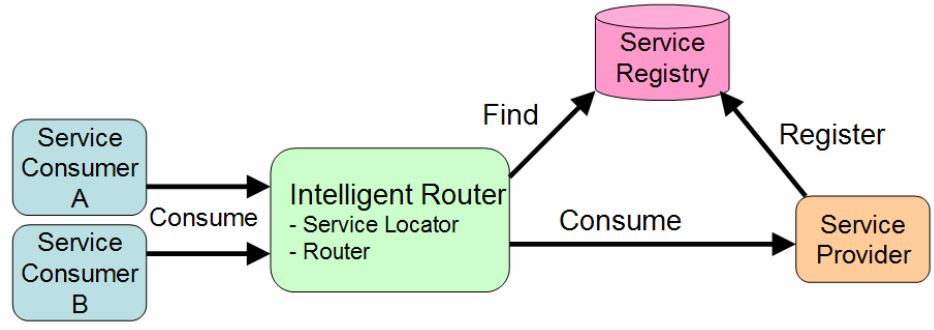

Figure 4: Utilizing an Intelligent Router to provide location transparency.

Compared to Figure 3, the design illustrated in Figure 4 simplifies the implementation of a service consumer. Furthermore, through the use of an Intelligent Router, location transparency is made available to both service consumers and service providers without them being concerned

with the implementations. Subsequently, the concept of a Virtual Platform is materialized (see Figure 5).

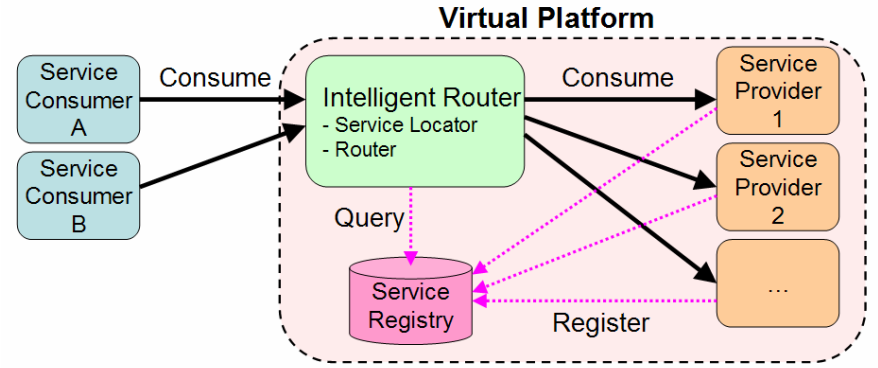

Figure 5: Creating a Virtual Platform through the use of an intelligent router and a service registry.

In Figure 5, all service providers may be dispersed throughout a network, implemented using different technologies, hosted in different environments, and removed/added to the registry at different times. However, from the perspective of a service consumer, all service providers appear as residing within the same *machine*, and all activities occurring in the *machine* are transparent to the service consumer. More importantly, when changes take place on the service providers' side, such as physical relocation of or update to a service, there is no need to make changes to the service consumers provided that the same service contracts are preserved.

4. An implementation of location transparency

The implementation described in this section assumes that Web services SOAP[4] or RESTful[5] services utilizing the HTTP protocol to transport messages, as they are currently the primary vehicles to implement SOA in the industry.

As discussed in the previous section, the core implementation of location transparency consists of two components: Service Registry and Intelligent Router. The Service Registry is well understood in the SOA community. For examples, ebXML[6] and UDDI are two industry initiatives that support the construction of a Service Registry. However, the concept of an Intelligent Router has not been fully entertained by researchers. Of the two sub-components of an Intelligent Router, the Service Locator component is relatively straightforward to construct,

[4] SOAP, or Simple Object Access Protocol, is a specification for exchanging structured information in the implementation of Web services. It relies on Extensible Markup Language (XML) as its message format, and other application layer protocols such as Remote Procedure Call (RPC) and Hypertext Transfer Protocol (HTTP) for transporting messages.

[5] REST, or Representational State Transfer, is a style of software architecture for distributed hypermedia systems. A RESTful Web service requires developers to use HTTP methods explicitly. Service contents are treated as resources that can be accessed and managed using the four basic HTTP methods – GET, POST, PUT, and DELETE.

[6] ebXML refers to Electronic Business using Extensible Markup Language and is a family of XML-based standards to provide an open, XML-based infrastructure that enables the global use of electronic business information in an interoperable, secure and consistent manner. The capabilities that it provides include publication and discovery of services electronically.

because a registry such as UDDI has a well-designed API that supports service publication and discovery. It is the Router component that poses a real challenge.

The Router must not only make intelligent routing decisions on the fly but also must act as a faithful *middle-man* between a service consumer and a service provider. From the point of view of a service consumer, the Router is a service provider, and from the point of view of a service provider, the Router is a service consumer. To perform this task, the Router must achieve *content-based routing*, meaning the content of a service request must be examined before a routing decision is made.

A traditional network router works in a very different way, which relies on a pre-defined *routing table* to perform its job, where the *routing table* is a set of fixed routing decisions, that contains lists of address mappings instructing the network router where to forward a message. In so doing, the content of a message never needs to be looked at.

With content-based routing, a router must: first, examine an incoming service request to extract information regarding *what* service contract the request applies to; second, search a service registry to discover any service providers who have implemented that service contract and *where* they are located; third, decide on a provider; fourth, create a new service request based on the original request; and finally, forward the service request to the chosen service provider. In principle, when the Intelligent Router constructs a new request from the original one, the payload of the request remains unchanged, with only the address information (i.e., addressee and return address) altered. However, there are cases where the Intelligent Router must modify the payload such as encrypting or decrypting the request or injecting security information into the message. One such example is illustrated in Figure 6.

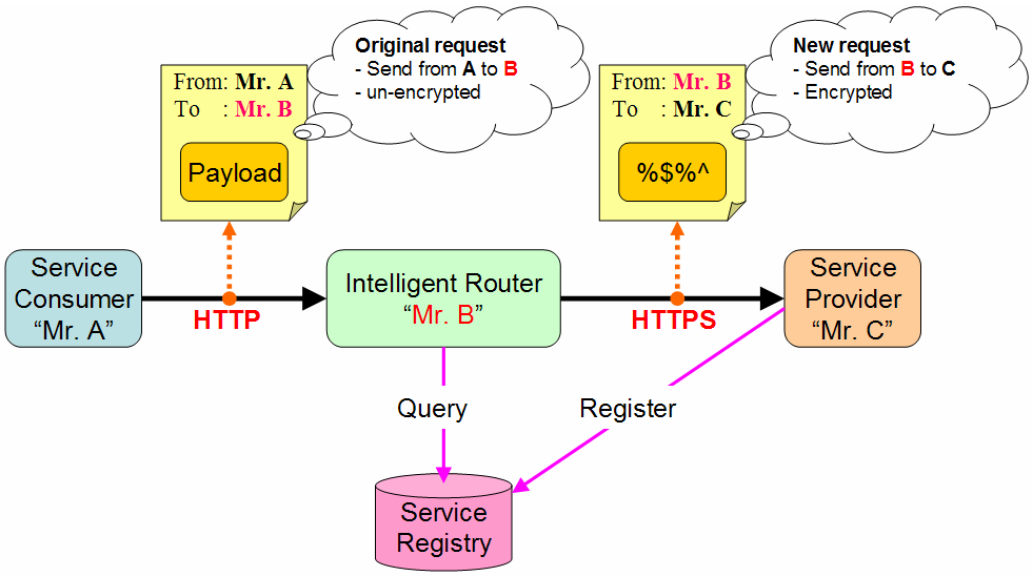

Figure 6: When performing 'content-based' routing, an Intelligent Router needs to construct a new request out of the original one.

In Figure 6, after the Intelligent Router (i.e., "Mr. B") receives a request from a Service Consumer (i.e., "Mr. A"), it makes the following modifications to the request:

- The address information of the request is changed from "From Mr. A To Mr. B" to "From Mr. B To Mr. C". "Mr. C" is the service provider chosen by the Router.

- The new request is encrypted using HTTPS, while the original request is not encrypted. This encryption step is necessary because the chosen Service Provider mandates that an incoming request be encrypted.

Similarly, when a response is received from the Service Provider, the Intelligent Router needs to construct a new response accordingly and forward it to the Service Consumer. All of the work that the Intelligent Router performs is transparent to both the Service Consumer and the Service Provider, and the Service Consumer and the Service Provider are not aware of each other's existence.

The following sections describe the implementation of a Service Registry and an Intelligent Router. The latter is composed of two sub-components: a Service Locator; and, a Router.

3.1.Service registry

The implementation discussed in this section uses OpenUDDI[7] as its service registry. OpenUDDI offers the following Application Programming Interfaces (API):

- **Publish**. This API allows a service provider to register a service with the registry so that the service can be discovered by a service consumer. In addition, this API allows a service provider to modify an existing entry in the registry.
- **Inquiry**. This API allows a service consumer to discover service providers that can satisfy its needs.

At a minimum, to publish a service instance to UDDI, a service provider must submit the following information to the UDDI registry through the Publish API: 1). Service provider's name, description and POC; 2). Service interface's name, description, contract (e.g., WSDL), and type; and, 3). Service instance's name, description, and physical end-point. An example is given as the follows.

Service provider:

Name:	Omega Cooperation
Description:	A software company that works on the Singularity technology
POC:	Dr. Omega, omega@singularity.com, Tel.: 1800.344.3444

Service interface:

Name:	Singularity Search Interface
Description:	A Web search interface into the Singularity knowledge base
Service contract:	available at https://www.singulariry-inc.com/search?WSDL
Service type:	SOAP-HTTP-Stateless

Service instance:

Name:	Singularity Search Service
Description:	A Web service that implements the Singularity Search Interface
End-point:	https://192.34.43.01:443/search-service/

Once the above information is submitted, the UDDI registry assigns a unique provider-key, interface-key, and service-key to the service provider, the service interface and the service instance, respectively. A service provider can modify the above information through the same API later on. For example, if the service provider would like to bring down the "Singularity

[7] OpenUDDI is a high performance UDDI v3 compliant service registry implementation. More information about OpenUDDI is available at: http://openuddi.sourceforge.net/

Search Service" at the location https://192.34.43.01:443 for maintenance without interrupting the consumers, the provider could do the following:

1. Activate a copy of the "Singularity Search Service" at another location, for example, https://84.32.45.03:443 – which is hosted at a different location.
2. Modify the UDDI entry such that the End-point of the service is https://84.32.45.03:443/search-service.
3. Bring down the service at https://192.34.43.01:443 and perform the maintenance.

Through the use of the Intelligent Router (introduced in the following section), the service requests previously hitting the service located at 192.34.43.01:443 would be routed to the new location at 84.32.45.03:443. Note that this location change is transparent[8] to the service consumers of the "Singularity Search Service" (see Figure 7). It is also worth noting that although the above scenario is easily achievable for stateless services more effort is required to accomplish the same for stateful services. To guarantee no service interruption to service consumers when working with a stateful service, the service consumer needs to detect a possible termination of a stateful interaction and re-send the stateful request(s) to the Intelligent Router.

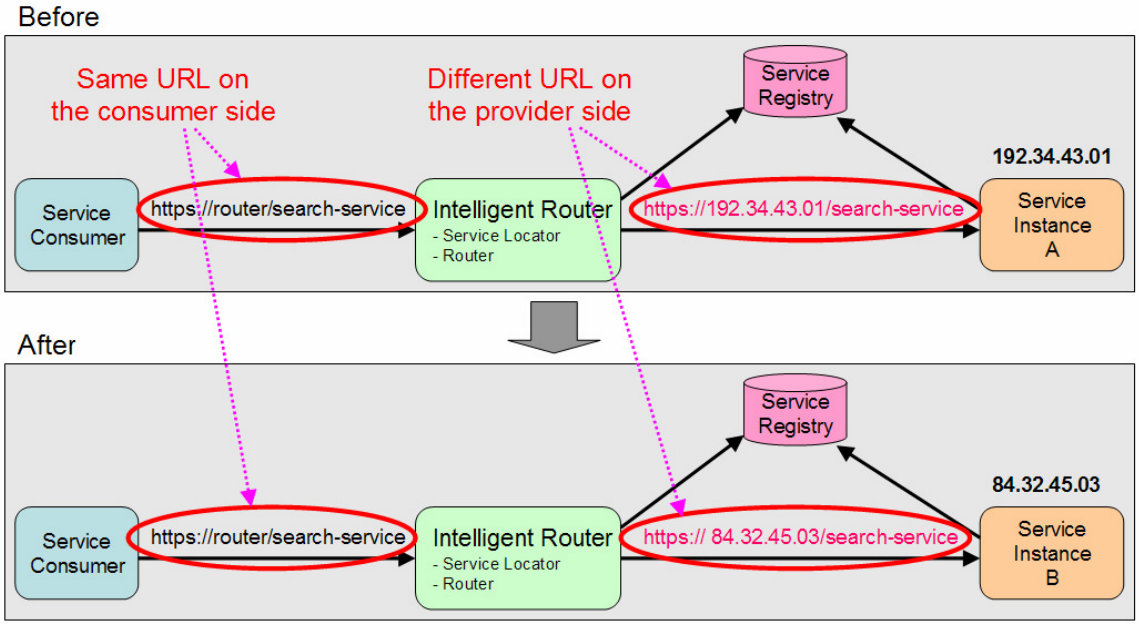

Figure 7: A service provider 'swaps' out a service instance without causing interruption to the service consumers.

3.2.Intelligent router

The Intelligent Router is composed of two sub-components: a Service Locator and a Router. Given a service request as the input, the former performs run-time queries to the OpenUDDI registry to discover service providers. The latter makes a routing decision, forwards the request to the chosen provider, and handles error conditions in the process.

[8] In order to maintain total continuity of the service, it is assumed the one of the following conditions is true: 1). the service is stateless, meaning the service does not maintain the state information of its consumers; or, 2). the service is stateful, however all state information is replicated when the copy of the service is activated.

3.2.1. Service locator

The Service Locator utilizes the UDDI Inquiry API to discover a service provider. A service request coming from a service contains a URL (Uniform Resource Locator), which is structured as follows:

[**Protocol**]://[**IP** or **DNS Name**]:[**Port**]/[**Resource URI**]

An actual example would be:

https://192.34.43.01:443/search-service/

Where "https" is the Protocol, "192.34.43.01" is the server IP, "443" is the Port, and "/search-service/" is the Resource URI (Uniform Resource Identifier).

In this implementation, a service consumer is not restricted to using Resource URI in the URL. It can send a service request to the Intelligent Router using any of the following URL formats:

1. https://router/**search-service/** → using a URI to identify a service
2. https://router/**interface-key-23432/** → using an interface-key to identify a service
3. https://router/**service-key-10009/** → using a service-key to identify a service

The Service Locator will resolve #1 and #2 above to discover the service instances that match the URI "/search-service/" and the interface-key "interface-key-23432". However, #3 above will match to exactly one service instance because each service-key is uniquely assigned to a service instance in UDDI.

Assume that there are two service instances (implementing a same service contract that has the key "interface-key-23432") registered with the following end-points:

1. https://192.34.43.01:443/search-service
2. https://84.32.45.03:443/search-service

Then a service request sent to either "https://router/search-service/" or "https://router/interface-key-23432/" will cause the Service Locator to find both service instances. Another Service Locator function is to sort service instances based on their performance metrics such that a more responsive service instance would show up higher in the list. The Service Locator obtains its service metrics by sending the *testing packets*, and determining *up* or *down* status along with service responding times. A more sophisticated performance metric may be obtained if the service has a service API allowing the Service Locator to collect detailed information about the usage of CUP, heap space, physical memory, and virtual memory of the machine where the service is hosted.

3.2.2. Router

The Router performs two functions: choosing a service instance if multiple instances are found by the Service Locator; and, forwarding a service request onto a chosen service instance. If a stateful service is involved, then the Router will ensure that the service requests with the same stateful session are routed to the same service instance. The Router accomplishes this by

maintaining a cache in memory to keep track of any stateful communication between consumer and provider[9].

The following procedure describes the logic performed by the Router:

PROCEDURE: Router Logic

1. Receive a service request ***R*** from a service consumer ***C***;
2. IF ***R*** is engaged in a stateful communication with an end-point ***E***
3. THEN GOTO #14;
4. ELSE GOTO #6;
5. END IF;
6. Invoke the Service Locator and receive a list of service end-points ***L***;
7. IF ***L*** is empty
8. THEN send a 404 error response to consumer ***C***, END;
9. ELSE
10. FOR each end-point ***E*** in ***L***
11. Establish connection with ***E***
12. IF the connection fails,
13. THEN GOTO #10;
14. ELSE Construct a new request based on the original request;
15. Forward the new request to ***E***;
16. Receive a response from ***E***;
17. Construct a new response based on the original response;
18. Send the new response back to the consumer ***C***, END;
19. END IF;
20. END FOR;
21. END IF;

Although the above procedure is generic in the sense that it is applicable to most types of services in a SOA environment, the implementation of steps #14 through #18 must be protocol-specific. The following elaborations are specific to the HTTP protocol.

The general form of a HTTP request is as follows:

[HTTP Method] [URI] [Protocol/Version]
[HTTP Headers]
[Message Body]

Figure 8 depicts a sample HTTP request message.

[9] At the time of registration, a service must specify whether it is a stateful. When the Service Locator finds a service instance for a service consumer, it informs the Router if the service instance is stateful. Therefore, the Router is able to determine whether the consumer and the service instance are engaged in stateful communication.

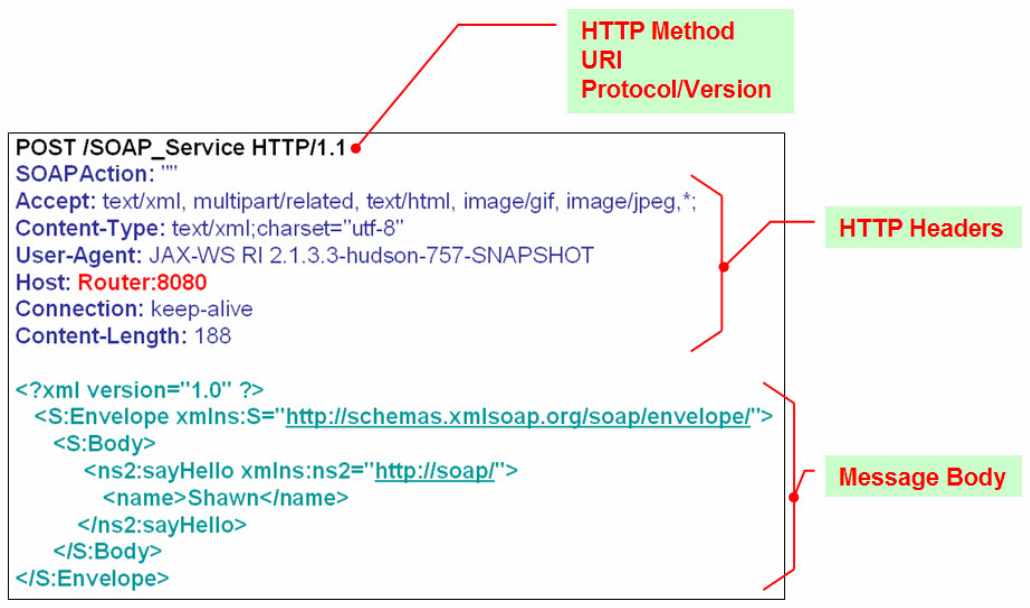

Figure 8: A sample HTTP request using SOAP.

The general form of a HTTP response is as follows:

[Protocol/Version] [Response Status]
[HTTP Headers]
[Message Body]

Figure 9 depicts a sample HTTP response message.

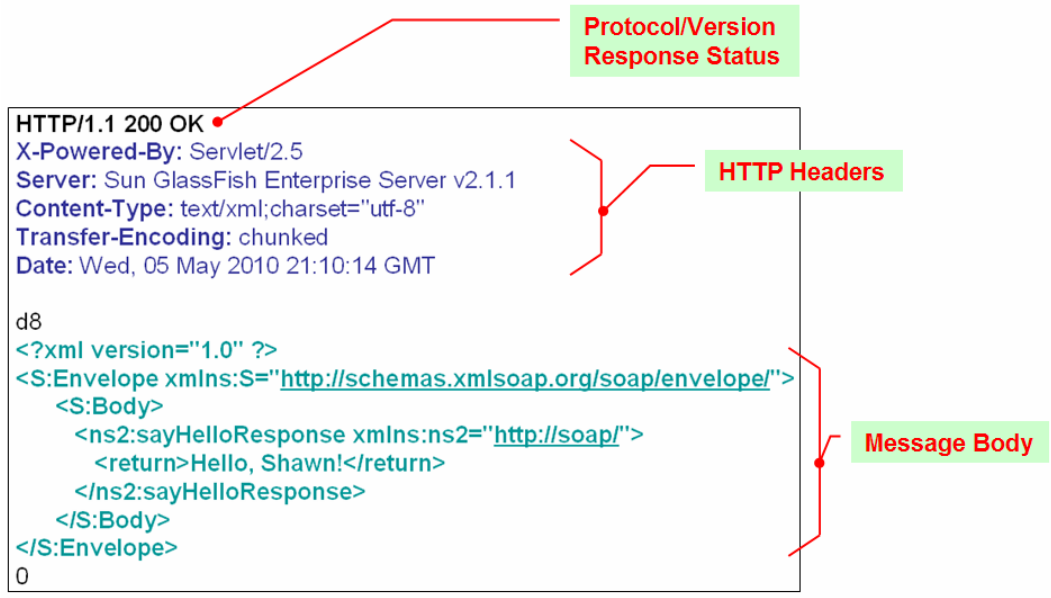

Figure 9: A sample HTTP response using SOAP

To implement step #14 and step #17 (i.e., constructing a new request and a new response), the Router needs to make changes to the *HTTP Headers* portion of a message. For example, if the service provider is hosted at "ProviderServer:9090," then the header "Host" in Figure 8 must be modified from "Router:8080" to "ProviderServer:9090," so that the correct service host is

reflected in the request. For a typical case, the HTTP headers for both a request and a response need to be modified, including:

- **Host** – specifies the Internet host and port number of the resource being requested;
- **Location** – is used to redirect the recipient to a location other than the one specified in the Request-URI;
- **Referer** – allows the client to specify the URI of the resource from which the Request-URI was obtained; and
- **Server** – is a Server response header field that contains information about the software used by the origin server to handle the request.

The implementation of step #15 and step #16 (i.e., sending a request to a provider and receiving a response) is relatively straight-forward. It requires the Router to write the service request to the *OutputStream* and read the service response from the *InputStream*, respectively, of the socket used by the Router to connect to a provider.

To implement step #11 (i.e., connecting to a provider), the Router establishes a connection with the provider using a network socket[10]. For example, using the Java language, a HTTP connection between the Router and a Provider can be created using the *java.net.Socket* class as shown in the following code sample:

```
Socket remoteServer = new Socket();
remoteServer.bind(null);
remoteServer.connect(new InetSocketAddress(IP, PORT),  TIMEOUT);
```

Where *IP* and *PORT* specify the network address of the provider, and *TIMEOUT* specifies the waiting time before a connection is terminated, in case the connection cannot be established.

For creating an HTTPS connection in Java, the *javax.net.ssl.SSLSocketFactory* class should be used to configure the Router with a proper server certificate and a certificate trust-store (to support Secure Socket Layer security),:

```
SocketFactory socketFactory = SSLSocketFactory.getDefault();
Socket remoteServer = socketFactory.createSocket();
remoteServer.bind(null);
remoteServer.connect(new InetSocketAddress(IP, PORT),  TIMEOUT);
```

Because the Router implementation described in this section does not need to examine the message body of an HTTP request or an HTTP response (other than performing encryption and decryption), the solution works generally for all HTTP-based messages (e.g., BlazeDS[11] messages).

[10] A network socket is an endpoint of a bi-directional inter-process communication flow across a computer network. Its address is identified by the combination of an IP address and a port number.

[11] BlazeDS is a server-based Java remoting and Web message technology that enables developers to easily connect to back-end distributed data and push data in real-time to Adobe Flex applications for responsive Rich Internet Application experiences.

5. Experimental results

The experiments described in this section involve using a stateless Web service and a stateful Web service as test services. The first service, *Compute_Prime_Stateless*, has one operation:

Operation: computePrime
Input: a positive integer number
Output: a list of prime numbers and the server IP

This service is stateless because it does not need to maintain any state information about the consumer of the service – the service receives an integer number and returns a list of prime numbers within the range as defined by the integer. There is no correlation between two separate service requests. In addition, the server IP that indicates the location of the server is returned for the sake of the experiment. For example, if the input is "7", then the service would return the list "2, 3, 5, 7" and "192.168.2.1", where the latter is the IP of the server that processes the request.

The second service, *Compute_Prime_Stateful*, has two operations:

Operation 1: sendInput
Input: an ID and a positive integer number
Output: none

Operation 2: compute
Input: an ID
Output: a list of prime numbers and the server IP

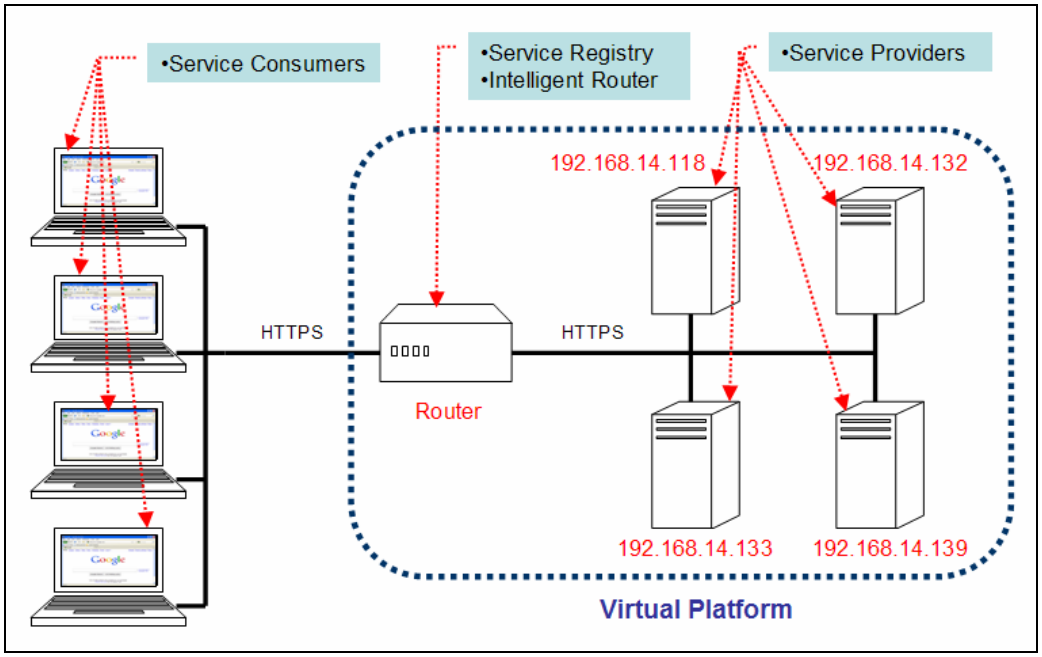

Figure 10: Experimental environment setup.

In order to utilize this service, a service consumer must send two consecutive requests to the service. The first request contains an ID and an integer number. After the service receives the request, it stores the ID and the number in its memory. The second request contains only an ID that the service uses to retrieve the corresponding integer in memory and to compute the prime numbers for that integer. If the ID does not exist in the memory, the service responds to the

consumer with an error. This is a stateful service, because the service must keep track of the state information across two separate service requests, and the two consecutive requests must be processed by the same service instance.

Figure 10 illustrates the configuration of the experimental environment. A Service Registry and an Intelligent Router are deployed to a server named *Router*. Four service providers are registered with the Service Registry. Each service provider has a unique IP address and hosts a *Compute_Prime_Stateless* service and a *Compute_Prime_Stateful* service. A service client sends service requests to the *Router* server only. The *Router* server is responsible for locating service providers to fulfill a service request. This configuration represents a *Virtual Platform*, because from the perspective of the service consumer all service providers reside on the *Router* server.

For the first experiment, four service consumer machines were configured to invoke the *Compute_Prime_Stateless* service concurrently. Each consumer machine sent out 1,000 consecutive requests (4,000 requests total), and each request caused a *Compute_Prime_Stateless* service to compute and return prime numbers between 1 and 100,000, along with the IP of the server that performed the computation. All consumers sent their requests to the following end-point (where the Intelligent Router resides):

https://Router:443/Compute_Prime/Compute_Prime_StatelessService

The *Router* server received the requests and performed load-balancing – distributing the requests to the four service providers based on their run-time performance scores. Table 1 shows the distributions of the requests across the four providers.

Table 1: Distribution of 4,000 stateless service requests across four providers

	Consumer 1	Consumer 2	Consumer 3	Consumer 4	Total
Provider 192.168.14.118	104	107	106	105	422
Provider 192.168.14.132	216	216	217	216	865
Provider 192.168.14.133	360	356	349	354	1287
Provider 192.168.14.139	320	321	328	325	1294

Similarly, the *Computer_Prime_Stateful* service was used for the second experiment. Each of the four consumer machines sent out 1,000 pairs of requests to the Router machine at the following end-point:

https://Router:443/Compute_Prime/Compute_Prime_StatefulsService

Each pair of requests consists of two consecutive requests that share the same HTTP session ID, which allows the Router to deliver the two requests to the same provider. In so doing, stateful interactions between consumers and providers are maintained. Table 2 shows the distributions of 4,000 pairs of stateful requests across the four providers.

Table 2: Distribution of 8,000 (i.e., 4,000 pairs) stateful service requests across four providers

	Consumer 1	Consumer 2	Consumer 3	Consumer 4	Total
Provider 192.168.14.118	143	142	141	142	568
Provider 192.168.14.132	231	233	230	232	926
Provider 192.168.14.133	323	321	323	322	1494
Provider 192.168.14.139	303	304	306	304	1217

The data shown in the Table 1 and the Table 2 leads to the following observations:

- Location transparency has been achieved for both the stateful and stateless services in the experiments. As far as a service consumer is concerned, there was only one provider and it resided on the server named *Router*. However, in the experiment there were multiple providers, and each was hosted on a different server.
- A performance-based load balancing capability has been achieved in these experiments. The provider with IP *192.168.14.133* has the best run-time performance, and the provider with IP *192.168.14.118* has the worst run-time performance.
- Location transparency is a suitable strategy for making a service scalable. If the demand of a service increases, more provider machines that host the service can be stood up to meet the demands. To make an additional service instance available to the consumers, the only configuration required is to register the service instance with the Service Registry.

Another significant feature supported by location transparency is failover. Specifically at runtime when multiple providers are available to support the same service contract, if one provider fails to process a request, the subsequent requests can be routed to other providers. Moreover, if a service consumer is configured to resend a stateless service request, or all requests involved in a stateful session, when a server error is detected while processing the request, then subsequent requests along with any failed requests can be recovered. In this way, it is possible to swap service providers at runtime without causing service interruptions.

In the next experiment, using the same environment illustrated in Figure 10, two service consumer machines were configured to send stateless requests (1,000 consecutive requests for each consumer) and the other two service consumer machines were configured to send stateful request pairs (1,000 pairs for each consumer) to the Router machine for processing. Each stateless request or stateful request pair will cause a service provider to compute all prime numbers between 1 and 100,000. In addition, the service consumers were configured to resend a stateless request or stateful request pair if a server error was detected. To simulate server error conditions, every 30 seconds a service provider was randomly chosen to disconnect from the network and reconnect back to the network 10 seconds later. Table 3 lists the distribution of both stateless and stateful requests that were successfully processed even though all the service providers failed to respond occasionally. As the results indicate, no single request failed to be processed even when error conditions took place.

Table 3: Distribution of both stateless and stateful requests that were successfully processed when service providers failed to respond occasionally

	Stateless Consumer 1	Stateless Consumer 2	Stateful Consumer 1	Stateful Consumer 2	Total
Provider 192.168.14.118	200	205	239	239	883
Provider 192.168.14.132	451	458	384	390	1683
Provider 192.168.14.133	115	106	140	148	2566
Provider 192.168.14.139	234	231	237	223	925
Requests re-sent	10	12	12	14	48

The data shown in the Table 3 demonstrates that a robust failover capability can be developed based on location transparency. When the failover capability works together with the load balancing capability, improved service availability can be achieved in a potentially unreliable computing environment, characterized by fluctuating network connectivity and occasional server failures.

6. Conclusions

Although the significance of location transparency is recognized in the areas of middleware, SOA, and Cloud Computing research, methods for achieving location transparency in a Web service environment are scarce. This paper presents such a method by describing a design and HTTP protocol-based implementation of location transparency in a Web service environment. In the design, the utilization of a service registry and an intelligent router is elaborated. An HTTP protocol-based implementation is presented and some experimental results are discussed. The benefits of location transparency demonstrated, include: 1) support for the creation of virtual platforms; 2) increased mobility, availability and scalability for service providers; and, 3) the elimination of service location as a concern for service consumers. In addition, two significant capabilities are established through the use of location transparency and are demonstrated, namely: performance-based load balancing; and, failover.

References:

Brown, P., Implementing SOA: Total Architecture in Practice. Addison-Wesley: Boston. 2008. ISBN 0321504720.

Berbner, R., Grollius, T., Repp, N., Heckmann, O., Ortner, E., Steinmetz, R., “An Approach for the Management of Service-Oriented Architecture (SOA) Based Application Systems. Proceedings of the Workshop Enterprise Modeling and Information Systems Architectures (EMISA 2005). October 2005, 208–221.

Belle, W., Verelst, K., and D’Hondt, T., “Location Transparent Routing in Mobile Agent Systems—Merging Name Lookups with Routing,” Proc. 7th IEEE Workshop Future Trends of Distributed Computing Systems (FTDCS 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 207-212.

Chen, J., “Reroute of a Web Service in a Web Based Application,” Patent, Greenblum & Bernstein PLC, 2009. Available at: http://www.faqs.org/patents/app/20090094314

Channabasavaiah, K., Holley, K., and Tuggle, E., “Migrating to a service-oriented architecture,” white paper, IBM, April 2004.

Erl, T., SOA: Principles of Service Design. Prentice Hall: New York. 2008. ISBN0132344823.

Fiege, L., Gartner, C., Kasten, O., and Zeidler, A., "Supporting Mobility in Content-Based Publish/Subscribe Middleware," in Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003). Rio de Janeiro, Brazil, 2003, pp. 103-122.

Keen, M., Acharya, A., Bishop, S., Hopkins, A., Milinski, S., Nott, C., Robinson, R., Adams, J., and Verschueren, P., “Patterns: Implementing an SOA using an Enterprise Service Bus”. IBM Redbook, July 2004.

Liupia, D., “Method of Redirecting Client Requests to Web Services,” Patent, Hoffman Warnick LLC, 2009. Available at: http://www.faqs.org/patents/app/20090019106

Mei, L., Chan, W., and Tse, T., "A Tale of Clouds: Paradigm Comparisons and Some Thoughts on Research Issues," Asia-Pacific Services Computing Conference (APSCC '08), Yilan, Taiwan, December 2008, 464-469.

Srinivasan, L., and Treadwell, J., "An overview of service-oriented architecture, web services and grid computing," November 2005. Available at: http://devresource.hp.com/drc/technical papers/grid soa/SOA-Grid-HP.pdf.

Stal, M., "Web Services: Beyond Component-based Computing". Communications of the ACM, October 2002. Vol. 45, No. 10, pp. 71-76.

Vaquero, L., Rodero-Marino, L., Caceres, J., Lindner, M., "A break in the clouds: towards a cloud definition," SIGCOMM Computer Communication Review, 39 (2009), 137–150.