

WEB SERVICE DISCOVERY PROCESSING STEPS

Wolfgang Hoschek
CERN IT Division,
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

ABSTRACT

The most straightforward but also most inflexible configuration approach for invocation of remote services is to hard wire the location, interface, behavior and other properties of remote services into the local application. Loosely coupled decentralized systems call for solutions that are more flexible and can seamlessly adapt to changing conditions. While advances have recently been made in the field of web service specification, invocation and registration, the problem has so far received little systematic conceptual attention. In this paper, we outline seven web service problem areas and their associated processing steps, namely *description*, *presentation*, *publication*, *request*, *discovery*, *brokering* and *execution*. We propose a simple grammar (SWSDL) for *describing* network services as collections of service interfaces capable of executing operations over network protocols to endpoints.

A service must *present* its current (up-to-date) description so that clients from anywhere can retrieve it at any time. A registry for *publication* and *query* of service and resource presence information is outlined. Reliable, predictable and simple distributed registry state maintenance in the presence of service failure, misbehavior or change is addressed by a simple and effective soft state mechanism. The notions of *request*, *resource* and *operation* are clarified. We outline the *discovery* step, which finds services implementing the operations required by a request. The *brokering* step determines an invocation schedule, which is a mapping over time of unbound operations to service operation invocations using given resources. Finally, the *execution* step implements a schedule, by using the supported protocols to invoke operations on remote services.

KEYWORDS

Service Discovery, Web Services.

1. INTRODUCTION

An enabling step towards increased Internet software execution flexibility is the *web services* vision [1,2] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible, adaptive and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. This helps to enable distributed WAN applications spanning administrative domains, such as electronic market places, distributed auctions, Peer-to-Peer (P2P) file sharing systems, monitoring infrastructures for large-scale clusters of clusters, as well as instant messaging and news services.

For example, the European DataGrid (EDG) [3,4] is a global software infrastructure that ties together a massive set of people and computing resources spread over hundreds of laboratories and university departments. This includes thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [5]. A data-intensive High Energy Physics analysis application sweeping over Terabytes of data looks for remote services that exhibit a suitable combination of characteristics, including appropriate interfaces, operations and network protocols as well as network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost. It is thus of critical importance to develop capabilities for rich service discovery as well as a query language that can support advanced resource brokering. Examples for a service are:

- A replica catalog implementing an interface that, given an identifier (logical file name), returns the global storage locations of replicas of the specified file.
- A replica manager supporting file replica creation, deletion and management as well as remote shutdown and change notification via publish/subscribe interfaces.
- A storage service offering GridFTP transfer, an explicit TCP buffer size tuning interface as well as administration interfaces for management of files on local storage systems. An auxiliary interface supports queries over access logs and statistics kept in a registry that is deployed on a centralized high availability server, and shared by multiple such storage services of a computing cluster.
- A gene sequencing, language translation or an instant news and messaging service.

While advances have recently been made in the field of web service specification [6], invocation [7] and registration [8], the problem has so far received little systematic conceptual attention. A key question then is:

- *What distinct problem areas and processing steps can be distinguished in order to enable flexible remote invocation in the context of web service discovery?*

This paper outlines seven web service problem areas and their associated processing steps, namely *description, presentation, publication, request, discovery, brokering* and *execution*.

2. DESCRIPTION

As communications protocols and message formats are standardized on the Internet, it becomes increasingly possible and important to be able to describe communication mechanisms in some structured way. A service description language such as the Web Service Description Language (WSDL) [6] addresses this need by defining a grammar for describing network services as collections of service interfaces capable of executing operations over network protocols to endpoints. Service descriptions provide documentation for distributed systems and serve as a recipe for automating the details involved in application communication. In contrast to popular belief, a web service is neither required to carry XML messages, nor to be bound to SOAP [7] or the HTTP protocol, nor to run within a .NET hosting environment, although all of these technologies may be helpful for implementation. For clarity, service descriptions in this paper are formulated in the Simple Web Service Description Language (SWSDL), which is a modified and strongly simplified version of WSDL.

Description is the process of defining metadata for a thing that allows one to reason about the thing itself. In our context, sufficient service description metadata needs to be defined (and published) that allows a client to start communicating with the service. In support of this goal one describes in a structured manner

- Which interfaces a service offers
- Which operations and arguments are defined on an interface
- How operations and arguments are bound (mapped) to network protocols and endpoints
- Other related metadata relevant for discovery, such as Quality of Service descriptions, current network load, host information, etc.

We note that the concept of a service is a logical rather than a physical concept. The service interfaces of a service may, but need not, be deployed on the same host. They may be spread over multiple hosts across the LAN or WAN and even span administrative domains. This notion allows speaking in an abstract manner about a coherent interface bundle without regard to physical implementation or deployment decisions. We speak of a *distributed (local) service*, if we know and want to stress that service interfaces are indeed deployed across hosts (or on the same host). Typically, a service is persistent (long lived), but it may also be transient (short lived, temporarily instantiated for the request of a given user).

We now use an example to informally introduce a modified and strongly simplified form of WSDL. SWSDL describes the interfaces of a distributed service object system. For simplicity, it offers neither a class concept nor interface inheritance. In SWSDL, a service description defines a *service* as a set of related

service interfaces. A service interface has an *interface type*. An interface type defines a set of *operations* and *arguments*. The interface type can be used to check whether a service interface conforms to some well-known standard. An operation is bound to one or more protocols and network endpoints via *binding* definitions. As an example, assume we have a simple scheduling service that offers an operation `submitJob` that takes a job description as argument. The function should be invoked via the HTTP protocol. A valid service description reads as follows:

```
<service>
  <interface type = "http://gridforum.org/interface/Scheduler-1.0">
    <operation>
      <name>void submitJob(String jobdescription)</name>
      <allow> http://cms.cern.ch/everybody </allow>
      <bind:http verb="GET" URL="https://sched.cern.ch/submitjob"/>
    </operation>
  </interface>
</service>
```

The description above states that the service interface is of a scheduler type. The precise scheduler type, including syntax and semantics of operations, is identified by the URL `http://gridforum.org/interface/Scheduler-1.0`. Next, we define `submitJob` as being the name of the submit operation and input and output arguments of type `String`. The operation is bound to the HTTP protocol, and can be invoked by sending an HTTP GET request to the URL `https://sched.cern.ch/scheduler/submitjob` (subject to local security policy). Only members of the CMS virtual organization are allowed to invoke this operation.

3. PRESENTATION

Having outlined the structure of a service description, we now turn to the problem of description presentation. Clearly, clients from anywhere must be able to retrieve the current (up-to-date) description of a service. Hence, a service needs to present (make available) to clients the means to retrieve the service description. To enable clients to query in a global context, some identifier for the service is needed. Further, a description retrieval mechanism is required to be associated with each such identifier. Together these are the bootstrap key (or handle) to all capabilities of a service.

In principle, identifier and retrieval mechanisms could follow any reasonable convention. In practice, however, a fundamental mechanism such as service discovery can only hope to enjoy broad acceptance, adoption and subsequent ubiquity if integration of legacy services is made easy. The introduction of service discovery as a new and additional auxiliary service capability should require as little change as possible to the large base of valuable existing legacy services, preferable no change at all. It should be possible to implement discovery-related functionality without changing the core service. Further, to help easy implementation the retrieval mechanism should have a very narrow interface and be as simple as possible. The service description concept comes to our help. In support of these requirements, we logically separate core service functionality and presentation functionality into separate service interfaces. Further, the identifier is chosen to be a URL, and the retrieval mechanism is chosen to be HTTP(S). We define that an HTTP(S) GET request to the identifier must return the current service description (subject to local security policy). In other words, a simple hyperlink is employed. In the remainder of this paper, we will use the term *service link* for such an HTTP URL identifier enabling service description retrieval.

4. PUBLICATION

Publication is the process of making the presence of services, resources, user communities and other metadata known to potential clients. In this specific context, it is the process of making a service identifier and description retrieval mechanism (in practice a service link) known to potentially interested clients so that they can retrieve the service description and use it. We are concerned with the basic capability of

making a service link (and hence service description) *reachable* for clients. To this end, service links are collected in one or more well-known registries, which are databases that can be queried by clients. Registries for organizations or communities with special interests serve a similar purpose as link list web pages and top-level organizational web pages: Many clients use well known and authoritative websites (registries) as entry points for browsing and searching because they mostly contain relevant hyperlinks (service links) for the given target community. Consequently, a particular community can discover information relevant to its interests.

When a service starts up, it announces its presence by invoking a `publish` operation on the registry. The operation takes as argument a service link to identify the service attempting publication. The registry appends the service link if it is not already present. Conversely, when a service shuts down it announces its unavailability by invoking a `depublish` operation on the registry. The registry removes the service link if it is present.

Clients can query the registry by invoking a query operation, for example using the powerful and expressive XQuery and XPath languages standardized by the W3C. For simplicity of exposition, here we describe the simplest possible query operations (“select all” style), forming the basis on top of which sophisticated query interfaces can be layered. The simplest query operation (`getLinks`) takes no arguments and returns the set of all known service links. An example result set for a query reads:

```
<tupleset>
  <tuple link="http://sched001.cern.ch/getServiceDescription"/>
  <tuple link="http://sched.infn.it:8080/getServiceDescription"/>
  <tuple link="http://repcat.cern.ch/getSrvDescription?id=4711"/>
</tupleset>
```

To retrieve the service descriptions of a result set, a client needs to establish a network connection for each service link in the result set. In principle, this is no problem, but in practice it can lead to prohibitive latency, in particular in the presence of large result sets. This is due to the very expensive nature of secure (and even insecure) connection setup. To address this problem, we define an additional query operation (`getTuples`) that returns service descriptions instead of associated service links. A registry implementation can use caching to reduce the number of connection setups and/or can use keep-alive connections to minimize setup time. As a consequence, this query operation may return outdated descriptions. More importantly, it may return only a partial result set; excluding any service descriptions the registry is not authorized to retrieve (service links are returned instead). This is the case if a security sensitive publisher refuses to delegate authorization to the registry, but only allows select clients to retrieve its service description. An example result set with two normal services and one replica catalog service refusing trust delegation reads:

```
<tupleset>
  <tuple link="http://sched001.cern.ch/getServiceDescription">
    <content>
      <service> service description A goes here </service>
    </content>
  </tuple>

  <tuple link="http://sched.infn.it:8080/getServiceDescription">
    <content>
      <service> service description B goes here </service>
    </content>
  </tuple>

  <tuple link="http://repcat.cern.ch/getSrvDescription?id=4711">
  </tuple>
</tupleset>
```

5. SOFT STATE PUBLICATION

This section discusses mechanisms for reliable, predictable and simple distributed registry state maintenance. In a system composed of a very large number of services, the mean time between failures is small. Recall that the previous section proposed a model in which services explicitly publish and de-publish as appropriate. We ignored the fact that services often fail or misbehave, leaving a registry in an inconsistent state. For example, a service that crashes may not de-publish with the registry, and hence clients may unnecessarily discover and try to contact an unavailable service repeatedly. Similarly, a service may be reconfigured to change its service link (and service description), yet it may forget to update all registries with which it is already associated. Further, a registry may change its authorization policy and, as a result, an already published service may suddenly no longer be in the position to de-publish itself. All these situations leave inconsistent or stale registry state behind. It is difficult for a registry to detect such situations and to determine when and how they can be resolved. For example, one can envision a strategy in which a registry drops a service link if a client (or perhaps itself) finds a service to be unavailable. However, the unavailability may be due to an authorization policy denying access to some but not all clients, or due to problems in a small network segment or simply due to service reboot. The service owner may be offended and claim violation of a service level agreement (SLA), because, in his opinion, there is no reason for dropping its service. Even worse, the service owner might not even notice for quite some time that he has been dropped. To summarize, so-called *hard state* based distributed information systems populated from many independent autonomous and heterogeneous distributed sources typically evolve quickly into garbage dumps where valid information is hard to distinguish from trash, decreasing overall utility dramatically.

Elaborate mechanisms can be designed to cope with the problems of reliable and consistent state maintenance. Such mechanisms typically face many complex and subtle problems. However, one can elegantly avoid much complexity by using a simple *soft state* mechanism for reliable distributed garbage collection: State established at a remote location may eventually be discarded unless refreshed by a stream of subsequent confirmation notifications [9]. In this manner, component failures and changes are tolerated in the normal mode of operation rather than addressed through a separate recovery procedure [10]. Lack of refresh indicates service failure, shutdown or change.

The responsibility for state maintenance is displaced by moving it from the registry to the publishing services. Registries keep service links (and perhaps also descriptions) as soft state, that is, they are kept for a limited amount of time only. Service links are tagged with time-to-live tokens (TTLs). Service links are expired and dropped unless explicitly renewed via periodic publication, henceforth termed *refresh*. Services refresh by essentially saying, “*I am still here*”. Consequently, services can crash, stop, be added or changed without leaving stale state behind indefinitely.

6. REQUEST AND DISCOVERY

Let us clarify some terminology surrounding the formulation of requests from clients to use network attached third party functionality. Clients formulate *requests*. Examples are: “submit job”, “compute Pi with accuracy of 10000 decimals”, and “retrieve result of HTTP GET to a given URL”. Another example is an analysis job consisting of a sequence of operations. It first uses a file transfer service (to stage input data from remote sites), next a replica catalog service (to locate an input file replica with good data locality), then a job execution service (to run the analysis program), and finally again a file transfer service (to stage output data back to the user desktop).

Resources are things that can be used for a period of time, and may or may not be renewable. They have owners, who may charge others for using resources, and they can be shared or be exclusive. Examples include disk space, network bandwidth, specialized device time, and CPU time [11]. Resources are made accessible through the operations of services. *Operations* are consumers of resources.

Requests are hierarchical entities, and may have recursive structure; i.e., requests can be composed of sub requests or operations, and sub requests may themselves contain sub requests. The leaves of this structure are operations. The simplest form of a request is one containing a single operation. The definition is derived

from [11]. Sometimes complex constraints and preferences formulated in a request description language accompany requests.

Having formulated a request, the discovery step finds services implementing the operations required by a request. More precisely, for each operation of a request of a given user, the discovery step searches one or more registries and produces *candidate services*, which are services (more precisely: service descriptions) that implement the operation on top of a given set of protocols. The simplest form of candidate contains a single service implementing a single operation on top of a single protocol.

It is often necessary to use several services in combination to implement the operations of a request. For example, a request may involve the combined use of a file transfer service (to stage input and output data from remote sites), a replica catalog service (to locate an input file replica with good data locality), a request execution service (to run the analysis program), and finally again a file transfer service (to stage output data back to the user desktop). Hence, discovery often involves querying for several types of operations or services.

7. BROKERING

For each operation of a request of a given user, the previous discovery step produces a set of candidate services that implement the operation. In the following brokering step, more or less sophisticated techniques are used to refine the selection and determine an invocation schedule. *Schedules* (also termed *execution plans*) are mappings over time of unbound operations to service operation invocations using given resources. One maps operations, not requests, because requests are containers for operations, and operations are the actual resource consumers [11]. The simplest schedule contains a single service operation.

The brokering step can be as simple as randomly picking a single service from the candidates, or as sophisticated as initiating a complex auction where participating services place bids and negotiate a resolution based on economic models, Quality of Service (QoS) and/or Service Level Agreements (SLAs). Consider a less ambitious example where, in an attempt to minimize response time, the brokering step of a job scheduler compares the current CPU load of candidate job execution services.

As mentioned above, it is often necessary to use several services in combination to implement the operations of a request. In such cases it is often helpful to consider correlations. For example, a scheduler for data-intensive requests may look for input file replica locations with a fast network path to the execution service where the request would consume the input data. If a request involves reading large amounts of input data, it may be a poor choice to use a host for execution that has poor data locality with respect to an input data source, even if it is very lightly loaded.

An advanced job scheduler typically also matches execution services against query patterns describing job requirements such as desired operating system and computer architecture type, minimum main memory size, disk quota, availability and connectivity to third party services like database engines, etc. Requests with complex constraints and preferences, for tasks like job submission, are augmented with a structured request description language for matching and ranking.

As can be seen, advanced brokering for tasks like job scheduling often requires additional information not available as part of service descriptions. Such additional information must be gained from other data sources. Such data sources for brokering may follow and respect a single globally standardized data and query model. In practice, however, non-uniform special-purpose data sources are often involved. This is due to the heterogeneous nature of large distributed cross-organizational systems such as the Grid, the large variety of use cases, brokering strategies and query types as well as strongly varying data freshness and data aggregation requirements. For example, brokering information includes very slowly changing data, such as the type of operating system, or more frequently changing quantities, such as the number of running jobs or the current CPU utilization. In addition, the brokering process may be highly application specific

and due to its complexity not expressible in any known query language. For example, it is hard to envisage that a negotiation process involved in distributed auctions can be expressed as a query (rather than an algorithm). In special-purpose areas, special matchmaking mechanisms have been developed [12].

In some cases more or less advanced resource reservations on a set of services can accompany the brokering step in order to help ensure consistent execution semantics for the operations of a request, or to guarantee a certain Quality of Service (QoS). For a detailed discussion of Quality of Service and advanced reservation see [13]. Note that services controlling a domain can commit resources with authority, whereas services outside a control domain cannot do so. For example, a local scheduler managing a cluster is in the position to make definitive statements about its resource usage and policy in general. A cross-organizational global scheduler, on the other hand, does not own any cluster, and hence cannot commit resources with authority. Such a global scheduler can only compute schedules based on assumptions and educated guesses.

8. EXECUTION

In the previous brokering step, more or less sophisticated techniques are used to determine an invocation schedule from candidate services. The execution step implements a schedule. The service descriptions of the operations of a schedule are parsed, and the supported protocols are used to invoke operations on remote services. In an attempt to cover a broad range of existing and future protocols, *invocation* is understood very broadly. For example, the operation to be invoked may be a SOAP operation carried over BEEP or HTTP(S), but it may also simply mean issuing one of the standard commands supported by the FTP or SMTP protocol (e.g. DELETE, GET).

9. RELATED WORK

The ANSA project was an early collaborative industry effort to advance distributed computing. It defined trading services [14] for advertisement and discovery of relevant services, based on service type and simple constraints on attribute/value pairs. The CORBA Trading service [15] is an evolution of these efforts.

UDDI (Universal Description, Discovery and Integration) [8] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL service descriptions. It does not offer a dynamic data model. It is not based on soft state, which limits its ability to dynamically manage and remove service descriptions from a large number of autonomous third parties in a reliable, predictable and simple way. Only key lookups with primitive qualifiers are supported, limiting the kind of discovery patterns that can be implemented.

The Jini Lookup Service [16] is located by Java clients via a UDP multicast. The network protocol is not language independent because it relies on the Java-specific object serialization mechanism. Publication is based on soft state. Clients and services must renew their leases periodically. The query “language” allows for simple string matching on attributes.

The Service Location Protocol (SLP) [17] uses multicast, softstate and simple filter expressions to advertize and query the location, type and attributes of services. The query “language” is more simple than Jini's. An extension is the Mesh Enhanced Service Location Protocol (mSLP) [18], increasing scalability through multiple cooperating directory agents. Both assume a single administrative domain and hence do not scale to the Internet and Grids.

The Service Discovery Service (SDS) [10] is also based on multi cast and soft state. It supports a simple XML based exact match query type. SDS is interesting in that it mandates secure channels with authentication and traffic encryption, and privacy and authenticity of service descriptions. SDS servers can be organized in a distributed hierarchy. For efficiency, each SDS node in a hierarchy can hold an index of the content of its sub-tree. The index is a compact aggregation and custom tailored to the narrow type of query SDS can answer. Another effort is the Intentional Naming System [19]. Like SDS, it integrates name resolution and routing.

10. CONCLUSIONS

While advances have recently been made in the field of web service specification, invocation and registration, the problem has so far received little systematic conceptual attention. This paper outlines seven web service problem areas and their associated processing steps, namely *description*, *presentation*, *publication*, *request*, *discovery*, *brokering* and *execution*. We propose a simple grammar (*SWSDL*) for *describing* network services as collections of service interfaces capable of executing operations over network protocols to endpoints. A service must *present* its current (up-to-date) description so that clients from anywhere can retrieve it at any time. A registry for *publication* and *query* of service and resource presence information is outlined. Reliable, predictable and simple distributed registry state maintenance in the presence of service failure or misbehavior or change is addressed by a simple and effective soft state mechanism. The notions of *request*, *resource* and *operation* are clarified. We outline the *discovery* step, which finds services implementing the operations required by a request. The *brokering* step determines an invocation schedule, which is a mapping over time of unbound operations to service operation invocations using given resources. Finally, the *execution* step implements a schedule, by using the supported protocols to invoke operations on remote services.

REFERENCES

1. Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery. PhD Thesis, Technical University of Vienna, March 2002.
2. Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int'l. Journal of Supercomputer Applications, 15(3), 2001.
3. Ben Segal. Grid Computing: The European Data Grid Project. In IEEE Nuclear Science Symposium and Medical Imaging Conference, Lyon, France, October 2000.
4. Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000), Bangalore, India, December 2000.
5. Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001.
6. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note 15, 2001. www.w3.org/TR/wsdl.
7. World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. W3C Note 8, 2000.
8. UDDI Consortium. UDDI: Universal Description, Discovery and Integration. www.uddi.org.
9. S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald. The grid notification framework. Technical report, Grid Forum Working Draft GWD-GIS-019, June 2001. <http://www.gridforum.org>.
10. Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In Int'l. Conf. on Mobile Computing and Networks, Seattle, WA, August 1999.
11. Rajkumar Buyya, Steve Chapin, and David DiNucci. Architectural Models for Resource Management in the Grid. In 1st IEEE/ACM Int'l. Workshop on Grid Computing (GRID 2000), Bangalore, India, December 2000.
12. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In Int'l. Symposium on High Performance Distributed Computing (HPDC'98), Chicago, IL, July 1998.
13. I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In 8th Int'l. Workshop on Quality of Service, 2000.
14. Ashley Beitz, Mirion Bearman, and Andreas Vogel. Service Location in an Open Distributed Environment. In Proc. of the Int'l. Workshop on Services in Distributed and Net-worked Environments, Whistler, Canada, June 1995.
15. Object Management Group. Trading Object Service. OMG RPF5 Submission., May 1996.
16. J. Waldo. The Jini architecture for network-centric computing. Communications of the ACM, 42(7), July 1999.
17. Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. IEEE Internet Computing Journal, 3(4), 1999.
18. Weibin Zhao, Henning Schulzrinne, and Erik Guttman. mSLP - Mesh Enhanced Service Location Protocol. In Proc. of the IEEE Int'l. Conf. on Computer Communications and Networks, Las Vegas, USA, October 2000.
19. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In Proc. of the Symposium on Operating Systems Principles, Kiawah Island, USA, 1999.