The Web Service Discovery Architecture

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

Abstract

In this paper, we propose the Web Service Discovery Architecture (WSDA). At runtime, Grid applications can use this architecture to discover and adapt to remote services. WSDA promotes an interoperable web service discovery layer by defining appropriate services, interfaces, operations and protocol bindings, based on industry standards. It is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The architecture is open and flexible because each primitive can be used, implemented, customized and extended in many ways. It is powerful because the individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies.

1 Introduction

An enabling step towards increased Internet and Grid software execution flexibility is the *web services* vision [1, 2, 3] 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.

For example, the European DataGrid (EDG) [4, 5] 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 [6]. 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 of 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.

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.

[*]Proc. IEEE/ACM Supercomputing Conference (SC 2002), Baltimore, USA, 0-7695-1524-X/02 $17.00 (c) 2002 IEEE

A service description language 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 [7]. In contrast to popular belief, a web service is neither required to carry XML [8] messages, nor to be bound to SOAP [9] or the HTTP [10] protocol, nor to run within a .NET [11] 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), as introduced in our prior studies [1]. SWSDL describes the interfaces of a distributed service object system. It is a compact pedagogical vehicle trading flexibility for clarity, not an attempt to replace the WSDL [7] standard.

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 SWSDL service description reads as follows[1]:

```
<service>
  <interface type = "http://gridforum.org/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>
```

It is important to note that the concept of a service is a logical rather than a physical concept. For efficiency, a so-called *container* of a virtual hosting environment such as the Apache Tomcat servlet container may be used to run more than one service or interface in the same process or thread. 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).

[1]To avoid name space collisions, an interface type is a universally unique URI such as `http://gridforum.org/interface/Consumer-1.0`. Interfaces expressed with XML Schema [12] (e.g. WSDL based interfaces) must use the XML namespace of that schema as type name. Sometimes we omit prefixes and version postfixes for compact exposition.

In this paper we define a web service layer that promotes interoperability for existing and future Internet software. Such a layer views the Internet as a large set of services with an extensible set of well-defined interfaces. A discovery architecture defines appropriate services, interfaces, operations and protocol bindings for discovery. The key problems are:

- *Can we define a discovery architecture that promotes interoperability, embraces industry standards, and is open, modular, flexible, unified, nondisruptive and simple yet powerful?*
- *What kind of query and data model as well as query language can support simple and complex dynamic service and resource discovery with as few as possible architecture and design assumptions?*

This paper makes the following contributions: We propose and specify a discovery architecture that addresses these problems, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It specifies a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. The individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. An interface for XQueries allows for powerful searching, which is critical for non-trivial applications. State maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. A dynamic data model allows for a wide range of dynamic content freshness policies.

This paper is organized as follows. Section 2 defines four interfaces, namely `Presenter`, `Consumer`, `MinQuery` and `XQuery`. The `Presenter` interface allows clients to retrieve the current (most up-to-date) service description. The `Consumer` interface allows content providers to publish a tuple set to a consumer. The `MinQuery` interface provides the simplest possible query support (*"select all"*-style); It returns tuples including or excluding cached content. The `XQuery` interfaces provides powerful XQuery support.

Section 3 specifies default transport protocol bindings for these interfaces. Section 4 describes how two example services, the *hypermin registry* and the *hyper registry*, use and combine these interfaces. Section 5 discusses the desirable properties the architecture exhibits. Section 6 gives a detailed comparison with the emerging Open Grid Services Architecture [2, 13]. Section 7 compares our approach with other related work. Section 8 concludes this paper. We also outline interesting directions for future research.

2 Interfaces

Presenter. The `Presenter` interface allows clients to retrieve the current (most up-to-date) service description. Clearly clients from anywhere must be able to retrieve the current description of a service (subject to local security policy). 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, suggesting the use of any arbitrary URI. 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.

Thus, for generality, we define that an identifier may be any URI. However, in support of the above requirements, the identifier is most commonly chosen to be a URL [14], and the retrieval mechanism is chosen to be HTTP(S) [10]. If so, 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 identifier enabling service description retrieval. Like in the WWW, service links (and content links, see below) can freely be chosen as long as they conform to the URI and HTTP URL specification [14]. Hence, they may contain the usual URL encoded attribute-value pairs. Examples of links are:

```
urn:/iana/dns/ch/cern/cn/techdoc/94/1642-3
urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6
http://sched.cern.ch:8080/getServiceDescription.wsdl
https://cms.cern.ch/getServiceDesc?id=4712&cache=disable
http://phone.cern.ch/lookup?query=
    "select phone from phonebook where phone=0450-1234"
http://repcat.cern.ch/getPFNs?lfn="myLogicalFileName"
```

Because service descriptions should describe the essentials of the service, it is recommended[2] that the service link concept be an integral part of the description itself. As a result, service descriptions may be retrievable via the `Presenter` interface, which defines an operation `getServiceDescription()` for this purpose. The operation is identical to service description retrieval and is hence bound to (invoked via) an HTTP(S) GET request to a given service link. Additional protocol bindings may be defined as necessary.

Consumer. The `Consumer` interface allows content providers to publish a tuple set to a consumer (e.g. a registry service), via the `publish` operation. A *tuple* can embed arbitrary content such as a service description, Quality of Service description, a file, file replica location, current network load, host information, stock quotes, etc. A tuple follows the *Dynamic Data Model (DDM)* (also see our prior studies [1]). It has as attributes a content link, a type, a context, four soft state time stamps, and (optionally) two arbitrary-shaped extensibility elements, namely metadata and content. A tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time, as depicted in Figure 1 and 2.

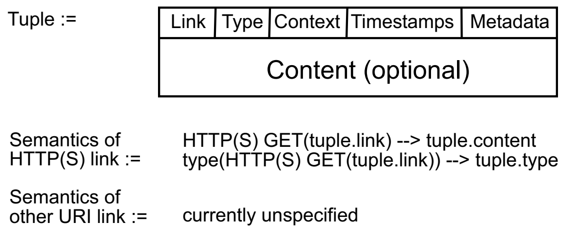
Tuple :=
Link
Type
Context
Timestamps
Metadata
Content (optional)
Semantics of HTTP(S) link :=
HTTP(S) GET(tuple.link) --> tuple.content
type(HTTP(S) GET(tuple.link)) --> tuple.type
Semantics of other URI link :=
currently unspecified

Figure 1: Tuple is an annotated multi-purpose soft state data container, and allows for dynamic refresh.

A content link (e.g. service link) is a URI. If it is an HTTP(S) URL then the current (most up-to-date) content can be retrieved (pulled) at any time via an

[2]In general, it is not mandatory for a service to implement any “standard” interface.

```
<tupleset>
    <tuple link="http://registry.cern.ch/getDescription"
           type="service" ctx="parent"
           TS1="10" TC="15" TS2="20" TS3="30">
       <content>
         <service>
           <interface type="http://cern.ch/Presenter-1.0">
             <operation>
               <name>XML getServiceDescription()</name>
               <bind:http verb="GET"
                URL="https://registry.cern.ch/getDesc"/>
             </operation>
           </interface>

           <interface type = "http://cern.ch/XQuery-1.0">
             <operation>
               <name> XML query(XQuery query)</name>
               <bind:beep
                     URL="beep://registry.cern.ch:9000"/>
             </operation>
           </interface>
         </service>
       </content>
       <metadata>  <owner name="http://cms.cern.ch"/>
       </metadata>
    </tuple>

    <tuple link="http://repcat.cern.ch/getDesc?id=4711"
           type="service" ctx="child"
           TS1="30" TC="0" TS2="40" TS3="50">
    </tuple>

    <tuple link="urn:uuid:f81d4fae-11d0-a765-00a0c91e6bf6"
           type="replica" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <replicaSet LFN="urn:/iana/dns/ch/cern/cms/myFile"
                size="10000000" type="MySQL/ISAM"
                creator="fred@cern.ch">
             <PFN URL="ftp://storage.cern.ch/file123"/>
             <PFN URL="ftp://se01.infn.it/file456"/>
         </replicaSet>
       </content>
    </tuple>
</tupleset>
```

Figure 2: Tuple Set from Dynamic Data Model.

HTTP(S) GET request to the link, turning the embedded tuple content into a *cache*. The type describes *what* kind of content is being published. The context describes *why* the content is being published or *how* it should be used[3]. The optional metadata element may further describe the content and/or its retrieval beyond what can be expressed with the previous attributes. For example it may describe retrieval from an UDDI [16] registry, formulated in the Web Service Inspection Language (WSIL) [17], or it may be a secure digital XML signature [18].

Given this tuple information, a content *retriever* module can retrieve the current content from the provider. Content and metadata can be structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment[4]. An individual element may, but need not, have a schema (XML Schema [12]), in which case it must be valid according to the schema. All elements may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

Based on embedded soft state time stamps defining life time, a tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. An existing tuple can be "updated" by publishing other values under the same tuple key. A tuple set must not contain equal tuples. That is, it must not contain tuples with the same tuple key. As usual, the rules governing comparison and equality of content links and contexts are the ones defined in the specification of the relevant URI and URL schemes. URIs from different schemes are never equal.

For detailed motivation, justification and discussion of the Dynamic Data Model and the semantics of soft state time stamps, see [1]. The dynamic data model is open and allows for a wide range of powerful caching policies. The publish operation has the signature `(TS4, TS5) publish(XML tupleset)`.

MinQuery. The `MinQuery` interface provides the simplest possible query support (*"select all"*-style). It returns tuples including or excluding cached content. As a minimum, clients can query by invoking minimalist query operations. The `getTuples()` query operation takes no arguments and returns the full set of all tuples "as is". That is, query output format and publication input format are the same (see Figure 2). If supported, output includes cached content. The

[3]For clarity of exposition, the examples given here use short strings as values for type and context (e.g. `service, parent`). However, strictly speaking, this is not legal. To avoid namespace pollution and ambiguities, the value of a type must be a universally unique URI [14], a MIME content-type [15] (e.g. `application/octet-stream, image/jpeg, audio/mpeg`) or the empty string. For XML content observing an XML Schema [12] the type must equal the URI of the schema namespace. The value of a context must be a URI or the empty string. For example, a service description really is of type `http://gridforum.org/content-type/service-1.0` whereas the `parent` context really has the value `http://gridforum.org/content-context/parent-1.0`.

[4]For clarity of exposition, the content is an XML element. In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME [15] object. The XML based publication input tuple set and query result tuple set is augmented with an additional MIME multipart object, which is a list containing all content. The content element of a tuple is interpreted as an index into the MIME multipart object.

getLinks() query operation is similar in that it also takes no arguments and returns the full set of all tuples. However, it always substitutes an empty string for cached content. In other words, the content is omitted from tuples, potentially saving substantial bandwidth. The second tuple in Figure 2 has such a form.

Advanced query support can be expressed on top of the minimal query capabilities. Such higher-level capabilities conceptually do not belong to a consumer and minimal query interface, which are only concerned with the fundamental capability of making a content link (e.g. service link) *reachable*[5] for clients. As an analogy, consider the related but distinct concepts of web hyper-linking and web searching: Web hyper-linking is a fundamental capability without which nothing else on the Web works. Many different kinds of web search engines using a variety of search interfaces and strategies can and are layered on top of web linking. The kind of XQuery support we propose below is certainly not the only possible and useful one. It seems unreasonable to assume that a single global standard query mechanism can satisfy all present and future needs of a wide range of communities. Multiple such mechanisms should be able to coexist. Consequently, the consumer and query interfaces are deliberately separated and kept as minimal as possible, and an additional interface type (XQuery) for answering XQueries is introduced below.

XQuery. The XQuery interface provides powerful XQuery support, which is important for realistic service and resource discovery use cases. For a detailed motivation and justification, including a discussion of a wide range of discovery queries and an evaluation of various query languages, see our prior studies [1]. XQuery [19, 20] is the standard XML query language developed under the auspices of the W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL [21] can also be expressed with XQuery. However, XQuery is a more expressive language than SQL, for example, because it supports path expressions for hierarchical navigation. Example XQueries for service discovery are depicted in Figure 3. XQuery can dynamically integrate external data sources via the document(URL) function, which can be used to process the XML results of remote operations invoked over HTTP. For example, given a service description with a getPhysicalFileNames(LogicalFileName) operation, a query can match on values dynamically produced by that operation. The same rules that apply to minimalist queries also apply to XQuery support. An implementation can use a modular and simple XQuery processor such as Quip [22] for the operation XML query(XQuery query). Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

[5] *Reachability* is interpreted in the spirit of garbage collection systems: A content link is reachable for a given client if there exists a direct or indirect retrieval path from the client to the content link.

Interface Summary. The four interfaces and their respective operations are summarized in Table 1. Figure 4 depicts the interactions of a client with implementations of these interfaces.

3 Network Protocol Bindings

The operations of the interfaces are bound to (carried over) a default transport protocol. The XQuery interface is bound to the *Peer Database Protocol (PDP)* proposed in our previous work [1, 23]. PDP support database queries for a wide range of database architectures and response models such that the stringent demands of ubiquitous Internet discovery infrastructures in terms of scalability, efficiency, interoperability, extensibility and reliability can be met. In particular, it allows for high concurrency, low latency as well as early and/or partial result set retrieval, both in pull and push mode. For all other operations and arguments we assume for simplicity HTTP(S) GET and POST as transport, and XML based parameters. *Additional* protocol bindings may be defined as necessary. An example service description of a registry implementing all four interfaces, formulated in SWSDL, is depicted in Figure 5.

4 Services

In [24] we defined two kinds of example registry services: The so-called *hypermin registry* must (at least) support the three interfaces Presenter, Consumer and MinQuery (excluding XQuery support). A *hyper registry* must (at least) support these interfaces plus the XQuery interface. Put another way, any service that happens to support, among others, the respective interfaces qualifies as a hypermin registry or hyper registry. As usual, the interfaces may have endpoints that are hosted by a single container, or they may be spread across multiple hosts or administrative domains.

It is by no means a requirement that only dedicated hyper registry services and hypermin registry services

Interface	Operations	Responsibility
`Presenter`	`XML getServiceDescription()`	Allows clients to retrieve the current description of a service and hence to bootstrap all capabilities of a service.
`Consumer`	`(TS4,TS5) publish(XML tupleset)`	A content provider can publish a dynamic pointer called a content link, which in turn enables the consumer (e.g. registry) to retrieve the current content. Optionally, a content provider can also include a copy of the current content as part of publication. Each input tuple has a content link, a type, a context, some time stamps, and (optionally) metadata and content.
`MinQuery`	`XML getTuples()` `XML getLinks()`	Provides the simplest possible query support (*"select all"*-style). The `getTuples` operation returns the full set of all available tuples "as is". The minimal `getLinks` operation is identical but substitutes an empty string for cached content.
`XQuery`	`XML query(XQuery query)`	Provides powerful XQuery support. Executes an XQuery over the available tuple set. Because not only content, but also content link, context, type, time stamps, metadata etc. are part of a tuple, a query can also select on this information.

Table 1: WSDA Interfaces and their Respective Operations.

may implement WSDA interfaces. Any arbitrary service may decide to offer and implement none, some or all of these four interfaces. For example, a job scheduler may decide to implement, among others, the `MinQuery` interface to indicate a simple means to discover metadata tuples related to the current status of job queues and the supported Quality of Service. The scheduler may not want to implement the `Consumer` interface because its metadata tuples are strictly read-only. Further, it may not want to implement the `XQuery` interface, because it is considered overkill for its purposes. Even though such a scheduler service does not qualify as a hypermin or hyper registry, it clearly offers useful added value. Other examples of services implementing a subset of interfaces are consumers such as an instant news service or a cluster monitor. These services may decide to implement the `Consumer` interface to invite external sources for data feeding, but they may not find it useful to offer and implement any query interface.

In a more sophisticated scenario, the example job scheduler may decide to publish its local tuple set also to an (already existing) remote hyper registry service (i.e. with XQuery support). To indicate to clients how to get hold of the XQuery capability, the scheduler may simply copy the `XQuery` interface description of the remote registry service and advertise it as its own interface by including it in its own service description. This kind of *virtualization* is not a "trick", but a feature with significant practical value, because it allows for minimal implementation and maintenance effort on the part of the scheduler.

Alternatively, the scheduler may include in its local tuple set (obtainable via the `getLinks()` operation) a tuple that refers to the service description of the remote registry service. An interface referral value for the context attribute of the tuple is used, as follows:

```
<tuple link="https://registry.cern.ch/getServiceDescription"
       type="service" ctx="x-ireferral://cern.ch/XQuery-1.0"
       TS1="30" TC="0" TS2="40" TS3="50">
</tuple>
```

5 Properties

WSDA has a number of key properties:

- **Standards Integration.** The architecture embraces and integrates solid and broadly accepted industry standards such as XML [8], XML Schema [12], the Simple Object Access Protocol (SOAP) [9], the Web Service Description Language (WSDL) [7] and XQuery [19]. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL) [17].
- **Interoperability.** WSDA promotes an interoperable web service layer on top of existing and future Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. We do not introduce new Internet standards. Rather, we judiciously combine existing interoperability-proven open Internet standards such as HTTP(S) [10], URI [14], MIME [15], XML [8], XML Schema [12] and BEEP [25, 26, 27].
- **Modularity.** The architecture is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, publication, as well as minimal and powerful query support. The responsibility, definition and evolution of any given primitive is distinct and independent of that of all other primitives.
- **Ease-of-use and Ease-of-implementation.** Each communication primitive is deliberately designed to avoid any unnecessary complexity. The

- *Find all (available) services.*

```
RETURN /tupleset/tuple[@type="service"]
```

- *Find all services that implement a replica catalog service interface that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN).*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
FOR $tuple in /tupleset/tuple[@type="service"]
LET $s := $tuple/content/service
WHERE
  SOME $op IN $s/interface[@type = $repcat]/operation
  SATISFIES ($op/name="XML getPFNs(String LFN)" AND
             $op/bindhttp/@verb="GET" AND
             contains($op/allow, "http://cms.cern.ch"))
RETURN $tuple
```

- *Find all replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*

```
LET $repcat := "http://cern.ch/ReplicaCatalog-1.0"
LET $s := /tupleset/tuple[@type="service"]
          /content/service[interface@type = $repcat]
RETURN
  FOR $pfn IN invoke($s, $repcat,
      "XML getPFNs(String LFN)",
      "http://myhost.cern.ch/myFile")
      /tupleset/PFN
  WHERE starts-with($pfn, "ftp://")
  RETURN $pfn
```

- *Return the number of replica catalog services.*

```
RETURN count(/tupleset/content/service
 [interface/@type="http://cern.ch/ReplicaCatalog-1.0"])
```

- *Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (Job wants to read and write locally).*

```
LET $exeType := "http://cern.ch/executor-1.0"
LET $stoType := "http://cern.ch/storage-1.0"
FOR $exec IN /tupleset/tuple[content/service/
                    interface/@type=$exeType],
    $stor IN /tupleset/tuple[content/service/
                    interface/@type=$stoType
                    AND domainName(@link) =
                    domainName($exec/@link)]
RETURN <pair> {$exec} {$storage} </pair>
```

Figure 3: Example XQueries for Service Discovery.

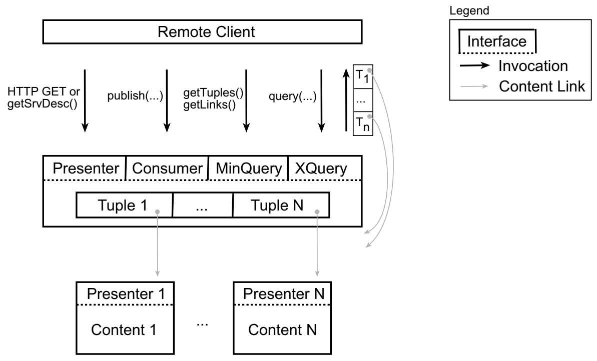

Figure 4: Interactions of Client with Interfaces.

```
<service>
  <interface type = "http://cern.ch/Presenter-1.0">
    <operation>
      <name>XML getServiceDescription()</name>
      <bind:http verb="GET"
       URL="https://registry.cern.ch/getDescription"/>
    </operation>
  </interface>

  <interface type = "http://cern.ch/Consumer-1.0">
    <operation>
      <name> (TS4,TS5) publish(XML tupleset)</name>
      <bind:http verb="POST"
            URL="https://registry.cern.ch/publish"/>
    </operation>
  </interface>

  <interface type = "http://cern.ch/MinQuery-1.0">
    <operation>
      <name> XML getTuples()</name>
      <bind:http verb="GET"
            URL="https://registry.cern.ch/getTuples"/>
    </operation>
    <operation>
      <name> XML getLinks()</name>
      <bind:http verb="GET"
            URL="https://registry.cern.ch/getLinks"/>
    </operation>
  </interface>

  <interface type = "http://cern.ch/XQuery-1.0">
    <operation>
      <name> XML query(XQuery query)</name>
      <bind:beep URL="beep://registry.cern.ch:9000"/>
    </operation>
  </interface>
</service>
```

Figure 5: SWSDL description of a registry service implementing all four interfaces.

design principle is to *"make simple and common things easy, and powerful things possible"*. In other words, solutions are rejected that provision for powerful capabilities yet imply that even simple problems are complicated to solve. For example, service description retrieval is by default based on a simple HTTP(S) GET. Yet, we do not exclude, and indeed allow for, alternative identification and retrieval mechanisms such as the ones offered by UDDI (Universal Description, Discovery and Integration) [16], RDBMS or custom Java RMI registries (e.g. via tuple metadata specified in WSIL [17]). Further, tuple content is by default given in XML, but advanced usage of arbitrary MIME [15] content (e.g. binary images, files, MS-Word documents) is also possible. As another example, the minimal query interface requires virtually no implementation effort on the part of a client or server. Yet, where necessary, also powerful XQuery support may, but need not, be implemented and used.

- **Openness and Flexibility.** WSDA is open and flexible because each primitive can be used, implemented, customized and extended in many ways. For example, the interfaces of a service may have endpoints spread across multiple hosts or administrative domains. However, there is nothing that prevents all interfaces to be co-located on the same host or implemented by a single program. Indeed, this is often a natural deployment scenario. Further, even though default network protocol bindings are given, additional bindings may be defined as necessary. For example, an implementation of the `Consumer` interface may bind to (carry traffic over) HTTP(S) [10], SOAP/BEEP [27], FTP [28], or RMI [29]. The tuple set returned by a query may be maintained according to a wide variety of cache coherency policies, resulting in static to highly dynamic behavior. A consumer may take any arbitrary custom action upon publication of a tuple. For example, it may interpret a tuple from a specific schema as a command or an active message [30], triggering tuple transformation and/or forwarding to other consumers such as loggers. For flexibility, a service maintaining a tuple set may be deployed in any arbitrary way. For example, the database can be kept in a XML file, in the same format as returned by the `getTuples` query operation. However, tuples can also be dynamically recomputed or kept in a relational database.

- **Expressive Power.** The architecture is powerful because its individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors. Each single primitive is of limited value all by itself. The true value of simple orthogonal multi-purpose communication primitives lies in their potential to generate powerful emerging synergies. For example, combination of WSDA primitives enables building services for replica location, name resolution, distributed auctions, instant news and messaging, software and cluster configuration management, certificate and security policy repositories, as well as Grid monitoring tools.

  As another example, the consumer and query interfaces can be combined to implement a Peer-to-Peer (P2P) database network for service discovery. In a large distributed system spanning many administrative domains such as a DataGrid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. However, in such a database system, the set of information tuples in the universe is partitioned over multiple distributed nodes, for reasons including autonomy, scalability, availability, performance and security. Here, P2P nodes maintain a local database and implement the consumer and query interfaces. Clients and P2P nodes publish (their) service descriptions and/or other metadata to one or more P2P nodes. Publication enables distributed node topology construction (e.g. ring, tree or graph) and at the same time constructs the database to be searched. When any *originator* wishes to search the P2P network with some query, it sends the query to an *agent node*. The node applies the query to its local database and returns matching results; it also forwards the query to select *neighbor nodes*. These neighbors return their local query results; they also forward the query to select neighbors, and so on. We have extensively discussed this in [1, 31, 23], where the *Unified Peer-to-Peer Database Framework (UPDF)* and corresponding *Peer Database Protocol (PDP)* are devised, which are unified in the sense that they allow to express specific applications for a wide range of data types (typed or untyped XML, any MIME type [15]), node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies (in the form of an XQuery), pipelining characteristics, timeout and other scope options.

- **Uniformity.** WSDA is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single *semi-transparent* umbrella. It allows for multiple competing distributed systems concepts and implementations to coexist and to be integrated. Clients can dynamically adapt their behavior based on rich service introspection capabilities. Clearly there exists no solution that is optimal in the presence of the heterogeneity found in real-world large cross-organizational distributed systems such as Data Grids, electronic market places and instant Internet news and messaging services. Introspection and adaption capabilities increasingly make it unnecessary to mandate a single global solution to a given problem, thereby enabling integration of collaborative systems.

- **Non-Disruptiveness.** WSDA is non-disruptive because it offers interfaces but does not mandate that every service in the universe must comply to a set of "standard" interfaces.

6 Comparison with Open Grid Services Architecture

We have recently learned about the emerging *Open Grid Services Architecture (OGSA)* [2, 13]. OGSA exhibits striking similarities with our architecture, in spirit and partly also in design. We stress that this paper and OGSA have so far been mutually independent work in their entirety. Future work is likely to be collaborative and convergent due to shared interest. Due to the recent circulation of early OGSA material, our understanding of it is limited and not necessarily accurate. Nevertheless, in this section we attempt a preliminary comparison of OGSA concepts with WSDA concepts.

OGSA is work-in-progress, but an important first step towards enabling powerful, flexible yet also interoperable large cross-organizational Grid systems. Like WSDA, OGSA defines and standardizes a set of (mostly) orthogonal multi-purpose communication primitives that can be combined and customized by specific clients and services to yield powerful behavior. Like WSDA, OGSA embraces solid and broadly accepted industry standards such as XML [8], XML Schema [12], the Simple Object Access Protocol (SOAP) [9], the Web Service Description Language (WSDL) [7] and XQuery [19].

Service Link, Service Description and Presenter. In OGSA, a service instance is identified by a *Grid Service Handle (GSH)*, which is an immutable, globally unique HTTP(S) URL that distinguishes a specific service instance from all other service instances that have existed, exist now, or will exist in the future. By means of an HTTP GET or the `HandleMap` interface, the handle can be resolved to a *Grid Service Reference (GSR)*, which is typically a WSDL document containing descriptions of all supported service interfaces. A reference may change over time. That is, the contents of the WSDL document may change over time. A reference is based on soft state, hence expires unless periodically renewed.

A GSH corresponds to a WSDA *service link*. However, unlike a GSH, a service link is neither required to be immutable nor globally unique. A GSR corresponds to a WSDA *service description* given in WSDL. Both are soft state documents. Although it is unclear from the complex presentation, it appears that a `HandleMap` corresponds to the WSDA `Presenter` interface. In WSDA, the operation `Presenter.getServiceDescription()` or a simple HTTP(S) GET to a content link (e.g. service link) may be used to retrieve the current content (e.g. service description). WSDA allows arbitrary-shaped content retrieval via MIME encoding (e.g. textual, binary), including mapping a service link to a service description. It appears that OGSA is restricted to mapping a GSH to a GSR. Further not every legal HTTP(S) URL is a legal GSH. In WSDA, every legal HTTP(S) URL is a legal content link and hence also a legal service link. OGSA defines implicit and unclear URL suffix mapping and `GSHomeHandleMapID` semantics that we believe would better be omitted or expressed as part of the `HandleMap` interface. The OGSA `HandleMap` operation `GSR FindByHandle(GSH)` corresponds to the WSDA `Presenter` operation `XML getServiceDescription()`. The additional GSH argument is unnecessary in our approach.

Tuple, Tuple Set and Query. In OGSA, a *grid service instance* maintains so-called *service data* XML elements. A service data element is a multi-purpose soft state data container that may contain arbitrary content. A service data element has a name attribute and three soft state time stamp attributes (`goodFrom`, `goodUntil`, `notGoodAfter`) and may contain an arbitrary extensibility element as content. In contrast, a WSDA *tuple* follows the Dynamic Data Model (DDM). It has as attributes a content link, a type, a context, four soft state time stamps, and (optionally) two arbitrary-shaped extensibility elements, namely metadata and content. A WSDA tuple is an annotated multi-purpose soft state data container that may contain a piece of arbitrary content and allows for refresh of that content at any time (see Figure 1).

The dynamic nature of our data model allows keeping access control for security sensitive content in the hands of the authoritative content provider. While it may be harmless that potentially anybody can learn that some content exists (content link), stringent trust delegation policies may dictate that only a few select clients, not including `Consumers` and registries, are allowed to retrieve the content from a provider. Consider for example, that a detailed service description may be helpful for launching well-focused security attacks. A data model that is not dynamic, on the other hand, requires trust delegation, which potentially opens the door for a malicious audience to learn detailed enough service descriptions to launch well-focused virus or denial of service attacks.

A collection of OGSA service data elements corresponds to a WSDA *tuple set*. In WSDA, publication input data and query output is uniformly expressed as a tuple set[6].

The OGSA operation `FindServiceData` of the `GridService` interface allows querying the collection of service data elements (roughly corresponding to the WSDA `MinQuery` and `XQuery` interfaces). It attempts to support multiple query languages by accepting and returning an arbitrary XML element. The schema of the input XML element indicates the query language. If the service instance supports the desired query language, the query is executed against the collection of service data elements. Every OGSA grid service must support a simple query "language" that returns a list of all service data elements whose name equals a given name (exact match). The name *"root"* is reserved and must return at least a list of "standard" service data elements such as handle, reference, primary key, a list of the supported query languages, etc.

The OGSA `FindServiceData` operation takes arbitrary XML input and returns arbitrary XML output. It remains to be seen how useful it is to coerce distinct query capabilities into a single generic operation. Consider that modern software systems rarely coerce distinct capabilities into a single generic handler function of the form `Object do(Object)`. Further consider that the very purpose of separate interface types and names is to allow for independence, separate evolution, flexibility, clarity, predictability, type safety and straightforward introspection. In essence, this is what web services and service descriptions are about. In this light, generic functions for distinct capabilities appear counter-intuitive to the spirit of web services.

[6]The output tuple set of a constructive query may contain arbitrary content, whereas all other queries output a tuple set with tuples from the dynamic data model.

For comparison, in WSDA, trivial and powerful query support are cleanly separated. The `MinQuery` interface is indeed minimal and requires only the simplest possible query support (*"select all"*-style) via the operations `getTuples()` and `getLinks()`. The WSDA `XQuery` interface, on the other hand, allows extremely powerful queries. Finally, it is unclear from the material whether OGSA intends in the future to support either or both XQuery, XPath, or none.

Data Publication. An OGSA *registry service instance* maintains a collection of handles. Typically, an OGSA registry service offers a `Registry` interface, which supports registering and unregistering handles. The `registerService` operation takes as input a handle, a timeout, and optionally, an abstract and an arbitrary extensibility element. The `unregisterService` operation takes as input the handle to be removed.

In contrast, a WSDA consumer and query can accept and return arbitrary textual and binary soft state data in the form of a tuple set, including content links, service links (handles), cached content and metadata (e.g. a WSIL [17] fragment). The OGSA `registerService` operation roughly corresponds to the WSDA `publish` operation. However, the latter operation is a unified multi-purpose operation, supporting a set of arbitrary-shaped soft state tuples. The functionality to publish information other than handles (e.g. references) to a registry or service currently appears to be missing from OGSA. The OGSA notification interface appears only useful for interaction patterns based on subscription by notification sinks. Invitation for subscription appears to be missing. The `unregisterService` operation is not necessary in WSDA, because a tuple is based on soft state. Explicit (immediate) unregistration can be achieved by using the WSDA `publish` operation with tuples carrying zero-valued soft state time stamps.

An OGSA registry service supports discovery queries via the `FindServiceData` operation of the `GridService` interface. The relationship between discovery queries and the maintained collection of handles is unclear. More precisely, it is unclear whether handles are maintained as service data elements, and if they can be queried in the same way as described above.

The OGSA `NotificationSource` and `NotificationSink` interfaces allow for publish-subscribe functionality based on message type and interest statements. A notification source may offer a set of topics, which notification sinks may use for subscription. A notification source may send notification messages to a subscribed notification sink. Subscription requests are soft state based and expire unless

Concept	**OGSA**	**WSDA**
Interfaces	GridService, NotificationSource, NotificationSink, Registry, Factory, PrimaryKey, HandleMap	Presenter, Consumer, MinQuery, XQuery
Interfaces required to be implemented by every service	GridService interface; defines operations for query and life cycle maintaince	None
Service identifier	Grid Service Handle (GSH)	Service link (i.e. content link)
Service description	Grid Service Reference (GSR) (e.g. WSDL)	Service description (e.g. WSDL)
Service description retrieval	via HTTP(S) GET or `HandleMap.findByHandle(GSH)`	via HTTP(S) GET or `Presenter.getServiceDescription()`
Multi-purpose data container	Service data	Tuple
Set of data containers	Service data list	Tuple set
Query capability	`GridService.FindServiceData(XML query)`	`MinQuery.getLinks()`, `MinQuery.getTuples()`, `XQuery.query(XQuery)`
Data publication	`Registry.RegisterService(handle)`, `NotificationSink.deliverNotification(sdata)`	`Consumer.publish(XML tupleset)`

Table 2: Open Grid Services Architecture vs. Web Service Discovery Architecture.

periodically renewed. The OGSA `NotificationSink` interface corresponds to the WSDA `Consumer` interface. However, it appears that it is not foressen that an OGSA notification message may carry a set of more than one service data elements[7]. In contrast, a WSDA consumer message explicitly carries zero or more tuples in a tuple set, resulting in potentially much improved efficiency. Consider that all production quality database management systems we are aware of support batching of tuples over the network. This is because inserting a million tuples (e.g. CPU load samples) into a database should not involve the same number of network round-trips. In addition to efficiency by design, WSDA offers an open and precisely specified dynamic data model that allows for a wide range of powerful caching policies. We are working on a multi-purpose interface for persistent XQueries (i.e. server-side trigger queries), which will roughly correspond to the OGSA `NotificationSource` interface, albeit in a more general and powerful manner. The Peer Database Protocol [23] already supports, in a unified manner, all messages and network interactions required for efficient implementations of Peer-to-Peer publish-subscribe and event trigger interfaces (e.g. synchronous pull and asynchronous push, as well as invitations and batching).

Grid Service. In OGSA, a `GridService` interface supports discovery queries (`findServiceData`), setting and prolonging of shutdown time (`setTerminationTime`) as well as explicit (immediate) shutdown of the service (`destroy`). OGSA mandates that every grid service must implement the `GridService` interface. In contrast, WSDA does not require a service to implement any "standard" interface. A specific service (e.g. a registry service) may, of course, mandate implementation of certain interfaces, but there is no global requirement that any and all services in the universe must satisfy. Historical evidence suggests that the acceptance of ubiquitous Internet infrastructures and their flexible and successful evolution strongly depends on being conservative with the term *MUST*. A design rarely turns out right the first time. Typically, several design revisions and refactorings over time are needed. Consider that once a fundamental interface is introduced as mandatory, one is "stuck" with it forever, at least if compatibility and stability are of concern.

The problem is subtle and comparable to the design of the base class of single-rooted object oriented programming languages. A uniform and well designed base class clearly offers strong advantages because everyone can safely assume certain essential features to be available everywhere. Many C++ reusability problems stem from the fact that the C++ language has no single common base class. For example, integration of third-party frameworks is problematic at best. On the other hand, a controversial or flawed base class feature such as the (potentially very useful) `clone()` method of the Java base class is bound to lead to dissatisfaction.

Another comparable problem-in-the-large is the definition of the Java platform. Typically, Java interfaces and frameworks are not designed, specified, revised, standardized and hardened *within* the Java platform. Rather, they are born and live externally for some two years to allow enough time for a community process, deployment feedback from reference implementations, and for separation of wheat from chaff. Only then may some of them be considered to be merged into the core

[7]Nesting service data elements inside a service data element is possible but undefined and left without semantics.

Java platform definition.

It is often desirable to include features only if considered non-disruptive and absolutely essential for a wide range of communities. While certainly interesting and often useful, justification is missing why query, shutdown and other lifetime maintenance of services should be absolutely essential for *every* service. Further, while certainly well designed, justification is missing why the chosen definition of these features is superior to other approaches. Consider that a large variety of server administration and monitoring products with related but not equivalent interfaces has been introduced and marketed in the past [32]. In addition, the OGSA query interface certainly is, just like our query interfaces, not the only possible and useful one. Finally, we believe that a main idea behind web services is service description introspection and dynamic adaption. This capability increasingly makes it unnecessary to mandate a global "service base class or interface".

Other. The OGSA `Factory` interface supports dynamically creating short or long-lived service instances. An OGSA service instance may be associated with a *primary key*, which is used to locate and shut down service instances created by a factory. It is unclear what the added value of a primary key over a handle is. The concept may perhaps be related to the WSDA *tuple key*, which is the pair `(content link, context)`, uniquely identifying a tuple within a tuple set.

Table 2 summarizes the comparison of corresponding OGSA and WSDA concepts.

7 Other Related Work

WSDL. The Simple Web Service Description Language (SWSDL) describes the interfaces of a distributed service object system. For simplicity, it offers neither a class concept nor interface inheritance. Service descriptions could also be formulated in the Web Service Description Language (WSDL) [7]. WSDL is a rigorous, expressive and flexible industry standard. However, WSDL trades clarity for expressiveness and flexibility. The example stated in [7] requires 66 XML lines and 7 levels of XML nesting even though it merely describes a stock quote service with a trivial operation that returns the trading price of a given stock. We estimate that WSDL based service descriptions are about one order of magnitude larger in size and structural complexity than corresponding SWSDL based descriptions. We stress that SWSDL is a pedagogical vehicle, not an attempt to replace the standard. All features of SWSDL can be (and in practice will be) mapped to WSDL. Both languages are not mutually exclusive. SWSDL is more useful in the high-level architecture and design phase of a software project whereas WSDL is more useful for the detailed specification and implementation phase.

LDAP and MDS. The Lightweight Directory Access Protocol (LDAP) [33] defines a network protocol in which clients send requests to and receive responses from LDAP servers. LDAP is an extensible network protocol, not a discovery architecture. It does not offer a dynamic data model, is not based on soft state and does not follow an XML data model. The expressive power of the LDAP query language is insufficient for realistic service discovery use cases [1].

The Metacomputing Directory Service (MDS) [34] is a specific service based on LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS does not offer a dynamic data model. However, it is based on soft state. MDS is not a web service, because it is not specified by a service description language. It does not offer interfaces and operations that may be bound to multiple network protocols. However, it appears that MDS is being recast to fit into the OGSA architecture. Indeed, the OGSA registry and notification interfaces could be seen as new and abstracted clothings for MDS.

UDDI. UDDI (Universal Description, Discovery and Integration) [16] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL service descriptions. UDDI is a definition of a specific service class, not a discovery architecture. 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. Query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases.

ANSA and CORBA. The ANSA project was an early collaborative industry effort to advance distributed computing. It defined trading services [35] for advertizement and discovery of relevant services, based on service type and simple constraints on attribute/value pairs. The CORBA Trading service [36] is an evolution of these efforts.

Jini, SLP, SDS, INS. The Jini Lookup Service [37] 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. Content freshness is not addressed. The query "language" allows for simple string matching on attributes, and is even less powerful than LDAP.

The Service Location Protocol (SLP) [38] 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) [39], 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) [40] 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 subtree. 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 [41]. Like SDS, it integrates name resolution and routing.

8 Conclusions

We propose and specify an open discovery architecture, the so-called *Web Service Discovery Architecture (WSDA)*. WSDA views the Internet as a large set of services with an extensible set of well-defined interfaces. The architecture has a number of key properties. It promotes an interoperable web service layer on top of existing and future Internet software, because it defines appropriate services, interfaces, operations and protocol bindings. It embraces and integrates solid industry standards such as XML, XML Schema, SOAP, WSDL and XQuery. It allows for integration of emerging standards such as the Web Service Inspection Language (WSIL). It is modular because it defines a small set of orthogonal multi-purpose communication primitives (building blocks) for discovery. These primitives cover service identification, service description retrieval, data publication as well as minimal and powerful query support. Each communication primitive is deliberately designed to avoid any unnecessary complexity.

The architecture is open and flexible because each primitive can be used, implemented, customized and extended in many ways. It is powerful because the individual primitives can be combined and plugged together by specific clients and services to yield a wide range of behaviors and emerging synergies. It is unified because it subsumes an array of disparate concepts, interfaces and protocols under a single semi-transparent umbrella. It is non-disruptive because it offers interfaces but does not mandate that every service in the universe must comply to a set of "standard" interfaces. Finally, we compare in detail the properties of WSDA with the emerging Open Grid Services Architecture.

The results presented in this paper open three interesting research directions.

First, it would be valuable to rigourously assess, review and compare the Web Service Discovery Architecture and the Open Grid Services Architecture in terms of concepts, design and specifications. A strong goal is to achieve convergence by extracting best-of-breed solutions from both proposals. Future collaborative work could further improve current solutions, for example in terms of simplicity, orthogonality and expressiveness.

Second, Tim Berners-Lee designed the World Wide Web as a consistent interface to a flexible and changing heterogeneous information space for use by CERN's staff, the High Energy Physics community, and, of course, the world at large. The WWW architecture [42] rests on four simple and orthogonal pillars: URIs as identifiers, HTTP for retrieval of content pointed to by identifiers, MIME for flexible content encoding, and HTML as the primus-inter-pares (MIME) content type. Based on our Dynamic Data Model (DDM), we hope to proceed further towards a self-describing meta content type that retains and wraps all four WWW pillars "as is", yet allows for flexible extensions in terms of identification, retrieval and caching of content. Judicious combination of the four WWW pillars, DDM, WSDA, the Hyper Registry [24], the Unified Peer-to-Peer Database Framework (UPDF) [31] and its associated Peer Database Protocol (PDP) [23] are used to define how to bootstrap, query and publish to a dynamic information space maintained by self-describing network interfaces.

Third, we are starting to build a system prototype with the aim of reporting on experience gained from application to an existing large distributed system such as the European DataGrid.

Acknowledgments. We would like to thank Peter Kunszt for an early high-level analysis. Ben Segal supported this work with great enthusiasm, encouragement and proof-reading.

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, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.

[3] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[4] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[5] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[6] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://lhc-computing-review-public.web.cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

[7] 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.

[8] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, October 2000.

[9] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. *W3C Note 8*, 2000.

[10] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *IETF RFC 2616*. UC Irvine, Digital Equipment Corporation, MIT.

[11] Oracle. J2EE and Microsoft .NET, April 2002. Oracle Corp., White Paper.

[12] World Wide Web Consortium. XML Schema Part 0: Primer. *W3C Recommendation*, May 2001.

[13] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, and Carl Kesselman. Grid Service Specification, February 2002.

[14] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *IETF RFC 2396*.

[15] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.

[16] UDDI Consortium. UDDI: Universal Description, Discovery and Integration. www.uddi.org.

[17] P. Brittenham. An Overview of the Web Services Inspection Language, 2001. www.ibm.com/developerworks/webservices/library/ws-wsilover.

[18] World Wide Web Consortium. XML-Signature Syntax and Processing. *W3C Recommendation*, February 2002.

[19] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[20] World Wide Web Consortium. XML Query Use Cases. *W3C Working Draft*, December 2001.

[21] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[22] Software AG. The Quip XQuery processor. http://www.softwareag.com/developer/quip/.

[23] Wolfgang Hoschek. A Unified Peer-to-Peer Database Protocol. Technical report, DataGrid-02-TED-0407, April 2002.

[24] Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 2002.

[25] Marshall Rose. The Blocks Extensible Exchange Protocol Core. *IETF RFC 3080*, March 2001.

[26] Marshall Rose. Mapping the BEEP Core onto TCP. *IETF RFC 3081*, March 2001.

[27] E. O'Tuathail and M. Rose. Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP). *IETF RFC 3288*, June 2002.

[28] J. Postel and J. Reynolds. File Transfer Protocol (FTP). *IETF RFC 959*, October 1985.

[29] Madhusudhan Govindara, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Supercomputing Conference (SC'00)*, Dallas, Texas, November 2000.

[30] David Culler, Kim Keeton, Lok Tim Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. The Generic Active Message Interface Specification, August 1994. Computer Science Division, University of California at Berkeley, White Paper.

[31] Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework and its Application for Scalable Service Discovery. In *Proc. of the 3rd Int'l. IEEE/ACM Workshop on Grid Computing (Grid'2002)*, Baltimore, USA, November 2002. Springer Verlag.

[32] Maite Barroso. Grid Fabric Management Work Package Report on Current Technology. Technical report, DataGrid-04-TED-0101, May 2001.

[33] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[34] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[35] 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 Networked Environements*, Whistler, Canada, June 1995.

[36] Object Management Group. Trading Object Service. *OMG RPF5 Submission:*, May 1996.

[37] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7), July 1999.

[38] Erik Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing Journal*, 3(4), 1999.

[39] 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 (ICCCN'00)*, Las Vegas, USA, October 2000.

[40] Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int'l. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.

[41] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. of the Symposium on Operating Systems Principles*, Kiawah Island, USA, December 1999.

[42] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD Thesis, University of California, Irvine, 2000.