Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 1–16. http://www.scpe.org
ISSN 1895-1767

ENABLING RICH SERVICE AND RESOURCE DISCOVERY WITH A DATABASE FOR DYNAMIC DISTRIBUTED CONTENT

WOLFGANG HOSCHEK*

Abstract. In a distributed system such as a Data Grid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. However, it is not obvious how a database (registry) should maintain information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. In particular, how can one avoid sacrificing reliability, predictability and simplicity while allowing to express powerful queries over time-sensitive dynamic information? We propose the so-called *hyper registry*, which has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client.

Key words. Dynamic Database, XQuery, Service Discovery

1. Introduction. The next generation Large Hadron Collider project at CERN, the European Organization for Nuclear Research, involves thousands of researchers and hundreds of institutions spread around the globe. A massive set of computing resources is necessary to support it's data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [1]. To make collaboration viable, it was decided to share in a global joint effort—the European Data Grid (EDG) [2, 3]—the data and locally available resources of all participating laboratories and university departments.

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [4]. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

An enabling step towards increased Grid software execution flexibility is the (still immature and hence often hyped) *web services* vision [5, 6, 7] 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, 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 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.

More generally, in a distributed system, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. This enables information discovery and collective collaborative functionality that operate on the system as a whole, rather than on a given part of it. For example, it allows a search for descriptions of services of a file sharing system, to determine its total download capacity, the names of all participating organizations, etc. Example systems include a service discovery system for a (worldwide) Data Grid, an electronic market place, or an instant messaging and news service. In all these systems, a variety of information describes the state of autonomous remote participants residing within different administrative domains. Participants frequently join, leave and act on a best effort basis. In a large distributed system spanning many administrative domains, predictable, timely, consistent and reliable global state maintenance is infeasible. The information to be aggregated and integrated may be outdated, inconsistent, or not available at all. Failure, misbehavior, security restrictions and continuous change are the norm rather than the exception.

*CERN IT Division, European Organization for Nuclear Research, 1211 Geneva 23, Switzerland(wolfgang.hoschek@cern.ch).

In this paper, a design and specification for a type of database is developed that is conditioned to address the problem, the so-called *hyper registry*. A hyper registry has a database that holds a set of tuples. A *tuple* may contain an arbitrary piece of arbitrary *content*. Examples of content include a service description expressed in WSDL [8], a Quality of Service description, a file, file replica location, current network load, host information, stock quotes, etc., as depicted in Figure 1.1. A tuple is annotated with a *content link* pointing to the authoritative data source of the embedded content.

```
<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/scheduler/submitjob"/>
    </operation>
  </interface>
</service>
```

```
<hostInfo>
   <host name="fred01.cern.ch" os="i386_redhat 7.2"   MHz="1000" cpus="2"/>
   <host name="fred02.cern.ch" os="sparc_solaris 2.7" MHz="400"  cpus="64"/>
</hostInfo>
```

```
<replicaSet LFN="urn:/iana/dns/ch/cern/cms/higgs" size="1000000" type="MySQL">
   <PFN URL="ftp://storage.cern.ch/file123" readCount="17"/>
   <PFN URL="ftp://se01.infn.it/file456"    readCount="1"/>
</replicaSet>
```

FIG. 1.1. *Example Content.*

This paper is organized as follows. Section 2 identifies query types and requirements by means of examples. Section 3 discusses content description and publication. A *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). Section 4 illustrates how a remote client can query a hyper registry, obtaining a set of tuples as answer. The use of XQuery [9, 10] as a rich and expressive query language is motivated and discussed by means of examples. Section 5 proposes solutions to the cache coherency issues arising from content *caching* for improved client efficiency. For reliable, predictable and simple state maintenance, we propose to keep a registry tuple as soft state. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from the content provider. Section 6 shows that content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. Section 7 compares our approach with related work. Finally, Section 8 summarizes and concludes this paper.

2. Query Examples and Types. To concretize discussion and to identify query types and requirements, we now give several example queries related to service discovery. Queries are initially expressed in prose. Some of them will later be formalized in a suitable query language. One can distinguish three types of queries: *simple, medium* and *complex*. The latter are more powerful than the former. Nevertheless, even a simple query is a powerful tool.

2.1. Simple Query. Simple queries are most often used for discovery. A simple query finds all tuples (services) matching a given predicate or pattern. The query visits each tuple (service description) in a set individually, and generates a result set by applying a function to each tuple. The function usually consists of a predicate and/or a transformation. Individual answers are added to a (initially empty) result set. An empty answer leaves the result set unchanged. A simple query has the following form:

```
R = {}
for each tuple in input
```

```
  R = R UNION { function(tuple) }
endfor
return R
```

Example simple queries are:

- *(QS1) Find all (available) services.*
- *(QS2) Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*
- *(QS4) Find all local services (all service interfaces of any given service must reside on the same host).*
- *(QS5) Find all services and return their service links (instead of descriptions).*
- *(QS6) Find all CMS replica catalogs and return their physical file names (PFNs) for a given logical file name (LFN); suppress PFNs not starting with "ftp://".*
- *(QS7) Within the domain "cern.ch", find all execution services and their CPU load where cpuLoad* < 0.5 *(Assuming the operation cpuLoad() is defined on the execution service).*

In support of the wide variety of real-life questions anticipated, it should be possible to arbitrarily combine and nest all capabilities exposed in these examples. Note that the first four queries return service descriptions, whereas the others return additional or entirely different information (service links, physical file names or CPU load). We term the former queries *predicate (or filter) queries*. The structure of the result set is predetermined in the sense that query output must be a subset of query input. We term the latter queries *constructive queries*, because they construct answers of arbitrary structure and content. Predicate queries are a subset of constructive queries. A constructive query function that always returns `"Hello World"` or an empty string is legal, but not very useful.

Further, note that the queries *QS6, QS7* involve multiple independent data sources and match on dynamically delivered content (via remote invocation of operations `getPFNs` and `cpuLoad`), rather than on values being part of service descriptions. We call these queries *dynamic queries*, as opposed to *static queries*. To support dynamic queries, a query language must provide means to dynamically retrieve and interpret information from diverse remote or local sources.

Dynamic queries can sometimes be reformulated as static queries. For example, the LFN/PFN database information of query *QS6* could be made available as part of the tuple set. In practice, this is typically infeasible for reasons including database size, consistency, information hiding, security and performance. Publishing highly volatile attributes such as CPU load as part of tuples leads to stale data problems. Clearly dynamic invocation is a more appropriate vehicle to deliver CPU load. Alternatively, custom push protocols can be used [11].

2.2. Medium query. A medium query computes an answer over a set of tuples (service descriptions) as a whole. For example, it can compute aggregates like number of tuples, maximum, etc. Examples of medium queries are:

- *(QM1) Find the CMS storage service with the largest network bandwidth to my host "dummy.cern.ch" (assuming there exists a service estimating bandwidth from A to B).*
- *(QM2) Return the number of replica catalog services.*
- *(QM3) Find the two CMS execution services with minimum and maximum CPU load and return their service descriptions and load.*
- *(QM4) Return a summary of all replica catalogs and schedulers residing within the domains "cern.ch", "infn.it" and "anl.gov", grouped in ascending order by owner, domain and service type, with aggregate group cardinalities.*

The query is applied to the set as a whole. For example, *QM4* is interesting in that it involves crossing tuple boundaries, which simple hierarchical query languages typically do not support. Like a simple query, a medium query can be static or dynamic. It can be a predicate query or a constructive query.

2.3. Complex query. Complex queries are most often used for advanced discovery or brokering. Like a medium query, a complex query computes an answer over a set of tuples (service descriptions) as a whole. However, it has powerful capabilities to combine data from multiple sources. For example, it supports all database join flavors. Like any other query, a complex query can be static or dynamic. It can be a predicate query or a constructive query. Example complex queries are:

- *(QC1) 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).*
- *(QC2) Find all domains that run more than one replica catalog with CMS as owner. (Want to check for anomalies).*
- *(QC3) Find the top 10 owners of replica catalog services within the domains "cern.ch", "infn.it" and "anl.gov", and return their email, together with the number of services each of them owns, sorted by that number.*

3. Content Link, Content Provider and Publication. A *content provider* can publish a dynamic pointer called a *content link*, which in turn enables the hyper registry and third parties to retrieve (pull) the current content presented from the provider at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push).

3.1. Content Link. A *content link* may be any arbitrary URI. However, most commonly, it is an HTTP(S) URL, in which case it points to the content of a content provider, and an HTTP(S) GET request to the link must return the current (up-to-date) content. In other words, a simple hyperlink is employed. In the context of service discovery, we use the term *service link* to denote a content link that points to a service description. Content links can freely be chosen as long as they conform to the URI and HTTP URL specification [12]. Examples of content 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 book where phone=4711"
http://repcat.cern.ch/getPFNs?lfn="myLogicalFileName"
```

3.2. Content Provider. A *content provider* offers information conforming to a homogeneous global data model. In order to do so, it typically uses some kind of internal mediator to transform information from a local or proprietary data model to the global data model. A content provider can be seen as a gateway to heterogeneous content sources. The global data model is the Dynamic Data Model (DDM) introduced in Section 3.3. Content can be structured or semi-structured data in the form of any arbitrary well-formed XML document or fragment. Individual content may, but need not, have a schema (XML Schema [13]), in which case content must be valid according to the schema. All content may, but need not, share a common schema. This flexibility is important for integration of heterogeneous content.

A content provider is an umbrella term for two components, namely a presenter and a publisher. The *presenter* is a service and answers HTTP(S) GET content retrieval requests from a hyper registry or client (subject to local security policy). The *publisher* is a piece of code that publishes content link, and perhaps also content, to a hyper registry. The publisher need not be a service, although it uses HTTP(S) POST for transport of communications. The structure of a content provider and its interaction with a hyper registry and a client are depicted in Figure 3.1 (a). Note that a client can bypass a hyper registry and directly pull current content from a provider. Figure 3.1 (b) illustrates a hyper registry with several content providers and clients.

Just as in the dynamic WWW that allows for a broad variety of implementations for the given protocol, it is left unspecified how a presenter computes content on retrieval. Content can be static or dynamic (generated on the fly). For example, a presenter may serve the content directly from a file or database, or from a potentially outdated cache. For increased accuracy, it may also dynamically recompute the content on each request. Consider the example providers in Figure 3.2. A simple but nonetheless very useful content provider uses a commodity HTTP server such as Apache to present XML content from the file system. A simple `cron` job monitors the health of the Apache server and publishes the current state to a hyper registry. Another example for a content provider is a Java servlet that makes available data kept in a relational or LDAP database system. A content provider can execute legacy command line tools to publish system state information such as network statistics, operating system and type of CPU. Another example for a content provider is a network service such as a replica catalog that (in addition to servicing replica lookup requests) publishes its service description and/or link so that clients may discover and subsequently invoke it.

Providers and registries can be deployed and configured arbitrarily. For example, in a strategy for scalable administration of large cluster environments, a single shared Apache web server can easily be configured to

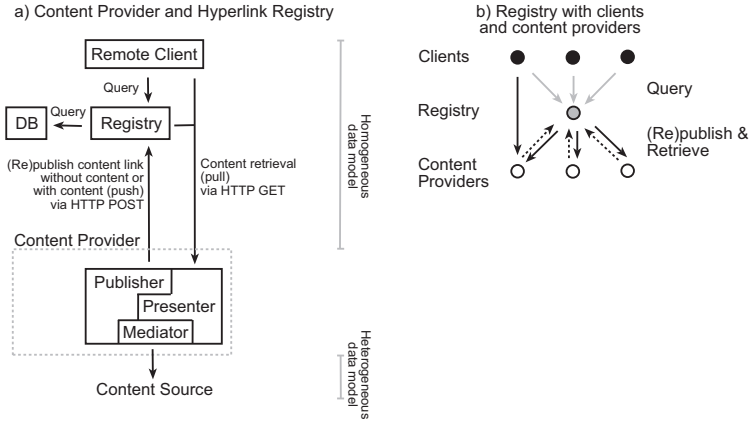

FIG. 3.1. *Content Provider and Hyper Registry.*

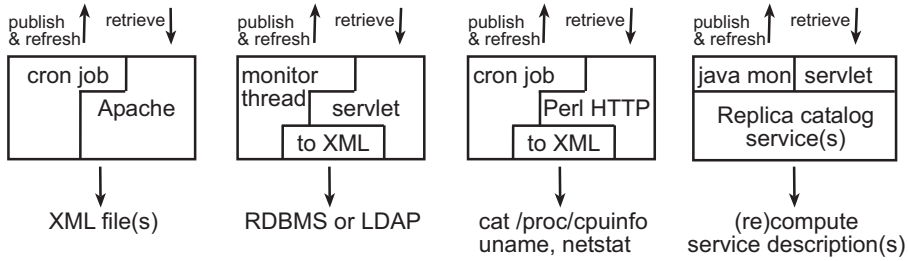

FIG. 3.2. *Example Content Providers.*

serve XML descriptions of thousands of services on hundreds of hosts. For example, via a naming convention we can assign a distinct web server directory and corresponding service link for each host-service combination. To serve descriptions it is sufficient to have some administrative `cron` job run periodically on each service host, which writes the current service description into an appropriate XML file in the appropriate directory on the web server.

Presenter and publisher are not required to run in the same process or even on the same host. For efficiency, a so-called *container* of a virtual hosting environment may be used to run more than one provider in the same process or thread. For example, a highly efficient and scalable container such as the Apache Tomcat servlet engine [14] not only can serve many hundreds to a thousand (light-weight) concurrent requests per second on a commodity PC, but it can also embed any number of dynamic provider types in the same process, with each provider type being capable of serving any number of provider instances. Typically, a provider is remote to the hyper registry. This is not a requirement, however. A provider may also be local to the hyper registry and connect through a loop-back connection. For further efficiency, a hyper registry may internally host any number of built in providers. Using commodity servlet technology, any number of hyper registries can be hosted within the same container.

3.3. Publication. In a given context, a content provider can publish content of a given type to one or more registries. More precisely, a content provider can publish a dynamic pointer called a content link, which in turn enables the hyper registry (and third parties) to retrieve the current content. For efficiency, the `publish` operation takes as input a set of zero or more tuples. In what we propose to call the *Dynamic Data Model (DDM)*, each XML tuple has a content link, a type, a context, some soft state time stamps, and (optionally) 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. Consider the dynamic tuple set depicted in Figure 3.3.

- **Link.** The content link is an HTTP(S) URL as introduced above. Given the link the current content of a content provider can be retrieved (pulled) at any time.
- **Type.** The type describes *what* kind of content is being published (e.g. `service`,

```
<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/higgs" size="1000000" type="MySQL">
             <PFN URL="ftp://storage.cern.ch/file123" readCount="17"/>
             <PFN URL="ftp://se01.infn.it/file456"    readCount="1"/>
         </replicaSet>
       </content>
    </tuple>
    <tuple link="http://hosts.cern.ch" type="hosts" TC="65" TS1="60" TS2="70" TS3="80">
       <content>
         <hosts>
             <host name="fred1.cern.ch" os="rh7.2"  arch="i386"  mem="512M" MHz="1000"/>
             <host name="fred2.cern.ch" os="sol2.7" arch="sparc" mem="8192M" MHz="400"/>
         </hosts>
       </content>
    </tuple>
</tupleset>
```

FIG. 3.3. *Example Tuple Set from Dynamic Data Model.*

`application/octet-stream`, `image/jpeg`, `networkLoad`, `hostinfo`).

- **Context.** The context describes *why* the content is being published or *how* it should be used (e.g. `child, parent, x-ireferral, gnutella, monitoring`). Note that it is often not meaningful to embed context information in the content itself, because a given content can be published in a different context at different hyper registries (1:N association). For example, a service can be a child of some nodes and at the same time be a parent for some other nodes. However, its service description (content) should clearly remain invariant. In addition, context and type allow a query to differente on crucial attributes even if content caching is not supported or not authorized.
- **Timestamps TS1, TS2, TS3, TC.** Discussion of timestamps is deferred to Section 5.2 below.
- **Metadata.** The optional metadata element further describes the content and/or its retrieval beyond what can be expressed with the previous attributes. For example, the metadata may be a secure digital XML signature [15] of the content. It may describe the authoritative content provider or owner of

the content. Another metadata example is a Web Service Inspection Language (WSIL) document [16] or fragment thereof, specifying additional content retrieval mechanisms beyond HTTP content link retrieval. The metadata argument may be any well-formed XML document or fragment. It is an extensibility element enabling customization and flexible evolution.

- **Content.** Given the link the current content of a content provider can be retrieved (pulled) at any time. Optionally, a content provider can also include a copy of the current content as part of publication (push). For clarity of exposition, the published content is an XML element[1].

The publish operation of a hyper registry has the signature `void publish(XML tupleset)`. Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. If a key does not already exist on publication, a tuple is inserted into the hyper registry database. An existing tuple can be updated by publishing other values under the same tuple key. An existing tuple (key) is "owned" by the content provider that created it with the first publication. It is recommended that a content provider with another identity may not be permitted to publish or update the tuple.

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

- *Return the number of replica catalog services.*

```
LET $repcat := "http://gridforum.org/interface/replicaCatalog-1.0"
RETURN count(/tupleset/tuple/content/service[interface/@type=$repcat])
```

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

FIG. 3.4. *Simple, Medium and Complex XQueries for Service Discovery.*

4. Query.

4.1. Minimalist Query. Clients can query the hyper registry by invoking minimalist query operations (*"select all"*-style). 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 3.3). If supported, output includes cached content. The `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 3.3 has such a form.

4.2. XQuery. A hyper registry node has the capability to execute XQueries over the set of tuples it holds in its database. XQuery [9, 10] is the standard XML query language developed under the auspices of the

[1]In the general case (allowing non-text based content types such as `image/jpeg`), the content is a MIME object. The XML based publication input set and query result set is augmented with an additional MIME multipart object [17], which is a list containing all content. The content element of the result set is interpreted as an index into the MIME multipart object. A typical hyper registry that supports caching can handle content with at least MIME content-type `text/xml` and `text/plain`.

W3C. It allows for powerful searching, which is critical for non-trivial applications. Everything that can be expressed with SQL [18] can also be expressed with XQuery. However, XQuery is a more expressive language than SQL. Simple, medium and complex XQueries for service discovery are depicted in Figure 3.4. 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 `getNetworkLoad()` operation, a query can match on values dynamically produced by that operation. For a detailed discussion of a wide range of discovery queries, their representation in the XQuery language, as well as detailed motivation and justification, see our prior studies [5]. 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` [19] 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.

4.3. Deployment. For flexibility, a hyper registry may be deployed in any arbitrary way (*deployment model*). 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 certainly also be dynamically recomputed or kept in a remote relational database (table), for example as follows:

Link	Context	Type	TS1	TC	TS2	TS3	Content
`http://sched001.cern.ch/ getServiceDescription`	Parent	Service	10	15	20	30	<service> A </service>
`http://sched.infn.it:8080/pub/ getServiceDescription`	Child	Service	20	25	30	40	<service> B </service>
`http://repcat.cern.ch/pub/ getServiceDescription?id=4711`	Child	Service	30	0	40	50	null
`http://repcat.cern.ch/pub/ getStatistics`	Null	RepStats	60	65	70	80	<repcatStats> ... </repcatStats>

5. Caching and Soft State.

5.1. Caching. Content *caching* is important for client efficiency. The hyper registry may not only keep content links but also a copy of the current content pointed to by the link. With caching, clients no longer need to establish a network connection for each content link in a query result set in order to obtain content. This avoids prohibitive latency, in particular in the presence of large result sets. A hyper registry may (but need not) support caching. A hyper registry that does not support caching ignores any content handed from a content provider. It keeps content links only. Instead of cached content it returns empty strings. Cache coherency issues arise. The query operations of a caching hyper registry may return tuples with stale content, i.e. content that is out of date with respect to its master copy at the content provider.

A caching hyper registry may implement a *strong* or *weak cache coherency policy*. A strong cache coherency policy is *server invalidation* [20]. Here a content provider notifies the hyper registry with a publication tuple whenever it has locally modified the content. We use this approach in an adapted version where a caching hyper registry can operate according to the client push pattern (*push hyper registry*) or server pull pattern (*pull hyper registry*) or a hybrid thereof. The respective interactions are as follows:

- **Pull Hyper Registry.** A content provider publishes a content link. The hyper registry then pulls the current content via content link retrieval into the cache. Whenever the content provider modifies the content, it notifies the hyper registry with a publication tuple carrying the time the content was last modified. The hyper registry may then decide to pull the current content again, in order to update the cache. It is up to the hyper registry to decide if and when to pull content. A hyper registry may pull content at any time. For example, it may dynamically pull fresh content for tuples affected by a query. This is important for frequently changing dynamic data such as network load.
- **Push Hyper Registry.** A publication tuple pushed from a content provider to the hyper registry contains not only a content link but also its current content. Whenever a content provider modifies content, it pushes the new content to the hyper registry, which may update the cache accordingly.

- **Hybrid Hyper Registry.** A hybrid hyper registry implements both pull and push interactions. If a content provider merely notifies that its content has changed, the hyper registry may choose to pull the current content into the cache. If a content provider pushes content, the cache may be updated with the pushed content. This is the type of hyper registry subsequently assumed whenever a caching hyper registry is discussed.

A non-caching hyper registry ignores content elements, if present. A publication is said to be *without content* if the content is not provided at all. Otherwise, it is said to be *with content*. Publication without content implies that no statement at all about cached content is being made (neutral). It does *not* imply that content should not be cached or invalidated.

A client must not assume that content is cached. For example, a hyper registry may not implement caching or it may be denied authorization when attempting to retrieve a given content, or it may ignore content provided on provider push. 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 the hyper registry, 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. In addition, a hyper registry may have an authorization policy. For example, depending on the identity of a client, a registry may return a subset of the full result set or hide cached content and instead return the tuple substituted by empty content. Similarly, depending on content provider identity, pushed content may be ignored or rejected, or publication denied altogether.

5.2. Soft State. For reliable, predictable and simple distributed state maintenance, a hyper registry tuple is maintained as *soft state*. A tuple may eventually be discarded unless refreshed by a stream of timely confirmation notifications from a content provider. To this end, a tuple carries timestamps. A tuple is expired and removed unless explicitly renewed via timely periodic publication, henceforth termed *refresh*. In other words, a refresh allows a content provider to cause a content link and/or cached content to remain present for a further time.

The strong cache coherency policy *server invalidation* is extended. For flexibility and expressiveness, the ideas of the Grid Notification Framework [21] are adapted. The publication operation takes four absolute time stamps `TS1, TS2, TS3, TC` per tuple. The semantics are as follows. The content provider asserts that its content was last modified at time `TS1` and that its current content is expected to be valid from time `TS1` until at least time `TS2`. It is expected that the content link is alive between time `TS1` and at least time `TS3`. Time stamps must obey the constraint $\texttt{TS1} \leq \texttt{TS2} \leq \texttt{TS3}$. `TS2` triggers expiration of cached content, whereas `TS3` triggers expiration of content links. Usually, `TS1` equals the time of last modification or first publication, TS2 equals `TS1` plus some minutes or hours, and `TS3` equals `TS2` plus some hours or days. For example, `TS1`, `TS2` and `TS3` can reflect publication time, 10 minutes, and 2 hours, respectively.

A tuple also carries a timestamp `TC` that indicates the time when the tuple's embedded content (not the provider's master copy of the content) was last modified, typically by an intermediary in the path between client and content provider (e.g. the hyper registry). If a content provider publishes with content, then we usually have `TS1=TC`. `TC` must be zero-valued if the tuple contains no content. Hence, a hyper registry not supporting caching always has `TC` set to zero. For example, a highly dynamic network load provider may publish its link without content and `TS1=TS2` to suggest that it is inappropriate to cache its content. Constants are published with content and `TS2=TS3=infinity, TS1=TC=currentTime`.

These soft state time stamp semantics are summarized in Figure 5.1.

Time Stamp	**Semantics**
`TS1`	Time provider last modified content
`TC`	Time hyper registry last modified content cache
`TS2`	Expected time while current content at provider is at least valid
`TS3`	Expected time while content link at provider is at least valid (alive)

FIG. 5.1. *Soft State Time Stamp Semantics.*

Insert, update and delete of tuples occur at the timestamp-driven state transitions summarized in Figure 5.2.

Within a tuple set, a tuple is uniquely identified by its *tuple key*, which is the pair `(content link, context)`. A tuple can be in one of three states: *unknown*, *not cached*, or *cached*. A tuple is unknown if it is not contained in the hyper registry (i.e. its key does not exist). Otherwise, it is known. When a tuple is assigned *not cached* state, its last internal modification time `TC` is (re)set to zero and the cache is deleted, if present. For a *not cached* tuple we have `TC < TS1`. When a tuple is assigned *cached* state, the content is updated and `TC` is set to the current time. For a *cached* tuple, we have `TC` $\geq$ `TS1`.

A tuple moves from *unknown* to *cached* or *not cached* state if the provider publishes with or without content, respectively. A tuple becomes *unknown* if its content link expires (`currentTime > TS3`); the tuple is then deleted. A provider can force tuple deletion by publishing with `currentTime > TS3`. A tuple is upgraded from *not cached* to *cached* state if a provider push publishes with content or if the hyper registry pulls the current content itself via retrieval. On content pull, a hyper registry may leave `TS2` unchanged, but it may also follow a policy that extends the lifetime of the tuple (or any other policy it sees fit). A tuple is degraded from *cached* to *not cached* state if the content expires. Such expiry occurs when no refresh is received in time (`currentTime > TS2`), or if a refresh indicates that the provider has modified the content (`TC < TS1`).

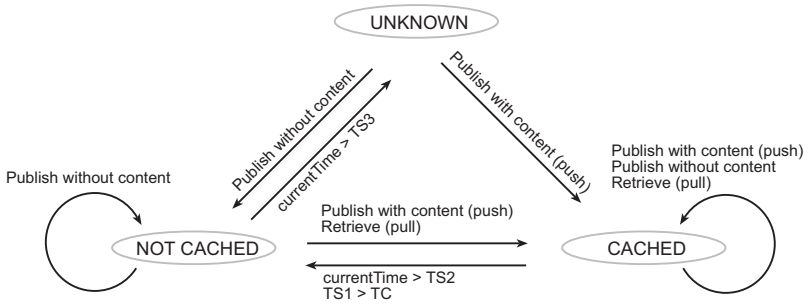

FIG. 5.2. *Soft State Transitions.*

6. Flexible Freshness. Content link, content cache, a hybrid pull/push communication model and the expressive power of XQuery allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness. For example, a content provider can model the freshness of its content via pushing appropriate timestamps and content. A hyper registry can model the freshness of its content via controlled acceptance of provider publications and by actively pulling fresh content from the provider. If a result (e.g. network statistics) is up to date according to the hyper registry, but out of date according to the client, the client can pull fresh content from providers as it sees fit. However, this is inefficient for large result sets. Nevertheless, it is important for clients that query results are returned according to their notion of freshness, in particular in the presence of frequently changing dynamic content.

Recall that it is up to the hyper registry to decide to what extent its cache is stale, and if and when to pull fresh content. For example, a hyper registry may implement a policy that dynamically pulls fresh content for a tuple whenever a query touches (affects) the tuple. For example, if a query interprets the content link URL as an identifier within a hierarchical name space (e.g. as in LDAP) and selects only tuples within a sub-tree of the name space, only these tuples should be considered for refresh.

6.1. Refresh-on-client-demand. So far, a hyper registry must guess what a client's notion of freshness might be, while at the same time maintaining its decisive authority. A client still has no way to indicate (as opposed to force) its view of the matter to a hyper registry. We propose to address this problem with a simple and elegant *refresh-on-client-demand* strategy under control of the hyper registry's authority. The strategy exploits the rich expressiveness and dynamic data integration capabilities of the XQuery language. The client query may itself inspect the time stamp values of the set of tuples. It may then decide itself to what extent a tuple is considered interesting yet stale. If the query decides that a given tuple is stale (e.g. if `type="networkLoad" AND TC < currentTime() - 10`), it calls the XQuery `document(URL contentLink)` function with the corresponding content link in order to pull and get handed fresh content, which it then processes in any desired way.

This mechanism makes it unnecessary for a hyper registry to guess what a client's notion of freshness might be. It also implies that a hyper registry does not require complex logic for query parsing, analysis, splitting, merging, etc. Moreover, the fresh results pulled by a query can be reused for subsequent queries. Since the query is executed within the hyper registry, the hyper registry may implement the `document` function such that it not only pulls and returns the current content, but as a side effect also updates the tuple cache in its database. A hyper registry retains its authority in the sense that it may apply an authorization policy, or perhaps ignore the query's refresh calls altogether and return the old content instead. The refresh-on-client-demand strategy is simple, elegant and controlled. It improves efficiency by avoiding overly eager refreshes typically incurred by a guessing hyper registry policy.

This basic idea could be used in variations that are more ambitious. For example, in an attempt to improve efficiency via "batching", a query may collect the content links of all tuples it considers stale into a set, and hand the set to a `documents(URL[])` function provided by the hyper registry, which then fetches fresh content in a batched fashion. Alternatively, a query may use this approach in a non-blocking manner, merely indicating that the hyper registry should soon refresh the given tuples (asynchronously), while the old tuples are still fine for the current query. The theme can be developed further. In a hierarchical Peer-to-Peer environment with caching nodes along the query route, it may be preferable to have the `documents(URL[])` function forward the refresh set as a query to the next node along the query route, instead of directly pulling from the content provider. This refreshes all node caches along the route, possibly at the expense of increased latency. As a different type of optimization, the hyper registry may reduce latency by keeping alive the TCP connections to content providers, which is often impractical to do for clients.

To summarize, a wide range of dynamic content freshness policies can be supported, which may be driven by all three system components: content provider, hyper registry and client. All three components may indicate how to manage content according to their respective notions of freshness.

6.2. Throttling. Clearly there is a tradeoff between the resource consumption caused by refreshes and state consistency. The higher the refresh frequency, the more consistent and up-to-date the state, and the more resources are consumed. High frequency refresh can consume significant network bandwidth, due to pathological client misbehavior, denial-of-service attacks, or sheer popularity. Implementations using high frequency refresh rates can encounter serious latency limitations due to the very expensive nature of secure (and even insecure) network connection setup for publication. Keep-alive connections should be used to minimize setup time. However, they only partly address the problem. Consider, for example, an automatically adapting search engine indexing one million services, each refreshing every minute with a message of 200 bytes. Just to stay up-to-date the search engine must scale to 17000 refreshes/sec and its maintainer must pay for a WAN bandwidth of at least 3.4 MB/sec.

To condition for overload, limit resource consumption and satisfy minimum requirements on content freshness, mechanisms to *throttle* refresh frequency are proposed, adaptively inviting more or less traffic over time. For example, the search engine hyper registry may ask the content providers to wait at least 100 minutes between refreshes. Conversely, a hyper registry of a job scheduler depending on very fresh CPU load measurements from job execution services may want to invite execution service providers to refresh at least every second. For example, the service administrator can set the aggregate bandwidth available for refresh to a fraction of the total available system bandwidth. [22, 23] suggest to determine the maximum refresh rate for a given source service from historic bandwidth statistics over refreshes. This avoids overload, yet helps to maintain scalability in the number of source services.

Accordingly, the publication operation returns two time stamps `TS4` and `TS5`, which we call *minimum idle time* and *maximum idle time*, respectively. The semantics of the minimum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next refresh"*. The semantics of the maximum idle time are as follows: *"Publication was successful, but in the future you may be dropped and denied service if you do not refresh before time* `TS5` *"*. We have `minimum idle time < maximum idle time`. A simple hyper registry always returns zero and infinity as minimum idle and maximum idle time, respectively. Content providers ignoring throttling warnings can be dropped and denied service without further notice, for example at the local, firewall or Internet Service Provider (ISP) level. Analogously, query operations return a minimum idle time (`TS4`) as part of the result set. The semantics are as follows: *"The query was successful, and here is the result set. However, in the future you may be dropped and denied service if you do not wait at least until time* `TS4` *before the next query"*.

7. Related Work.

RDBMS. Relational database systems [24] provide SQL as a powerful query language. They assume tight and consistent central control and hence are infeasable in Grid environments, which are characterized by heterogeneity, scale, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. They do not support an XML data model and the XQuery language. The relational data model does not allow for semi-structured data. A query must have out-of-band knowledge of the relevant table names and schemas, which themselves may not be heterogeneous but must be static, globally standardized and synchronized. This seriously limits the applicability of the relational model in the context of autonomy, decentralization, unreliability and frequent change. SQL lacks hierarchical navigation as a key feature and other capabilities such as dynamic data integration, leading to extremely complex queries over a large number of auxiliary tables [25]. Further, RDBMS do not provide dynamic content generation, soft state based publication and content caching. Our work does not compete with an RDBMS, though. A registry may well internally use an RDBMS for data management. A registry can accept queries over an XML view and internally translate the query into SQL [26, 25]. The relational data model and SQL are, for example, used in the Relational Grid Monitoring Architecture (RGMA) system [27] and the Unified Relational GIS Project [28].

Web Proxy Caches. A weak cache coherency policy popular with web proxy caches is *adaptive TTL* [20]. Here the problem is handled by adjusting the time-to-live of a content based on observations of its lifetime. Adaptive TTL takes advantage of the fact that content lifetime distribution tends to be bimodal; if a given content has not been modified for a long time, it tends to stay unchanged. Thus, the time-to-live attribute of a given content is assigned to be a percentage of the content's current "age", which is the current time minus the last modified time of the document.

The "web server accelerator" [29] resides in front of one or more web servers to speed up user accesses. It provides an API, which allows application programs to explicitly add, delete, and update cached data. The API allows the accelerator to cache dynamic as well as static data. Invalidating and updating cached data is facilitated by the Data Update Propagation (DUP) algorithm, which maintains data dependence information between cached data and underlying data in a graph [30].

ANSA and CORBA. The ANSA project was an early collaborative industry effort to advance distributed computing. It defined trading services [31] for advertizement and discovery of relevant services, based on service type and simple constraints on attribute/value pairs. The CORBA Trading service [32] is an evolution of these efforts.

UDDI. UDDI (Universal Description, Discovery and Integration) [33] is an emerging industry standard that defines a business oriented access mechanism to a registry holding XML based WSDL [8] service descriptions. It is not designed to be a registry holding arbitrary content. UDDI is not based on soft state, which implies that there is no way to dynamically manage and remove service descriptions from a large number of autonomous third parties in a reliable, predictable and simple way. It does not address the fact that services often fail or misbehave or are reconfigured, leaving a registry in an inconsistent state. Content freshness is not addressed. As such, UDDI only appears to be useful for businesses and their customers running static high availability services. Last, and perhaps most importantly, query support is rudimentary. Only key lookups with primitive qualifiers are supported, which is insufficient for realistic service discovery use cases (see Figure 3.4 for examples).

Jini, SLP, SDS, INS. The centralized Jini Lookup Service [34] 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) [35] 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) [36], 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) [37] 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 [38]. Like SDS, it integrates name resolution and routing.

LDAP. The Lightweight Directory Access Protocol (LDAP) [39] defines an access mechanism in which clients send requests to and receive responses from LDAP servers. The data model is not based on soft state. Content freshness is not addressed. LDAP does not follow an XML data model. An LDAP query is an expression that logically compares (`=, <=, >=`) the string value of an attribute (`email`) with a string constant, optionally with a substring match joker (`picture*.jpg`) and approximate string equality test (∼). Expressions can be combined with Boolean `AND, OR` and `NOT` operators. The expressive power of the LDAP query language is insufficient for service discovery use cases (see Figure 3.4 for examples) and most other non-trivial questions.

MDS. The Metacomputing Directory Service (MDS) [40, 41] is based on LDAP. As a result, its query language is insufficient for service discovery, and it does not follow an XML data model. MDS is based on soft state but it does not allow clients (and to some extent even content providers) to drive registry freshness policies.

The MDS consists of an unmodified OpenLDAP [42] server with value-adding backends, configured with a strong security library. The Grid Resource Information Service (GRIS) is an OpenLDAP backend that accepts LDAP queries from clients over its own LDAP namespace sub-tree. It is a backend into which a list of content providers can be plugged on a per attribute basis. An example content provider is a shell script or program that returns the operating system version. A provider owns a namespace sub tree of the GRIS and returns a set of LDAP entries within that namespace. Depending on the namespace specified in an LDAP query, the GRIS executes (and creates the processes for) one or more affected providers and caches the results for use in future queries. It then applies the query against the cache. A GRIS can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). An example GRIS configuration invokes the `/usr/local/bin/grid-info-cpu-linux` executable and caches CPU load results for 15 seconds. Content provider invocation follows a CGI like life cycle model. The stateless nature, heavy weight process forking and context switches of such a model render it unsuitable for use in dynamic environments with high frequency refreshes and requests [43].

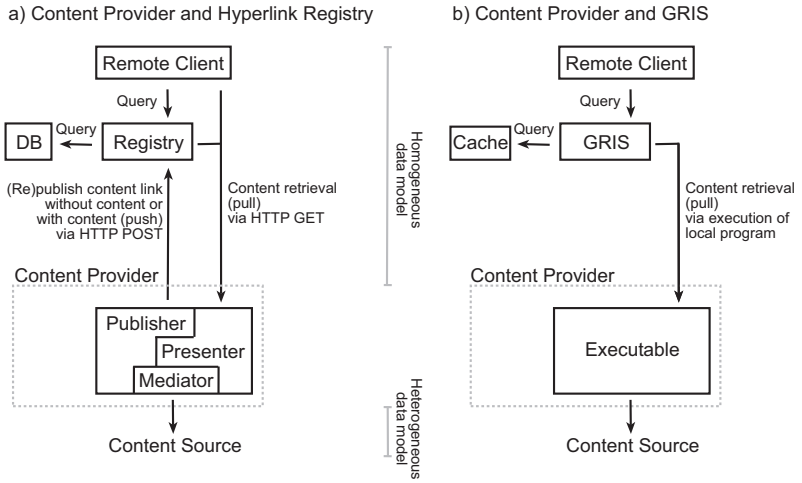

FIG. 7.1. *Hyper Registry and Grid Resource Information Service.*

Figure 7.1 contrasts the different architectures of a GRIS and a hyper registry. A hyper registry maintains content links and cached content in its database, whereas a GRIS maintains cached content only. The control paths from client to content provider and from content provider to the hyper registry are missing in the GRIS architecture, disabling cache freshness steering. A GRIS content provider is always local to the GRIS and cannot publish to a remote GRIS. In contrast, a content provider is cleanly decoupled from a hyper registry and only requires the ubiquitous HTTP protocol for simple communication with a local or remote registry. A GRIS requires implementing the complex LDAP protocol, including its query language, at every content provider location. In contrast, handling the powerful but complex XQuery language is only required within a

hyper registry, not at the content provider. Table 7.1 summarizes some commonalities and differences related to publication and content freshness.

TABLE 7.1
Comparison of Hyper Registry and Metacomputing Directory Service.

Question	**MDS**	**Hyper Registry**
Can a provider publish to a remote registry?	No. A provider is local to a GRIS.	Yes. A provider can publish to any hyper registry, no matter whether the hyper registry is deployed locally, remotely or in-process.
Can a provider actively steer registry freshness?	No. A GRIS is actively pulling content. It can be statically configured to cache the pulled content for a fixed amount of time (on a per provider basis). A content provider is passive. It cannot actively publish and refresh content and hence cannot steer the freshness of its content cached in the GRIS.	Yes. Content provider and hyper registry are active and passive at the same time. At any time, a hyper registry can actively pull content, and a content provider can actively push with or without content. Both components can steer the freshness of content cached in the hyper registry.
Can a client retrieve current content?	No. A client cannot retrieve the current content from a content provider. It has to go through a GRIS or GIIS, which normally return stale content from their cache.	Yes. A client can directly connect to a content provider and retrieve the current content, thereby avoiding stale content from a hyper registry.
Can a client query steer result freshness?	No. A client query cannot steer the freshness of the results it generates.	Yes. A client query can steer the freshness of the results it generates via the refresh-on-client-demand strategy.

8. Conclusions. We address the problems of maintaining and querying dynamic and timely information populated from a large variety of unreliable, frequently changing, autonomous and heterogeneous remote data sources. The hyper registry has a number of key properties. An XML data model allows for structured and semi-structured data, which is important for integration of heterogeneous content. The XQuery language allows for powerful searching, which is critical for non-trivial applications. Database state maintenance is based on soft state, which enables reliable, predictable and simple content integration from a large number of autonomous distributed content providers. Content link, content cache and a hybrid pull/push communication model allow for a wide range of dynamic content freshness policies, which may be driven by all three system components: content provider, hyper registry and client. A content provider is cleanly decoupled from the hyper registry and only requires the ubiquitous HTTP protocol for communication with a local or remote hyper registry.

As future work, it would be interesting to study and specify in more detail specific cache freshness interaction policies between content provider, hyper registry and client (query). Our specification allows expressing a wide range of policies, some of which we outline, but we do not evaluate in detail the merits and drawbacks of any given policy.

REFERENCES

[1] LARGE HADRON COLLIDER COMMITTEE, *Report of the LHC Computing Review,* Technical report, CERN/LHCC/2001-004, April 2001, `http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF`

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

[3] 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.

[4] 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.
[5] 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.
[6] 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, `http://www.globus.org.`
[7] 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.
[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1.* W3C Note 15, 2001, `http://www.w3.org/TR/wsdl`
[9] World Wide Web Consortium, *XQuery 1.0: An XML Query Language,* W3C Working Draft, December 2001.
[10] World Wide Web Consortium, *XML Query Use Cases* W3C Working Draft, December 2001.
[11] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany, *A Grid Monitoring Architecture,* Technical report, Global Grid Forum Informational Document, January 2002, `http://www.gridforum.org`
[12] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax,* IETF RFC 2396.
[13] World Wide Web Consortium, *XML Schema Part 0: Primer,* W3C Recommendation, May 2001.
[14] Apache Software Foundation, *The Jakarta Tomcat Project,* `http://jakarta.apache.org/tomcat/`
[15] World Wide Web Consortium, *XML-Signature Syntax and Processing,* W3C Recommendation, February 2002.
[16] P. Brittenham, *An Overview of the Web Services Inspection Language,* 2001, `www.ibm.com/developerworks/webservices/library/ws-wsilover`
[17] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,* IETF RFC 2045, November 1996.
[18] International Organization for Standardization (ISO), *Information Technology-Database Language SQL,* Standard No. ISO/IEC 9075:1999, 1999.
[19] Software AG, *The Quip XQuery processor,* `http://www.softwareag.com/developer/quip/`
[20] J. Wang, *A survey of web caching schemes for the Internet,* ACM Computer Communication Reviews, 29(5), October 1999.
[21] 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`
[22] C. Weider, A. Herron, A. Anantha, and T. Howes, *LDAP Control Extension for Simple Paged Results Manipulation,* IETF RFC 2696.
[23] M. P. Maher and C. Perkins, *Session Announcement Protocol: Version 2,* IETF Internet Draft draft-ietf-mmusic-sap-v2-00.txt, November 1998.
[24] M. Tamer Özsu and Patrick Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 1999.
[25] Daniela Florescu, Ioana Manolescu, and Donald Kossmann, *Answering XML Queries over Heterogeneous Data Sources,* In Int'l. Conf. on Very Large Data Bases (VLDB), Roma, Italy, September 2001.
[26] Mary Fernandez, Morishima Atsuyuki, Dan Suciu, and Tan Wang-Chiew, *Publishing Relational Data in XML: the SilkRoute Approach,* IEEE Data Engineering Bulletin, 24(2), 2001.
[27] Steve Fisher et al, *Information and Monitoring (WP3) Architecture Report,* Technical report, DataGrid-03-D3.2, January 2001.
[28] W. P. Dinda and B. Plale, *A Unified Relational Approach to Grid Information Services.* Technical report, Grid Forum Informational Draft GWD-GIS-012-1, February 2001, `http://www.gridforum.org`
[29] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias, *Design and performance of Web server accelerator.* In Proceedings of Infocom'99, 1999.
[30] J. Challenger, A. Iyengar, and P. Dantzig, *A scalable system for consistently caching dynamic Web data,* In Proceedings of Infocom'99, 1999.
[31] 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.
[32] Object Management Group, *Trading Object Service,* OMG RPF5 Submission, May 1996.
[33] UDDI Consortium, *UDDI: Universal Description, Discovery and Integration,* `http://www.uddi.org`
[34] J. Waldo, *The Jini architecture for network-centric computing,* Communications of the ACM, 42(7), July 1999.
[35] Erik Guttman, *Service Location Protocol: Automatic Discovery of IP Network Services,* IEEE Internet Computing Journal, 3(4), 1999.
[36] 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.
[37] 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.
[38] 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.
[39] W. Yeong, T. Howes, and S. Kille, *Lightweight Directory Access Protocol,* IETF RFC 1777, March 1995.
[40] 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.
[41] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke, *A Directory Service for Configuring High-Performance Distributed Computations,* In 6th Int'l. Symposium on High

Performance Distributed Computing (HPDC '97), 1997.
[42] The OpenLDAP project, *The OpenLDAP project,* `http://www.openldap.org`
[43] A. Wu, H. Wang, and D. Wilkins, *Performance Comparison of Web-To-Database Applications,* In Proceedings of the Southern Conference on Computing, The University of Southern Mississippi, October 2000.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: August 12, 2002
Accepted: December 13, 2002