

Service-Based Distributed Querying on the Grid

M. Nedim Alpdemir¹, Arijit Mukherjee², Norman W. Paton¹, Paul Watson²,
Alvaro A.A. Fernandes¹, Anastasios Gounaris¹, and Jim Smith²

¹ Department of Computer Science ² School of Computing Science
University of Manchester University of Newcastle upon Tyne
Oxford Road, Manchester M13 9PL Newcastle upon Tyne NE1 7RU
United Kingdom United Kingdom

Abstract. Service-based approaches (such as Web Services and the Open Grid Services Architecture) have gained considerable attention recently for supporting distributed application development in e-business and e-science. The emergence of a service-oriented view of hardware and software resources raises the question as to how database management systems and technologies can best be deployed or adapted for use in such an environment. This paper explores one aspect of service-based computing and data management, viz., how to integrate query processing technology with a service-based Grid. The paper describes in detail the design and implementation of a service-based distributed query processor for the Grid. The query processor is service-based in two orthogonal senses: firstly, it supports querying over data storage and analysis resources that are made available as services, and, secondly, its internal architecture factors out as services the functionalities related to the construction of distributed query plans on the one hand, and to their execution over the Grid on the other. The resulting system both provides a declarative approach to service orchestration in the Grid, and demonstrates how query processing can benefit from dynamic access to computational resources on the Grid.

1 Introduction

The Grid is an emerging infrastructure that supports the discovery, access and use of distributed computational resources [7]. Its name comes by analogy with the electrical power grid, in that the intention is that computational resources (by analogy with power generators) should be able to be accessed on demand, with the location and ownership of the resources being orthogonal to their manner of use. Although the Grid was originally devised principally to support scientific applications, the functionalities associated with middlewares, such as Globus [www.globus.org] and Unicore [www.unicore.de], are potentially relevant to applications from many domains, in particular those with demanding, but unpredictable, computational requirements. For the most part, Grid middlewares abstract over platform or protocol-specific mechanisms for authentication, file access, data movement, application invocation, etc., and allow dynamic deployment of applications on diverse hardware and software platforms.

In parallel with the development of Grid computing, *Web Services* (WSs) [9] have gained widespread acceptance as a way of providing language and platform-independent mechanisms for describing, discovering, invoking and orchestrating collections of networked computational services. For the most part, WSs are static, in that specific services are deployed, described and advertised with human intervention, and with fixed computational capabilities.

The principal strengths of Web and Grid services thus seem to be complementary, with WSs focusing on platform-neutral description, discovery and invocation, and Grid services focusing on the dynamic discovery and efficient use of distributed computational resources. This complementarity of Web and Grid Services has given rise to the proposed Open Grid Services Architecture (OGSA) [6, 18], which makes the functionality of *Grid Services* (GSs) available through WS interfaces. The Open Grid Services Infrastructure (OGSI) is the base infrastructure underlying the OGSA. It allows the dynamic creation of service instances on computational resources that are discovered and allocated as, and when, they are needed. The OGSI is currently undergoing a standardisation process through the Global Grid Forum [www.gridforum.org].

Although the initial emphasis in Grid computing was on file-based data storage [15], the importance of structured data management to typical Grid applications is becoming widely recognised, and several proposals have been made for the development of Grid-enabled database services (e.g., Spitfire [1], OGSA-DAI [www.ogsa-dai.org.uk]). To simplify somewhat, a Grid-enabled database service is a programmatic interface to a database that uses one or more GSs (e.g., for authentication or data transport).

The provision of facilities that support application development is relevant to both GSs and WSs. For example, in the Grid setting, applications can use GSs through toolkits, or Grid-enabled versions of parallel programming libraries such as MPI [5]. In the WS setting, tools exist to support the generation of client stubs (e.g., AXIS [www.apache.org]), but, more ambitiously, XML-based workflow languages have been developed to orchestrate WSs, of which BPEL4WS [www.ibm.com/developerworks/library/ws-bpel] is perhaps the most mature. However, all of these approaches are essentially procedural in their nature, and place significant responsibility on programmers to specify the most appropriate order of execution for a collection of service requests and to obtain adequate resources for the execution of computationally demanding applications. Furthermore, support for large-scale processing of structured data is minimal.

This paper argues that distributed query processing (DQP) can provide effective declarative support for service orchestration, and describes an approach to service-based DQP on the Grid that:

1. supports queries over *Grid Database Service* (GDSs) and over other services available on the Grid, thereby combining data access with analysis;
2. uses the facilities of the OGSA to dynamically obtain the resources necessary for efficient evaluation of a distributed query;
3. adapts techniques from parallel databases to provide implicit parallelism for complex data-intensive requests; and

- uses the emerging standard for GDSs to provide consistent access to database metadata and to interact with databases on the Grid.

As well as using the emerging GDS standard, the Grid distributed query processor described in this paper is itself a GDS, and thus can be discovered and invoked in the same way as other GDSs. Thus, the Grid stands to benefit from DQP, through the provision of facilities for declarative request formulation that complement existing approaches to service orchestration. Furthermore, a complementary claim is that DQP stands to benefit from GSs, since they facilitate dynamic discovery and allocation of computational resources, as required to support computationally demanding database operations (such as joins), and implicit parallelism for complex analyses.

The remainder of this paper is structured as follows. Section 2 describes how GSs evolved from WSs. Section 3 motivates and describes current efforts to provide high-level database access services in the Grid. Section 4 contains the technical contributions of the paper, viz., a detailed description of how a DQP engine can be realised that uses Grid services both as architectural components in the design of the engine itself and as nodes in distributed query execution plans. The engine described has been implemented for first public release in July 2003 and is referred to as **OGSA-DQP**. Section 5 draws contrasts with other work on distributed query processing. Finally, Section 6 states some conclusions.

2 Grid Services

Among the drivers for the evolution to service-based architectures are the networking infrastructure now available, of course, and, more importantly, the emergence, in both e-business and e-science, of a cooperation model referred to as a *virtual organisation* [7]. The service-based approach seems to many a good solution to the problem of modelling a virtual organisation as a distributed system. GSs build upon and extend the service-oriented architecture and technologies first proposed for WSs [9]. Figure 1 shows the structure of a GS instance. WSs cannot be created dynamically and are stateless from the viewpoint of the requester.

The Grid community has sought to address these limitations, thereby responding to the needs of applications in the high performance distributed computing area where dynamic allocation of highly shared, powerful resources is essential. Thus, the OGSA [6, 18] proposes interfaces for GSs that, unlike WSs, can be instantiated dynamically by a call to their factory and are stateful.

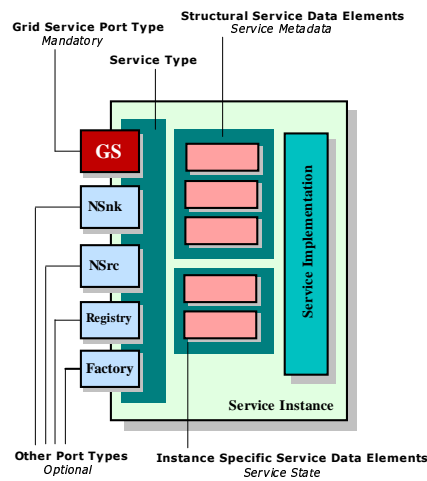


Fig. 1. Internal Structure of a GS

A GS must define a **Grid Service** port type, and may define optional port types, such as **Notification Sink** (NSnk) and **Notification Source** (NSrc) in Figure 1, as discussed below. Associated with a GS is a potentially dynamic set of *Service Data Elements* (SDEs). An SDE is a named and typed XML element that is used to represent information about GS instances, thereby opening the way for the discovery and management of potentially dynamic GS instance state.

The OGSA specifies the following functionalities:

- **Registration and Discovery** – A standard interface is defined for registering information about GS instances with registry services. A GS that also implements the **Registry** port type becomes a *Grid Service Registry* (GSR). The `findServiceData` operation can be used to query registry metadata.
- **Dynamic Service Creation** – One of the most important characteristics of the OGSA is its acknowledgement of the need to create and manage new GS instances dynamically. A **Factory** port type is defined (with the expected semantics) to achieve this. Once created, a GS instance is denotable by a globally unique *Grid Service Handle* (GSH).
- **Lifetime Management** – Because GSs can be created dynamically, they need to be disposed of. The **GS** port type defines operations for managing the lifetime of a GS instance and for reclaiming services and state associated with failed operations. The OGSA adopts soft-state protocols that enable the host itself to discard a GS instance if the stream of keep-alive messages that reaffirm the need for it to be retained dries up.
- **Notification** – Common abstractions and service interfaces for subscription to, and delivery of, event notifications are supported through the notification sink and source port types, so that dynamic, distributed GSs can asynchronously notify each other of relevant changes to their state.

The provision of an infrastructure with the properties above is an important step towards fulfilling the goal of supporting virtual organisations. Concretely, this goal requires that, among other tasks, the access to, and management of, Grid resources can be virtualised. By virtualised is meant, roughly, that the levels of cohesion and coupling that obtain among application components are such as to allow developers to express the needs of a virtual organisation for data and computational resources at the grain of complete, possibly federated, databases, and of complete, possibly orchestrated, applications. Initiatives to provide value-adding services in specific settings are being pursued, e.g., security, or transport. Of particular importance for the work presented in this paper, are the data access and integration services discussed in Section 3.

3 Grid Database Services

While query processing technology is characterised by high levels of cohesion, coupling it to applications has often required special attention. This need has occurred at several granularities, from programs to distributed systems.

The service-based approach to data access and integration offers very low levels of coupling (because of its high-level consistent interfaces) at a very coarse granularity (e.g., that of a, possibly federated, database system). From this fact stem challenges and opportunities as this and the remaining sections show.

The main objective of the OGSA-DAI initiative is to build upon the OGSA infrastructure to deliver added-value, high-level data management functionality for the Grid. OGSA-DAI components are either *data access components* or *data integration components*. The former give access to stored collections of structured data managed by database management systems implementing standard data models. The latter provide facilities for combining or transforming data from multiple data access components in order to present some integrated or derived view of the data. The DQP system introduced in this paper is an example of a data integration component providing combination facilities. OGSA-DAI extends GSs with the following services and port types:

- A *Grid Data Service Registry* (GDSR) is a facility for the publication of GDSs. Services are registered with a GDSR via a `registerService` operation. The data that is used to describe a service for registration is generally a subset of the SDEs exposed by the GDS. Registered services (and their capabilities) can be found using a `findServiceData` call. A GDSR is also capable of reporting registered service changes to service instances that have subscribed for this via a `subscribe` call.
- A *Grid Data Service Factory* (GDSF) is configurable to create GDS instances tailored to specific requests. A `createService` operation is passed configuration information for the requested GS instance, including database management system location, database and query language.
- The **Grid Data Service** (GDS) port type is the core contribution of OGSA-DAI. It accepts a request (as an XML document) which instructs the GDS instance to interact with a database to create, retrieve, update or delete data. The primary operation in the GDS port type is `perform`, through which a request can be passed to the GDS (e.g., for an SQL query to be evaluated).
- The **Grid Data Transport** (GDT) port type enables data transport between GDSs and between client processes and GDSs. It is used by the GDS to satisfy delivery requests and allows data to be pushed or pulled through `putData` and `getData` calls, respectively. The GDT port type is crucial for efficient and reliable data transport when large data volumes prevail.

Figure 2 illustrates the interaction of these services and port types in a typical scenario. Circled numbers (written inside parentheses in running text) denote the position of an interaction in a sequence. Solid arrows denote invocation. Dashed ones, instantiation. (1) A GDSF registers itself with one or more registries (possibly a virtual organisation registry) as part of its initialisation procedure. (2) A client discovers the GDSF by issuing a `findServiceData` request to the GDSR. (3) The client uses the GDSF to create a GDS instance. (4) The client submits queries using the `perform` call in the GDS port type. (5) The results of the query are then delivered by the GDS instance to the consumer (assumed in Figure 2, for illustration purposes, to be different from the client).

The current OGSA-DAI software (V 2.5) is layered on top of the OGSI reference implementation (initially known as OGSI) released as Globus Toolkit 3 (Beta) in June 2003. Both are available under open source licenses, from [www.globus.org/ogsa] and [www.ogsa-dai.org.uk].

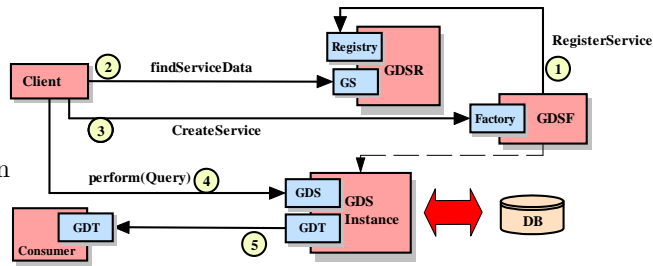


Fig. 2. OGSA-DAI Service Interactions

4 A Service-Based DQP Architecture

This section describes a query processing framework in which query compilation, optimisation and evaluation are viewed (and implemented) as invocations of OGSA-compliant GSs. Moreover, both the execution nodes and the data nodes of the distributed query execution plans are OGSA-compliant GSs. Benefits include the following: (i) Grid services can be used to identify lightly loaded resources that are suitable for query evaluation and to allocate query evaluators on these nodes; (ii) Grid security supports single sign-on for remote resources, simplifying authentication for distributed execution; (iii) since source databases and intermediate evaluators support consistent interfaces for data delivery, the design of the query processing framework is simplified; (iv) consistent resource discovery and allocation mechanisms can be used for both data sources and analysis tools accessed from a query.

The query processing framework presented in this section extends the OGSA and OGSA-DAI with two new services (and their corresponding factories):

- A **Grid Distributed Query Service (GDQS)** extends the GDS interface. When a GDQS is set up, it interacts with the appropriate registries to obtain the metadata and computational resource information that it needs to compile, optimise, partition and schedule distributed query execution plans over multiple execution nodes in the Grid. The implementation of the GDQS builds on our previous work on the Polar* distributed query processor for the Grid [17] by encapsulating its compilation and optimisation functionality.
- A **Grid Query Evaluation Service (GQES)** also extends the GDS interface. Each GQES instance is an execution node in the distributed query plans alluded to above. It is in charge of a partition of the query execution plan assigned to it by a GDQS. It implements a physical algebra over GDSs (encapsulated within which lie the actual data sources whose schemas were imported during GDQS set-up).

Figure 3 provides an overview of the interactions during the instantiation and set-up of a GDQS as well as those that take place when a query is received and processed. Conventions are as for Figure 2, with dotted arrows labelled by dark-background sequence numbers denoting interactions that take place in the set-up phase, which occurs only once in the lifetime of the GDQS instance. Solid arrows denote interactions that take place when a query is submitted. The 3-dot sequence in Figure 3 can, as usual, be read as ‘and so on, up to’.

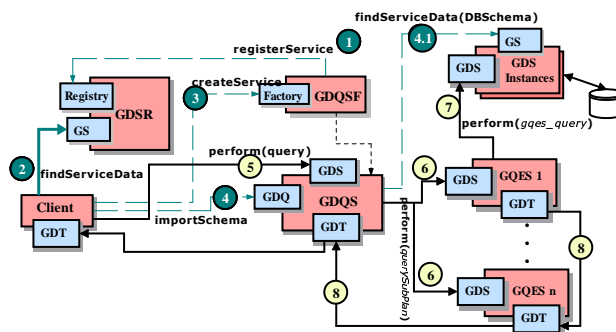


Fig. 3. GDQS Interactions: Set-Up and Querying

Note that since the GDQS is OGSA-DAI compliant, interactions (1) to (3) are the same as those illustrated in Figure 2. The fourth interaction is specific to a GDQS (and hence justifies the additional GDQ port type). It allows the GDQS to import logical and physical schemas of the participating data sources as well as information about computational resources into the metadata dictionary used by the query compiler and optimiser.

After importing the schemas of the participating data source, the client can submit queries (5) via the GDS port type using a `perform` call. This call is compiled and optimised into a distributed query execution plan, the partitions of which are scheduled for execution at different GQESs. The GDQS then uses this information to create GQES instances on their designated execution nodes (and these could be, potentially, anywhere in the Grid) and hands over to each (6) the plan partition assigned to it (as described in more detail in Section 4.2). This is what allows the DQP framework described in this paper to benefit from (implicitly) parallel evaluation even as the uniform service-based interfaces hide most of the low-level complexity necessary to achieve this. Finally, (some of the) GQES instances interact (7) with other GDS instances to obtain data, after which the results start to propagate (8) across GQES instances and, eventually, back to the client via the GDT port type.

In the remainder of the paper, the characteristic query in Figure 4 is used to illustrate issues and challenges and to describe the solutions to them that this paper contributes. The setting is that of the ODMG [2] data model and its associated query language, OQL. This query returns, for each protein annotated with the GO term ‘GO:0008372’ (i.e., unknown cellular component), those proteins that are similar to it. Assume that (as in [17]) the `protein` and `proteinTerm` ex-

```
select p.proteinId, blast(p.sequence)
from p in protein, t in proteinTerm
where t.termId = 'GO:0008372' and
       p.proteinId = t.proteinId
```

Fig. 4. Example Query

are similar to it. Assume that (as in [17]) the `protein` and `proteinTerm` ex-

tents are retrieved from two databases, respectively: the Genome Information Management System (GIMS) [img.cs.man.ac.uk/gims] and the Gene Ontology (GO) [www.geneontology.org], each running under (separate) MySQL relational database management systems. The query also calls the BLAST sequence similarity program [www.ncbi.nlm.nih.gov/BLAST/], which, given a protein sequence, returns a set of structures containing protein IDs and similarity scores. Note that the query retrieves data from two relational databases, and invokes an external application on the join results.

A service-based approach to processing this query over a distributed environment allows the query optimiser to choose from multiple providers (in the safe knowledge that most heterogeneities are encapsulated behind uniform interfaces), and to spawn multiple copies of a query-algebraic operator to exploit parallelism. In the example query, for instance, the optimiser could choose between different GO and GIMS databases, different BLAST services, and different nodes for running GQES instances¹. Moreover, a DQP engine built upon GSs can offer better long-term assurances of efficiency because dynamic service discovery, creation and configuration allow it to take advantage of a constantly changing resource pool which would be troublesome for other approaches.

4.1 Setting Up a Distributed Query Service

The GDQS service type (i.e., the collection of port types it supports) implements two port types from the OGSA, viz., **GS** and **NSnk**, and two from OGSA-DAI, viz., **GDS** and **GDT**. To these, it adds a **Grid Distributed Query (GDQ)** port type that allows source schemas to be imported. Note that this provides a context over which global models can be specified (e.g., using views), but the query processor itself makes no attempt at schema integration. Such capabilities could be provided by higher-level schema-integration services, potentially also implementing GDS port types. The steps involved in identifying and accessing the participating data sources depend on the lifetime model applied to GDS instances that wrap those data sources. We assume that the GDS instances are created per-client and can handle multiple queries but are still relatively short-lived entities. This model is also applicable to the lifetime of the GDQS instance. Note that other approaches to lifetime management are possible, each having particular advantages and disadvantages. For instance, the GDQS could be implemented to serve multiple queries from multiple users for a long period of time.

In that case the cost of setting up the system would be minimal, at the expense of somewhat increased complexity in handling query requests because of the need to manage multiple simultaneous requests and the need to manage resource allocation for the corresponding interactions. On the other hand, the GDQS instance could be designed to be a per-query, short-lived entity, in which case, the cost of setting up the system would constitute a considerable proportion of the total lifetime of the instance. The approach adopted for our

¹ The first release of **OGSA-DQP** does not support this facility because of limited functionality in the OGSi implementation it builds upon.

particular implementation (i.e. an instance per-user that is capable of responding to multiple requests), avoids the complexity of multi-user interactions while ensuring that the cost of system set-up phase is not the dominating one. We also assume that the client knows enough about the data sources to discover the GDS factories for them, and to hand over their GSHs to the GDQS. The GDQS can then request that those GDS instances be created (thereby obtaining control over the lifetimes of the GDS instances), and imports their schemas. This model balances the responsibilities of the client and those of the GDQS by assigning instance creation and lifetime management to the latter, while leaving identification and discovery of the data sources to the former. Figure 5 shows the detailed interaction sequence before and during an `importSchema` call. Notation is as in Figure 2.

(1) reflects the default behaviour for a GDSF of registering itself upon activation. In the figure, the client starts by discovering (2) a GDQS factory and creating from it (3) a GDQS instance *GDQS1*. The client then discovers (4) a GDS Factory for a particular data source, obtains (5) a configuration document by querying the GDSF), and passes (6) the handle *GSH:GDSF* of this factory and the configuration document to *GDQS1* via an `importSchema` call.

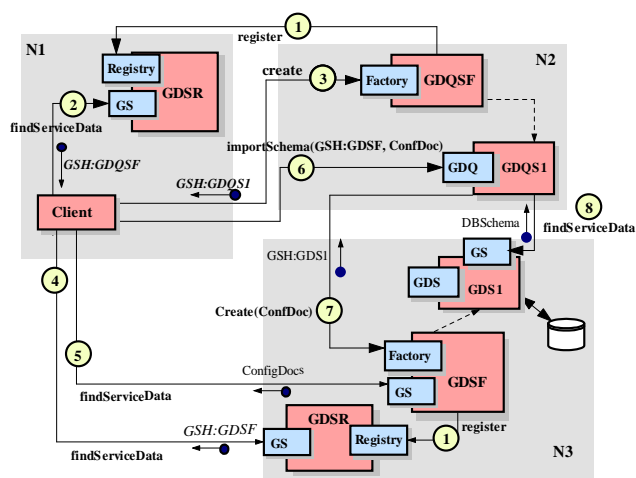


Fig. 5. Importing Service Metadata

The GDQS instance creates (7) a GDS instance *GDS1* using the factory handle and the configuration document provided by the client, and obtains (8) the database schema of the data source wrapped by that GDS.

It is also necessary to import metadata about external services (e.g., BLAST, in the example query) that are required to participate in the queries the GDQS supports. The participation of an external service occurs when the service is called from within the OQL query. For instance, in the example query, the call to the BLAST sequence similarity algorithm. As the service is described by a WSDL document, `importSchema` obtains the latter and incorporates the data types and operation definitions into the metadata collection.

It is also important for the GDQS to collect sufficient data about the available computational resources on the Grid to enable the optimiser to schedule the distribution of the plan partitions as efficiently as possible. Although the current OGSA reference implementation does not fully support this need, it does

provide a high-level **Index Service**, to enable collecting, caching and aggregating of computational resource metadata. Figure 6 illustrates the service-based architecture that enables a GDQS to collect resource metadata from multiple nodes on the Grid. Conventions are the same as in Figure 5. Similarly numbered interactions potentially take place concurrently.

In this experimental set-up, an index service collects dynamic information on the system it is deployed in using back-end information providers. Since index services implement the **Registry** port type, any service can be published (1) using them. The GDQS identifies a central index service as its server for caching and aggregating metadata, and causes (2) it to subscribe to other distributed index services.

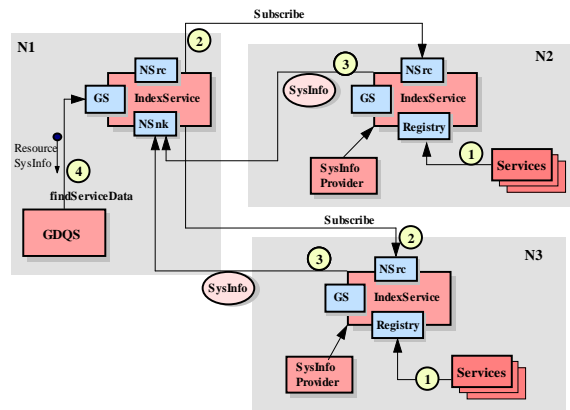


Fig. 6. Importing Resource Metadata

The remote index services send (3) notification messages at specified periods whose payload is resource metadata in a format determined by the back-end information provider. The GDQS can use (4) a `findServiceData` call to obtain the aggregated information as SDEs from its server. Note that one would expect the index service hierarchy to have been set up as part of a virtual organisation's infrastructure, since the identification of Grid nodes that constitute the organisation's resource pool is beyond the operational scope of the GDQS.

```

<GridNodeInfo
  hostsDataSource="1" hostsService="0"
  hasEvaluatorFactory="1">
  <nodeID>mach1.cs.man.ac.uk</nodeID>
  <CPUSpeedMHz>1400</CPUSpeedMHz>
  <CPULoadPercentage>10</CPULoadPercentage>
  <connectionSpeedMBperSec>1.0</connectionSpeedMBperSec>
  <hostedDataSource
    GDSFactoryHandle
    ="http://x.cs.man.ac.uk/ogsa/services/GDSFactory"
  <evaluatorFactory>
    http://x.cs.man.ac.uk/ogsa/services/EvaluatorFactory
  </evaluatorFactory>
</GridNodeInfo> <GridNodeInfo
  hostsDataSource="0" hostsService="1"
  hasEvaluatorFactory="0">
  <nodeID>mach1.ebi.co.uk</nodeID>
  <CPUSpeedMHz>1000</CPUSpeedMHz>
  <CPULoadPercentage>95</CPULoadPercentage>
  <connectionSpeedMBperSec>2.0</connectionSpeedMBperSec>
  <hostedService>
    http://www.ebi.ac.uk/.../AxisServlet/urn:emblfetch
  </hostedService>
  <hostedService>
    http://www.ebi.ac.uk/.../AxisServlet/urn:spooftblast
  </hostedService>
</GridNodeInfo>

```

Fig. 7. Computational Resource Metadata

The GDQS captures computational resource metadata for all machines hosting a schema it has imported.

The XML fragment in Figure 7 shows the canonical form in which computational resource metadata is maintained within the GDQS. Each `GridNode-Info` element in the fragment contains information such as the CPU speed, CPU load (as a percentage), the network bandwidth, the address of the GDSF (if there exists one) that wraps the data source hosted in that particular node, and the URLs of hosted services. Once a GDQS is set up as described, it is ready to accept queries against the schemas and resources that it ranges over. The XML fragment in Figure 8 shows, for the example query, the document that is passed as the parameter of the `perform()` call. The query request document consists of a header specifying the document name, the document version and the GSH of the originator service; and a body conveying the OQL query and indicating how results are to be delivered.

```

<GridDataServiceRequest>
  <Header>
    <RequestName>Example 1</RequestName>
    <Version>
      <Config>config</Config>
      <RequestEnvironment>
        environment
      </RequestEnvironment>
    </Version>
    <Originator>GSH of Originator</Originator>
  </Header>
  <Body>
    <Statement name="xyz"
      dataResource="MyDataResource">
      select p.proteinId, blast(p.sequence)
      from p in protein, t in proteinTerm
      where t.termId='GO:0008372' and
            p.proteinId=t.proteinId
    </Statement>
    <Delivery name="delivery">
      <Mechanism type="bulk"/>
      <Mode type="full"/>
      <From>xyz</From>
      <To>response</To>
    </Delivery>
    <Execute name="execute">xyz</Execute>
  </Body>
</GridDataServiceRequest>

```

Fig. 8. Query as a GDS Request Document

Due to lack of space, details about the compilation and optimisation are omitted. They can be found in [17]. Figure 9 depicts the execution plan produced by the compiler/optimiser. Three partitions (i.e., the dashed regions) have been decided upon whose intersections are marked by the exchange operators used to bind them, as explained below. Note that the partition containing the potentially expensive the BLAST `operation_call` has been scheduled to run on two of the four nodes N1-N4 harnessed for executing the query.

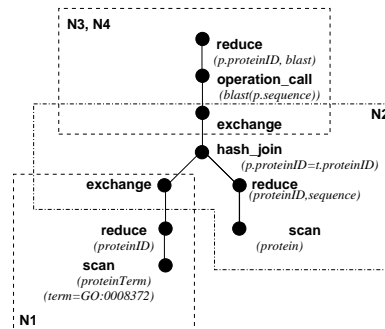


Fig. 9. Distributed Query Plan

has been scheduled to run on two of the four nodes N1-N4 harnessed for executing the query.

4.2 Evaluating Distributed Plans

Evaluator functionality is exposed via GQES instances that implement the GS, GDS and GDT port types from the OGSA and OGSA-DAI.

The GDQS creates as many GQESs instances as stipulated in the distributed query plan constructed by the compiler (see Figure 9). The GDQS had already obtained a handle on each GDS that contributes to the evaluation of a partition, as illustrated in Figure 5.

Each partition, along with the GDS handles it needs, is mapped to XML and sent to be evaluated by the GQES instance it is assigned to.

For the example query, this gives rise to the GS interaction diagram in Figure 10. Conventions are as in previous figures. The GQESs that scan stores, viz., N1 and N2, are instantiated in different hosts. Conditions at N2 (e.g., available memory) are such as to justify the GDQS having decided on using, say, a `hash_join`, assigning it to N2. For the BLAST operation `call`, the GDQS saw benefits in parallelising it over two GQESs N3 and N4.

The GDQS receives the request (1) and compiles it into a distributed query plan, each partition of which is assigned to one or more execution nodes. Each execution node corresponds to a GQES instance which is created by the GDQS (2). The GDQS then dispatches (3), as an XML document, each plan partition to its designated GQES instance. Upon receiving its plan partitions, each GQES instance initiates its evaluation.

The overall behaviour of a GQES instance is as follows. Execution is a data flow computation using an iterator model [10], with each operator implementing an `{open(), next(), close()}` interface. Each GQES starts executing its partition independently by calling `open()` on the topmost operator, so introducing parallelism between partitions. Every GQES instance that has an exchange leaf² is fed data by the GQES instance executing the partition whose root is

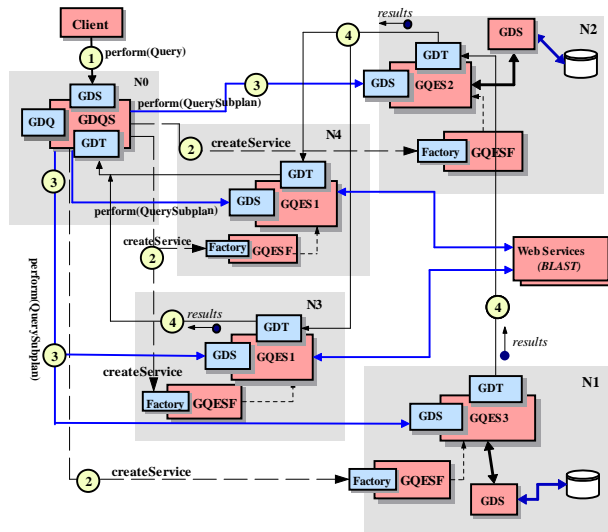


Fig. 10. GQES Instances and Query Partitions

² An exchange operator [10] encapsulates control flow, data distribution and inter-process communication

the corresponding exchange. Data flows from the GQES instances that execute partitions containing operators whose semantics requires access to stores.

Within each GQES instance, the initialisation procedure starts when an `open()` call reaches the topmost operator. This call propagates down the operator tree from parent to children at every level until it reaches the leaf operators. Then, interaction with other GDSs occurs. The handle for each such GDS will have been planted by the GDQS in the XML document passed to each GQES instance that needs it. For example, in node N2 (in Figure 10, but see also Figure 9), when this stream of `open()` calls reaches a, say, `scan` operator (see Figure 9), it causes the N2 GQES to interact with the GDS instance on N2, whereby data becomes ready to flow upwards from the `protein` extent in the GDS through which the GIMS database is accessed. Following the `open()`, a stream of `next()` calls again propagates from the topmost operator down to the leaf operators. When a `next()` call reaches a leaf, the standing-by GDS begins responding with tuples, and data flows upward.

Taking as example, again, node N2 in Figure 10, its topmost operator is an `exchange`, which, as alluded to, encapsulates inter-evaluator interaction semantics. Its effect is to cause tuples to flow (4) across this GQES instance's boundary and along the inter-GQES channels. When an `eof` is received in response to a `next()`, the flow is over and a `close()` is called, leading to clean up and dynamic resource de-allocation. Eventually, all operators have been closed, at which point that GQES instance has fulfilled its purpose. Across all GQES instances, results propagate up the tree, and exchanges ship (4) them to their destination.

5 Related Work

The distributed nature of Grid applications means that services to support co-ordinated use of Grid resources are important, and considerable attention has been given to functionalities for managing data derivation (e.g., [8]) and replication (e.g., [3]). However, such higher-level Grid data management functionalities are still targeted principally at file-based data, and the only previous work on distributed query processing in a Grid setting is the Polar* proposal from the authors [17]. Polar* differs from the approach presented in this paper in that it is not service-based; in Polar*, Grid middleware is accessed using a Grid-enabled version of MPI [5]. The absence of the service-based context in Polar* means that connection to external databases and computational services is much less seamless than in the OGSA setting.

In the Web Services setting, structured data representations, at least in the form of XML Schemas, have been much more prominent from the start. In addition, vendors have been quick to integrate Web Service and data management products (e.g. [13, 16]). However, we know of only one previous proposal for querying over collections of Web Services, viz. that of SkyQuery [14], which applies the classical wrapper-mediator architecture in a service-based setting. SkyQuery deploys WSs at each database store for handling metadata, performing queries, and cross matching partial results. However, the SkyQuery proposal

is less ambitious than that presented here, in a number of respects: (i) the only services that contribute to query evaluation are the data sources – there is no dynamic discovery and allocation of evaluators, for example, to support evaluation of large joins; (ii) the execution plan generated by the optimiser is a straightforward pipeline – there is no partitioned parallelism; and (iii) the query language supported is specialised for use with astronomical queries, and seems to assume that database nodes contain horizontal partitions of the overall database – there seem not to be generic facilities for joining data from multiple nodes, for example. Thus SkyQuery is an important early demonstration of the viability of Web Services for supporting distributed query processing, but it lacks dynamic allocation of resources to match the needs of specific requests. This latter feature is central to the ethos of the Grid, in which computational resources are made shareable, and thus can be deployed flexibly to support changing user needs and system loads.

How does the work presented here compare with other work on DQP, as surveyed in [12]? The principal differences derive from the context in which queries are executed. The aim of the current proposal is essentially the same as that of the developers of systems such as Garlic [11] and Kleisli [4], i.e., to support declarative query formulation over distributed data stores and analysis tools. However, the development of service-based Grids provides certain opportunities for the developers of DQP systems that were more elusive before. For example, Web Services promise to make available comprehensive discovery and access facilities for distributed resources that ease their integration into federated architectures. We note that no custom-built wrappers were developed to support the bioinformatics application illustrated in this paper – generic Grid Database Services were used to access the databases, and existing BLAST Web Services were used to perform sequence comparisons. This contrasts with both Garlic and Kleisli, where custom wrappers are constructed for interfacing the query engine to the external resources. Furthermore, we observe that the resources used to evaluate the example query were obtained dynamically, based on the anticipated needs of the request. Had the request required substantially greater resources to run efficiently, these would have been allocated from those available on the Grid. This contrasts with both Garlic and Kleisli, where query evaluation is shared between the central query evaluator and the source wrappers, with no dynamic resource discovery. Thus, although many characteristics of existing DQP systems carry over largely unchanged to the service-based setting, various features of relevance to DQP deployment and development are significantly affected by service-based architectures.

6 Conclusions

Web Services, in particular in conjunction with the resource access and management facilities of Grid computing, show considerable promise as an infrastructure over which distributed applications in e-business and e-science can be developed. However, to date, the emphasis has been on the development of core middleware

functionalities, such as for service description, discovery and access. Extensions to support the coordinated use of such services, for example using distributed transactions or workflow languages are still under development. This paper seeks to contribute to the corpus of work on higher-level services by demonstrating how techniques from distributed query processing can be deployed in a service-based Grid. The proposal is service-based in two respects:

- Queries are written with respect to and evaluated over distributed resources discovered and accessed using emerging Web Services and Grid Services standards. This is important because it is as yet far from clear how best to orchestrate collections of Web and Grid Services. Although it is likely that workflow languages will have a prominent role, DQP offers system-supported optimisation of declarative requests with implicit parallelism, a combination that should yield significant programmer productivity and performance benefits for large-scale, data intensive applications. As such, we believe that service-based architectures stand to benefit significantly from DQP. The proposal made in this paper is much the most comprehensive to date for a distributed query processor that acts over services.
- The query processor has been designed and implemented as a collection of cooperating services, using the facilities of the OGSA to dynamically discover, access and use computational resources to support query compilation and evaluation. This is important because although the OGSA has found widespread support within the academic and industrial Grid community, there are as yet few examples of higher-level services developed over the OGSA. This proposal can be seen to provide important validation of OGSA facilities for developing higher-level services. Furthermore, it has been shown how the combination of dynamic computational resource discovery and allocation can be used to match the requirements of a distributed query to the resources available in a heterogeneous distributed environment. As such, we believe that DQP stands to benefit significantly from the availability of service based grids. The proposal made in this paper is much the most comprehensive to date for a distributed query processor that uses grid services in its implementation.

The proposal described in this paper has been prototyped. The resulting software, referred to as **OGSA-DQP**, is scheduled for public release from [www.ogsa-dai.org.uk] in July 2003.

Acknowledgements — The work reported in this paper has been supported by the UK e-Science Programme through the OGSA-DAI project and the ^{my}Grid project, and by the UK EPSRC grant GR/R51797/01. We are grateful for that support. The ideas in this paper have benefited greatly from, and build upon, our collaboration with colleagues, from IBM, Oracle, the UK National e-Science Centre and the Edinburgh Parallel Computing Centre, in the OGSA-DAI project.

References

1. W. H. Bell, D. Bosio, W. Hoschek, P. Kunszt, G. McCance, and M. Silander. Project Spitfire - Towards Grid Web Service Databases. In *Global Grid Forum 5*, 2002.
2. R. G. G. Cattell and D. K. Barry. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
3. A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripenu, B. Schwartzkopf, H. Stocking, K. Stockinger, and B. Tierney. Giggie: A Framework for Constructing Scaleable Replica Location Services. In *Proc. Supercomputing*. IEEE Press, 2002.
4. S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Stoeckert. K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources. *IBM Systems Journal*, 40(2):512–531, 2001.
5. I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proc. Supercomputing*. IEEE Press, 1998.
6. I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6):37–46, 2002.
7. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. Supercomputer Applications*, 15(3), 2001.
8. I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *Proc. CIDR*, 2003.
9. K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Sys. Journal*, 41(2):170–177, 2002.
10. G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, pages 102–111, 1990.
11. V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In *Proc. SIGMOD*, pages 524–532, 2002.
12. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
13. S. Malaika, C. Nelin, R. Qu, B. Reinwald, and D. C. Wolfson. DB2 and Web Services. *IBM Systems Journal*, 41(4):666–685, 2002.
14. T. Malik, A. S. Szalay, T. Budavari, and A. R. Thakar. SkyQuery: A Web Service Approach to Federate Databases. In *Proc. CIDR*, 2003.
15. R. W. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan. Data-Intensive Computing. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 5, pages 105–129. Morgan Kaufmann, 1999.
16. R. M. Riordan, editor. *Microsoft ADO.NET Step by Step*. Microsoft Press, 2002.
17. J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Distributed Query Processing on the Grid. In *Proc. Grid Computing 2002*, pages 279–290. Springer, LNCS 2536, 2002.
18. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid Service Specification. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002. Draft 5, November 5, 2002.