The Relational Grid Monitoring Architecture: Mediating Information about the Grid

Andy Cooke, Alasdair Gray and Werner Nutt
Heriot-Watt University, Edinburgh, UK

James Magowan, Manfred Oevers and Paul Taylor
IBM-UK

Roney Cordenonsi
Queen Mary, University of London, UK

Rob Byrom, Linda Cornwall, Abdeslem Djaoui, Laurence Field*, Steve Fisher, Steve Hicks, Jason Leake†, Robin Middleton, Antony Wilson and Xiaomei Zhu
Rutherford Appleton Laboratory, UK

Norbert Podhorszki
SZTAKI, Hungary

Brian Coghlan, Stuart Kenny, David O'Callaghan and John Ryan
Trinity College, Dublin, Ireland

April 1, 2004

Abstract. We have developed and implemented the Relational Grid Monitoring Architecture (R-GMA) as part of the DataGrid project, to provide a flexible information and monitoring service for use by other middleware components and applications.

R-GMA presents users with a virtual database and mediates queries posed at this database: users pose queries against a global schema, and R-GMA takes responsibility for locating relevant sources and returning an answer. R-GMA's architecture and mechanisms are general and can be used wherever there is a need for publishing and querying information in a distributed environment.

We discuss the requirements and design of R-GMA as deployed on the DataGrid testbed. We also describe some of the ways in which R-GMA is being used.

Keywords: DataGrid, Grid information system, Grid monitoring, Grid monitoring architecture, R-GMA

1. Introduction

This paper describes the design and implementation of the Relational Grid Monitoring Architecture (R-GMA). R-GMA is a Grid informa-

* Now at CERN, Switzerland.
† Under contract from Objective Engineering Ltd.

tion and monitoring system built by Work Package 3 of the European DataGrid project [6].

Grids, which are put together by cooperating organisations to share computing resources, are designed to be *big*. The main aim of a Grid information and monitoring system is to provide a way for users to obtain information about Grid resources as quickly as possible. However, there can be a huge number of data sources scattered around the Grid. R-GMA takes a novel approach to solving this problem. To a user, R-GMA appears as a "virtual database". The user simply queries a relational global schema, and R-GMA takes the responsibility of locating relevant sources and returning an answer (this is called "mediation").

The advantage of R-GMA's approach is that users are offered all the flexibility that the relational model and SQL query language bring. The relational approach has a sound theoretical basis [3], and its framework has been extended recently to the world of streams by the database community [1]. In R-GMA, both "one-time" and continuous queries over streams are supported.

The flexibility gained by choosing a relational data model over, say, a hierarchical model does come with some costs. It is less obvious how a relational schema can be distributed across the Grid. Indexes still need to be chosen with certain queries in mind. However, the main advantage is that the relational model allows queries to explore complex relationships in data.

This paper details how the R-GMA approach has been realised. The requirements that drove the design of our system are described in Section 2. R-GMA's design builds on an architecture recommended by the Global Grid Forum for its scalability and flexibility: the Grid Monitoring Architecture. In Section 3 we describe this architecture, and look at other Grid monitoring systems that have been built from this. Section 4 describes the design of R-GMA while Section 5 describes how the task of "mediation" is achieved in R-GMA. Finally, Section 6 describes the experiences we have had in building and deploying R-GMA in DataGrid, and the plans for developing the R-GMA system in EGEE, the European Union's follow-on project.

2. Grid Monitoring

We begin by describing how DataGrid's middleware components interact, before discussing the requirements of a Grid information and monitoring system.

On a Grid, one can distinguish between rapidly changing data on the one hand and more static data on the other. Often the term Grid monitoring refers to managing data of the first kind, while a Grid information system would handle data of the second kind. As we argue in the present paper, Grid components need a unified system that is able to deal with both kinds of data. Thus, when using the term *Grid monitoring* system, we refer to what might more correctly be called a *Grid information and monitoring* system.

2.1. Overview of DataGrid Components

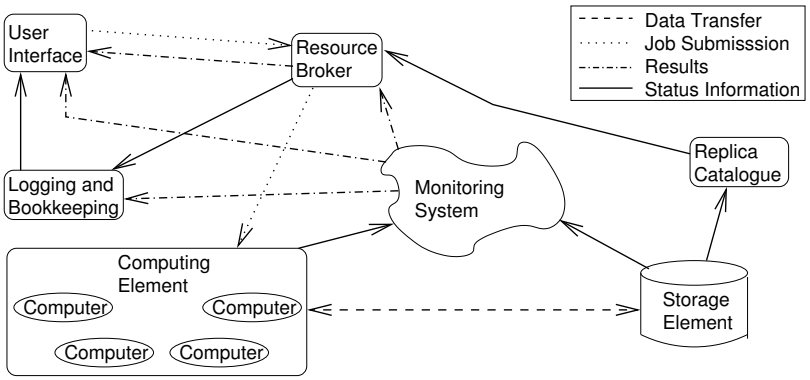

Figure 1. The major components of DataGrid.

Figure 1 shows a "bird's eye" view of the components that make up a Grid. Institutions participating in a Virtual Organisation [7] will contribute computing resources: computing elements, storage elements and network bandwidth. Distributed across the Grid are resource brokers, replica catalogues and other components, which coordinate and manage access to computing resources and experimental data. At the centre is a Grid monitoring system, which is used by these components to publish or query information about the state of resources on the Grid.

A user submits a job using the user interface. The user will specify certain requirements, e.g. that certain experimental data stored on the Grid is used, that the job should run in parallel using a number of CPUs on machines having the correct software, and that the output should be stored on at a particular location.

The resource broker has the responsibility of locating the best resources for running that job. It first consults the replica catalogue to locate the data requested by the job (the replica catalogue maps logical file names to physical locations). It then queries the monitoring system to find suitable computing elements close to the storage elements that

hold the experimental data, and then forwards the job on to the computing elements to run. As the job progresses, information on the job's status is published. The logging and bookkeeping service tracks and logs the jobs that are running on the Grid. Finally, users can monitor the progress of their job using the user interface, which will query the monitoring system.

2.2. Use Cases and Requirements

A Grid monitoring system must make information about the status of Grid resources and jobs available both to users and to other components of the Grid. We will now specify some typical use cases of a Grid monitoring system in order to highlight the requirements of such a system.

1. A resource broker needs to quickly locate (within a few seconds) a cluster with 8 CPUs and 1 GB of memory that is currently lightly loaded. The cluster must have ATLAS-6.0.4 software installed, and have a network link to a SE holding certain experimental data.

2. A visualisation tool enables users to monitor the progress of their jobs. Its display needs to be updated whenever the job status changes.

3. The logging and bookkeeping service needs to know what resources a job has used on the Grid and for how long, in order to be able to charge for usage.

2.2.1. *Publishing Data*

The act of publishing requires two abilities: the ability to advertise what data is available, and the ability to answer requests for that data.

Data about a Grid may be stored or generated in different ways. For example, details about the topology of a network might be held in a database (use case 1). Information about the current status of a job might be computed by a script running on a computing element (use case 3). A Grid monitoring system, therefore, needs suitable interfaces to allow applications to publish a range of data sources.

Data can be perceived in two different ways: as a stream of changing values, or as a static set (or pool) of values. Database management systems for example, create the *illusion* that data is static through concurrency control, even when in reality data might be flowing rapidly into the database. The stream metaphor is useful when users want to query for change. A Grid monitoring system should have publishing interfaces that support both of these perspectives.

2.2.2. *Locating Data*
Data will be distributed across the entire fabric of the Grid, and a Grid monitoring system should provide easy ways for users to locate sources with interesting data. In addition, a *global view* over all the data published in the system must be provided so that users can easily explore the relationships between components.

2.2.3. *Queries with Different Temporal Characteristics*
Since a Grid monitoring system should be able to publish both static and stream data, it should also be possible to query both types of published data.

From the use cases, we see that it must be possible to find out about the *latest-state* of a resource (use case 1), or to be notified *continuously* whenever the state of a component changes (use case 2). Likewise, the *history* of a stream is sometimes needed (use case 3). Thus, a Grid monitoring system should support three temporal types of query: continuous, latest-state and history.

To be accepted by users, the query language should capture most of the common use cases, but should not force a user to learn too many new concepts. The queries should also be processed in a timely manner, e.g. a resource broker must receive accurate up-to-date information within only a few seconds of its query (use case 1).

2.2.4. *Scalability, Performance and Robustness*
It is envisioned that Grids will contain many thousands of resources, spread over a large geographical area. For example, it is planned that thousands of CPUs will be made available via a Grid to physicists working on the Large Hadron Collider [13]. It is inevitable that the fabric of such a large Grid will be unreliable; network failures and other problems will be common place in Grids.

The monitoring system must be able to *scale* to cope with data being published by a large number of resources simultaneously and to respond with correct answers in a timely manner. It must not be a performance bottleneck for the entire Grid.

The monitoring system itself should be resilient to failure of any of its own components, otherwise the whole Grid could fail along with it. The monitoring system cannot have any sort of central control as resources will be contributed by organisations that are independent of each other.

2.2.5. *Security*
Users of the system should identify themselves so that unauthorised users are prevented from accessing data. This is known as *authentication*. Also, a tool that publishes data may want to restrict, for example, certain columns of its relations from particular users. This is called *authorisation*. The monitoring system, therefore, should support both authentication and authorisation.

2.2.6. *Interfaces*
Grid-enabled software will need a way of accessing the monitoring system so that information can be published or retrieved to support decision making. Thus, APIs should be provided in a number of languages. Other interfaces, e.g. a web page, would also be useful for casual monitoring of the Grid.

3. The Grid Monitoring Architecture

The Global Grid Forum [8] have identified a Grid Monitoring Architecture (GMA) [16], which could offer the scalability and flexibility needed by a Grid monitoring system. In this section we describe the GMA, and look at two implementations of this in light of the requirements identified in Section 3.

3.1. Overview of the GMA

The Grid Monitoring Architecture (GMA) is a simple architecture containing three main types of actors:

Producer: Makes monitoring data (events) available, e.g. a sensor reporting on a computing element's status.

Consumer: Requests monitoring data, e.g. a resource broker that wants to locate suitable computing elements.

Directory Service: A place where producers can advertise their data, and consumers can advertise their needs.

Grid components act out these roles using API interfaces.

Figure 2 shows the interactions of these actors. A producer registers a description of its event stream with the directory service. A consumer contacts the directory service to locate producers that have data relevant to its query. A communication link is then set up directly with each producer to acquire data, either by a publish/subscribe protocol, or by a query/response protocol. Consumers may also register with

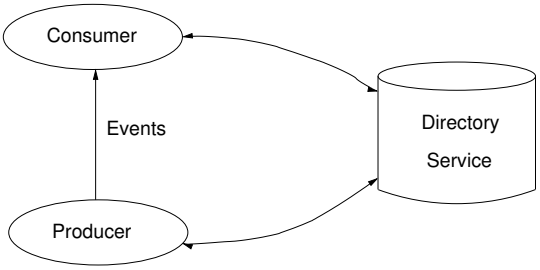

Figure 2. The components of the GMA and their interactions

the directory service. These are then notified whenever new producers become available.

Intermediary components may be set up that consist of both a consumer and a producer. Intermediaries may be used to forward, broadcast, filter, aggregate or archive data from other producers. The intermediary then makes this data available for other consumers from a single point in the Grid.

By separating the tasks of information discovery, enquiry, and publication, the GMA is *scalable*. However, it does not define a data model, query language, or a protocol for data transmission. Nor does it say what information should be stored in the directory service. There are no details of how the directory service should perform the task of matching producers with consumers.

3.2. GMA Implementations

Two notable systems that can be seen to fit the GMA architecture are Hawkeye [12] and the Monitoring and Discovery Service (MDS) [5].

Hawkeye was developed as part of the Condor project [15], and is used for monitoring and managing distributed systems. It consists of monitoring agents (GMA producers) at each computing node, which register and periodically stream data to a central manager (GMA directory service and intermediary). Hawkeye has a semi-structured data model ("ClassAds") and its query types resemble those outlined in Section 2.2.3. Users can query both the latest-state and the history of the stream (an archive is kept). Also, users can define triggers, such as "send an email if a machine is low on disk space or swap space". The main drawback of Hawkeye from the point of Grid monitoring is that the data model and query language is limited, and that the system has a central point of failure - the manager.

MDS continues to be developed as part of the Globus Toolkit [9]. It consists of information providers (GMA producers) and aggregate directories (GMA directory service and intermediary). Data is organised

using a hierarchical model, based on the Lightweight Directory Access Protocol (LDAP) in version 2, and on XML in version 3.

MDS has a scalable architecture, and is able to present to users a *global* view over the data. In version 2, latest-state queries are supported; however there is no assurance that the answers are up-to-date, as cached data is only refreshed when queries are received. Also, version 2 provides no easy way of supporting history queries. Some of these problems were addressed in version 3. However, the main drawback of MDS is its limited query language. Firstly, the schema must be designed with popular queries in mind. Secondly, there are no mechanism to capture relationships in the data that are not hierarchical, e.g. join operations.

4. R-GMA's Design

In this section we describe R-GMA's architecture and the rationale behind its design. R-GMA builds upon the GMA proposal (see Section 3.1) by specifying a data model, a query language and the functionality of the directory service.

The implementation of this design is on-going work, and the state of the final implementation by the end of DataGrid is discussed in Section 6.

4.1. R-GMA: A Virtual Database

Users need to be able to locate interesting information using the monitoring system; but the difficulty is that data is scattered across the whole Grid. It would be useful if the system could operate like a database, presenting a schema over which queries can be posed. However, it wouldn't be practical to stream all data into a central database, as this would become a single point of failure for the Grid. Therefore, R-GMA creates the illusion of a database through which data appears to flow.

To implement a virtual database requires techniques of data integration which have been developed within the database community over the last 15 years. Data integration systems use a mediator [17] for matching queries posed over the global schema with sources of relevant information. Several approaches have been suggested for performing this "matchmaking role" [11]. One approach is *local-as-view* where data sources describe their content as a view on the global schema. This provides the flexibility of being able to add and remove sources without reconfiguring the global schema, however query answering is

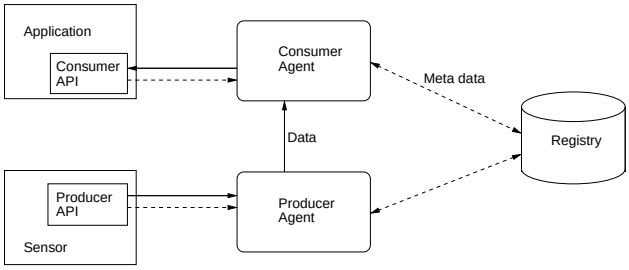

Figure 3. The roles and agents of R-GMA

not straight forward. We take the ideas of the local-as-view approach and apply them to data streams.

4.2. Software Design Considerations

R-GMA picks up the GMA metaphor of Grid components playing the roles of producer and consumer. We provide lightweight APIs to allow the components to act out their role. The functionality of the APIs is provided by *agents* which are created on their behalf and located on a web server. Figure 3 shows the interactions between the APIs, agents and the registry.

In our system, agents are realised using Java servlets hosted by web servers. A web-based architecture is convenient as only a small number of extra ports need then be opened across a firewall to accommodate R-GMA, and control messages can be encoded and decoded from XML using standard web service tools and transported by HTTP.

Protocols are needed to cope with the unreliability of Grid components and its networks. One common approach is for registration "heartbeats" to be sent from sender to receiver. If a problem occurs, and heartbeats fail to arrive within some termination time, the receiver can assume that the sender is no longer around. The Grid Resource Registration Protocol (GRRP) [5] is a heartbeat protocol proposed by the Globus project. In R-GMA, GRRP heartbeats are sent from API to agent, and from registered agent to registry.

4.3. R-GMA Components

Figure 4 shows the components that make up R-GMA. Like the GMA, R-GMA has producers, consumers, a registry (GMA directory service) and republishers (GMA intermediaries). We refer to producers and republishers collectively as publishers. Unlike the GMA, our system also has producer and consumer agents and a schema. Also, in R-GMA the registry is mostly hidden from our users—the consumer agent has

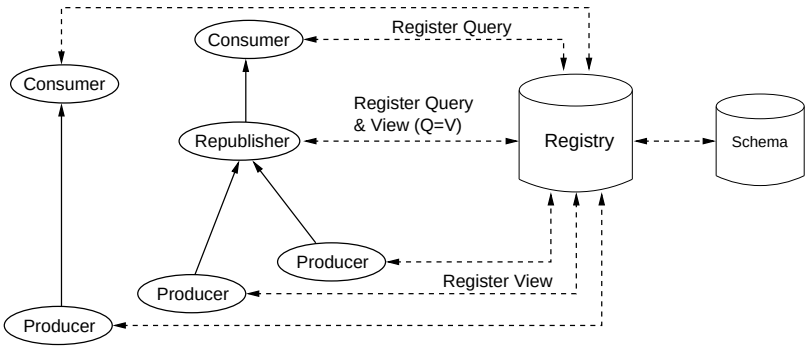

Figure 4. The components of R-GMA and their interaction with the Registry.

mediator functionality that will perform registry lookups on behalf of the user.

An analogy can be drawn with the market place. Any market place has buyers (consumers), manufacturers (producers) and wholesalers (republishers). Manufacturers and wholesalers are both types of sellers (publishers), which advertise their services (register) in the yellow pages (registry). Buyers might prefer to buy from wholesalers (to query republishers), as this reduces some transport overhead: otherwise they would have to visit manufacturers (producers) individually to obtain the same choice of goods.

We now give an overview of the components of R-GMA, detailing their functionality.

4.3.1. *Schema*

Just as buyers and sellers need a common language in order to understand each other, in R-GMA, a language is needed for describing what producers have to offer, and what consumers want. This language is based on SQL, and the vocabulary consists of relations that together make up a global schema. R-GMA's *schema* component has the responsibility of storing and maintaining this global schema. It acts as a catalogue that describes all of the products on offer in the market place, as well as their relationships. The registry and the agents will contact the schema whenever they need to know about the relations of the global schema, for example, when validating a query.

The Grid community have identified a core schema, known as GLUE [10], that describes the standard components of a Grid. R-GMA ships with a relational version of this; however users can easily introduce new relations into the global schema.

R-GMA relations have attributes and types, as in SQL. As each relation represents a stream, it also implicitly has a timestamp attribute

of type DateTime, indicating at what time the tuple was published. These timestamps are set with clocks that are synchronized with each other using the network time protocol."

A subset of the attributes can be singled out as the *primary key*, and this key usually identifies the parameters of a measurement. In R-GMA, any two tuples in the system that agree on the key attributes and the timestamp will also agree on the remaining attributes.

For instance, R-GMA's schema contains the core relation tp to publish the network throughput between sites connected to the Grid. The relation has the schema

$$\mathsf{tp}(\mathsf{from}, \mathsf{to}, \mathsf{tool}, \mathsf{psize}, \mathsf{value}, \mathsf{timestamp}),$$

to measure the network throughput according to readings by a certain tool, to transport packets of a specific size from one site to another. The from, to, tool and psize attributes make up the primary key of tp.

A particular set of values for a primary key can be thought of as representing a *channel* in a stream. A tuple in the tp relation might be

$$\mathsf{tp}('\mathtt{hw}', '\mathtt{ral}', '\mathtt{UDPmon}', 1000, 2.4, 2004\text{-}01\text{-}12\ 11{:}26{:}42)$$

giving a measurement of 2.4 ms for a 1000 byte UDPmon packet between Heriot-Watt and Rutherford Appleton Laboratories on 12 January 2004 at 11:26am. The channel would be identified by the values

$$('\mathtt{hw}', '\mathtt{ral}', '\mathtt{UDPmon}', 1000).$$

In R-GMA, keys are not enforced, but are used to identify channels.

4.3.2. *Producer Role*

The role of a producer is to publish data. This involves two tasks: (i) advertising the tuples that they make available, and (ii) answering requests for data. When a job runs on the Grid, scripts are invoked on different machines to control the execution of the computation. To be able to perform the first task of a producer, these scripts need to be enhanced so that they can register a description of their data. For the second task, the scripts need to be able to process queries and return answers. R-GMA offers these abilities through the producer API.

One method in the API is provided for registering a view on the global schema. This relates the local stream relation of the producer to a relation in the global schema, and has the form of a simple SQL select query, e.g.

$$\mathtt{SELECT}\ *\ \mathtt{FROM}\ \mathsf{tp}\ \mathtt{WHERE}\ \mathsf{from} = '\mathtt{hw}'\ \mathtt{and}\ \mathsf{tool} = '\mathtt{UDPmon}'.$$

Another method allows the producer to insert a collection of tuples. The producer agent supports a component playing the role of a producer by taking the responsibility of storing tuples and answering queries.

4.3.3. *Consumer Role*

The role of a consumer is to locate and collect together monitoring data of interest. For example, the resource broker will act as a consumer to access information about resources. To perform this role it needs the ability of posing an SQL query of a certain type and of retrieving an answer. R-GMA offers these abilities through the consumer API.

In R-GMA, consumers are defined by both an SQL query, and a query type. The query type may either be *history* or *latest-state* (these queries are collectively called *one-time* queries), or *continuous*. Orthogonal to this is the distinction between *local* and *global* queries. A local query is directed towards one or more publishers, while a global query is posed without specifying any publishers and needs to be translated into a local query in order to be executed. This translation process is called *mediation*. Thus, the consumer API allows a user to pose a query and declare its type. The consumer agent takes on the task of planning the execution of a query and retrieving tuples.

Once the consumer agent starts to retrieve tuples, these need to be passed on to the consumer. The consumer API offers a method that allows tuples to be retrieved one at a time, or as a collection. With this mechanism, a user who is only interested in receiving, say one tuple, need not wait for a whole answer set to arrive.

4.3.4. *Producer Agents*

A producer agent helps Grid components to play the role of a producer. It acts on behalf of the producer by registering a view describing the data, and by answering requests from consumers for that data.

When a producer inserts tuples using the API, these are forwarded to the agent. The agent then checks that the tuples conform to the producer's descriptive view, before storing the tuples in a buffer. A "retention period" is defined by the producer, which determines how long tuples will be held in the buffer before they are discarded.

In R-GMA, all producer agents are capable of answering continuous queries. In addition, producer agents may be configured, at a small performance cost, to be able to answer latest-state or history queries. A database is used for this, and one-time queries are simply passed on to the supporting database.

The producer agent registers the view, the retention period (which is useful for query planning), and the query types that are supported on behalf of the producer. The registration acts as an advertisement,

and consumers that are interested (either in its local relation, or in the global relation) will contact it with a suitable query.

4.3.5. *Consumer Agents*
A consumer agent assists Grid components in playing the role of a consumer. The first stage of query answering is to identify which publishers have relevant information, and to decide which of these to contact. The consumer agent cooperates with the registry for this, and details will be given in Section 5.

Currently, queries are processed in the following way. Once the consumer agent has decided on a query plan, it contacts each publisher in the plan in turn with a local query. As answers arrive, the consumer agent stores these in a single queue. The consumer can then retrieve the answer by "popping" tuples from the queue, as described earlier.

The improvement of R-GMA's query answering capabilities is ongoing work. We hope to take advantage of a distributed query processor that is currently being developed as part of the OGSA-DAI project [14]. When this is ready, R-GMA's consumer agent will be adapted to use it.

4.3.6. *Republishers*
As the Grid grows in size, mechanisms are needed in R-GMA that ensure that popular queries are still answered efficiently; republishers (c.f. GMA intermediaries) are used for this purpose. A republisher is similar to the view mechanism of a relation database—it poses a continuous query over the global schema and publishes the answer stream.

For continuous queries, it can be expensive (more network traffic, more socket connections) to contact many producers. If a republisher is available that has already merged these streams, then a consumer agent will answer the query more efficiently by contacting this instead.

For one-time queries, it can be expensive to perform a query that involves distributed query processing, such as a distributed join. However, if a republisher is available that has all the relevant data in a database, then such a query can be answered efficiently by passing the query onto that database.

In principle, as both the input and output of a republisher is a stream, *hierarchies* of republishers can be built. For example, republishers could be set up at sites, and these could all feed into a top-level global republisher. Queries that only request data from an individual site could be served by a site republisher, whereas the global republisher could serve more general queries.

The advantages that republisher hierarchies could bring depend on whether relations can be partitioned in a sensible way. The current

system allows hierarchies to be set up manually. However, we are developing algorithms and protocols that automatically generate and maintain such a hierarchy [4].

4.3.7. *Registry*
Just as the yellow pages directory helps buyers in a market place to locate sellers, the registry allows producers and republishers to advertise their "goods", and helps consumer agents to find publishers. However, the registry is hidden from the users of the system. Instead, their agents interact with the registry on their behalf.

Consumer agents interact differently with the registry, depending on whether they have a continuous or a one-time query.

For one-time queries, the simplest approach is for the agent to consult the registry each time the query is run. The registry is able to identify publishers that are relevant to the query, and returns their address, type, and other information (e.g. retention periods) that are useful for planning the query.

In contrast, as continuous queries are long-lived, the query is registered by the agent. The registry then ensures that throughout the lifetime of the consumer, it can receive all of the data the query asks for: the consumer is notified as new relevant publishers are registered or dropped.

5. Answering Queries in R-GMA

With R-GMA, clients are relieved from the task of locating sources of information. The schema models what kind of information about the Grid is available in principle. A client poses its query against the schema, which is translated by the system into a set of queries over the local relations of relevant publishers and then executed. *This process is called mediation.*

In this section, we first discuss the semantics of a query for each of the three temporal types introduced beforehand. That is, we define which is the set or stream of tuples that make up the answer of a query. This is not completely straightforward since queries are not evaluated over a relational database. It will turn out that sometimes queries can only be answered partially if the data needed for a full answer are not available.

Then, we discuss the mediation process. It consists of two stages, (i) *matchmaking,* which is the role of the registry, and (ii) *query planning*, which is the role of the consumer agent. As the semantics and the

approach to mediation differs between one-time and continuous queries, we discuss each case separately.

5.1. Query Semantics

5.1.1. *Semantics of One-Time Queries*

One-time queries are arbitrary SQL queries, which are flagged either as a history or as a latest-state query. Suppose that Q is such a query. We indicate its query type using the superscripts h and l, respectively, thus writing $Q^{(h)}$ or $Q^{(l)}$. We denote the set of answers obtained by evaluating the SQL query Q over some database $\mathcal{D}$ as $Q(\mathcal{D})$.

Suppose the query is being posed at a given point in time t_0. Up until time t_0, the producers of R-GMA—including those that have ceased to exist—will have published a certain set of tuples, all of which have a timestamp that is less or equal to t_0. Conceptually, we load all these tuples into one database, which we denote as $\mathcal{D}^{(h)}$. Then the set of answers to the *history query* $Q^{(h)}$ is defined as the result of evaluating Q over $\mathcal{D}^{(h)}$, formally, $Ans(Q^{(h)}) = Q(\mathcal{D}^{(h)})$.

To define the semantics of a latest-state query, we have to go back to the concept of a channel. Consider as an example the relation tp with the schema

$$\mathsf{tp}(\underline{\mathsf{from}}, \underline{\mathsf{to}}, \underline{\mathsf{tool}}, \underline{\mathsf{psize}}, \mathsf{value}, \mathsf{timestamp}),$$

where the first four attributes form the primary key. A set of values for these attributes defines a channel, and all the tuples that agree on these attributes are the tuples that have been sent across the channel. We assume that different producers always produce for different channels. We say that a channel is *alive* at time t_0 if its producer is registered at this point in time. Now, we construct conceptually a database $\mathcal{D}^{(l)}$ that contains for each channel that is alive the tuple with the latest timestamp less or equal to t_0. Then the set of answers to the *latest-state query* $Q^{(l)}$ is defined as the result of evaluating Q over $\mathcal{D}^{(l)}$, formally, $Ans(Q^{(l)}) = Q(\mathcal{D}^{(l)})$.

Note that the answers to $Q^{(l)}$ cannot be computed by evaluating Q over the subset of $\mathcal{D}^{(h)}$ that contains the latest tuple for each channel present in $\mathcal{D}^{(h)}$, because some of these channels may no longer be alive. The restriction to channels that are alive is important since, for instance, a resource broker would not be helped if it received information about a computing element that has been shut down in the meantime.

5.1.2. *Semantics of Continuous Queries*
Currently, R-GMA only supports continuous queries that are selections over a single relation, that is, queries that in SQL are written as

$$\texttt{SELECT * FROM } r \texttt{ WHERE } C, \tag{1}$$

where r is a relation and C is a condition. We will also use the relational algebraic notation $\sigma_C(r)$ for such a selection query.

Suppose that Q is a selection query. If Q is posed as a continuous query, we write $Q^{(c)}$, using the superscript c to indicate a continuous query. If $Q^{(c)}$ posed at time t_0, then the answer $Ans(Q^{(c)})$ is a stream that consists of all tuples of relation r that satisfy condition C and have a timestamp greater than t_0.

Since tuples are published in a distributed fashion it is impossible to require that the answer stream be chronologically ordered. However, we impose the weaker requirement that the tuple making up a channel are chronologically ordered. We say that a stream with this property is *weakly ordered.*

We also plan to support two hybrid query types, which we call *continous+latest* and *continous+time* queries. To indicate these types, we use the superscripts *cl* and *ct*.

If Q is a selection query then $Ans(Q^{(cl)}) = Ans(Q^{(l)}) \cup Ans(Q^{(c)})$, that is, the answer stream ships first the latest-state answers to Q and then starts with the regular answer stream. Such a query would for instance be useful for defining a republisher maintaining a latest-state cache of its stream.

A "continuous+time" query is specified by a selection query Q with an additional parameter s, which stands for the start time. It is abstractly denoted as $Q_s^{(ct)}$. The answer stream for this query consists of all tuples of the relation of Q that satisfy the condition of Q and have a timestamp greater than s. If $s > t_0$, then the answer stream of $Q_s^{(ct)}$ starts in the future, and if $s < t_0$, then the answer stream starts in the past. The latter version can be seen as a combination of a continuous and a (simple) history query.

5.2. Answering Continuous Queries

When answering a query Q, we distinguish between (i) matchmaking, that is, the task of identifying publishers that can potentially contribute to answering the query (called *relevant publishers*), and (ii) query planning, that is, the task of deciding which publishers to contact, posing queries over them, and combining the answers.

5.2.1. *Matchmaking*
Suppose a consumer with a continuous query $Q = \sigma_C(r)$ has been generated. The consumer agent then contacts the registry to register the consumer and to obtain a list of relevant publishers.

Consider a publisher for the relation r that has registered a descriptive view $\sigma_D(r)$. When is such a publisher relevant to Q? Clearly, if it is logically impossible to satisfy both C and D, then the producer can never contribute a tuple to the query. However, if the condition C `AND` D is satisfiable, then the publisher must be relevant; answer tuples would be lost it were not contacted.

In general, checking the satisfiability of unnested conditions, formed using `AND` and `OR`, is NP-hard in the worst-case, which could turn matchmaking into a computationally expensive task. The current version of R-GMA, however, only accepts queries and views that have simple conditions. For queries, these conditions are of the form

$$Attr_1 Op_1 Val_1 \texttt{ AND} \ldots \texttt{AND } Attr_n Op_n Val_n, \tag{2}$$

where $Attr_i$ is an attribute, Val_i is a value and Op_i one of the operators "$\leq$", "$=$", or "$\geq$"; view conditions have the same form, except that only the operator "$=$" is supported. For such conditions the satisfiability check is simple. To perform it efficiently, the registry contains a relational database that stores in a structured manner both consumer queries and publisher views. Then, for a consumer query Q, R-GMA generates a query over the registry database that retrieves all relevant publishers for Q.

Similarly, when a new producer is registered by its agent, then R-GMA creates a query that retrieves all consumers for which that publisher is relevant and notifies their agents of the new producer.

5.2.2. *Query Planning and Execution*
Again, a simple approach has been chosen for planning continuous queries in the current implementation of R-GMA, in that only producers are used to answer a query—republishers are not used. We have developed theoretical foundations for a more general approach [4], but we are waiting for a clear use case before implementing this.

A plan that delivers all answer tuples for a selection query $\sigma_C(r)$ consists of contacting all relevant producers, querying them for those tuples of the relation in question that satisfy the query condition, and finally merging the answer streams. When new producers are created or exisiting ones die, the plan is adapted in a straightforward manner.

Techniques that make use of republishers need to be considerably more sophisticated. For one, republishers introduce redundant data, and a choice has to be made as to which publisher to use in a query

plan. Moreover, plans have to be modified to take advantage of new republishers or to compensate for a republisher that is no longer available. When switching plans, special care must be taken to avoid the loss of tuples or duplicate tuples. We expect a need for republishers to arise as R-GMA is used in larger Grids.

5.3. Answering One-Time Queries

There are two reasons why one-time queries in R-GMA are more difficult to answer than continuous queries.

The main difficulty is due to the fact that history or latest-state queries refer to data that have been produced in the past. Therefore, such a query can only be answered if a producer agent or a republisher maintains the necessary data in a latest-state or a history database for its streams. To ease our presentation, we say that a producer is a *latest-state* or *history* producer if its agent maintains a latest-state or history database. Similarly, we talk about latest-state and history republishers. Note that producers and republishers can be set up to support both types.

A second difficulty arises if the data are distributed over more than one database. In such a situation query answering would require some specialised mechanism for distributed query processing. Until now, no production-strength distributed query processors were available in the public domain, and so R-GMA has only relatively simple mechanisms for distributed query processing.

They are based on the observation that sometimes a global query Q can be translated into several local queries $Q_1, \ldots, Q_n$ over databases $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that $Ans(Q) = Q_1(\mathcal{D}_1) \cup \cdots \cup Q_n(\mathcal{D}_n)$. If this is possible, the query can essentially be executed locally and only the results need to be merged. This approach works for example for selection queries.

To apply such an approach to more complex situations, more meta-information about the databases would be needed. For instance, equality joins often involve attributes where one is a foreign key referring to the other. If a foreign key constraint is maintained locally, then the corresponding join could also be executed locally. However, R-GMA does not support foreign keys so that this optimisation technique is not feasible at present.

5.3.1. *Matchmaking*

When a consumer agent receives a one-time query it checks whether it is a selection with a condition as in Equation 2. If so, it is called a *simple* query; if not, it is called *complex query.*

If it is a complex query, R-GMA cannot process it locally. Thus, the agent asks the registry for a list of all publishers that are of the same type as the query and are capable of processing the query alone. We say that a publisher publishes *fully* for a relation r if its descriptive view for that relation has an empty `WHERE` clause, i.e., if it is of the form "`SELECT * FROM` r." The registry returns a list of publishers each of which satisfies the following criteria: the publisher is publishing for all relations that occur in the query and,

- in case it is a *republisher*, it publishes the query relations fully, or
- in case it is a *producer*, there is no other producer for any of the relations in the query.

In both cases the publisher can answer the query alone. In the first case, it is a republisher that collects all the data for all the relations in the query. In the second case, it is a producer that holds all the data needed because it is the only source of data for the query relations. If none of the two cases holds, R-GMA does not attempt to answer the query and the consumer agent returns a warning.

If the query is simple, then it can be processed locally. The agent asks the registry for relevant publishers that are of the same type as the query. They are determined as for a continuous query.

5.3.2. *Query Planning and Execution*

The guiding principle for answering complex queries is to ensure that answers are always correct. Currently, R-GMA can only ensure this by handing the query over to a *complete* publisher. We say a publisher is complete with respect to its view if $\sigma_D(r)$ contains *all* tuples of relation r that satisfy D. In general, answers to queries involving negation or aggregation could be incorrect if executed by an *incomplete* publisher.

All the publishers in the consumer agent's list are complete. However, they may be closer or less close in terms of communication time. Therefore, the agent polls the publishers on its list to identify the one that can reply the fastest, and then sends the query to it. The agent forwards to its client the result set returned by the publisher agent.

For simple queries, correctness of answers is straightforward to guarantee. A consumer agent that has to set up a plan for a simple query receives a list of relevant republishers $R_1, \ldots, R_m$ and relevant producers $P_1, \ldots, P_n$.

Currently, the agent follows a simple approach. If there is a republisher that publishes fully for R, then it is chosen, otherwise, all producers are contacted. In the absence of full republishers, the approach is not guaranteed to deliver all answers because there may be producers that do not maintain latest-state or history databases.

Below we sketch how republishers that publish only a fragment of the relation r can be used to improve the coverage of a plan. Since a republisher's query is registered as its descriptive view, and since a republisher makes available all answers to its query, a republisher is always *complete* with respect to its view.[1]

Consider two republishers R and R' with views $\sigma_D(r)$ and $\sigma_{D'}(r)$, respectively. Then R contributes at least as many answers to the query $Q = \sigma_C(r)$ as R' if the conjunction $C \wedge D'$ logically entails D. In this case we say that R' *is subsumed by* R *with respect to* Q. Due to the simple structure of the conditions involved, subsumption is straightforward to check.

Now, to create a plan one chooses among the republishers delivered by the registry a collection $R_{i_1}, \ldots, R_{i_k}$ such that each R_i is subsumed by some R_{i_j}. Then no other republishers are needed for answering the query. Also, any producer that is subsumed by a republisher is not needed. However, a producer that is subsumed by another producer cannot be dropped from a plan because producers cannot be guaranteed to be complete with respect to their views.

Since the view conditions of republishers can overlap with those of other republishers and those of producers, the conditions in the local queries have to be refined so that no duplicate tuples are returned. For example, if a plan for the query $\sigma_C(r)$ uses republishers R_1, R_2, having views with conditions D_1, D_2, and a producer P, then the local queries, using logica notation, would be posed as follows: the query for R_1 would be $\sigma_C(r)$, the one for R_2 would be $\sigma_{C \wedge \neg D_1}(r)$, and the one for P would be $\sigma_{C \wedge \neg D_1 \wedge \neg D_2}(r)$. The condition $\neg D_1$ in the query over R_2 would filter out all tuples delivered by R_1, and $\neg D_1 \wedge \neg D_2$ would filter out those delivered by R_1 or R_2.

6. R-GMA in DataGrid and Beyond

From the outset, R-GMA was designed as a general purpose Grid information system that embodied a number of novel concepts. In DataGrid, R-GMA was mainly used for publishing network monitoring data and for providing information on resources. In addition, it was tested for

[1] The statement has to be taken with a grain of salt. Any republisher may miss tuples from some producers due to network failures. Depending on the frequency with which producers generate tuples, a latest state republisher may not contain a tuple for each of the channels it covers. Finally, a history republisher usually maintains only a history of a certain length, determined by its retention period. These sources of incompleteness are not taken into account in R-GMA's current query planning.

monitoring batch jobs. We shall discuss the last two applications to illustrate how the R-GMA concepts fitted the challenges arising in DataGrid and which additional efforts were needed to reach a solution. An important outcome of this work was that a user community was established around the system.

6.1. Collecting Resource Information

The first major use of R-GMA in DataGrid was to gather data about resources and to provide an up to date account to the resource brokers. At the end of the project, DataGrid's testbed comprised four resource brokers and 25 sites, each with a computing and a storage element.

The software to produce the resource information was developed within DataGrid and generated data in LDAP format. Similarly, the resource brokers were designed to pose queries against LDAP databases. The reason was that Globus MDS (Version 2), which is based on LDAP, had been foreseen as the initial information and monitoring system on the testbed. However, because of concerns about the performance and scalability of MDS, it was decided to migrate to R-GMA.

In this scenario, R-GMA was used as an information collection and distribution service. The resource information could be naturally classified into dynamic information produced by sensors, (e.g., the number of free CPUs in a computing element) and static information that would be updated occasionally (e.g., lists specifying access rights of users).

Using R-GMA producers was the method of choice for publishing the dynamic information at a site, which was generated at intervals of 30 seconds. However, for reasons discussed before, R-GMA did not offer mechanisms specifically designed for handling static data. Fortunately, the volume of the static information was moderate and it was feasible to ship each site's entire dataset on an hourly basis to the resource brokers. This was done using another producer. However, it is expected that this approach is unlikely to scale if sites need to publish larger sets of static data. Instead, mechanisms may be needed that forward only changes of "static" data sets.

In the vicinity of each resource broker, a latest-state republisher was installed to collect the streams from all the producers and to provide the information on which the broker would base its decisions.

Since neither the monitoring software nor the resource brokers had been designed to interface with R-GMA, interoperability became an issue. To bridge these gaps, two types of translators were provided, called Gadget In (GIn) and Gadget Out (GOut). A GIn would take LDAP output from the existing information providers and publish it

via the two producers, while GOut would map the relational data in a latest-state republisher to the resource broker's LDAP database.

This configuration had a clear drawback. On the one hand it created redundancy of data since the LDAP databases were essentially identical. On the other hand, since the resource brokers did not query R-GMA directly, no advantage was taken of R-GMA's ability to contact alternate republishers if the designated one was unavailable.

The DataGrid testbed challenged R-GMA's performance and stability and was crucial for hardening the code. Moreover, it was instrumental in creating a user community.

6.2. Job Monitoring

The particle physics experiments CMS and D0 used R-GMA in extended tests of real time job monitoring. In these experiments long running data analysis jobs—lasting up to 24 hours—are run, which require careful monitoring and bookkeeping. Since both tests used a similar set-up we only discuss the one conducted by CMS (see also [2]).

Within CMS, a Batch Object Submission System (BOSS) had been developed to submit jobs to a batch farm. It wraps a job into an extra layer of code, which spawns a separate process that produces information about the job's progress and sends it to a local database. In a Grid environment, a resource broker can pass the job to any remote computing element and information on the job's progress needs to be sent back to the submitter's site.

The goal of the tests was to establish whether R-GMA could be used for this purpose, the main question being whether it could cope with a sufficient number of jobs run concurrently. A set-up was chosen where the code wrapped around a job would create a stream producer, while a history republisher would collate the data about the jobs submitted at a site. The jobs themselves were only simulated, while stream producers would generate messages that were based on monitoring data collected from real jobs.

The tests revealed that the number of producers that a single R-GMA site could support was a bottleneck. Initially, stability was lost at just 10 producers. This led to a number of improvements, and eventually a single site could support more than 2000 producers without running into difficulties. This figure is very close to satisfying CMS requirements.

Since the relations needed for the tests were not part of the core schema, it proved to be helpful that users can add new relations to the global schema at any point in time. It turned out that this is a feature that significantly increases the usability of R-GMA and gives it the character of a generic information infrastructure.

6.3. Work Planned within the EGEE Project

Work on R-GMA will continue within the EGEE project (= "Enabling Grids for E-Science in Europe"). The aim will be to enhance R-GMA in several ways so that it can be deployed on large scale Grids. To this end we will make R-GMA's functionality available as a set of web services based on the emerging Web Service-Resource Framework standard.

Future Grids will be shared by several virtual organisations (VOs). For instance, the Large Hadron Collider Grid will be used by a number of experiments. This requires mechanisms to ensure one VO can only view its own information. In addition, more fine-grained authorisation schemes will be needed to improve security in such a setting.

The registry and schema, each of which are currently realised as a single component, will be replicated to eliminate a single point of failure.

Finally, it is planned that mediation capabilities will be improved to allow hierarchies of republishers to be created and exploited in answering continuous queries. Moreover, if public domain middleware for distributed query processing that is currently being developed in other Grid projects (OGSA-DAI) proves to be reliable, R-GMA will be extended to execute queries over multiple republishers.

7. Conclusion

We conclude with a discussion of how well R-GMA meets the requirements identified for Grid monitoring. Through the role of a producer, Grid components can publish their monitoring data. The schema provides a global view of all the monitoring data available. Grid components interested in monitoring data can locate and retrieve that data through the role of a consumer. The actual task of locating and retrieving the data is automated by the consumer's agent and the registry. By separating out the tasks of locating and retrieving data, the system will scale effectively.

Although R-GMA has been designed as a Grid monitoring and information system, the architecture is general and could be used for other applications that require the location and querying of distributed data streams.

References

1. Babcock, B., S. Babu, M. Datar, R. Motwani, and J. Widom: 2002, 'Models and Issues in Data Stream Systems'. In: *PODS-21*. pp. 1–16.

2. Bonacorsi, D., D. Colling, L. Field, S. Fisher, C. Grandi, P. Hobson, P. Kyberd, B. MacEvoy, H. Nebrensky, H. Tallini, and S. Traylen: 2003, 'Scalability Tests of R-GMA based Grid Job Monitoring System for CMS Monte Carlo Data Production'. In: *Proceedings of the IEEE 2003 Nuclear Science Symposium*. Oregon.
3. Codd, E. F.: 1970, 'A Relational Model of Data for Large Shared Data Banks'. *Communications of the ACM* **13**(6), 377–387.
4. Cooke, A., A. J. G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. A. Coghlan, S. Kenny, and D. O'Callaghan: 2003, 'R-GMA: An Information Integration System for Grid Monitoring'. In: R. Meersman, Z. Tari, and D. C. Schmidt (eds.): *CoopIS/DOA/ODBASE*, Vol. 2888 of *Lecture Notes in Computer Science*. pp. 462–481.
5. Czajkowski, K., S. Fitzgerald, I. Foster, and C. Kesselman: 2001, 'Grid Information Services for Distributed Resource Sharing'. In: *HPDC-10*.
6. DataGrid: 2003, 'The DataGrid Project'. http://www.eu-datagrid.org.
7. Foster, I., C. Kesselman, and S. Tuecke: 2001, 'The Anatomy of the Grid: Enabling Scalable Virtual Organization'. *The International Journal of High Performance Computing Applications* **15**(3), 200–222.
8. GGF: 2003, 'Global Grid Forum'. http://www.ggf.org.
9. Globus: 2003, 'Globus Toolkit'. http://www.globus.org.
10. Glue: 2003, 'High Energy Nuclear Physics InterGrid Collaboration Board'. http://www.hicb.org/glue/glue.htm.
11. Halevy, A. Y.: 2001, 'Answering queries using views: A survey'. *The VLDB Journal* **10**(4), 270–294.
12. HawkEye: 2004, 'HawkEye: A Monitoring and Management Tool for Distributed Systems'. http://www.cs.wisc.edu/condor/hawkeye.
13. LCG: 2004, 'LHC Computing Grid Project'. http://lcg.web.cern.ch.
14. Smith, J., A. Gounaris, P. Watson, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou: 2002, 'Distributed Query Processing on the Grid'. In: *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, Vol. 2536 of *Lecture Notes in Computer Science*. pp. 279–290.
15. Thain, D., T. Tannenbaum, and M. Livny: 2002, 'Condor and the Grid'. In: F. Berman, G. Fox, and T. Hey (eds.): *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc.
16. Tierney, B., R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski: 2000, 'A Grid Monitoring Architecture'. Revised January 2002.
17. Wiederhold, G.: 1992, 'Mediators in the Architecture of Future Information Systems'. *IEEE Computer* **25**(3), 38–49.

Appendix

A. Glossary

ComputingElement (CE): a Grid-enabled computing resource.

Globus, Monitoring and Discovery Service (MDS): originally Metacomputing Directory Service, a LDAP based information system.

Resource Broker: a Grid middleware component that brokers the running of Grid jobs, making use of the information service to obtain grid status information about available resources, and schedules jobs.

StorageElement (SE): a Grid-enabled storage system.

Virtual Organization (VO): A set of individuals defined by certain sharing rules - e.g. members of a collaboration.