The Relational Grid Monitoring Architecture: Mediating Information about the Grid

A. W. Cooke, A. J. G. Gray and W. Nutt
Heriot-Watt University, Edinburgh, UK

J. Magowan, M. Oevers and P. Taylor
IBM-UK

R. Cordenonsi
Queen Mary, University of London, UK

R. Byrom, L. Cornwall, A. Djaoui, L. Field*, S. M. Fisher, S. Hicks, J. Leake†, R. Middleton, A. Wilson and X. Zhu
Rutherford Appleton Laboratory, UK

N. Podhorszki
SZTAKI, Hungary

B. Coghlan, S. Kenny, D. O'Callaghan and J. Ryan
Trinity College, Dublin, Ireland

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, design and implementation 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, implementation and performance of the Relational Grid Monitoring Architecture (R-GMA). R-GMA is a unified Grid information and monitoring system built as part of the European DataGrid project [11].

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

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 and applications as quickly as possible. However, there can be a huge number of resources, running thousands of jobs, all 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 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 the flexibility that the relational model and SQL query language bring. The relational approach has a sound theoretical basis [8] and its framework has been extended recently to the world of streams by the database community [3]. In R-GMA, both traditional "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. Section 2 identifies the requirements that drove the design, which is presented in Section 3. We then describe how the task of "mediation" is achieved in Section 4. Section 5 describes the deployment of R-GMA in the DataGrid project while Section 6 puts our work into context with other Grid information systems. We present performance results for the scalability of R-GMA in Section 7. Section 8 contains our conclusions.

2. Grid Monitoring

We describe how DataGrid's middleware components interact. Then we discuss the requirements of a Grid information and monitoring system.

A *Grid monitoring system* often manages rapidly changing status data, e.g. the throughput of a network link, while a *Grid information system* would handle more static data, e.g. the hardware configuration of a machine. As we argue in the present paper, Grid components need a unified system that is able to deal with both kinds of data.

2.1. Overview of DataGrid Components

Figure 1 shows a "bird's eye" view of the components that make up a Grid. Institutions participating in a virtual organisation [12] 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 information and monitoring system, which is used by these components to publish or query information about the state of resources.

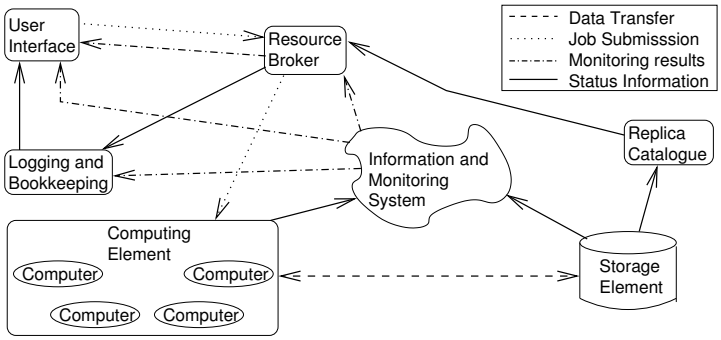

Figure 1. The major components of DataGrid.

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

The resource broker has to locate 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 information and monitoring system to find suitable computing elements (CE) close to the storage elements (SE) that hold the experimental data and forwards the job on to the computing elements to run. As the job progresses, information on the job's status is being published. The logging and bookkeeping service tracks and logs the jobs that are running on the Grid, by querying the information and monitoring system. Finally, users can monitor the progress of their job using the user interface, which queries the information and monitoring system.

2.2. Use Cases and Requirements

Typical use cases of a Grid information and monitoring system include:

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.

The main function of a Grid information and monitoring system is to allow both users and other components of the Grid to publish data, to locate data sources, and to query them. It must also meet certain performance and security constraints. These requirements are examined here.

2.2.1. *Publishing Data*

The act of publishing requires two abilities: (i) to advertise what data is available and (ii) 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 running on a computing element might be computed by a script (use case 3). A Grid information and 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 information and monitoring system should have publishing interfaces that support both of these perspectives.

2.2.2. *Locating and Querying Data*

Data will be distributed across the entire fabric of the Grid and a Grid information and 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.

Since a Grid information and monitoring system should be able to publish both static and stream data, it should also be possible to query both types of 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 information and monitoring system should support these three temporal query types.

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).

2.2.3. *Scalability, Performance and Robustness*

It is envisioned that Grids will contain many thousands of resources, spread over a large geographical area. For example, over 7000 CPUs are being made available via a Grid to physicists working on the Large Hadron Collider [18]. It is inevitable that the fabric of such a large Grid will be unreliable; network failures and other problems will be commonplace in Grids.

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

2.2.4. *Security*

Users of the Grid may only use resources they are authorized to use. Only if the they authenticate themselves, they are granted access to those resources to which they are authorized. The information and monitoring system should only provide access to information users are authorized to access. Similarly, resources should only be allowed to publish if they are authorized to publish.

3. R-GMA's Design

In this section we describe R-GMA's architecture and the rationale behind its design. We provide some details of the implementation of this design, but this is on-going work. Details of the deployment of R-GMA in the DataGrid project are discussed in Section 5.

3.1. The Grid Monitoring Architecture

The Global Grid Forum [14] have identified a Grid Monitoring Architecture (GMA) [23], which could offer the scalability and flexibility needed by a Grid monitoring system. It is a simple architecture consisting of 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.

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

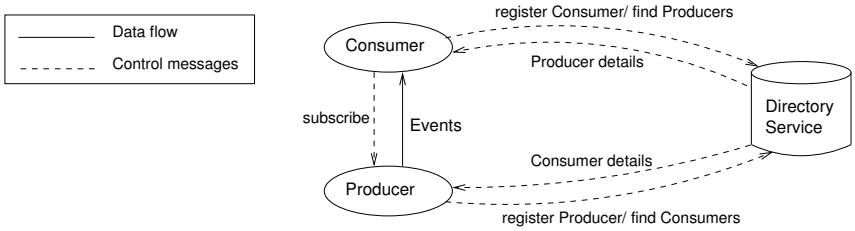

Figure 2. The components of the GMA and their interactions

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, the GMA does not define a data model, query language, nor a protocol for data transmission. Nor does it say what information should be stored in the directory service.

In the design of R-GMA we have taken the ideas from the GMA and extended them with the relational data model. This provides us with a query language, SQL, and a data format for transmitting data down the wire.

3.2. R-GMA: A Virtual Database

Users need to be able to locate interesting information using the information and 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 would not 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 [25] for matching queries posed over the global schema with sources of relevant information. Several approaches have been suggested for performing this "matchmaking role" [17]. 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, although query answering is not straightforward. We take the ideas of the local-as-view approach and apply them to data streams.

3.3. Software Design Considerations

The metaphor of Grid components playing the roles of producer and consumer, introduced by the GMA, is taken up and extended by R-GMA. Lightweight APIs are provided 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, registry and a consumer.

In our system, agents are realised as objects in Java servlets. A web-based architecture is convenient. It allows data to be streamed between agents as XML packets over a socket. Control messages can be exchanged between the agents and the registry and schema as HTTP requests.

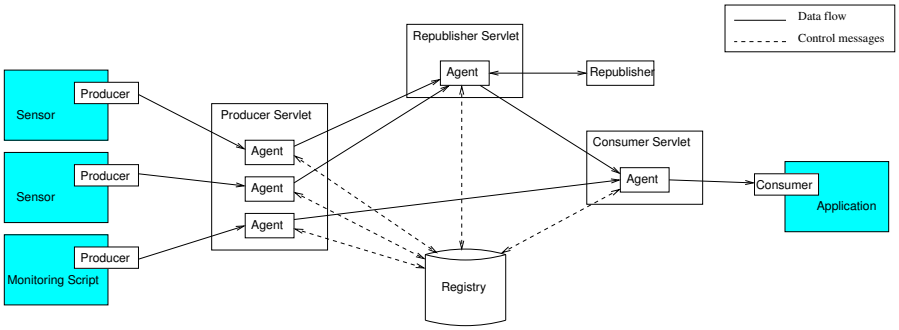

Figure 3. The roles and agents of R-GMA along with their interaction with the registry.

Protocols are needed to cope with the unreliability of Grid components and networks. We have adopted the common approach of the Grid Resource Registration Protocol (GRRP) heartbeats [10]. These are sent from API to agent and from agent to registry. If a heartbeat message should fail to arrive in a set time period then the receiver can assume that the sender is no longer alive.

3.4. R-GMA Components

Figure 3 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 (one per component) and a schema (omitted to reduce the complexity of the figure). Also, in R-GMA the registry is mostly hidden from our users: the consumer agent has mediator functionality that will perform registry lookups on behalf of the user. We now give an overview of the components of R-GMA, detailing their functionality.

3.4.1. *Schema*

In R-GMA a common 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 (i.e. table names) that together make up a global schema. R-GMA's *schema* component has the responsibility of storing and maintaining this global schema. 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 R-GMA schema is implemented as a servlet with a relational database to store details of the relations in the global schema. The

basic configuration has a set of relations that make up a core schema; however users can easily extend this by new relations.

R-GMA relations have attributes and types, as in SQL. As each relation represents a stream, it also implicitly has a timestamp attribute indicating at what time the tuple was published. A subset of the attributes can be singled out as the *primary key*. This is not enforced as a key constraint, but is used to identify the parameters of a measurement.

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

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

to record the *time* it took packets of a specific *size* to travel *from* one site *to* another, as measured by a certain *tool*. 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

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

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

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

3.4.2. *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.

The producer API provides a method for registering a view on the global schema that relates the stream of the producer to a relation in the global schema. Such a view is defined by a simple SQL query that ranges over a single relation and selects all attributes, e.g.

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

As a consequence, views for the same relation can only differ in their `WHERE` clause. By registering its view, the producer promises to publish only tuples for the table in the `FROM` clause that, moreover, satisfy the `WHERE` clause.

Another method in the API allows the producer to insert a collection of tuples into its stream. The producer agent supports a component

playing the role of a producer by taking the responsibility of storing tuples and answering queries.

3.4.3. *Consumer Role*

The role of a consumer is to locate and collect together monitoring data of interest. To perform this role it needs the ability of posing an SQL query of a certain type and of retrieving an answer, for which R-GMA offers methods in 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 *mediation*, i.e. translation into a local query, in order to be executed. Thus, the consumer API allows a client to pose a query and declare its type. The consumer agent takes on the task of planning the execution of the 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.

3.4.4. *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. The agent is realised as an object in a Java servlet.

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.

In R-GMA, all producer agents are capable of answering a continuous query. They do so by returning a stream consisting of all tuples inserted by the producer that satisfy the query. 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 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.

3.4.5. *Consumer Agents*
A consumer agent assists Grid components in playing the role of a consumer. Again, it is realise as an object in a Java servlet.

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, details are given in Section 4.

Queries are processed by the consumer agent constructing a query plan. It then 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 in Section 3.4.3.

3.4.6. *Republishers*
As the Grid grows in size, mechanisms are needed in R-GMA that ensure that popular queries are still answered efficiently. Republishers are used for this purpose. A republisher is similar to the view mechanism of a relational 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 on to 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 [9].

3.4.7. *Registry*
Consumer agents need to find publishers that can contribute to answering their query. This is facilitated by R-GMA's registry, which records all publishers that exist at any given point in time. The registry is another Java servlet which uses a database to store details of the publishers and consumers. However, the registry is hidden from the users/components of the Grid. 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. The mechanisms employed to perform the matchmaking within the registry are discussed in detail in Section 4.

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, by querying its database, and returns their address, type and other information, e.g. view conditions, 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 relevant publishers are registered or dropped.

4. Answering Queries in R-GMA

With R-GMA, clients are relieved of 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.

We will 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.

We also discuss the mediation process. As the semantics and the approach to mediation differs between continuous and one-time queries, we discuss each case separately.

4.1. Query Semantics

4.1.1. *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. In the following we will also use the relational algebra notation $\sigma_C(r)$ for such a selection query. If such a query is posed at time t_0, the answer is a stream that consists of all tuples of relation r that satisfy condition C and have a timestamp greater than t_0.

4.1.2. *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 can imagine one database, denoted as $\mathcal{D}^{(h)}$, which contains all these tuples. 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

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

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.

4.2. Answering Continuous Queries

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

4.2.1. *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 if it were not contacted.

In general, checking the satisfiability of un-nested 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, the registry generates a query over its database that retrieves all relevant publishers for Q.

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

4.2.2. *Query Planning and Execution*
Again, a simple approach has been chosen, in the current implementation of R-GMA, for planning continuous queries in that only producers are used to answer a query—republishers are not used. We have developed theoretical foundations for a more general approach [9] 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 existing ones die, the plan is adapted in a straightforward manner.

Techniques that make use of republishers need to be considerably more sophisticated. One reason is that 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.

4.3. 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 publisher 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 recently, 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 be executed by a "divide and conquer" approach, that

is, by processing it independently over each database and then merging the results. This approach works for example for selection queries.

To apply the "divide and conquer" 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 by each database, then the corresponding join could be executed by "divide and conquer." However, R-GMA does not support foreign keys so that this optimisation technique is not feasible.

4.3.1. *Matchmaking*

When a consumer agent receives a one-time query it checks whether it is a selection with a condition as in Formula 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 apply the "divide and conquer" strategy. 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 either it is a *republisher* and it publishes the query relations fully, or it is a *producer* and 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 "divide and conquer" is applicable. 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.

4.3.2. *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. It is technically possible, however, to improve R-GMA's mediator in such a way that it can also make use republishers which publish only fragments of the relation r to answer simple queries.

5. R-GMA in DataGrid

R-GMA was used within the DataGrid project [11] for publishing network monitoring information and providing information on resources. It was also tested for its suitability for providing job monitoring information. This served to help test and improve the implementation of R-GMA, and established a user community. In this section we discuss how R-GMA was used for collecting resource information and for providing information about the status of running jobs.

5.1. Collecting Resource Information

The first major use of R-GMA was to collect status information about Grid resources and make this available to the resource brokers. Status information may be obtained from the resources in an LDAP format. The resource brokers consume the data from their own local LDAP database. The LDAP data format was adopted as the Monitoring and Discovery Service (MDS 2) [10] was initially used as the information and monitoring system within DataGrid.

As stated in Section 3, R-GMA uses the relational data model. Thus, mechanisms were required to convert the published data from LDAP into relational tuples and back again for the resource broker to access. For this purpose Gadget In (GIn) and Gadget Out (GOut), respectively, were developed. These also allow for interoperability with Grids using the MDS 2 information system.

A latest-state republisher is used to populate and maintain a resource broker's LDAP database. This collects all the status information about resources into its own database. It is then converted, using the GOut tool, and copied into the resource broker's database. This approach has two drawbacks: (i) the resource broker's database is a copy of the republisher's database, and (ii) as the resource broker is not querying R-GMA directly it cannot take advantage of other latest-state republishers if its designated republisher should fail.

5.2. Job Monitoring

The scalability and stability of R-GMA were particularly tested when its suitability for job monitoring was tested. This was done in the context of batch jobs for the CMS and D0 experiments.

The batch jobs are wrapped so that they generate status information. This status information needs to be stored in a database located at the submitter's site. However, a resource broker may schedule the job to run on any site on the Grid so the status information must be returned to the submitter's site. This was achieved by the job wrapper creating a producer. A history republisher was then created which collated the information about all the jobs processed. The tests showed that R-GMA was then able to cope with batches of over 2000 jobs. Full details may be found in [6].

The facility to be able to extend R-GMA's schema at will proved to be useful for job monitoring. The information generated by the job wrappers did not fall in the core schema and thus R-GMA's schema was extended with suitable relations to capture this data.

6. Related Work

In this section we give an overview of existing Grid monitoring and Grid information systems. We also discuss work that has been conducted on data streams.

6.1. Grid Monitoring and Information Systems

Several Grid monitoring systems have been developed, each with a different emphasis and aim. Some have been developed for specific Grids, e.g. CODE [22] for NASA's Information Power Grid, while others are prototype systems to prove the correctness of the GMA approach, e.g. pyGMA [16].

Many Grid monitors only provide specific types of information. For example, Autopilot [19] used in the GrADS project and G-PM/OCM-G

[5] in the CrossGrid project have been developed to track the progress of jobs running on the Grid, while systems such as the Network Weather Service (NWS) [26] have been developed to monitor the status of the resources on the Grid.

R-GMA was designed as a unified Grid information and monitoring system to cover all aspects of the Grid. Other such unified systems are Scalea-G [24], Mercury [4], and the Monitoring and Discovery Service (MDS) [10] of the Globus toolkit. These can all be seen to be implementations of the GMA, with Scalea-G and MDS both using standard data models.

Scalea-G uses an XML data format and makes use of the XPath and XQuery query languages. It consists of Sensor Managers, a Client Service and a Directory Service. However, it has currently only been deployed on a small testbed with no indication of how it will scale to a large Grid.

MDS continues to be developed as part of the Globus Toolkit [15]. It consists of information providers and aggregate directories. 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. The MDS schema is not easily extensible. 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. The ability to perform a join is required by the resource broker, which needs to relate latest-state information about computing elements, storage elements and network links. The only way this can be conducted in MDS is with separate queries. These can be captured by a single query in R-GMA.

A more comprehensive comparison of these monitoring systems, and many others, can be found in the survey [13].

6.2. Data Streams

In recent years the database community has been actively researching the idea of processing stream data. Stream data occurs in many different scenarios including financial tickers, telecommunication logs and monitoring applications. A data stream is a time varying, unbounded, append-only sequence of data.

Standard database techniques require the entire data set to be loaded in before querying. For example, some techniques for processing joins block until the entire data set has been read once. This approach cannot be used for data streams as they are potentially infinite.

Several research data stream management systems (DSMS) such as STREAM [3], Telegraph [20] and Aurora [7] have been developed to deal with the characteristics of stream data. New ways of processing data which require only one pass and do not require the entire data set to have been processed have been devised, e.g. eddies [2]. Also, new query languages have been designed such as the relational based CQL language [1]. However, the work on DSMSs has been conducted for centralised systems, which are not suitable for a Grid information and monitoring system and many of the concepts for the query languages are beyond the requirements for a Grid monitoring system.

7. Performance Measurements

We have made measurements to explore the scalability of R-GMA. Our measurements are inspired by the work of Zhang et al. who conducted a performance evaluation of Grid information and monitoring systems [27]. They investigated four questions, which rephrased in R-GMA terminology read as follows:

1. How does the performance of a producer scale with the number of consumers?

2. How does the performance of the registry scale with the number of consumers?

3. How does the performance of a producer scale with the amount of data it is publishing?

4. How does a republisher scale with the number of producers it is aggregating?

We expect R-GMA to be deployed with information being published via producers, which stream tuples. Republishers then collect the information together. As a consequence, the load on the producers is very low as they are only feeding the republishers. Thus, Question 1 is not relevant for R-GMA. Moreover, by deploying more republishers the load on individual republishers can be reduced as necessary. However, since consumers have to contact the registry and usually are consuming from republishers, Questions 2, 3 and 4 are highly relevant.

Being aware of the registry as a single point of failure, R-GMA supports multiple registry instances with a replication mechanism between them. As the replication overhead must have some impact, we would not expect to see two registries giving twice the performance of one. To answer Question 2, we have made measurements with both one and two registries as described in 7.1.

Section 7.2 presents measurements which address Question 4 on the aggregation performance of a republisher in R-GMA. This test also involves publishing from a large number of producers via a producer servlet and so addresses Question 3 at the same time. There will be a limit on the amount of data one can push into one node on a network. The only way to circumvent this limit is not to attempt to merge all information but leave it partially distributed. To do this transparently requires the introduction of distributed query processing into R-GMA. Meanwhile it has to be done manually by setting up multiple republishers that each collect only a part of the information; the application must then combine results of queries from all the republishers.

7.1. Testing the Performance and Scalability of the Registry Service

This test reflects the second test in the study of Zhang et al. [27]. We used a fixed number of producers—all publishing to the same virtual table and with no `WHERE` clause. The measurements were made with ten and with fifty producers created on a single node. Separate nodes were used for the producer servlet, the registry and the schema.

To test the registry, we wrote a program to create and destroy consumers at varying intervals, each running a continuous query to the producers. Each creation contacts the registry to find all producers and registers the consumer. The destruction merely unregisters the consumer.

We were able to increase the load on the registry by running more instances of the program. Again, we used separate nodes for the consumers and the consumer servlet, otherwise we risked being limited by the performance of these components, and not by the performance of the registry.

By varying the total consumer creation/destruction frequency, we were able to observe the total load on the registry servlet by noting the percentage CPU idle time as reported by the `top` command (there were no other processes placing a significant load on the system). We expected to see that the maximum frequency would be limited by overloading of the registry servlet. By repeating the measurements with a pair of registries we were able to see how close we could get to doubling

the frequency of registry accesses, with any shortfall being due to the overhead of registry replication. All tests were run with logging (log4j) disabled.

7.1.1. *Results*

Figure 4 shows a plot of the percentage idle time on the registry servlet machine as a function of the overall frequency against the average time to complete the consumer create and close operations, with all measurements taken in the steady state. The lines labelled "indirect" show the results of carrying out the tests just described. As can be seen, the idle time decreases almost linearly with the registration frequency, and the response time also increases.

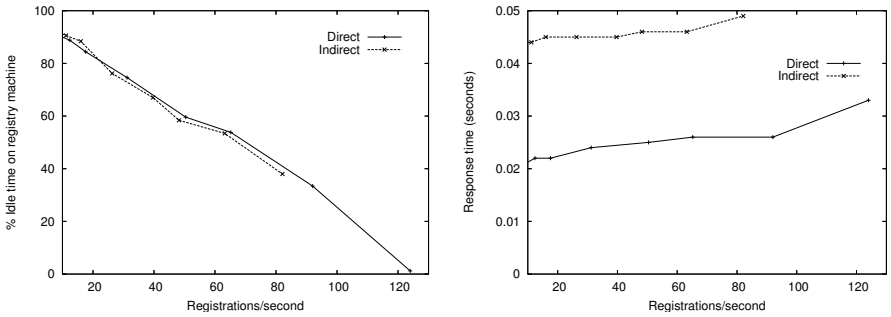

Figure 4. Idle time of the registry (left) and response time (right) vs rate of registration with direct and indirect connections from 4 machines.

We were, however, unable to saturate the registry with the available hardware. So to impose a greater load on the registry, we altered the test program to contact the registry directly, making the same calls as the consumer agent would normally do. This produced the lines labelled "direct", and the similarity between the "direct" and "indirect" lines in both plots justifies the simulation. Interestingly, the almost parallel offset in the response time graph (right) gives us an indication of the extra time taken by the consumer itself, with the "direct" connection being about 25ms faster. The main result is that the maximum rate of registration (with one registry) was around 125 registrations per second, at which point the CPU idle time was zero.

Repeating the test with a pair of registries produced the results plotted in Figure 5.

Note that even with eight direct connections to the pair of registries, we were unable to saturate them. An inspection of Figure 5 suggests that it is reasonable to extrapolate to a rate of 250 registrations per second for a zero idle time, though hardware limitations prevented us from demonstrating this.

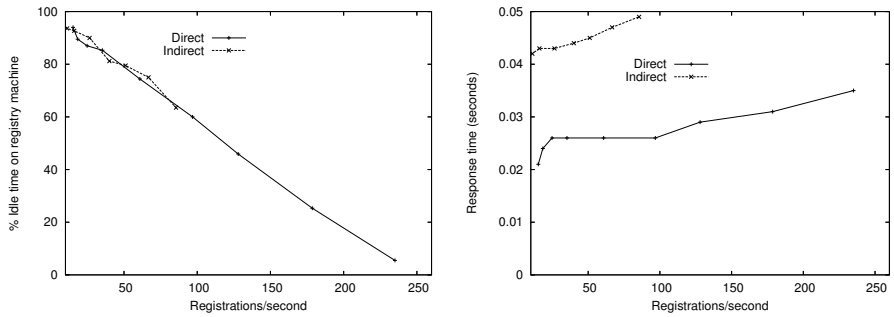

Figure 5. Idle time of the registry (left) and response time (right) vs rate of registration with direct connections to a pair of registries from 8 machines and indirect from 4 machines

For the response time, we note again that the consumer takes about 25ms in addition to the time taken by the registry.

The conclusion is that a single registry can cope with 125 consumers per second and that a second registry shows near-linear scaling behaviour. Although we would expect two registries to show slightly less than twice the performance of one (because of the overhead of replication), in this test, the effect was too small to measure.

7.2. Testing the Performance and Scalability of the Aggregation Process

This test used a simple consumer being fed by multiple producers, to assess the aggregation performance of a republisher. Since a republisher is essentially a consumer combined with a producer to publish the consumed tuples, this allowed us to concentrate on the aggregation part of the process which is carried out by the consumer. We investigated the effects of varying the tuple size and of varying the number of producers.

In each case, we monitored the tuples coming out of the consumer. We plotted both the idle time of the processor running the producer servlet and the propagation time of the data through the system, as a function of the rate of tuples consumed. We stopped when the tuples emerging were no longer up to date, as this indicated that data was beginning to be queued and so we had reached the maximum aggregation throughput. The machines were all synchronised with the Network Time Protocol (NTP), so measuring the propagation time merely required comparison of the published measurement time of a tuple with the time it was received by the consumer.

To ensure that we were limited only by the aggregation process, we used two producer servlets being fed by a program running multiple threads, each with a producer instance. This approach is identical, from

the consumer's point of view, to having many geographically dispersed producer agents, as there is a direct connection to the consumer from each producer instance.

7.2.1. *Results*

Figure 6 shows the effect on aggregation throughput, of varying the tuple size.

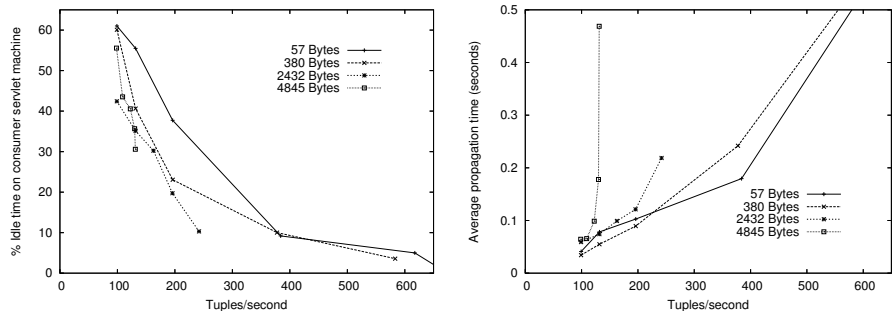

Figure 6. Idle time of the consumer servlet (left) and propagation time (right) vs rate of consumption for 100 producers over 2 producer servlet processors with tuple sizes of 57, 380, 2432 and 4845 Bytes

In each case, as the throughput increases, the idle time of the consumer servlet decreases. However the idle time drops close to zero only with the smaller of the tuple sizes, not for the 2432 byte and 4845 byte tuples. To understand why, we plotted the number of tuples queued on the consumer agent and the load on the consumer client. These results are shown in Figure 7, and they indicate that the consumer client is unable to keep up for the largest tuples, and in consequence, the data gets queued by the consumer agent. So, smaller tuple sizes are limited by the consumer agent, and larger ones by the consumer API.

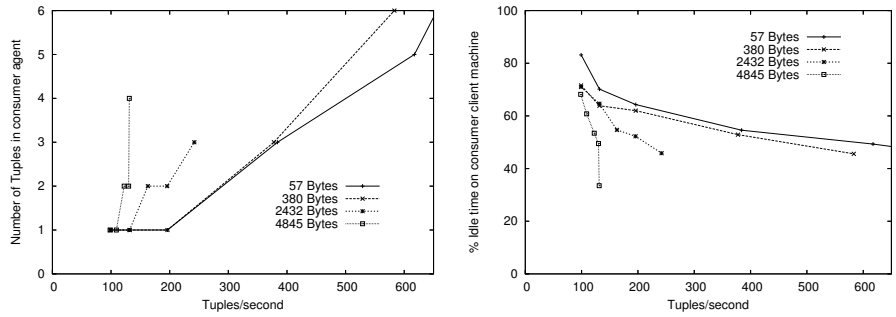

Figure 7. Number of tuples held by the consumer agent (left) and idle time of the consumer client (right) vs rate of consumption for 100 producers over 2 producer servlet processors with tuple sizes of 57, 380, 2432 and 4845 Bytes

Figure 8 shows the maximum consumption rate of tuples plotted against tuple size, fitted to a function of the form $f(n) = 1/(a + bn)$, assuming that the time to transfer n bytes is $a+bn$, for some parameters a and b. The curve shown has values of $a = 1404\mu$s and $b = 1.00\mu$s/byte. However, there are different limiting factors for small and large tuples and these would need to be considered when deploying on a real Grid.

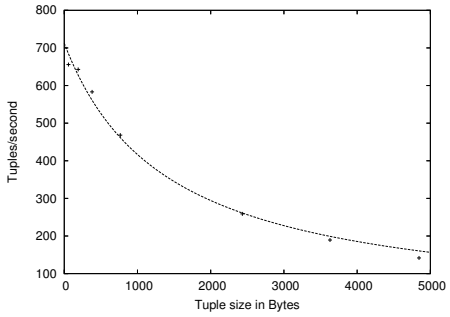

Figure 8. Maximum consumption rate of tuples vs tuple size

Finally, we tried varying the number of producers. As more producers were added, the rate of publishing for each one was reduced. The results for a tuple size of 765 bytes are shown in Figure 9 (where "SP" denotes a streaming producer).

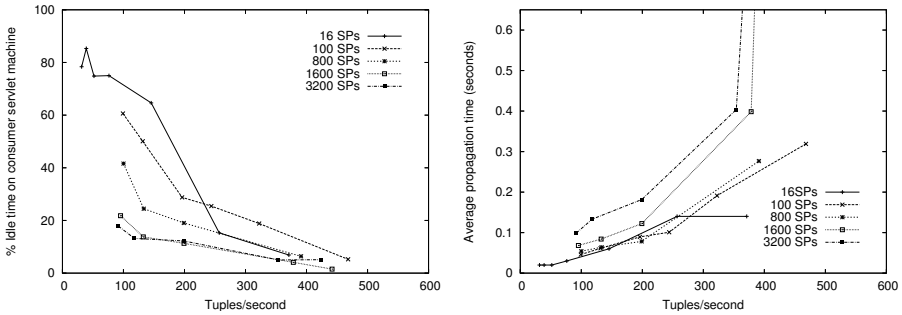

Figure 9. Idle time of the consumer servlet (left) and propagation time(right) vs rate of consumption for a tuple size of 765 bytes published via 2 producer servlet processors with 16, 100, 800, 1600 and 3200 streaming producers

We see that at low rates, consumer loading is strongly related to the number of producers, but the upper limit seems to be largely independent of the number of producers. We were again limited in testing by available hardware, and were not able to go beyond 3200 producers because of the number of Java threads running on the nodes with the producers and producer agents. The maximum rate for 3200 produc-

ers was 423 tuples per second. This involved having 1600 producers publishing via each of the producer servlets.

7.3. Hardware Configuration

The tests were carried out using release 4.0.0 of R-GMA on machines with an AMD Athlon XP 2400+. The machines had 1GB of RAM, and were connected through *Via RhineII 10/100 VT6102* NICs to a *Dlink 1016D 10/100* Ethernet Switch. We used Tomcat 4.1.18 and Sun's Java j2sdk1.4.2_05 with a maximum heap size of 512MB. The `ulimit` command was used to increase the limit on open files on the consumer servlet.

7.4. Deployment Experience

R-GMA is now being tested within the Large Hadron Collider Computing Grid Project (LCG) coordinated by CERN. It is currently installed on about 50 sites. The performance of R-GMA has not been a bottleneck. However, there can be problems when sites are not configured correctly, e.g. firewalls blocking the streaming of data. From this experience, the performance of R-GMA on a Grid is consistent with the measurements.

8. Conclusions and Future Work

We conclude with a discussion of how well R-GMA meets the requirements identified for a Grid information and monitoring system and describe the work that will be conducted to extend the implementation.

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.

Our measurements indicate that we have good scaling behaviour. A registry can cope with 125 consumers per second being created and destroyed. A second registry shows perfect scaling behaviour within the accuracy of our measurements. A consumer, aggregating tuples can process a tuple in a time of $1404\mu s + 1.00\mu s$/byte. We were able to aggregate data from 3200 producers with no indication of an approaching limit. We were able to publish data from 1600 producers connected to a single producer servlet.

Work on R-GMA is continuing within the Enabling Grids for E-Science in Europe (EGEE) project where it is being re-engineered as a set of Web Services and where great emphasis is being placed on the quality and portability of the code.

As Grids will be shared by increasing numbers of virtual organisations (VOs), each will need their own name space and control of security. We will allow each VO to have its own virtual database, each having authorization rules to control publishing and consuming of information. These rules will allow fine grained authorization.

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 [21]) proves to be suitable, R-GMA will be extended to execute queries over multiple producers.

References

1. Arasu, A. and J. W. S. Babu: 2003, 'CQL: A Language for Continuous Queries over Streams and Relations'. In: *10th International Workshop on Database Programming Languages*. Potsdam, Germany, pp. 1–19.
2. Avnur, R. and J. Hellerstein: 2000, 'Eddies: Continuously Adaptive Query Processing'. In: *Proc. 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, USA, pp. 261–272.
3. Babcock, B., S. Babu, M. Datar, R. Motwani, and J. Widom: 2002, 'Models and Issues in Data Stream Systems'. In: *Proc. 21st Symposium on Principles of Database Systems*. Madison, USA, pp. 1–16.
4. Balaton, Z. and G. Gombás: 2003, 'Resource and Job Monitoring in the Grid'. In: *Proc. 9th International Euro-Par Conference*, Vol. 2790 of *LNCS*. Klagenfurt, Austria, pp. 404–411.
5. Balis, B., M. Bubak, W. Funika, T. Szepieniec, R. Wismüller, and M. Radecki: 2003, 'Monitoring Grid Applications with Grid-enabled OMIS Monitor'. In: *Proc. First European Across Grids Conference*, Vol. 2970 of *LNCS*. Santiago de Compostela, Spain, pp. 230–239.
6. 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: *Proc. IEEE 2003 Nuclear Science Symposium*. Portland, USA.
7. Carney, D., U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik: 2002, 'Monitoring streams—a new class of data management applications'. In: *Proc. 28th International Conference on Very Large Data Bases*. pp. 215–226.
8. Codd, E.: 1970, 'A Relational Model of Data for Large Shared Data Banks'. *Communications of the ACM* **13**(6), 377–387.
9. Cooke, A., A. Gray, and W. Nutt: 2004, 'Stream Integration Techniques for Grid Monitoring'. *Journal on Data Semantics*. To appear.

10. Czajkowski, K., S. Fitzgerald, I. Foster, and C. Kesselman: 2001, 'Grid Information Services for Distributed Resource Sharing'. In: *10th International Symposium on High Performance Distributed Computing*. San Francisco, USA, pp. 181–194.
11. DataGrid: 2004, 'The DataGrid Project'. http://www.eu-datagrid.org.
12. 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.
13. Gerndt, M., R. Wismüller, Z. Balaton, G. Gombás, P. Kacsuk, Z. Németh, N. Pordhorszki, H. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef, 'Performance Tools for the Grid: State of the Art and Future'. Technical report, Research Report Series, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen, Vol. 30, Shaker Verlag, ISBN 3-8322-2413-0.
14. Global Grid Forum: 2004, 'Global Grid Forum'. http://www.ggf.org.
15. Globus: 2004, 'Globus Toolkit'. http://www.globus.org.
16. Gunter, D.: 2004, 'An overview of the pyGMA'. http://sourceforge.net/docman/display_doc.php?docid=8662&group_id=28724.
17. Halevy, A.: 2001, 'Answering queries using views: A survey'. *The VLDB Journal* **10**(4), 270–294.
18. LCG: 2004, 'LHC Computing Grid Project'. http://lcg.web.cern.ch.
19. Ribler, R., J. Vetter, H. Simitci, and D. Reed: 1998, 'Autopilot: Adaptive control of distributed applications'. In: *7th International Symposium on High Performance Distributed Computing*. Chicago, USA, pp. 172–179.
20. Shah, M., S. Madden, M. Franklin, and J. Hellerstein: 2001, 'Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System'. *SIGMOD Record* **30**(4), 103–114.
21. Smith, J., A. Gounaris, P. Watson, N. Paton, A. Fernandes, and R. Sakellariou: 2002, 'Distributed Query Processing on the Grid'. In: *Proc. of 3rd Int'l Grid Computing - GRID 2002*, Vol. 2536 of *LNCS*. Baltimore, USA, pp. 279–290.
22. Smith, W.: 2002, 'A System for Monitoring and Management of Computational Grids'. In: *Proc. 31st International Conference on Parallel Processing*. Vancouver, Canada, pp. 55–64.
23. Tierney, B., R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski: 2000, 'A Grid Monitoring Architecture'. Technical report, Global Grid Forum. Revised January 2002.
24. Truong, H. and T. Fahringer: 2004, 'SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid'. In: *Proc. Second European Across Grids Conference*. Nicosia, Cyprus. To appear.
25. Wiederhold, G.: 1992, 'Mediators in the Architecture of Future Information Systems'. *IEEE Computer* **25**(3), 38–49.
26. Wolski, R., N. Spring, and J. Hayes: 1999, 'The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing'. *Journal of Future Generation Computing Systems* **15**(5–6), 757–768.
27. Zhang, X., J. Freschl, and J. Schopf: 2003, 'A Performance Study of Monitoring and Information Services for Distributed Systems'. In: *12th International Symposium on High Performance Distributed Computing*. Seattle, USA, pp. 270–282.