R-GMA: An Information Integration System for Grid Monitoring

Andy Cooke[1], Alasdair J G Gray[1], Lisha Ma[1], Werner Nutt[1], James Magowan[2], Manfred Oevers[2], Paul Taylor[2], Rob Byrom[3], Laurence Field[3], Steve Hicks[3], Jason Leake[3], Manish Soni[3], Antony Wilson[3], Roney Cordenonsi[4], Linda Cornwall[5], Abdeslem Djaoui[5], Steve Fisher[5], Norbert Podhorszki[6], Brian Coghlan[7], Stuart Kenny[7], and David O'Callaghan[7]

[1] Heriot-Watt University, Edinburgh, UK
[2] IBM-UK
[3] PPARC, UK
[4] Queen Mary, University of London, UK
[5] Rutherford Appleton Laboratory, UK
[6] SZTAKI, Hungary
[7] Trinity College Dublin, Ireland

Abstract. Computational Grids are distributed systems that provide access to computational resources in a transparent fashion. Collecting and providing information about the status of the Grid itself is called Grid monitoring.

We describe R-GMA (Relational Grid Monitoring Architecture) as a solution to the Grid monitoring problem. It uses a local as view approach to information integration and will be a component of the European Union's DataGrid.

The R-GMA architecture and mechanisms are general and could be used in other areas where there is a need for publishing and querying information in a distributed fashion.

1 Introduction

In this paper we discuss how to monitor the state of a dynamically changing computational Grid, and our approach to this—a data integration system called R-GMA (Relational Grid Monitoring Architecture). Grid monitoring requires the publication of static and dynamic data, a global view of this data, and a query mechanism capable of dealing with "latest-state", "continuous", and "history" queries. It also needs to be scalable to allow hundreds of nodes to publish and be resilient if any node fails. There are also issues of privacy of data that need to be addressed.

We have designed a data integration system that meets most of these requirements, within the European Union's DataGrid project [1]. It aims to develop a computational Grid to allow three major user groups to process and analyse the results of their scientific experiments: (1) high energy physics to allow them to

distribute and analyse the vast amounts of data that will be produced by the Large Hadron Collider at CERN, (2) biological and medical image processing as part of exploitation of genomes, and (3) the European Space Agency's Earth Observation project to analyse images of atmospheric ozone.

During the past two years, we have implemented a working R-GMA system within DataGrid. Our aim was to develop functionality that had a firm theoretical basis and that was flexible enough to quickly respond to requirements as they became clearer. We will describe the status of our implementation as DataGrid enters its final year. R-GMA has an open-source license, and can be downloaded from [2].

In section 2 we describe what a computational Grid is, and outline the requirements of a Grid monitoring system. In section 3 we describe possible approaches to Grid monitoring, including existing systems, and discuss why these do not meet the requirements identified. We then present our R-GMA architecture for a Grid monitoring system in section 4. Hierarchies of republishers will allow queries to be answered efficiently. We discuss the query planning needed to automatically maintain these hierarchies in section 5. The state of the current implementation is presented in section 6.

2 Grid Monitoring: Overview and Requirements

We shall introduce the idea of a computational Grid and describe the components of a Grid. We explain what is meant by Grid monitoring, and identify requirements for a Grid monitoring system.

2.1 Computational Grids

A *computational Grid* is a collection of connected, geographically distributed computing resources belonging to several different organisations. Typically the resources are a mix of computers, storage devices, network bandwidth and specialised equipment, e.g. supercomputers or databases. A computational Grid provides instantaneous access to files, remote computers, software and specialist equipment [10]. To a user, a Grid behaves like a single virtual supercomputer.

The concept of a computational Grid has existed since the mid 1990s and has grown out of the distributed and high performance computing communities. There have now been several projects to construct computational Grids to perform different tasks. These include: the Globus Toolkit [12], NASA's Information Power Grid [13], CrossGrid [9] and TeraGrid [6].

To make a computational Grid behave as a virtual computer requires various components that mimic the behaviour of a computer's operating system. The components of DataGrid, and their interactions, can be seen in Fig. 1 and are similar to those presented in [11].

User Interface: allows a human user to submit jobs, e.g. "analyse the data from a physics experiment, and store the result".

Resource Broker: controls the submission of jobs, finds suitable available resources and allocates them to the job.

Logging and Bookkeeping: tracks the progress of jobs, informs users when jobs are completed, which resources were used, and how much they will be charged for the job.

Storage Element (SE): provides physical storage for data files.

Replica Catalogue: tracks where data is stored and replicates data files as required.

Computing Element (CE): performs the processing of jobs, taking data from storage elements.

Monitoring System: monitors the state of the components of the Grid and makes this data available to other components.

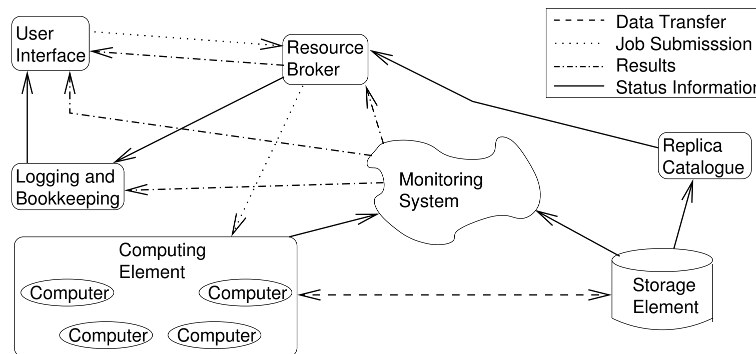

Fig. 1. The major components of DataGrid.

2.2 Grid Monitoring Requirements

The purpose of a Grid monitoring system is to make information about the status of a Grid available to users and to other components of the Grid. The following are typical use cases:

1. A resource broker needs to locate a computing element (CE), that has 5 CPUs available each with at least 200 MB of memory. The CE should have the right software installed, and the user must be authorised to use it. The throughput to an SE needs to be greater than 500 Mbps.
2. A visualisation tool is used by users to monitor the progress of their jobs needs to be updated whenever the status of a job changes.
3. Network administrators need to interrogate the past state of the network so that typical behaviour can be ascertained and anomalies identified.

Static and Stream Data. Information about a Grid comes from many different sources, e.g.

- Statistics from the resources on the Grid about themselves, e.g. the data about a machine delivered by a UNIX command such as `w`.
- Measurements of network throughput, e.g. made by sending a `ping` message across the network and publishing the runtime (use cases 1 and 3 above).
- Job progress statistics, either generated by annotated programs or by the resource broker (use case 2).
- Details about the topologies of the different networks connected (use cases 1 and 3).
- Details about the applications, licenses, etc., available at each resource (use case 1).

This monitoring data can be distinguished into two types based on the frequency with which it changes and depending on the way in which it is queried.

Static data (pools): This is data that does not change regularly or data that does not change for the duration of a query, e.g. data in a database with concurrency control. This is typically data about the operating system on a CE, or the total space on an SE (use case 1).

Dynamic data (streams): This is data that can be thought of as continually changing, e.g. the memory usage of a CE (use case 1), or data that leads to new query results as soon as it is available, for example the status of a job (use case 2).

A requirement, then, of a Grid monitoring system is that it should allow both static and streaming data to be published.

Locating Data. Monitoring data on a Grid will be published by the distributed components of the Grid. The monitoring system must provide mechanisms for users of the Grid to locate data sources. Users need a *global view* over these data sources, in order to understand how the data relates and to query it.

Queries with Different Temporal Characteristics. A monitoring system should support queries posed over data streams, over data pools, or over a mix of these (use case 1). It should be possible to ask one-time queries about the state of a stream right now (a *latest-state* query – use case 1), continuously from now on (a *continuous* query – use case 2) or in the past (a *history* query – use case 3). Up to date answers should be returned quickly—the resource broker needs information that is no more than 10 seconds old in use case 1. 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.

Scalability and Performance. A Grid is potentially very large: DataGrid's testbed currently has hundreds of resources, and will scale up by the end of the project. The fabric of the Grid will be unreliable: network connections will fail, resources will become in-accessible.

It is important that the monitoring system can *scale* to handle the large amounts of data published and still return correct answers in a timely manner. It should not become a performance bottleneck for the entire Grid. It should be able to cope with large numbers of queries received at the same time.

The monitoring system itself should be resilient to failure of any of its 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.

Security. An information source must be able to control who can "see" its data and this must also be respected by the monitoring system. A user should be able to identify themselves so that they can make use of the resources that they are entitled to. Resources should be able to prevent access to users who are not authorised.

3 Possible Approaches to Grid Monitoring

Peer to Peer systems allow resources to be shared across the Internet. However, as such systems do not offer a global view over all the resources available, they could not be used for Grid monitoring.

Another related technology are data stream management systems, for which several prototypes now exist. Here we will examine whether these could suitable for Grid monitoring. The Grid community have proposed the Grid Monitoring Architecture as a general architecture for a Grid monitoring system. We will discuss this architecture and existing systems such as the Monitoring and Discover Service which ships with Globus.

3.1 Data Stream Management Systems.

Data streams show up in many different situations where dynamically changing data can be collected, e.g. stock market prices, sensor data, monitoring information. Recently, the idea of a centralised data stream management system (DSMS) has been developed, and some preliminary systems have been implemented, such as STREAM [5], Aurora [7], Tribeca [18] and AIMS [16]. They support querying and management of relations, akin to a relational database management system, only these relations may be either streaming or static.

The existing DSMS do not meet all the requirements of Grid monitoring (section 2.2). The centralised systems developed today would not cope dynamically with the creation and removal of geographically distributed streams nor coordinate the communication of data from sources to clients. This central point would become single point of failure as all information sources and clients of the systems would need to interact with it.

3.2 Grid Monitoring Architecture

The Grid Monitoring Architecture (GMA) was proposed by Tierney *et al.* [19] and has been accepted as a standard for Grid monitoring systems by the Global Grid Forum [3]. It is a simple architecture comprising of three main components:

Producers: A source of data on the Grid, e.g. a sensor, or a description of a network topology.

Consumers: A user of data available on the Grid, e.g. a resource broker, or a system administrator wanting to find out about the utilisation of a Grid resource.

Directory Service: Stores details of producers and consumers to allow consumers to locate relevant producers of data.

The interaction of these components can be seen in Fig. 2. A producer informs the directory service of the kind of data it has to offer. A consumer contacts the directory service to discover which producers have data relevant to its query. A communication link is then set up directly with each producer to acquire data. Consumers may also register with the directory service. This allows new producers to notify any consumers that have relevant queries.

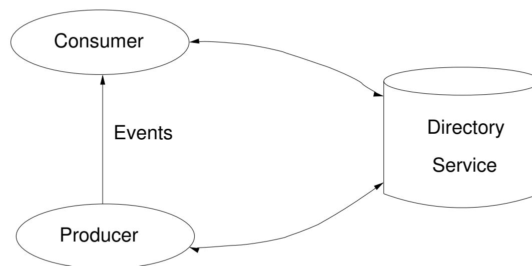

Fig. 2. The components of the GMA and their interactions

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, 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.

3.3 Existing Systems

There are several research systems implementing the GMA: AutoPilot [15], CODE [17], Monitoring and Discovery Service (MDS) [8], etc. However, only MDS is well known, because it is included in the widely-used Globus Toolkit [12].

Monitoring and Discovery Service. Although MDS[8] was designed and implemented before the GMA was proposed, it can still be seen to fit into the architecture. It consists of information providers (GMA producers) and aggregate directories (GMA directory service and intermediary). Data is organised in a hierarchical system based on the Lightweight Directory Access Protocol (LDAP); this provides the MDS system a name space, data model, wire protocol, and querying capabilities.

Although MDS is *scalable*, it does not meet other requirements outlined in section 2.2. Firstly, the LDAP query language has limitations. The hierarchy must be designed with popular queries in mind. Also, there is no support for users who want to relate data from different sections of the hierarchy—they must process these queries themselves.

To be able to offer a global view of the Grid to users, a hierarchy of intermediaries must be set up manually—providers and intermediary aggregate directories need to know which directory to register with. The system does not automate this, nor does it recover if any component in the hierarchy fails.

Lastly, MDS only supports latest-state queries with no assurance that the answers are up to date. It is claimed that you can create an archive of information by storing the latest-state values in a database and providing an LDAP interface to allow the system to access it. However, this would require considerable user effort.

4 The R-GMA Approach

R-GMA builds upon the GMA proposal (see section 3.2), but specifies a data model, a query language, and the functionality of the directory service. It also adds further components. Below, we introduce the main elements of the R-GMA architecture and explain the rationale of their design: consumers, producers, consumer and producer agents, schema, republishers, and registry.

We plan to finalise the implementation of the architecture presented by the end of this year. The current state of the implementation is discussed in section 6.

4.1 R-GMA as a Virtual Data Warehouse

The idea underlying R-GMA is to let the Grid monitoring data appear as being stored in or streaming through one large relational data warehouse.

Although it has occasionally been suggested to use a relational database management system for Grid monitoring [14], such an approach would not meet the requirements of section 2.2. For instance, the loading of data takes time; there may not be sufficient space to store all the data; connections to the database may fail so that the information will no longer be accessible; and finally, monitoring data often flow as data streams and queries ask for data streams as output, which is not supported by current database management systems.

[8] We discuss here the details of version 2, broadly similar to version 1 but the hierarchical LDAP structure has been decentralised.

In R-GMA, the warehouse is only virtual. Clients query the system via a global schema, but behind the scenes a data integration system directs data from the sources to the client.

4.2 Roles and Agents

R-GMA takes up the consumer and producer metaphors of the Grid Monitoring Architecture (see section 3.2) and refines them. An R-GMA installation allows clients, which may be Grid components or applications running on the Grid, to play the *roles* of information *producers* or *consumers*.

Producers. The producer role comes in two variants, the roles of *database producer* and of *stream producer,* depending on whether the data sets they make available are pools or streams. More specifically, a database producer publishes a collection of relations maintained in a relational database while a stream producer publishes several streams of tuples, each of which complies with the schema of a specific relation. We refer to these static or streamed relations as the *local relations* of a producer.

Consumers. A consumer is defined by a relational query. If the query is posed over a stream, the consumer has to declare whether it is to be interpreted as a continuous, a history, or a latest-state query (see section 2.2). Upon request, the consumer receives answers to its query.

Agents. R-GMA provides *agents* that support clients in their roles, for instance as consumer agents, or stream producer agents.

Currently, the role metaphor is implemented by an API that allows one to create objects for the various roles, like consumer or stream producer objects. Agents are realised as objects accessible via a Web server. Details are discussed in section 6.

4.3 The Global Schema

Producers and consumers can only interact with each other if there is a *language* and a *vocabulary* in which producers describe the information they supply and consumers the information for which they have a demand. In R-GMA, both the *language* for announcing supply and the one for specifying demand—that is, the query language—are essentially fragments of SQL.

The relations and attributes that make up the vocabulary are part of a global schema, which is stored in R-GMA's *schema* component.

The global schema distinguishes between two kinds of relations, *static* and *stream* relations. The two sets are disjoint. The global schema contains a collection of core relations that exist during the entire lifetime of an installation.

In addition, producers can introduce new relations to describe their data, and withdraw them again if they stop publishing.

Relations have attributes with types as in SQL. In addition to the attributes that are declared explicitly, stream relations have an additional attribute timestamp, which is of a type DateTime and records the point in time when a tuple was published.

For both kinds of relations, a subset of the attributes can be singled out as the *primary key*. We interpret primary keys as follows: any two tuples in the system that agree on the key attributes and the timestamp also agree on the remaining attributes. For the stream relations, the keys usually identify the parameters of a measurement. For instance, R-GMA's schema contains the core relation tp to publish measurements of the throughput of network links. The relation has the schema

tp(from, to, tool, psize, value, [timestamp]),

to record the time it took, according to measurements by a certain tool, to transport packets of a specific size from one node to another one. All attributes except value make up the primary key of tp. Intuitively, the key attributes of a stream relation identify a *channel* along which measurements are communicated.

Due to the distributed origin of information in R-GMA, the key constraints cannot be strictly enforced. However, R-GMA can check the views of producers to ensure that no two producers of a relation publish for the same keys. Consequently, key constraints hold globally if they hold locally.

Consumers pose queries over the global schema. Similarly, producers describe their local relations as views on the global schema. In Ullman's terminology [20], this means that R-GMA takes a "local as view" approach to data integration.

4.4 Producers and Consumers: Semantics

R-GMA requires that producers declare their content using views *without projections*. Thus, each producer contributes a set of tuples to each relation. This allows us to give an R-GMA installation an intuitive semantics: a static relation is interpreted as the union of the contributions published by the database producers; a stream relation is interpreted as a global stream obtained by merging the streams of all the stream producers.

Actually, the semantics of stream relations is not as well-defined as it may seem because it does not specify an order for the tuples in the global stream. We do not guarantee a specific order on the entire global stream. However, we require, that for a given channel the order of tuples in the global stream is the same as in the producer stream where they originated. We explain in section 5 how R-GMA enforces this constraint.

We are aware that our semantics of stream relations causes difficulties for some kinds of queries, for instance, aggregate queries over sliding windows where the set of grouping attributes is a strict subset of the keys. In such a case, different orderings of a stream can give rise to different query answers. We have not yet dealt with this issue.

Among the three temporal interpretation of stream queries, two are supported by stream producer agents, continuous and latest-state queries. For latest-state queries, the agent maintains a pool with the latest values of each channel for which its client produces values.

4.5 Republishers

Republishers in R-GMA correspond to the intermediaries in the GMA and resemble materialised views in database systems. Their main usage is to reduce the cost of certain query types, like continuous queries over streams, or to set up an infrastructure that enables queries of that type in the first place, like history queries.

A republisher is defined by one or more queries over the global schema and publishes the answers to those queries. The queries either have to be all continuous or all one-time queries.

A republisher combines the characteristics of a consumer and a producer. Due to the redundancy of information created by republishers, there are often several possibilities to answer a query. Section 5 describes how this is taken into account in the construction of query execution plans for simple stream queries.

Since R-GMA supports essentially two types of queries (one-time and continuous) and two types of producers (database and stream producers), in principle four main types of republishers are conceivable, depending on which query type is combined with which producer type. Currently, R-GMA supports two out of these four possible combinations: *stream republishers* and *archivers*.

Stream Republishers. Stream republishers output the query answers as they are generated, as a stream. In addition, similar to a stream producer agent, the stream republisher agent maintains a pool of latest-state values so that it can answer both continuous and latest-state queries.

Since both input and output are streams, one can build *hierarchies* of stream republishers over several levels. An important usage for such hierarchies is to bundle small flows of data into larger ones and thus reduce communication cost.

Stream producers often publish data obtained from sensors, such as the memory usage of computing elements, which are distributed all over the Grid. While such primary flows of data, to elaborate on the metaphor, tend to be trickles, with stream republishers they can be combined into streams proper. For instance, stream republishers may be used to collect first the memory usage of CEs at one site and, then at the next level up, of those belonging to an entire organisation participating in a Grid. Thus a consumer asking for the memory usage of the CEs at some site only needs to contact the republisher for that site instead of all the individual stream producers.

Archivers. An archiver stores the answers to a collection of continuous queries in a database. For each query, one can specify for how long query answers are stored. If a consumer poses a history query over one or more streams, it is answered using the data in archivers.

4.6 The Registry

We refer to producers and republishers together as *publishers.* 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 and consumers that exist at any given point in time. Publishers and consumers send a heartbeat to the registry at predefined intervals to maintain their registration.

When a new publisher is created, its agent contacts the registry to inform it about the type of that publisher. If the publisher is a producer, the agent registers its local relations together with the views on the global schema that describe their content. The registration is only accepted if the producer's view is disjoint from the view of all other producers. If it is a republisher, the agent registers its queries. Similarly, when a consumer is created, the consumer's agent contacts the registry with the consumer's query.

The registry cooperates with the consumer agent in constructing a query plan. It identifies publishers that can contribute to the answers of that query, called the *relevant* publishers. Due to the existence of republishers, there may be some redundancy among the relevant publishers. Therefore, the registry selects those which can contribute maximally, called *maximal* relevant publishers, and returns them to the consumer agent. Then the agent constructs the concrete plan.[9] For each consumer, the registry remembers the query and the list of maximal relevant publishers that it has returned.

When a consumer's query is registered, R-GMA ensures that during the entire lifetime of the consumer it can receive all the data the query asks for. To achieve this, whenever a new producer registers the registry identifies the consumers to which this producer is relevant. Then it inspects the the current list of maximal relevant publishers for the consumer to see whether the data from the new producer will be delivered by some republisher. If not, the new producer is added to the list and the consumer's agent is notified. Similarly, query plans need to be checked when a republisher goes offline because then a consumer may miss data that it has received via that republisher.

Finally, the registry informs consumer agents if a new relevant republisher is created and when a producer goes offline.

5 Republisher Hierarchies

Most queries in R-GMA require the latest values of several stream relations to be joined and aggregated. Consider for example the needs of the resource broker (section 2.2, use case 1). Such a query can be answered efficiently by setting up a hierarchy of republishers to collect the data needed. The complex query can then be answered by the top level republisher, with the help of a DBMS that holds a pool of latest-state values.

[9] In section 5 we discuss in detail how the registry selects those publishers for the case of continuous selection queries over streams and how the consumer agent creates the plan.

Each republisher poses a simple continuous query, which is a selection over a single relation. The currently deployed version of R-GMA supports only republishers defined by queries whose selection conditions are conjunctions of equalities of the form "$attr = val$". The next version will also allow comparisons such as "$attr \leq val$" or tests of the form "$attr$ in $(val_1, \ldots, val_n)$".

The creation and maintenance of such republisher hierarchies requires reasoning about the query conditions. We describe a mechanism that constructs and executes plans for the simple continuous queries defined above.

Consider a consumer that poses a continuous query Q over a stream relation r that selects all tuples satisfying a condition C where C involves comparisons and equalities between attributes of r and arbitrary values. Using relational algebra, we can write such a query as $Q = \sigma_C(r)$.

For such a consumer we discuss the query *planning* problem, which consists of locating relevant publishers, and planning how to query these and merge the results. Since republishers also play the role of consumers, the discussion applies to them as well.

Another important problem is the plan *adaptation* problem, which arises when a new publisher comes into existence or when a publisher from which the consumer is streaming ceases to exist. Our theory for query planning provides also a basis for adapting plans to a changed environment. However, due to space limitations we are unable to discuss this point in the present paper.

5.1 Properties of Streams and Stream Publishers

To design a suitable query processing mechanism, we first collect essential properties that all stream publishers in R-GMA are required to have.

Some of these are actually properties of the streams they publish. To ease our presentation, we assume that every stream publisher has only one local relation. This allows us to identify a publisher with its local relation and to denote both with the same letter, like P, R, S. The local relation is described by a view on the global schema, which has the form $\sigma_D(r)$.

We say that two streams are *disjoint* if they do not share any tuples. We say that two publishers are *disjoint* if their streams are disjoint. One can enforce that any two stream producers are disjoint by allowing a new stream producer S to register only if its descriptive view $\sigma_D(r)$ is disjoint from the view $\sigma_{D'}(r)$ of any existing stream producer S', i.e. if $D \wedge D'$ is unsatisfiable.

A publisher is *sound* w.r.t. its descriptive view $\sigma_D(r)$ if its output always consists of tuples that comply with the schema of relation r and satisfy the condition D. In R-GMA soundness of stream producers is enforced by the producer agents, which screen the tuples that are being published.

A publisher is *complete* w.r.t. its view if it outputs every tuple in the system that satisfies the view. Since stream producers are sound and their views are mutually disjoint, a stream producer S_1 can never output a tuple that satisfies the view of another producer S_2. Thus, stream producers are complete by design. For republishers, completeness has to be ensured via appropriate query planning.

A stream is *duplicate free* if it never outputs a tuple twice. This property is important when we compute aggregate values that are sensitive to multiplicities, like counts or sums. A stream is *weakly ordered* if for each channel, the tuples output in the order of their timestamps. We say that a publisher is duplicate free or weakly ordered if its stream has the corresponding property. Again, in R-GMA stream producers are duplicate free and weakly ordered by design because the producer agents attach a timestamp to a tuple when it is published, so that any two tuples are distinct. For republishers, these two properties have to be guaranteed by their query plans.

5.2 Query Plans and their Properties

A query Q posed against the global schema is executed by querying suitable publishers and combining the results of those local queries. If the global query selects tuples from a single relation the execution is relatively easy: an agent poses selection queries over a collection of publishers and merge the answer streams. This process can be described by a simple kind of query plan. Note that the merging can produce duplicate tuples in the result stream.

Definition 1 (Query Plan). *A plan for a query $Q = \sigma_C(r)$ is an expression*

$$\sigma_{C_1}(P_1) \uplus \cdots \uplus \sigma_{C_m}(P_m) \tag{1}$$

where $P_1, \ldots, P_m$ are stream publishers, each described by a view $\sigma_{D_i}(r)$.

We say that a *plan* of the form (1) is *sound* or *complete* if the answer stream resulting from the plan is sound or complete w.r.t. query Q, provided the publishers $P_1, \ldots, P_n$ are sound or complete w.r.t. their views $D_1, \ldots, D_n$. Similarly, we say that a plan is *duplicate free* or *weakly orderered* if the answer stream inherits this property whenever the P_i have it. Our next goal is to identify sufficient or, if possible equivalent, formal criteria for each of the four properties.

Soundness. A plan is sound if its output always consists of tuples that answer the query. A *sufficient* criterion for this is that for each $i \in 1..m$, the condition on P_i in the query plan and the condition D_i in the view describing P_i together entail the query condition, i.e.

$$C_i \wedge D_i \models C. \tag{2}$$

Completeness. A plan is complete if every answer to the query can be obtained by executing the plan. More precisely, if t is a tuple satisfying the query condition and S is a stream producer that possibly outputs t, then there is publisher P_i in the plan such that t satisfies C_i and D_i, the view condition of P_i. Logically, this is *equivalent* to the fact that for every stream producer S with view $\sigma_E(r)$ we have

$$C \wedge E \models \bigvee_{i=1}^{m} C_i \wedge D_i. \tag{3}$$

Duplicate Free Plans. A plan as in Definition 1 is certainly duplicate free if two components cannot produce the same tuple, that is, if any two components are disjoint. This means that the conditions

$$(C_i \land D_i) \land (C_j \land D_j) \tag{4}$$

are unsatisfiable for all $i, j \in 1..m$ with $i \neq j$.

Weakly Ordered Plans. If two or more publishers in a plan contribute to the same channel then the order of tuples in a channel can be disturbed due to the merging of their streams. However, the resulting stream of a plan is weakly ordered if the publishers in a plan are weakly ordered and if tuples for a given channel come always from the same publisher.

The latter requirement can be formalised. We write the query condition C as $C(x, y)$, where x stands for the vector of key attributes of r, which identifies a channel, and y for the non-key attributes, including timestamp. Similarly, we write C_i and D_i as $C_i(x, y)$ and $D_i(x, y)$ and abbreviate the conjunction $C_i(x, y) \land D_i(x, y)$ as $F_i(x, y)$. Then publisher P_i contributes *all* values y on channel x to the plan, if it contributes *some* values y, provided the following entailment holds:

$$\exists y. (C(x, y) \land F_i(x, y)) \models \forall y. (C(x, y) \rightarrow F_i(x, y)) \tag{5}$$

A plan with this property is called *faithful*. Clearly, a faithful plan is weakly ordered.

Because of the universal quantifier, the entailment (5) is difficult to check in general. However, it can be simplified considerably if, as in R-GMA, conditions on key and on non-key attributes are decoupled, that is, if every condition $C(x, y)$ can be written equivalently as $C^k(x) \land C^v(y)$ (and C_i, D_i, F_i analogously). The following proposition can easily be verified using straightforward transformations of logical formulas.

Proposition 1. *Suppose* $C(x, y) \equiv C^k(x) \land C^v(y)$ *and* $F_i(x, y) \equiv F_i^k(x) \land F_i^v(y)$. *Then*

$$\exists y. (C(x, y) \land F_i(x, y)) \models \forall y. (C(x, y) \rightarrow F_i(x, y))$$

holds if and only if one of the following holds:

1. $C^k(x) \land F_i^k(x)$ *is unsatisfiable;*
2. $C^v(y) \land F_i^v(y)$ *is unsatisfiable;*
3. $C^v(y) \models F_i^v(y)$.

In summary, all plans run by R-GMA have to satisfy four *essential* properties: they have to be sound and complete with respect to the query, as well as duplicate free and weakly ordered. A plan is guaranteed to have these essential properties if it satisfies the properties specified in Formulas (2), (3), (4), and (5).

A plan without the essential four properties above would be outright useless. Other properties are less crucial, but make plans more useful. For instance, a

plan should not have a disjunct $\sigma_{C_i}(r)$ that can never contribute an answer, i.e. a disjunct where $C_i \wedge D_i$ is unsatisfiable. More generally, a plan should be *irreducible*, in the sense that it is impossible to create a correct plan with a subset of the publishers involved.

5.3 Query Planning

To create a plan for a query, R-GMA goes through two phases. First, it retrieves a set of candidate publishers from the registry, each of which is maximally general w.r.t. the query. Then, it chooses a subset of the candidates and decides which local query to pose over each of them.

The registry is responsible for the first task, while the consumer (or republisher) agent takes over the second task. A reason why the second task is not left to the registry is that the agent actually contacts the publishers and it may need to modify its plan if some publishers cannot be reached.

Our algorithm works for selection queries and views where the conditions on key and non-key attributes are decoupled, that is, conditions have the form $C^k(x) \wedge C^v(y)$. The algorithm can be extended to the case of disjunctions $\bigvee_i C_i(x,y)$, where each disjunct $C_i(x,y)$ decouples the conditions on keys and non-keys. If no confusion can arise, we drop the arguments x, y of conditions.

Subsumption of Publishers. We want to define when one publisher is more general than another one with regard to a given query. If P and P' are publishers with views $\sigma_D(r)$ and $\sigma_{D'}(r)$, respectively, and $Q = \sigma_C(r)$ is a query, then we say that P is *subsumed* by P' w.r.t. Q and write $P \preceq_Q P'$ if every answer for Q that satisfies the view of P also satisfies the view of P'. Clearly, P is subsumed by P' w.r.t. to Q iff

$$C \wedge D \models D'. \qquad (6)$$

We say that P is *strictly subsumed* by P' w.r.t. Q and write $P \prec_Q P'$ if P is subsumed by P' but not vice versa. We say that P is *subsumed* by P' and write $P \preceq P'$ if P is subsumed by P' w.r.t. the universal query $\sigma_{true}(r)$ over r.

Note that the definition of subsumption does not actually refer to the information currently available, since it is only based on the views describing the publishers. It may well be the case that a republisher R' strictly subsumes another republisher R, but in a given environment does not publish any more tuples than R because there are no stream producers generating tuples that R would publish but not R'.

Relevant Publishers. The first step in query planning is to identify publishers that can possibly contribute to a plan for Q. We consider a publisher P with view $\sigma_{D^k \wedge D^v}(r)$ and a query $\sigma_{C^k \wedge C^v}(r)$. Then P can potentially contribute to a plan for Q if its view condition satisfies the following two properties:

1. $C^k \wedge D^k$ is satisfiable (Channel Consistency);
2. $C^v \models D^v$ (Value Entailment).

The first property states that P offers *at least one* channel that is requested by Q, while the second states that *all* values requested by Q are offered by P. We call a publisher with these two properties *relevant* to Q.

For a given query Q, the registry retrieves all relevant publishers. We note that for restricted classes of conditions relevance can be checked efficiently (see section 6).

Maximal Relevant Publishers. The second step is to single out those publishers among the relevant ones that, with regard to Q, are not strictly subsumed by another relevant publisher. We call such a publisher *maximal.*

We note that due to the value entailment property of relevant publishers, the subsumption test involves only the key attributes. More precisely, for any relevant publishers P_1, P_2 with conditions D_1, D_2 we have that $P_1 \preceq_Q P_2$ if and only if $C^k \wedge D_1^k \models D_2^k$. Again, for sufficiently simple conditions, this can be checked efficiently.

It may be that P_1, P_2 are both maximal and that $P_1 \preceq_Q P_2$. However, in this case we also have $P_2 \preceq_Q P_1$. Obviously, "$\preceq_Q$" is an equivalence relation on the set of maximal publishers. As far as answers to Q are concerned, there is no reason to prefer one of P_1, P_2 to the other for inclusion in a query plan.

To break the tie between equivalent maximal publishers, we apply a heuristic. It is possible that P_1, P_2 subsume each other w.r.t. Q, but P_2 publishes for a more general view than P_1, that is $P_1 \prec P_2$. In such a case, we prefer to contact P_1 rather than P_2, since it is more likely that P_2 will be relevant for another query than P_1.

In summary, when the registry is contacted by a consumer agent with a query Q, then it returns

– all maximal relevant publishers, grouped into equivalence classes w.r.t. "$\preceq_Q$",
– where each equivalence class is topologically sorted w.r.t. "$\preceq$", that is, if P_1 precedes P_2 then it is not the case that $P_2 \prec P_1$.

Constructing Query Plans. The consumer agent receives from the registry the maximal relevant publishers for its query and constructs a plan of the form (1). If among them there are equivalent publishers, then the agent has a choice as to which one to use in its plan. It chooses one publisher from each equivalence class, giving preference to those which are least general w.r.t. subsumption, thus obtaining a sequence of publishers $\langle P_1, \ldots, P_m \rangle$.

We call a sequence of publishers $\langle P_1, \ldots, P_m \rangle$, obtained by choosing one representative from each class of maximal relevant publishers a *supplier sequence.* Suppose $\langle P_1, \ldots, P_m \rangle$ is a supplier sequence where each P_i is described by $\sigma_{D_i}(r)$. We define the *canonical plan* for the sequence as

$$\sigma_{C_1}(P_1) \uplus \cdots \uplus \sigma_{C_m}(P_m) \tag{7}$$

where $C_1 = C$ and $C_i = C \wedge \neg D_1 \wedge \cdots \wedge \neg D_{i-1}$.

Proposition 2 (Canonical Plans). *If $\langle P_1, \ldots, P_m \rangle$ is a supplier sequence for query Q, then the canonical plan for $\langle P_1, \ldots, P_m \rangle$ is sound and complete for Q, and it is duplicate free and weakly ordered.*

Proof. (Sketch) The plan (7) is sound because the condition C_i of each component contains the query condition C as a conjunct. The plan is complete because every stream producer that can contribute a channel is relevant and, due to the construction of a supplier sequence, for each relevant stream producer S the sequence contains a publisher P_i such that $S \preceq_Q P_i$. The plan is duplicate free because all components are disjoint. The plan is weakly ordered because it is faithful. It is faithful because all participating publishers satisfy the definition of relevant publishers.

The canonical plan (7) can often be simplified considerably. Firstly, the conjuncts $\neg D_j$ can be replaced with $\neg D_j^k$, the negated condition on the key attributes. Secondly, the conjunct $\neg D_j$ can be dropped altogether if $D_i \wedge D_j$ is unsatisfiable, which is often the case in practice, e.g. if both publishers are stream producers. For the limited types of conditions that are used in the current implementation of R-GMA (see section 6) this test can be performed efficiently.

Finally, we like to point out that the query planning algorithm can still be used for classes of conditions where entailment and unsatisfiability tests are intractable. In such a case it is sufficient to use checkers that are possibly incomplete, but sound, that is, checkers that recognise some cases of entailment and unsatisfiability, but not all of them. Proposition 2 would still hold in such a case because only the positive outcome of such a test allows us to replace a stream producer by a republisher in a query plan. If the outcome is negative, the stream producer will be used. Thus, incomplete checkers would only forgo opportunities for optimisation.

Irreducible Plans. In general, canonical plans are not irreducible. In fact, in a supplier sequence there is no publisher P_0 that is subsumed (w.r.t. the query) by another publisher P_1. However, it may be the case that P_1 together with another publisher P_2 can contribute the same tuples as P_0.

Example 1. Suppose R_0, R_1, R_2 are republishers for the throughput relation tp, where R_0 republishes measurements for traffic from Heriot-Watt University, R_1 republishes measurements for packets of size at most 10, and R_2 for packets of size at least 10. More formally, each R_i is described by a view $\sigma_{D_i}(\mathsf{tp})$, where $D_0 = (\mathsf{from} = \texttt{'hw'})$, $D_1 = (\mathsf{psize} \leq 10)$, and $D_2 = (\mathsf{psize} \geq 10)$.

Consider the query $Q = \sigma_{true}(\mathsf{tp})$ and suppose R_0, R_1, R_2 are all the publishers available. Then all three republishers are maximal w.r.t. "$\preceq_Q$". But this set of republishers is not irreducible because R_1 and R_2 together "subsume" R_0.

We have proved that irreducibility of plans is already NP-hard if conditions are conjunctions of comparisons between attributes and values, i.e. comparisons of the form "*attr* $\{\geq, \leq\}$ *val*", as in the example above. However, if the selection

conditions can be expressed in Horn logic, one can show that a set of publishers is already irreducible provided no publisher subsumes another one. This is the case, for instance, if conditions are conjunctions of the form "$attr \{=,<\}$ *val*" or "$attr \{=,>\}$ *val*", where all comparisons have the same orientation.

It seems to us for this reason that in most practical cases an irreducibility test would not yield any improvement over the simpler subsumption check, which is the reason why R-GMA does not perform such a test.

6 R-GMA Implementation

We describe here the current implementation of R-GMA as the DataGrid project enters its final year. An implementation of the full system outlined in sections 4 and 5 is planned for the end of the project.

6.1 Overall Approach

R-GMA offers APIs in several programming languages: C, C++, Java, Perl, and Python. These support the roles described in section 4, e.g. a stream producer. A method invocation on an API object results in a request being sent to an agent object, running in a web server. This then acts on behalf of the component using the API.

The system presents the illusion of have a single registy and schema. However, these components are replicated to meet the resilience requirement (section 2.2). Registry replication is based on the gossip architecture: all copies of the registry swap messages periodically to keep each other up to date. A component can thus interact with any copy of the registry as if it were the only registry, and this ensures that the load is shared across registry instances.

The gossip architecture cannot be used for the schema because otherwise two users who simultaneously access the schema could introduce conflicting relations. Instead, a master-slave model is used: one of the schema instances is elected the leader and only the leader can allow the creation of a new relation. The other schema instances act as backups, but can also answer queries about relations.

6.2 R-GMA Components

Consumers. The consumer API allows users to pose continuous, latest-state and history queries in SQL. Generally, global queries may be posed over just one table of the schema, and are restricted to select-project queries. However, arbitrary one-time SQL queries can be answered if a republisher for all the relations of the query can be located. Results are *streamed* to the agent and merged so that quick partial answers can be returned, say, to a resource broker.

Producers. An ideal R-GMA system would offer two types of producer, stream and database producers. Our implementation only offers stream producers, and currently static information is published as streams. This was found to be the quickest and simplest way of providing minimal functionality for DataGrid.

Republishers. At present, our implementation only supports streams so it only offers two types of republisher: stream republishers and archivers. A hierarchy of stream republishers can be set up manually to merge streams together. Work is underway to automate this by implementing the algorithms presented in section 5.

Registry. Publishers may only register views over single relations with conditions that are conjunctions of equalities of the form "$attr = val$". With this simplification, the satisfiability and entailment tests of section 5 can be expressed as queries over the registry's database. So far, the entailment tests are trivial, as only republishers of entire relations are considered at the moment for query planning.

Schema. Built into R-GMA are a set of core relations known as the GLUE schema [4]. These relations were defined by a number of Grid projects, including DataGrid, and describe the components of a Grid, such as computing elements. However, publishers may also introduce new relations into the global schema.

Our implementation only supports a restricted set of types: Varchar, Real and Int. However, this simplifies the query planning task.

6.3 Performance of the Current R-GMA

Our limited implementation of R-GMA is currently undergoing resilience and performance testing on testbeds within DataGrid. An early result is that a consumer is able to merge data published every 30 seconds from up to 40 typical sites (one SE, three CEs). The performance is likely to benefit from republisher hierarchies.

7 Conclusions

We conclude with a brief discussion about how far R-GMA meets the requirements outlined at the beginning.

R-GMA allows for the publication of streaming and static data, although the current implementation only offers stream producers. The schema provides a global view of all the data available in the system. The registry can locate data sources that would be of interest to a user. It does so by matching a query that is posed against the global schema with suitable publishers. Scalability is provided through the replication of the registry and schema components, and the merging of small streams of data by hierarchies of stream republishers.

We have not yet addressed the issue of security. An approach using views describing which data a user is authorised to access may be appropriate.

Although R-GMA has been designed for monitoring the components of a computational Grid, it is a general architecture and could be used for other applications that require querying distributed pools and streams of data.

References

1. The DataGrid Project. http://www.eu-datagrid.org, June 2003.
2. DataGrid WP3 Information and Monitoring Services. http://hepunx.rl.ac.uk/edg/wp3/, June 2003.
3. Global grid forum. http://www.ggf.org, June 2003.
4. High energy nuclear physics intergrid collaboration board. http://hicb.org/, June 2003.
5. B. Babcock, Sh. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS-21*, pages 1–16, 2002.
6. F. Berman. From TeraGrid to Knowledge Grid. *Communications of the ACM*, 44(11):27–28, 2001.
7. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB-28*, pages 215–226, 2002.
8. K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *HPDC-10*, 2001.
9. L. Dutka and J. Kitowski. Application of component-expert technology for selection of data-handlers in CrossGrid. In *Proc. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *LNCS*, pages 25–32. Springer, 2002.
10. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2: Computational Grids, pages 15–51. Morgan Kaufmann, 1999.
11. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
12. Globus Toolkit. http://www.globus.org, June 2003.
13. W.E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: the engineering aspects of NASA's Information Power Grid. In *HPDC-11*, pages 197–204. IEEE, 1999.
14. B. Plale, P. Dinda, and G. von Laszewski. Key concepts and services of a Grid information service. In *ICSA PDCS-15*, 2002.
15. R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: adaptive control of distributed applications. In *HPDC-7*, pages 172–179, 1998.
16. C. Shahabi. AIMS: an Immersidata management system. In *CIDR-2003*, 2003.
17. W. Smith. A system for monitoring and management of computational Grids. In *ICPP-31*, pages 55–, 2002.
18. M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB-22*, page 594, 1996.
19. B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A Grid monitoring architecture. Global Grid Forum Performance Working Group, March 2000. Revised January 2002.
20. J.D. Ullman. Information integration using logical views. In *ICDT-6*, volume 1186 of *LNCS*, pages 19–40. Springer, 1997.