GridRM: An Extensible Resource Monitoring System

Mark Baker and Garry Smith

{mark.baker, garry.smith} @computer.org

Distributed Systems Group,

University of Portsmouth, United Kingdom

June 2003.

Abstract

GridRM is an open and extensible resource monitoring system, based on the Global Grid Forum's Grid Monitoring Architecture (GMA). GridRM is not intended to interact with applications; rather it is designed to monitor the resources that an application may use. This paper focuses on the pluggable driver infrastructure used by GridRM to interact with heterogeneous data sources, such as local SNMP or Ganglia agents, and how it provides a homogeneous view of the underlying heterogeneous data. This paper discusses the local infrastructure and details early work on implementing and deploying a number of drivers.

1   Introduction

In any wide-area distributed system there is a need to communicate and interact with networked devices and services ranging from computer-based ones (CPU, memory and disk), to network components (hubs, routers, gateways) and specialised data sources (embedded devices, sensors, data-feeds). While many of these resources use the Internet protocols for underlying communication, a diverse range of application level protocols are used to provide resource specific interaction. For example, information about the state of a computer can be provided by a large number of different agents such as SNMP (v1, 2, 3) [1], the Network

1

Weather Service [2], Ganglia [3], SCMS [4], Net Logger [5], and many others. Furthermore, not only do agent types differ in the way in which they communicate, but also by the types and format of the data they produce.

The heterogeneous data that can be collected from these data sources can only be truly useful if it can be normalised, so that a homogeneous view of the data can be produced. This transformation of raw data into marked-up metadata creates the possibility of equipping the data with semantic meaning, that can then be used by a range of high-level tools for tasks such as intelligent system monitoring, scheduling, load-balancing, and task-migration.

1.1   A Framework for Data Harvesting

GridRM [6] is an open source, generic system that is designed for monitoring resources distributed over a wide-area. GridRM's overall aim is to harvest resource data and provide it to a variety of clients in a form that is useful for their individual requirements. GridRM's objectives [6] include:

- ??   The provision of a generic resource monitoring framework for data harvesting,
- ??   The design of an scalable and fault-tolerant architecture that enables efficient monitoring across globally distributed resources,
- ??   Seamless and transparent client access to information,
- ??   The translation of heterogeneous data into common homogeneous view,
- ??   Multi-level and granularity of security for data access.

GridRM Gateways [7] are used to coordinate the management and monitoring of resources at each Grid site. This includes the controlled access to real-time and historical data harvested from local resources.
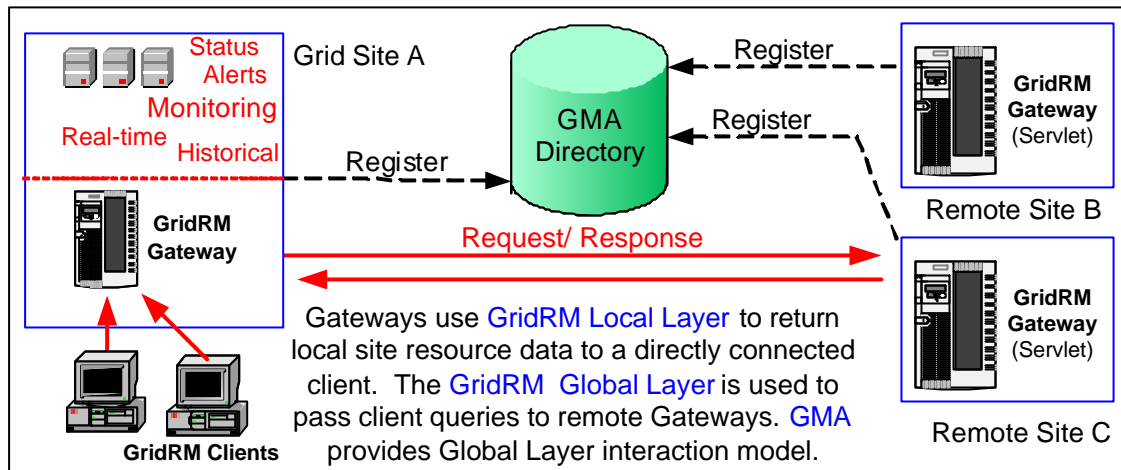
Figure 1: The GridRM Architecture showing Global and Local Layers

GridRM consists of two layers, a Global and Local layer (see Figure 1). The Global layer, which provides inter Grid site, or Virtual Organisation, interaction is based on the Global Grid Forum's (GGF) Grid Monitoring Architecture (GMA) [8]. GridRM gateways collaborate to provide an efficient and consistent view of the available resource data. At the Local layer, a GridRM gateway provides an access point to local resource data within its local control. Clients are free to connect to any Gateway; requests for remote resource data are routed through to the Global layer for processing by the gateway that owns the required data.

This paper discusses the Local layer of the GridRM. In section 2, we briefly introduce the extensible management architecture used within a Gateway to provide pluggable information harvesting. In section 3, we describe how the pluggable monitoring infrastructure has been implemented and provide practical experiences from the implementation and deployment of a number of data drivers. In section 4 we discuss driver management issues for GridRM clients. Finally, Section 5 concludes with a summary and details of on going and near future work.

2   The Extensible Management Architecture

GridRM provides an extensible approach to resource monitoring through the use of client abstraction and data source plug-ins (see Figure 2). The Abstract Client Interface Layer (ACIL) provides a clear separation between client specific APIs and the data model used within GridRM.

The Coarse Grained Security Layer (CGSL) provides controlled access to management data; each Gateway is responsible for the security of the resources it controls and interacts with. In a hierarchy of GridRM Gateways, security decisions can be deferred to the local Gateway responsible for a given resource

The Gateway's Local layer consists of modules for querying data sources, event handling, and storing/retrieving historical data. Request Handling modules coordinate the retrieval of data from not only local resources, but also resources controlled by remote GridRM Gateways. Naming schema modules are used to transform data retrieved from heterogeneous sources into a format that complies with a central naming schema, thereby providing a homogeneous view of the monitored data.
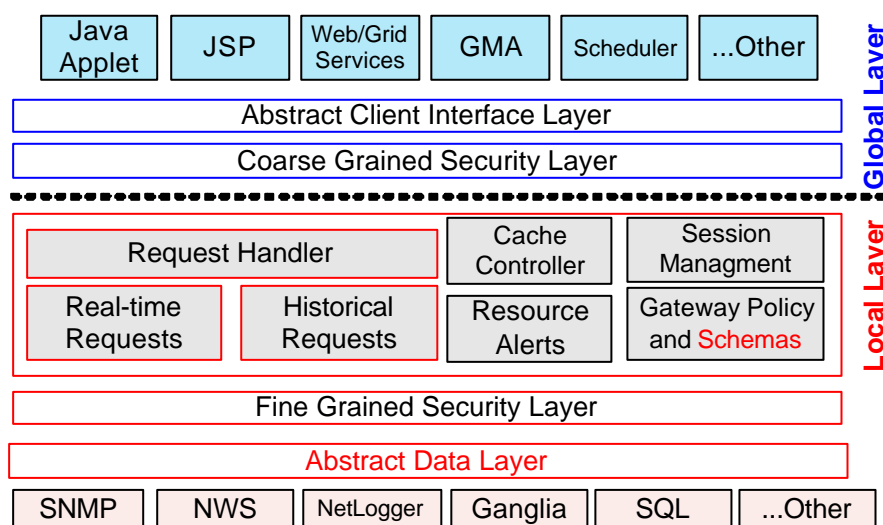

Figure 2: The Extensible Management Architecture

The Abstract Data Layer (ADL) provides an abstraction for the underlying mechanisms used to retrieve and normalise heterogeneous resource data. Data source driver plug-ins are used to extend the resource gathering capabilities of the architecture in a generic way, thereby allowing any type of data source to be incorporated into the GridRM framework. Plug-ins are dynamic, drivers can be added or removed at runtime without affecting normal Gateway operation.

3   Local Layer Implementation

The Structured Query Language (SQL) [9] is used extensively throughout GridRM. Queries for resource data are submitted (Figure 3) as SQL statements and pass down to the data source drivers in the same format. The correct driver to be used for a given data source can be selected automatically at runtime, thus providing a dynamic interaction with underlying resources. The drivers, which are implemented using the Java JDBC API [9], are passed a query, and in response, return a standard Java SQL object (a `javax.sql.ResultSet`). The approach used in GridRM, is simple and standard, yet powerful and expressive due to the nature of SQL. The core of a gateway's functionality benefits from the simplicity of the approach used, where there are `String` queries in, and `ResultSets` out. All low-level protocol details for a resource are contained within the resource's plug-in driver. The driver is responsible for the conversion between SQL, and the native protocol, and populating a JDBC `ResultSet` with the returning resource data.

JDBC was selected due to a number of important characteristics:

   ??   The JDBC goal is uniform access to a wide range of relational databases,

   ??   JDBC is a widely used de facto standard,

   ??   JDBC provides a set of classes for Java with a standard SQL database access interface,

5

?? JDBC provides an API for database "drivers" to make actual connections and transactions to database products,

?? JDBC is a "low-level" interface; SQL commands are passed directly to it. The API provides a base for higher-level interfaces.
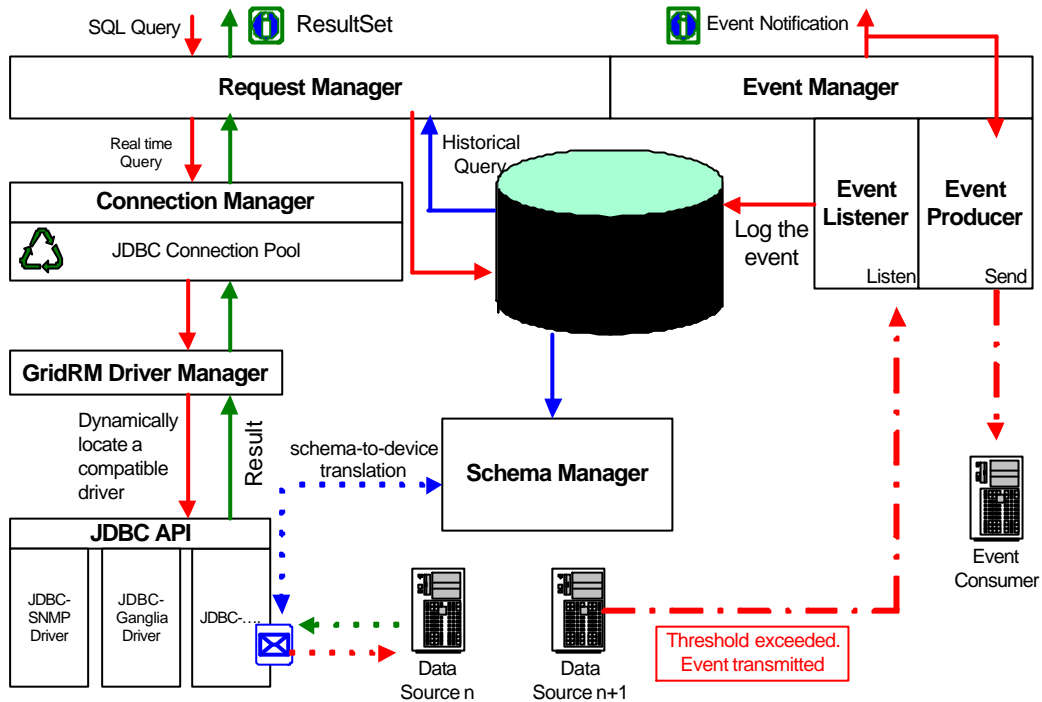

Figure 3: The path of a query for resource data within the local Gateway

3.1   The Main Components

This section outlines briefly the main components that make up the local GridRM layer, as shown in Figure 3. The following components operate in a layered fashion where each level uses the services of the level directly below.

3.1.1   The Request Manager

SQL requests are received from the Abstract Client Interface Layer (ACIL), the queries are processed and the results returned to the ACIL. The `RequestManager` coordinates queries across multiple data sources and consolidates results.   Furthermore, the manager is responsible for executing queries that span real-time resource requests and historical (or

cached) data. The `RequestManager` uses the `ConnectionManager` to execute real-time queries, while historical data is retrieved from the Gateway's internal database.

3.1.2 The Connection Manager

The `ConnectionManager` receives SQL queries from the `RequestManager`. A connection object is used to execute a query against a resource driver. Driver connections typically incur an overhead when a data source is first connected, especially if drivers are dynamically mapped to the data source. Therefore the `ConnectionManager` provides pooling of driver connections to reduce the overhead effects. The `ConnectionManager` calls the `GridRMDriverManager` to return a new connection if a suitable pooled instance does not exist. All new connections are registered with the connection pool before use.

3.1.3 The GridRM Driver Manager

The `GridRMDriverManager` registers and un-registers resource drivers, and performs driver-to-resource allocation. Driver connection requests are received from the `ConnectionManager`, an appropriate driver is selected, and the `Connection` returned. Drivers can be selected either:

- ?? Statically: using driver preferences registered in advance by the user,
- ?? Dynamically: when the `GridRMDriverManager` selects a compatible driver at runtime to execute the request.

For performance, the `GridRMDriverManager` maintains a cache containing details of the driver last successfully used for a data source. Configuration rules determine the actions that should occur, if a cached driver reference is no longer valid. For example retry the driver, try another, report the error.

3.1.4 The Naming Schema Manager

GridRM uses a common naming schema, from the Grid Laboratory Uniform Environment (GLUE) [11], to provide a consistent view of data across heterogeneous resources. The

`SchemaManager` provides mapping and translation services for data source drivers. The GLUE schema [12] provides minimum, common, conceptual schemas to allow interoperability between Grid implementations for resource monitoring and discovery. GLUE defines consistent naming and structure for resource attributes and relationships, for example, Compute Elements (CE), Storage Elements (SE) and Networks Elements (NE), independently of implementation details. The GLUE Schema is the emerging de facto standard, with support from many EU and US projects, including iVDGL [13], GriPhyN [14], DataTag [15], EDG [16], and PPDG [17]. Currently a number of GLUE implementations are underway, including relational [18], XML [19] and LDAP [20] versions.
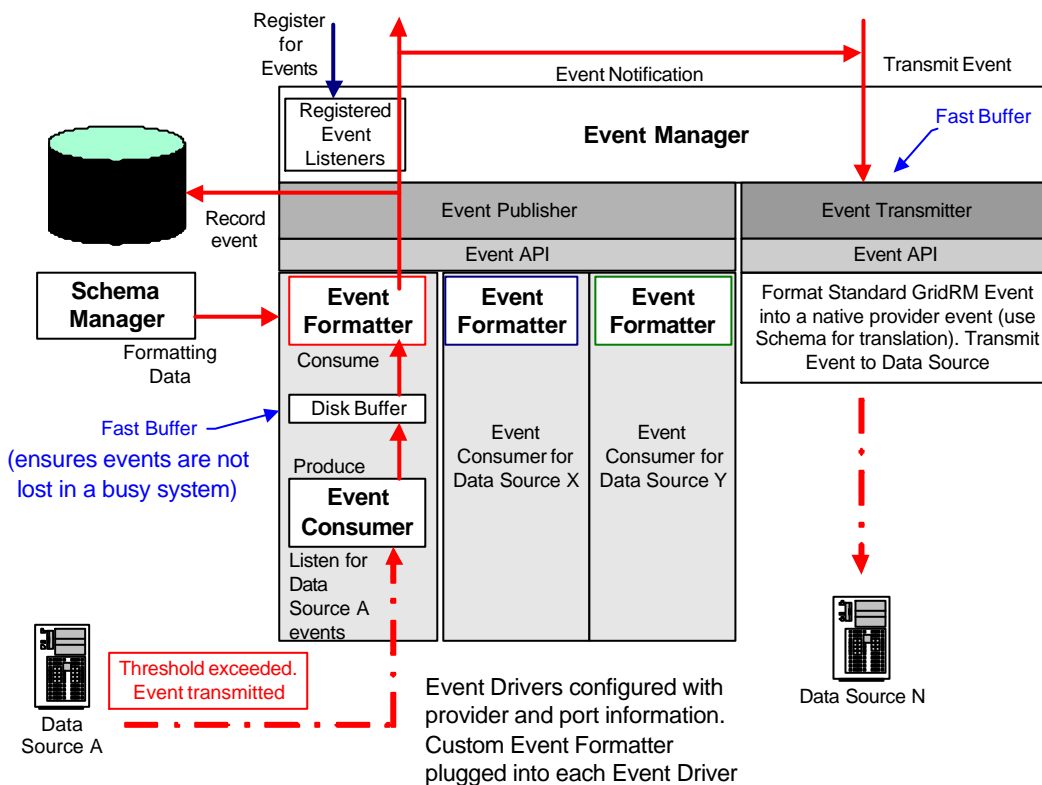
3.1.5 The Event Manager

Figure 4: The Event Manager Architecture

The `EventManager` provides a bridge between the native events issued by data sources and GridRM. Event drivers are used to receive native events and translate them into a standard format used by GridRM. Incoming events are recorded for historical analysis and forwarded

to all components that registered interest in the particular event. In addition to receiving events, the `EventManager` can pass events back out to data sources as required. The GridRM internal event format is translated to the data source's native format. This behaviour allows GridRM to propagate events between Gateways and groups of diverse data sources.

3.2 The GridRM Driver Infrastructure

A key element of GridRM is the driver layer for interacting with data sources. The drivers are modular plug-ins that can be installed or removed at runtime. GridRM can be extended to work with any number of data sources, all communicating via native protocols and supplying data in a variety of formats and forms. In order to create efficient and effective drivers, GridRM includes guidelines for driver development and implementation.

3.2.1 Driver Implementation

GridRM drivers are based on the JDBC API. JDBC provides a common interface to query and update relational data sources in Java using SQL syntax. Although the standard JDBC 3.0 API [21] is relatively small, comprising of 18 interfaces and 7 classes, there are quite a number of methods that must be implemented in order to meet the API's interface requirements. For example, the `ResultSet` object has 139 methods mainly concerned with returning field data as a prescribed Java data type. Furthermore, the `DatabaseMetaData` class has 165 methods. Although this example is extreme (the main `Driver` class has only 6 methods and the `DriverManager`, 14) it does highlight an important issue for driver development. In order to implement a driver with adequate functionality, potentially only a small subset of the overall JDBC API requires implementing. To permit an incremental approach to driver development, the JDBC API interfaces were implemented to return `nulls` or throw `SQLExceptions`. The resulting classes are then used as super-classes for driver implementations. For example, if a call is made to a `ResultSet` method that is not implemented, an `SQLException` is thrown, as one would expect from a fully implemented driver that had experienced errors while

attempting to retrieve the required data. At a later stage, if necessary, further methods can be implemented in a manner that allows drivers to be incrementally extended to meet the needs of the environment.

To create a minimal driver, a subset of the methods require implementing in the following classes/interfaces:

- ?? `java.sql.Driver,` determines if driver is capable of operating with the specified data source,
- ?? `java.sql.Connection,` creates a session with the data source and initialise schema settings for the session,
- ?? `java.sql.Statement,` translation of SQL queries and submission to data source,
- ?? `java.sql.ResultSet,` the results from a SQL query,
- ?? `java.sql.ResultSetMetaData,` data describing how to access the returned result fields.

In addition, the following functionality is required, typically implemented in separate classes within the driver:

- ?? A class to parse the SQL query strings, this is supplied as part of a GridRM driver development API,
- ?? A class to perform mapping of data requests to the data source, based on the presence of a particular naming schema (e.g. GLUE). This class is required to interact with the `SchemaManager,`
- ?? Code to interact with the data source agent, using the agent's native protocols,
- ?? Code to translate result data into the format required by GLUE.

3.2.2 Querying a GridRM Data Source

Before a GridRM Gateway can use a driver, it must be locally registered. Driver registration can occur:

?? When the Gateway is started up,

?? During runtime, when a GridRM client presents a driver for registration.

Upon start-up, the GridRM Gateway registers a number of drivers that come as default with the site. In addition, any drivers previously presented during runtime are also registered; registration details are cached persistently within the Gateway. Section 4 describes a Java Server Pages interface to the GridRM Gateway, that allows users to manage drivers. Regardless of the way drivers are registered, the `GridRMDriverManager` handles all registration requests. The code shown in Table 1, demonstrates the driver registration process. It should be noted that any driver implementing the `java.sql.Driver` interface could be registered. The registration component remains generic by avoiding any direct reference to the driver's actual class name. The driver package and class names are determined when the driver's JAR file is first submitted to the Gateway. This information is subsequently cached for future registrations.

```
//DriverMetaData driver
Class driverClass = Class.forName(driver.getDriverName());
DriverManager.registerDriver((Driver) driverClass.newInstance());
```

Table 1: The GridRMDriverManager: Driver registration

When a client issues a SQL request, the request consists of two parts, the network address of the data source and the query to be executed. Under normal circumstances the Gateway will know the correct driver to use with the data source. However, in some cases the appropriate driver will not be known a priori, in which case the Gateway must dynamically locate a driver that is suitable for that data source. Figure 5 shows the sequence of events that occur when a query is received for a given data source.
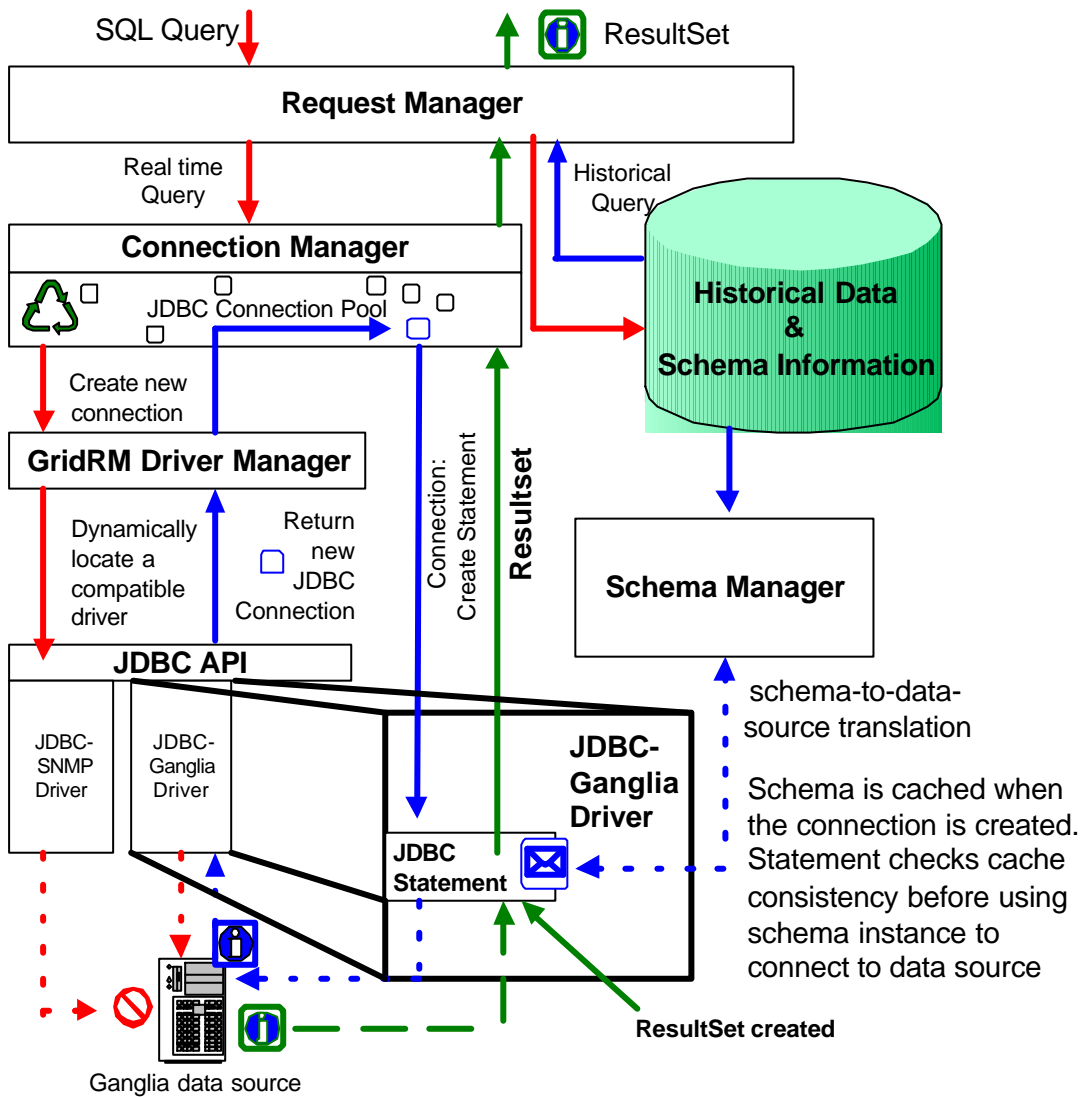
SQL Query ↓   ↑ 🔵 ResultSet

Request Manager

Real time
Query ↓   Historical
                                           Query

Connection Manager

♻ □   □   □ □ □
     JDBC Connection Pool   □   □
            □

Create new
connection ↓

GridRM Driver Manager

Dynamically   Return
locate a   □ new
compatible   JDBC
driver   Connection

JDBC API

| JDBC-SNMP Driver | JDBC-Ganglia Driver | | Connection: Create Statement | Resultset | **JDBC-Ganglia Driver** |

Historical Data & Schema Information

Schema Manager

schema-to-data-source translation

Schema is cached when
the connection is created.
Statement checks cache
consistency before using
schema instance to
connect to data source

JDBC Statement ✉

🔵

🚫

🔵

ResultSet created

Ganglia data source

Figure 5: The sequence of events that dynamically locate a GridRM data source

If the `ConectionManager`'s cache does not contain a `Connection` to the desired data source, the `GridRMDriverManager` is called to locate an appropriate driver and return the `Connection` object.

```
//locate the driver based on the URL string
String urlString = request.getURLString();
Enumeration driverEnum = DriverManager.getDrivers();
//Iterate the list of drivers. The first that returns true to
//acceptsURL() is returned as the driver to use for this request
Boolean match=false;
Driver current = null;
While driverEnum.hasMoreElements() && ! match){
  current = (Driver) driverEnum.nextElement();
  // Have we found a driver that supports the URL AND can connect to
  // the data source?
  match = current.acceptsURL(urlString);
}
//Obtain a Connection object from this driver.
```

Table 2: The code for dynamically locating a driver

In some cases a data source may support multiple protocols, in which case the client can help to guide the driver selection process by including protocol information within the JDBC URL. For example in order to use the first available driver the JDBC URL may be of the form: `jdbc:://snowboard.workgroup/PerfData`. Whereas, to specify the Network Weather Service driver, the URL would become: `jdbc:NWS://snowboard.workgroup/PerfData`.

3.2.3   Driver Data to Naming Schema Translations

GridRM normalises the data it harvests by describing the data using the GLUE naming schema. All data returned by the data source drivers should conform to GLUE and is thus abstracted away from the native form used by individual data sources. This abstraction is key to meeting GridRM's aim of providing a homogeneous view of data harvested from heterogeneous sources.

SQL is used to query data sources. GLUE logically organises data into groups. The schema prescribes the data fields for each group. The essence of a group can be directly compared to the tables of a relational database. Clients select one or more GLUE group names to query.

For example, to retrieve all data values associated with the group 'Processor', the SQL statement issued to a specific data source would be "`SELECT * FROM Processor`". The data source driver is responsible for mapping between GLUE and the actual values to be retrieved. Essentially GLUE provides the values that must be utilised by the data source's native API in order to execute the request. To perform the mapping, the driver must connect to the `SchemaManager` and retrieve metadata describing that driver's GLUE implementation.

While the mapping used provides a seamless view of data, it does not address the issue of its equivalence across data sources. In some cases, the drivers may connect to data sources that already adhere to GLUE, in which case little or no further processing would be required. However, in the general case drivers will need to translate data values, so that meaning and value correspond to the format defined by GLUE. Depending on the data source, it may not be practical or possible to translate a value to GLUE's definition. Therefore, if data required by the naming schema is unavailable directly from a data source, drivers can return `null` values, indicating a translation was either not possible or currently not implemented. As GLUE evolves, support for a wide range of resources will emerge easing any data translation issues.

3.2.4 Experiences with a range of GridRM Drivers

Experience has been gained whilst developing and deploying a number of GridRM drivers. The initial set of drivers consisted of SNMP, Ganglia, NWS, Net Logger and SCMS. This set was selected for their data representation characteristics and as they are commonly used systems. In some cases, for example SNMP and Net Logger, fine grained native requests for data are possible, with generally little or no parsing required to read the native data value into the GridRM driver. For other data sources, for example Ganglia and NWS, responses are

typically coarse grained. A greater overhead is required to parse values from the response, which is typically XML or plain text. Therefore, on a driver-by-driver basis, implementations should address these issues by, using caching policies, within the plug-in, as appropriate for the characteristics of a particular type of data source.

The main issues a driver developer must be concerned with include:

- ?? The speed at which the driver responds to a request, and the efficiency of contacting the data source and caching of data,
- ?? How to represent data within the ResultSet, including lazy or eager parsing mechanisms,
- ?? The manner in which requests for data are mapped to a common naming schema.

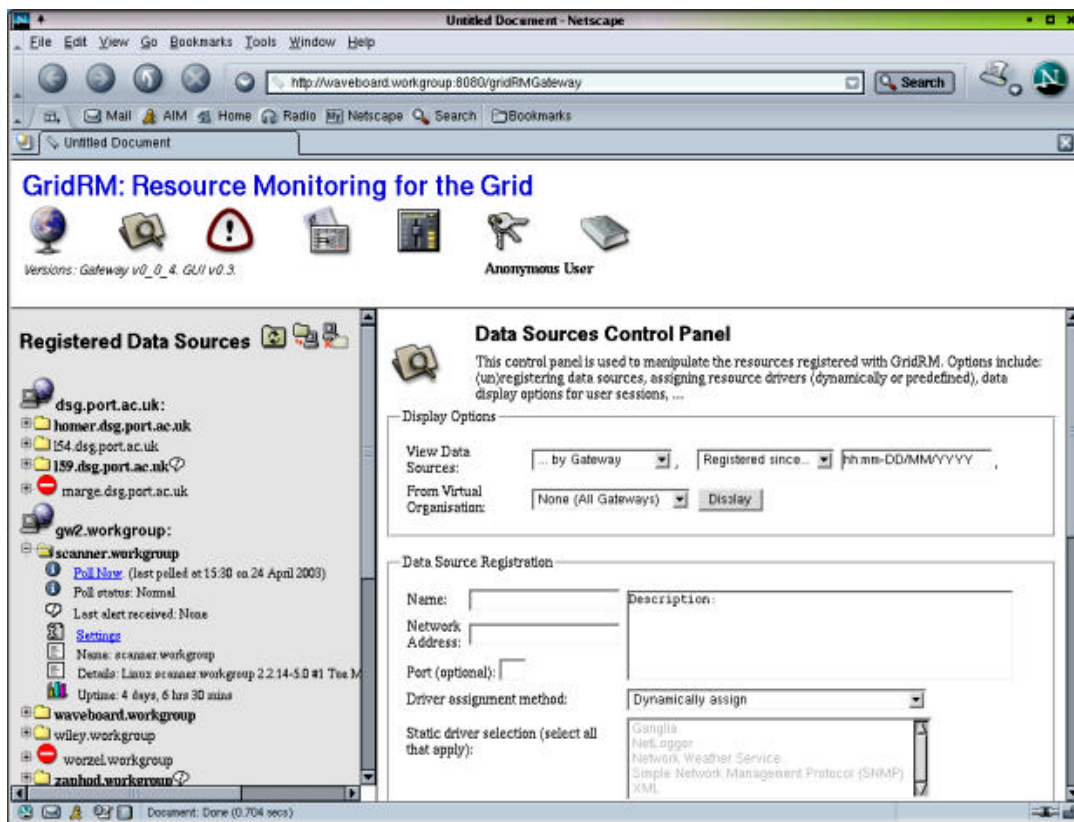## 4   Managing Drivers within GridRM

Figure 6: The JSP interface to GridRM data sources

Figure 7: The JSP navigation icons

Data sources are discovered by scanning a network, or they can be configured selectively based on a network address, or specific range of addresses. GridRM can select a driver dynamically, based on the success or failure of each query to that resource. Given a successful connection, a dynamically selected driver can be cached for direct use, the next time the resource is queried. If a subsequent query fails, GridRM resorts back to dynamic driver selection. A system user is free to specify a single driver to connect to a given data source, or register a number of drivers to be used in prioritised order. See Figure 8.

If the specified driver(s) are unable to connect to the data source for a given request, the user can determine the action that should follow, for example:

?? Provide notification of a connection failure or,

?? Retry the specified drivers for n iterations or,

?? Dynamically select a new driver from the set of registered drivers.

Figure 8: The Data Source Driver Registration Panel

The JSP tree view shown in Figure 9 is populated with cached data from queries issued within the local gateway. The cached data describes local resources, as well as any remote resource data, that was queried from the local gateway. By utilising the cache, a heavily used GridRM Gateway can return a view of the recent status of a site while limiting resource intrusion. This approach is used between gateways to increase scalability by reducing unnecessary requests. To obtain real-time data either the user must explicitly poll a given resource or refresh their tree view after other users have initiated a poll (See Figure 9).
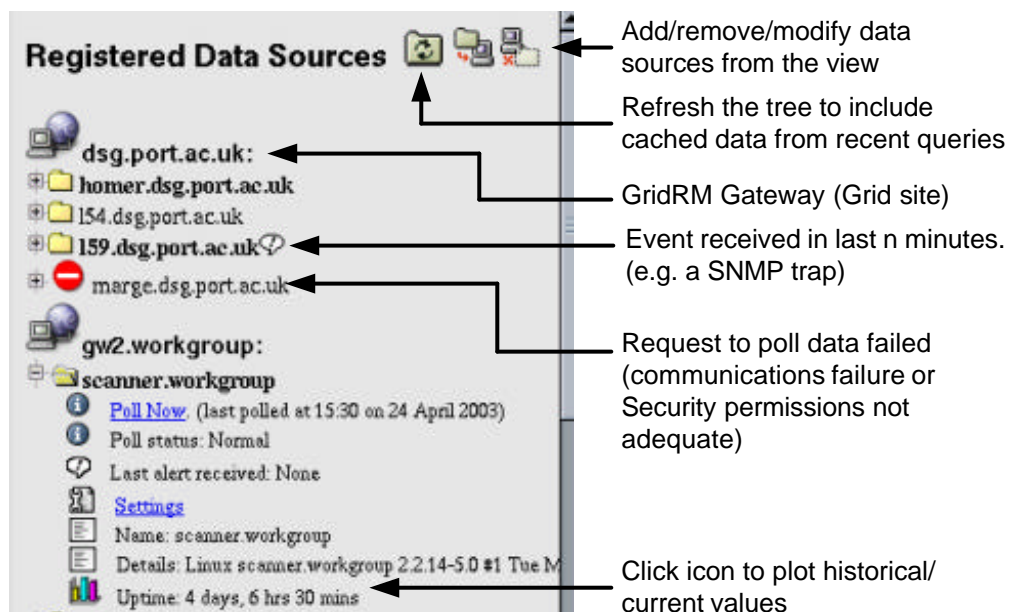

Figure 9: A View of Data Sources

5   Conclusion

GridRM is a generic open-source resource-monitoring framework that has been specifically designed for the Grid. It is designed to harvest resource data and provide it to a variety of clients in a form that is useful for their needs. GridRM is not intended to interact with applications; rather it is designed to monitor the resources that an application may use. Initial work has focused on using GridRM to harvest data for the monitoring of computer-based resources via their local agents. The main objective of the project is to produce ubiquitous and seamless mechanisms to communicate and interact with heterogeneous data sources using a

17

standardised (GMA, SQL) and extensible architecture. This paper has focused on extensible data gathering at the GridRM Local Layer where a variety of plug-in data source drivers have been implemented using the JDBC API.

5.1   Current Status and Near Future Work

Currently GridRM development is concentrated on completing the local layer of the systems architecture. This includes implementing further drivers and investigating caching and translation, and optimisation strategies. Over the summer of 2003, a first implementation of the Global Layer will take place. It is anticipated that by the fall of 2003, GridRM will be deployed across global test sites for early evaluation purposes.

6   References

[1] The Simple Network Management Protocol (SNMP), http://net-snmp.sourceforge.net/, 12th March 2003.
[2] The Network Weather Service, http://nws.cs.ucsb.edu, April 2003.
[3] Ganglia, http://ganglia.sourceforge.net/, April 2003.
[4] The Scalable Cluster Management System (SCMS), http://www.opensce.org, February 2003.
[5] NetLogger, http://www-didc.lbl.gov/NetLogger/, February 2003.
[6] GridRM: Grid Resource Monitoring, http://dsg.port.ac.uk/projects/research/grid/grid-monitoring/, April 2003.
[7] M.A. Baker and G. Smith, GridRM: A Resource Monitoring Architecture for the Grid, the proceedings of the 3d International Workshop on Grid Computing, LNCS, Springer-Verlag, pp268-273, ISBN 3-540-00133-6, November 2002. http://dsg.port.ac.uk/garry/pubs/papers/conferences/grid2002/gridrm-arch.pdf
[8] Global Grid Forum (GGF), Grid Monitoring Architecture (GMA) Working Group, http://www-didc.lbl.gov/GGF-PERF/GMA-WG/.
[9] Structured Query Language (SQL), http://www.sql.org, March 2003.
[10] JDBC, http://java.sun.com/products/jdbc/, March 2003.
[11] GLUE, http://www.hicb.org/glue/glue.htm, April 2003.
[12] GLUE-Schema, http://www.hicb.org/glue/glue-schema/schema.htm, April 2003.
[13] iVDGL, http://www.ivdgl.org, April 2003.
[14] GriPhyN, http://www.griphyn.org/index.php, April 2003.
[15] DataTag, http://datatag.web.cern.ch/datatag/, April 2003.
[16] The EU Data Grid, http://eu-datagrid.web.cern.ch/eu-datagrid/, April 2003.
[17] PPDG, http://128.3.182.66/, April 2003.
[18] The EU Data Grid, WP3, Relational Grid Monitoring Architecture (R-GMA), http://hepunx.rl.ac.uk/edg/wp3/, April 2003.
[19] Globus Project, OGSA, http://www.globus.org/ogsa/, April 2003.
[20] DataTAG, GLUE Schemas Activity, Computing Element, http://www.cnaf.infn.it/~sergio/datatag/glue/v11/CE/index.htm, April 2003.

[21] J. Ellis, L. Ho, and M. Fisher, 3.0 Specification, Final Release, Sun Microsystems, http://java.sun.com/products/jdbc/download.html#corespec30, October 2001.