SANTA-G: an instrument monitoring framework for the Grid

Rob Byrom[4], Brian Coghlan[7], Andrew Cooke[1], Roney Cordenonsi[3], Linda Cornwall[5], Ari Datta[3], Abdeslem Djaoui[5], Laurence Field[4], Steve Fisher[5], Steve Hicks[4], Stuart Kenny[7], James Magowan[2], Werner Nutt[1], David O'Callaghan[7], Manfred Oever[2], Norbert Podhorszki[6], John Ryan[7], Manish Soni[4], Paul Taylor[2], Antony Wilson[4], and Xiaomei Zhu[4]

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

Abstract. In this paper we describe SANTA-G (Grid-enabled System Area Networks Trace Analysis) an instrument monitoring framework that uses the R-GMA (Relational Grid Monitoring Architecture). We describe the R-GMA component that allows for instrument monitoring, the CanonicalProducer. We also describe an example use of this approach in the European CrossGrid project, the SANTA-G Network Tracer, a network monitoring tool.

1 The R-GMA

The Grid Monitoring Architecture (GMA) [2] of the Global Grid Forum (GGF), as shown in Figure 1, consists of three components: *Consumers*, *Producers* and a directory service, which in the R-GMA is referred to as a *Registry*.

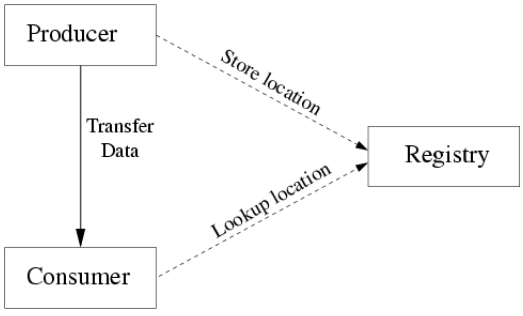

Fig. 1. Grid Monitoring Architecture

In the GMA Producers register themselves with the Registry and describe the type and structure of information they want to make available to the Grid. Consumers can query the Registry to find out what type of information is available and locate Producers that provide such information. Once this information is known the Consumer can contact the Producer directly to obtain the relevant data. By specifying the Consumer/Producer protocol and the interfaces to the Registry one can build inter-operable services. The Registry communication is shown on Figure 1 by a dotted line and the main flow of data by a solid line.

The current GMA definition also describes the registration of Consumers, so that a Producer can find a Consumer. The main reason to register the existence of Consumers is so that the Registry can notify them about changes in the set of Producers that interests them. Although the GMA architecture was devised for monitoring, the R-GMA uses it as a basis for a *combined* information and monitoring system. The case for this was argued in [6]; that the only thing which characterises monitoring information is a time stamp, so in the R-GMA there is a time stamp on all measurements, saying that this is the time when the measurement was made, or equivalently the time when the statement represented by the tuple was true.

The GMA does not constrain any of the protocols nor the underlying data model, so the implementation of the R-GMA was free to adopt a data model which would allow the formulation of powerful queries over the data.

R-GMA is a relational implementation of the GMA, developed within the European DataGrid (EDG), which harnesses the power and flexibility of the relational model. R-GMA creates the impression that you have one RDBMS per Virtual Organisation (VO). However it is important to appreciate that the system is a way of using the relational model in a Grid environment and *not* a general distributed RDBMS with guaranteed ACID properties. All the producers of information are quite independent. It is relational in the sense that Producers announce what they have to publish via an SQL CREATE TABLE statement and publish with an SQL INSERT and that Consumers use an SQL SELECT to collect the information they need. For a more formal description of R-GMA see [3].

R-GMA is built using servlet technology and is being migrated rapidly to web services, specifically to fit into an OGSA/OGSI [7] framework.

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. It must be possible to find out about the latest-state of a resource or to be notified continuously whenever the state of a component changes. Likewise, the history of a stream is sometimes needed. Thus, a Grid information and monitoring system should support these three temporal query types. In order to allow for this there have so far been defined not just a single Producer but four different types: a DataBaseProducer, a StreamProducer, a LatestProducer and a CanonicalProducer. All appear to be Producers as seen by a Consumer, but they have different characteristics. The StreamProducer allows for information to be streamed continously to a Consumer. The LatestProducer only stores the

most recent tuple of information for a given primary key, thus providing the latest-state information, whereas a DataBaseProducer stores the entire history of a stream of information. The CanonicalProducer is described in detail in Section 3.

The producers are instantiated and given the description of the information they have to offer by an SQL `CREATE TABLE` statement and a `WHERE` clause expressing a predicate that is true for the table. Currently this is of the form `WHERE (column_1=value_1 AND column_2=value_2 AND ...)`. To publish data, in all but the CanonicalProducer, a method is invoked which takes the form of a normal SQL `INSERT` statement. The CanonicalProducer, though in some respects the most general, is somewhat different due to the absence of a user interface to publish data via an SQL `INSERT` statement; instead, it triggers user code to answer an SQL query.

A Consumer uses the Registry to find out what type of information is there, and where it is. The R-GMA Registry stores information about all producers currently available. The R-GMA, uniquely, includes a mediator (a kind of broker that is hidden behind the Consumer interface) specifically to make the R-GMA easy to use. The mediator knows that Producers are associated with views on a virtual data base. Currently views have the form:

SELECT * FROM <table> WHERE <predicate>

This view definition is stored in the Registry. When queries are posed by a Consumer, the Mediator uses the Registry to find the right Producers and then combines information from them.

The set of queries that a particular producer supports is recorded in the registry. There are three types of query: continuous, latest and history. A continuous query causes all new tuples that match the query to be streamed into the consumers tuple storage, as soon as they are inserted into the virtual table by the producers. Streaming continues until the consumer requests it to stop. Latest and history queries are one-time queries: they execute on the current contents of the virtual table, then terminate. In a history query, all versions of any matching tuples are returned; in a latest-query, only those representing the "current state" are returned, where current state is the most recent version of the tuples.

2 R-GMA Architecture

R-GMA is currently based on Servlet technology. Each component has the bulk of its implementation in a Servlet. Multiple APIs in Java, C++, C, Python and Perl are available for user code to communicate with the servlets. The R-GMA makes use of the Tomcat Servlet container. Most of the R-GMA code is written in Java and is therefore highly portable. The only dependency on other EDG software components is in the security area.

Figure 2 shows the communication between the APIs and the Servlets. When a Producer is created its registration details are sent via the Producer Servlet

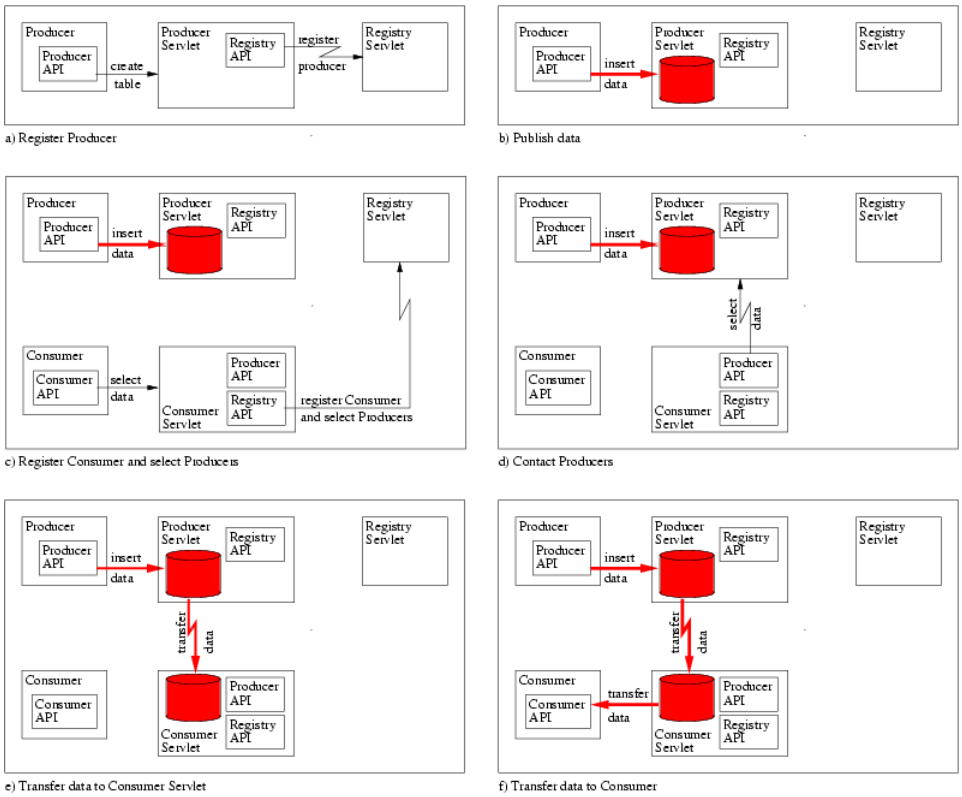

Fig. 2. Relational Grid Monitoring Architecture

to the Registry (Figure 2a). The Registry records details about the Producer, which include the description and view of the data published, *but not the data itself*. The description of the data is actually stored as a reference to a table in the Schema. In practise the Schema is co-located with the Registry. Once registration is completed, then whenever the Producer publishes data, the data are transferred to a local Producer Servlet (Figure 2b).

When a Consumer is created its registration details are also sent to the Registry although this time via a Consumer Servlet (Figure 2c). The Registry records details about the type of data that the Consumer is interested in. The Registry then returns a list of Producers back to the Consumer Servlet that match the Consumer's selection criteria.

The Consumer Servlet then contacts the relevant Producer Servlets to initiate transfer of data from the Producer Servlets to the Consumer Servlet as shown in Figures 2d-e. The data are then available to the Consumer on the Consumer Servlet, which should be close in terms of the network to the Consumer (Figure 2f).

As details of the Consumers and their selection criteria are stored in the Registry, the Consumer Servlets are automatically notified when new Producers are registered that meet their selection criteria.

The system makes use of soft state registration to make it robust. Producers and Consumers both commit to communicate with their servlet within a certain time. A time stamp is stored in the Registry, and if nothing is heard by that time, the Producer or Consumer is unregistered. The Producer and Consumer servlets keep track of the last time they heard from their client, and ensure that the Registry time stamp is updated in good time.

3 The CanonicalProducer

If we have to deal with a large volume of data it may not be practical to convert it all to a tabular storage model. Moreover, it may be inefficient to transfer the data to a Producer servlet with SQL INSERT statements. It may be judged better to leave the data in its raw form at the location where it was created. The CanonicalProducer is able to cope with this by accepting SQL queries and using user supplied code to return selected information in tabular form when required.

In general the R-GMA producers are sub-classes of the Insertable class, the class that provides the insert method. The insert method is used by the producers to send data to the servlets as an SQL INSERT string. The CanonicalProducer is different however; it is a subclass of the Declarable class. This means that it inherits the methods for declaring tables, but not inserting data. The user's producer code is responsible for obtaining the data requested. Figure 3 shows the communication between the servlets for a CanonicalProducer. When the other producer types publish data, the data is transferred to a local producer servlet via a SQL INSERT. The CanonicalProducer Servlet, however, is never sent raw data, which is instead retained local to the user's CanonicalProducer code.

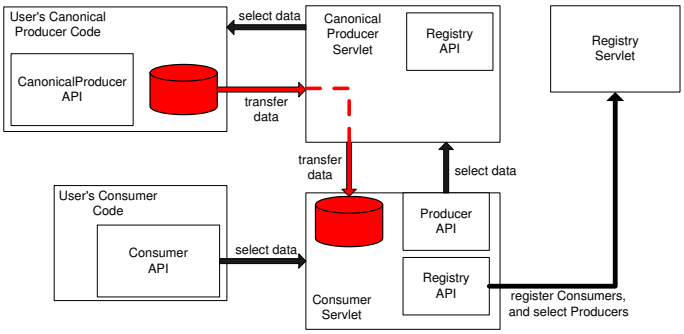

Fig. 3. CanonicalProducer Servlet Communication

A CanonicalProducer is instantiated by calling the API constructor method:

```
CanonicalProducer myProducer =
new CanonicalProducer
( 8998, CanonicalProducer.HISTORY );
```

This creates a new CanonicalProducer object, which registers itself with the CanonicalProducerServlet. The first parameter is a port number. The CanonicalProducerServlet expects to be able to connect, by way of a socket connection, to the CanonicalProducer code on this port in order to satisfy SQL queries. The second parameter describes the type of query that this producer code can satisfy, HISTORY or LATEST.

The table, or tables, that this producer publishes are then declared using the `declareTable` method.

```
myProducer.declareTable
( "cpuLoadUsage",
  " WHERE (ipAddress='"
  + this.ipAddress + "')",
  "CREATE TABLE cpuLoadUsage(
  ipAddress VARCHAR(50)
  NOT NULL PRIMARY KEY,
  cpuLoad REAL)"
)
```

When the servlet receives a query it opens a socket connection on the given port number to the CanonicalProducer code and forwards the SQL SELECT query to the producer code. The producer code must then execute the query, in whatever way it likes, and return a ResultSet to the servlet. The servlet can then return this ResultSet to the consumer. With the other producer types the producer is never aware of the SQL SELECT queries, they simply push the data to the servlet, and it is the servlet that carries out the SQL query. With a CanonicalProducer, however, the servlet has only the very minimum functionalty. To satisfy the query, it simply acts as an intermediary, forwarding the query to the correct CanonicalProducer instance and waiting for results to be returned.

A typical implementation of CanonicalProducer code would consist of several components, as shown in Figure 4. Although the figure shows the data being collected by an instrument and stored in log files, the data source could be anything.

CanonicalProducer Code would be the main class implemented by the user, which would use the CanonicalProducer API to instantiate a producer object, and declare the tables that the producer publishes. It would then start a Listener to wait for connections from the servlet.

Listener would be created by the main class. It would need to create a ServerSocket and then listen on this socket for connections from the servlet. When a connection is obtained it would be passed to a processing thread to execute the query and then continue listening for new connections.

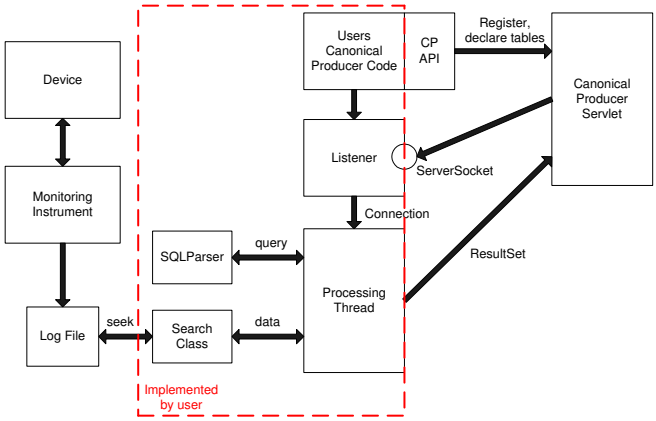

Fig. 4. An example of CanonicalProducer user code

Processing Thread would receive the connection to the servlet from the Listener. The processing thread would read the SQL SELECT query from the socket connection, and process it over the available data. When the results had been accumulated they would then be returned to the servlet, over the same socket connection.

SQL Parser Some additional classes would have to be used by the processing thread. A class would be needed to parse the SQL SELECT received from the servlet.

Search Class A class would also be needed to search the data for the required results to satisfy the query. This class might, for example, perform seek operations on a binary log file to find the data, or possibly invoke a script to collect the data.

Results should be returned to the servlet as XML ResultSets. The form of these is as follows:

```
<?xml version = '1.0' encoding='UTF-8' "standalone='no'?>
<edg:XMLResponse xmlns:edg='http://www.edg.org'>
    <XMLResultSet>
        <rowMetaData>
            <colMetaData>ColumnName</colMetaData>
        </rowMetaData>
        <row><col>ColumnValue</col></row>
    </XMLResultSet>
</edg:XMLResponse>
```

An important issue with the CanonicalProducer is the following. For the other producer types one can estimate how often the producer will contact the servlet, as it should be regularly inserting data. This is not the case with the CanonicalProducer. Because the CanonicalProducer never actually inserts data,

the servlet will never be informed as to whether the producer is still alive, and therefore will not inform the registry. After the R-GMA *termination interval* the CanonicalProducer would be presumed to be dead and its details would be removed from the registry. To avoid this a CanonicalProducer implementation should ensure that it regularly sends a sign of life to the servlet. This can be achieved by a thread that periodically, at intervals less than the termination interval, contacts the servlet.

Because the user must write the code to parse and execute the query, the CanonicalProducer can be used to carry out any type of query on any type of data source.

R-GMA is being further developed within the EU EGEE project [19]. A new Static query type has been added. The concept of the Canonical Producer has been extended as an On-Demand Producer that can encompass large databases as well as instruments, and which specifically supports the static query type.

4 SANTA-G CrossGrid Demonstrator: Network Tracer

SANTA-G (Grid-enabled System Area Networks Trace Analysis) is a generic template for ad-hoc, non-invasive monitoring with external instruments, see Figure 5.

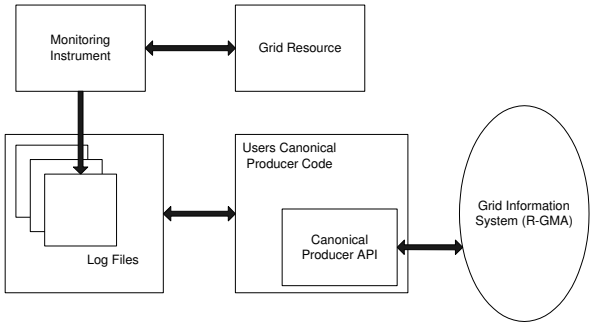

Fig. 5. SANTA-G monitoring framework

The template allows for the information captured by external instruments to be introduced into the Grid Information System. It is possible for these intruments to be anything, from fish sonars to PCR Analysers. The enabling technology for the template is the CanonicalProducer. The demonstrator of this concept, developed within the CrossGrid [13] project, is a network tracer that allows a user to analyse the Ethernet traffic at a site. The information obtained is useful for both the validation and calibration of intrusive monitoring systems and also for performance analysis.

The SANTA-G NetTracer is composed of three components that allow for the monitoring data to be accessed through the R-GMA: a Sensor (which is installed

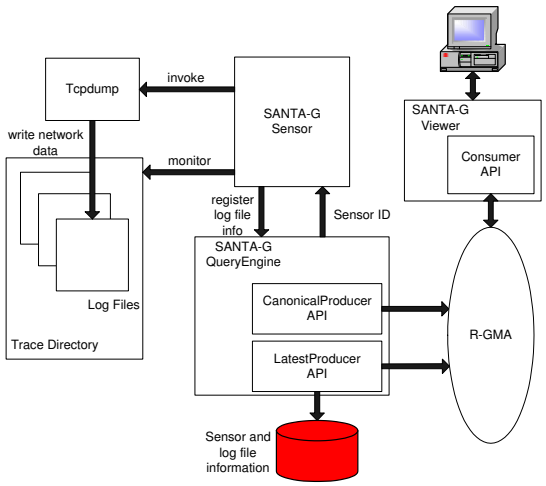

Fig. 6. SANTA-G NetTracer

on the node(s) to be monitored), a QueryEngine, and a Viewer GUI, see Figure 6. The Sensor invokes Tcpdump (an open-source packet capture application), and then monitors the log files created. The Sensor notifies the QueryEngine when new log files are detected. The QueryEngine records these events in a database, which is published to users through the R-GMA (by using the LatestProducer API). The QueryEngine also includes the interface to the R-GMA by using the CanonicalProducer API. Data is viewed via the R-GMA by submitting an SQL SELECT statement, as if querying a relational database. Through the CanonicalProducer this query is forwarded to the QueryEngine, which then parses the query, searches the appropriate log file to obtain the data required to satisfy the query, and returns the dataset to the GUI through the R-GMA.

It is the SANTA-G QueryEngine that implements the components of the CanonicalProducer code as described in Section 3. Figure 7 shows how the QueryEngine executes a SQL query received from the R-GMA (i.e. from the CanonicalProducerServlet). The QueryEngine listens on a socket, waiting for connections from the Servlet. When a connection is made the SQL query is read from the socket and passed to an SQLParser class. The parser breaks the query into three separate lists; a select list that contains the network header fields to be read, a from list that contains the table the fields belong to, and a where list that contains the values used to match the packets to. The Search class searches the log file for network packets that match the WHERE predicates specified in the query, and extracts the required packet header fields from them. The data that satisfies the query is accumulated into a ResultSet in XML format and returned to the Servlet over the socket connection. For example, the following query:

```
SELECT source_address, destination_address,
packet_type
```

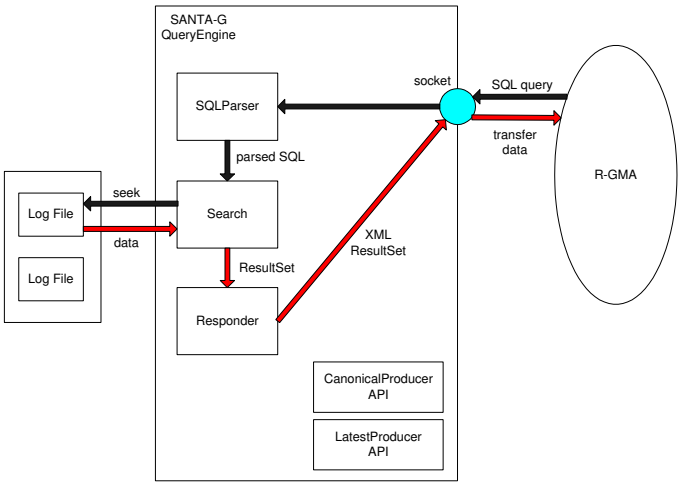

Fig. 7. SANTA-G QueryEngine Query Processing

```
FROM Ethernet
WHERE sensorId = 'some.machine.com:0'
AND fileId = 0
AND packetId < 100
```

would return the source address, destination address, and packet type fields of the Ethernet header for the first 100 packets in the log file assigned ID 0 and stored on 'some.machine.com'.

The SANTA-G NetTracer has been deployed in both the CrossGrid development and production testbed. The deployment of the system within a site is summarised in Figure 8. Within a CrossGrid testbed site the QueryEngine is installed on the storage element, while the sensors are installed on one or more of the worker nodes.

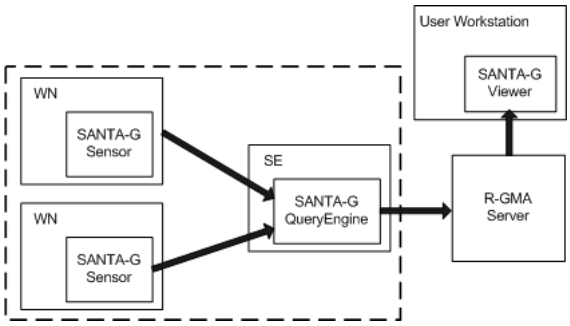

Fig. 8. SANTA-G NetTracer Site Deployment

TCD hosts a central R-GMA server for the whole CrossGrid testbed. This hosts a central registry for use by the CrossGrid VO.

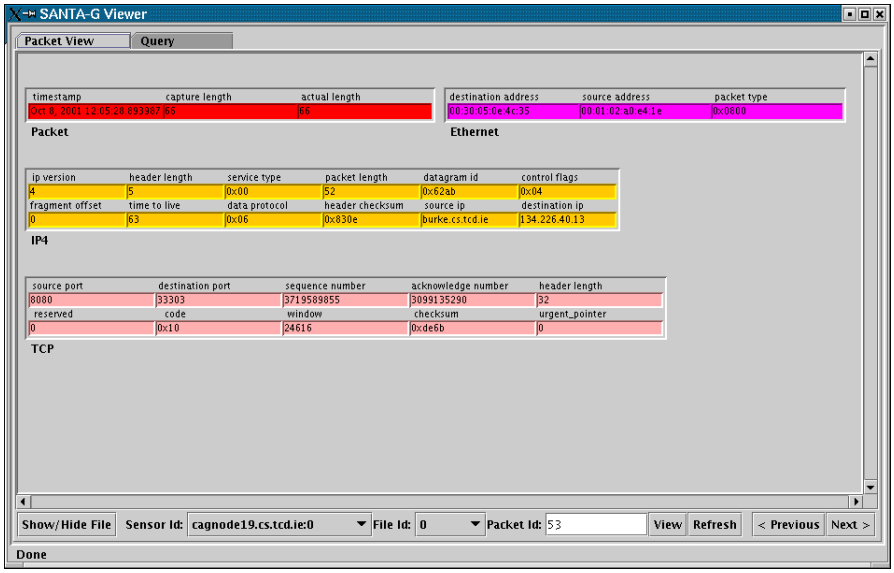

Fig. 9. SANTA-G Viewer

The SANTA-G Viewer (Figure 9) provides a graphical user interface, which makes use of the R-GMA Consumer API, to allow users to graphically view network packets in the log files, and also to build and submit SQL queries that will be carried out on the log files.

This architecture is obviously amenable to extensions that understand the contents of the packet data in Tcpdump logfiles. As an example, SANTA-G has been extended for the SNORT network intrusion detection system, which can log alerts to a Tcpdump-compatible logfile when suspect packets are detected. A snort Sensor type was added to allow users to query and view log files created by SNORT. The snort Sensor monitors the alert file generated by SNORT. When a new alert is detected its details are sent to the QueryEngine, which records the alert. Users can then view these alerts, using the Viewer GUI. The full packet data of the packet that triggered the alert can also then be viewed by querying the packet log file generated by SNORT.

5 An Example Experiment

The purpose of a SANTA-G system is to allow for ad-hoc monitoring experiments in the grid environment. With the NetTracer these experiments would be to

determine the state of the network at a given point in time. In any experiment involving a SANTA-G system there are four steps that would need to be followed:

1. Define the experiment.
2. Configure the SANTA-G system to acquire the required monitoring data.
3. Write the software for a Consumer to select subsets of the data and to calculate the required metrics from this data.
4. Run the experiment.

The following chapter describes an example experiment using the NetTracer.

5.1 TCP Throughput Measurements

The example experiment is to obtain throughput measurements using the network monitoring data obtained from the SANTA-G NetTracer in order to observe the flow of data through the R-GMA system during a query submission. A test Consumer that submits a query to the R-GMA will be run on a node, whilst a dynamic sensor will be used to acquire the traffic between the appropriate servlets in the R-GMA system. A second custom Consumer (described below) will be used to calculate throughput values from the data acquired.

Configure the SANTA-G system There are two components that need to be configured, the SANTA-G system itself (the NetTracer in this example) and the external instrumentation being used.

In the case of the NetTracer the external instrumentation is the network monitoring application Tcpdump. Tcpdump is configured by specifying the arguments to be used by the NetTracer sensors when invoking the Tcpdump application. The arguments to be passed to Tcpdump are entered in the sensor configuration file. The sensor is installed on the node to be monitored, i.e. the node hosting the R-GMA Consumer Servlet. By running on this host the sensor will be able to acquire all traffic sent between the Consumer Servlet and the CanonicalProducer Servlet, as well as that sent from the test Consumer code's host.

In this experiment we wish to determine the TCP throughput in order to visualise the flow of data through the R-GMA system. To do this the TCP traffic on the network must be acquired by the sensor. In order to minimise the amount of data collected only the traffic of interest should be acquired. Tcpdump is therefore configured to collect only TCP packets sent between the two servlets in the R-GMA system, and the test Consumer host.

The nodes involved in the example experiment are as follows:

cagnode46.cs.tcd.ie: hosts the test consumer.
bordeaux.cs.tcd.ie: hosts the Consumer Servlet, and the dynamic sensor.
cagraidsvr15.cs.tcd.ie: hosts the CanonicalProducer Servlet, along with the other R-GMA servlets.

cagnode47.cs.tcd.ie: hosts the QueryEngine, along with the throughput Consumer.

The dynamic sensor running on `bordeaux` must acquire the traffic received from the node hosting the test consumer, as well as that sent and received from the CanonicalProducer Servlet host. In the R-GMA system all communication is done via http on port 8080. Therefore, the appropriate Tcpdump arguments are:

```
(tcp src port 8080 or dst port 8080) and
(src host (bordeaux.cs.tcd.ie or cagnode46.cs.tcd.ie
or cagraidsvr15.cs.tcd.ie) and dst host (bordeaux.cs.tcd.ie
or cagnode46.cs.tcd.ie or cagraidsvr15.cs.tcd.ie))
```

When configuring the NetTracer itself it is important to carefully configure the size of both the log files and the log file queue. If the files and the queue are too large the performance of the system will be too slow for realtime calculations. If, however, the log files and queue are too small, and a large amount of traffic is generated, the data will not be maintained in the queue long enough to perform the calculations. The optimum settings for log file and queue size must be determined through trial and error.

Write the Consumer code In order to access the data acquired by the SANTA-G system a custom R-GMA Consumer must be written that makes use of the Consumer API. The following code extract shows how a consumer is created in order to submit a SQL query to the R-GMA and to retrieve results:

```
Consumer consumer = new Consumer("SELECT * FROM Ethernet", Consumer.HISTORY);
ResultSet resultSet = null;

if(!consumer.isExecuting()){
consumer.start();
}

while(consumer.isExecuting()){
Thread.currentThread().sleep(1000L);
}

if(consumer.hasAborted()){
throw new Exception("Consumer has aborted the query");
}

resultSet = consumer.popIfPossible();

consumer.close();
```

A Consumer is created by calling the Consumer constructor, passing in the SQL query and the query type. The SANTA-G system publishes the ethernet packet data as a HISTORY producer. The only type of query that can be answered therefore is a HISTORY query.

```
Consumer consumer = new Consumer("SELECT * FROM Ethernet", Consumer.HISTORY);
```

This creates a Consumer that will select all the available data from the Ethernet table.

To start the Consumer the start() method is called:

```
consumer.start();
```

The code then enters a loop that waits for the Consumer to finish collecting the data. This is done by polling the isExecuting() method.

When the Consumer completes, the result set can be obtained by calling one of the Consumer's pop() methods. popIfPossible() will return a result set if available, or null if no result set was returned.

The Consumer API is available in a number of languages, C, C++, Perl, and Python, as well as Java.

For this experiment a Consumer calculates throughput values, in bytes per second, from the acquired TCP traffic data. This is done by calculating and publishing two tables of throughput values, `TCPThroughput` and `AverageTCPThroughput`. The throughput is calculated for each unique TCP connection seen. The TCPthroughput table stores the final throughput values when a connection is closed, whereas AverageTCPThroughput stores the average throughput values seen during the lifetime of the connection (i.e. total bytes seen divided by the duration of the connection so far). Both tables are published by the Consumer to the R-GMA by using the DatabaseProducer API.

The data required to perform the throughput calculations is published by the NetTracer in two separate tables, the IP4 and TCP tables. Since the R-GMA does not support joins, the Throughput Consumer must perform two separate queries on these tables to obtain the data from the TCP packets acquired during the required interval. The SELECT statements used to obtain the IP and TCP data are as follows:

```
SELECT sensorId, fileId, source_ip, destination_ip,
       packet_length, header_length, data_protocol
FROM IP4
WHERE timestamp_Secs > startTime and timestamp_Secs < stopTime

SELECT sensorId, source_port, destination_port, header_length,
       sequence_number, acknowledge_number, code
FROM TCP
WHERE timestamp_Secs > startTime and timestamp_Secs < stopTime
```

where `stopTime` is the current time, and `startTime` is the current time minus the required interval. The ResultSets obtained in response to these queries

are then passed to a processing thread that matches packets contained in the ResultSet to TCP connections and calculates the throughput for that connection. The output from the Consumer was validated using the program Tcptrace. Tcptrace is an open source application that analyses Tcpdump format log files and produces statistics from them, including throughput values. Then why use the NetTracer rather than Tcptrace? Because the SANTA-G NetTracer enables multiple real-time acquisitions at geographically dispersed sites, and consequent calculations to be performed in a grid-wide fashion, in contrast to a single off-line local equivalent.

Run the experiment The final step is to run the experiment. To do this a *static* sensor must be started on a node in the testbed. The *dynamic* sensor must be started on the Consumer Servlet's host node. The throughput consumer must also be started. The test consumer, which submits a query for 1000 tuples from the static sensor's logfile, should then be started.

The table below (Table 1) shows the throughput values published by the throughput consumer during the query processing, where:

ID: identifies a unique connection seen.
AtoB: is the throughput from host A to host B in bytes/sec.
BtoA: is the throughput from host B to host A in bytes/sec.
timestamp: corresponds to the timestamp of the last packet in the connection.

ID	HostA	HostB	Throughput		Timestamp
			AtoB	BtoA	
1	cagnode46.cs.tcd.ie:33370	bordeaux.cs.tcd.ie:8080	26355	18985	12:12:51.532487
2	cagnode46.cs.tcd.ie:33371	bordeaux.cs.tcd.ie:8080	17630	26888	12:12:51.547510
3	cagnode46.cs.tcd.ie:33372	bordeaux.cs.tcd.ie:8080	21402	37216	12:12:51.561563
5	cagnode46.cs.tcd.ie:33373	bordeaux.cs.tcd.ie:8080	27396	41656	12:12:51.572828
6	cagnode46.cs.tcd.ie:33374	bordeaux.cs.tcd.ie:8080	16507	25100	12:12:52.595250
7	cagnode46.cs.tcd.ie:33375	bordeaux.cs.tcd.ie:8080	9959	15188	12:12:53.653846
8	cagnode46.cs.tcd.ie:33376	bordeaux.cs.tcd.ie:8080	20724	31657	12:12:53.669331
9	cagnode46.cs.tcd.ie:33377	bordeaux.cs.tcd.ie:8080	235	242015	12:12:54.636282
10	cagnode46.cs.tcd.ie:33378	bordeaux.cs.tcd.ie:8080	17723	32701	12:12:55.061768
4	bordeaux.cs.tcd.ie:34226	cagraidsvr15.cs.tcd.ie:8080	249	285529	12:12:57.968999

Table 1. Throughput values obtained during query submission.

It is possible to match each of these connections to a call in the code of the test Consumer that is used to submit a query to the R-GMA system, see Table 2. The code used is the same as that outlined in Section 5.1. The duration of the connection is obtained from the `AverageTCPThroughput` table, which stores the throughput seen during the lifetime of the connection.

The remaining connection (connection ID 4) is between the Consumer Servlet and the CanonicalProducer Servlet. It can be seen from Table 1 that the connection is not closed until after the call to close the Consumer API object. The

Connection ID	API Call	Duration (secs)
1	new Consumer(...)	0.017909
2	if(!consumer.**isExecuting()**)	0.012942
3	consumer.**start()**	0.010560
5	while(consumer.**isExecuting()**)	0.007994
6	while(consumer.**isExecuting()**)	0.013453
7	while(consumer.**isExecuting()**)	0.024611
8	if(consumer.**hasAborted()**))	0.011282
9	consumer.**popIfPossible()**	0.961396
10	consumer.**close()**	0.012018

Table 2. Connections and the API calls associated with them.

HostA	HostB	Duration	Throughput		Timestamp
			AtoB	BtoA	
bordeaux.cs.tcd.ie:34226	cagraidsvr15.cs.tcd.ie:8080	0.001168	294521	0	12:12:51.564081

Table 3. SQL query transmission measurement.

connection is initially opened by the Consumer Servlet in order to send the SQL query to the CanonicalProducer Servlet. This can be seen from the connection's entries in the `AverageTCPThroughput` table. Table 3 shows the first measurement for the connection. This corresponds to the transmission of the SQL query.

From the `AverageTCPThroughput` table the last packet transmitting the SQL query was sent at **12:12:51.564081**. Although the connection was not in fact closed until **12:12:57.968999** it was seen from the `AverageTCPThroughput` table that the last data packet on the connection was received on the Consumer Servlet host at **12:12:52.945843**. The time from completion of transmission of the SQL query, until completion of reception of the result set from the CanonicalProducer servlet was therefore **1381.76**ms. It is known from the QueryEngine logs that it took **938**ms for the QueryEngine to complete the query. This implies the CanonicalProducer Servlet, not taking into account network delays, added an additional **443.76**ms.

The final stage in the flow of data through the R-GMA system is the return of the result set to the Consumer API object. As stated the Consumer Servlet received the last data packet of the result set from the CanonicalProducer Servlet at **12:12:52.945843**. The Consumer Servlet completed transmitting this result set to the Consumer API at **12:12:54.636282**, a delay of **1690.44**ms, which corresponds to approximately 55% of the total **3072**ms (as calculated from the connection times) taken to answer the query. Table 4 summarises the times observed.

The additional time measured by the test Consumer (427ms) can be explained by the fact that the time measurement was taken in the code after the call to `popIfPossible()` had returned, thus taking into account the final conversion from XML to ResultSet object performed by the Consumer API.

Measurement	Time (mS)
time from query transmission by Consumer Servlet until resultset reception at Consumer Servlet	1381.8
time from resultset reception at Consumer Servlet until completion of resultset transmission to Consumer API	1690.4
time from query transmission by Consumer Servlet until completion of resultset transmission to Consumer API	3072.2
total query time as measured by test Consumer	3449

Table 4. Summary of calculated times.

6 Conclusion

The R-GMA is a relational implementation of the GMA architecture. It is built using servlet technology. In the R-GMA most producers of information publish data by transferring the data to servlets. This may not always be suitable for all applications. When dealing with instruments which produce a large volume of data it may not be practical to convert it all to a tabular storage model nor efficient to transfer the data to the servlets. It may be preferrable to leave the data where it was created, and only transfer it across the network when specifically requested by a user. In order to allow for this a special type of producer was included in R-GMA, the CanonicalProducer. This allows a user to customize the way the producer responds to a user request, i.e. a SQL query. The SANTA-G network monitoring tool developed within the CrossGrid project demonstrates the CanonicalProducer concept by publishing Ethernet trace data. In reality the underlying concepts have wider applicability; they allow information from a great variety of instruments to be accessed through the Grid information system. For example the SANTA-G NetTracer has been extended to publish logs generated by the SNORT network intrusion detection system.

References

1. Andrew Cooke, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Ari Datta, Roney Cordenonsi, Rob Byrom, Laurence Field, Steve Hicks, Manish Soni, Antony Wilson, Xiaomei Zhu, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian Coghlan, Stuart Kenny David O'Callaghan, John Ryan. RGMA: First Results After Deployment CHEP03, La Jolla, California, March 24-28, 2003.
2. 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.

3. Andy Cooke, Alasdair J G Gray, Lisha Ma, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Rob Byrom, Laurence Field, Steve Hicks, Jason Leake, Manish Soni, Antony Wilson, Roney Cordenonsi, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian Coghlan Stuart Kenny, David O'Callaghan. *R-GMA: An Information Integration System for Grid Monitoring* Proceedings of the Tenth International Conference on Cooperative Information Systems, 2003.
4. Andrew Cooke, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Ari Datta, Roney Cordenonsi, Rob Byrom, Laurence Field, Steve Hicks, Manish Soni, Antony Wilson, Xiaomei Zhu, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian Coghlan, Stuart Kenny, Oliver Lyttleton, David O'Callaghan, John Ryan. *Information and Monitoring Services Within a Grid* Proceedings of the Eleventh International Conference on Cooperative Information Systems, 2004.
5. Brian Coghlan, Andrew Cooke, Roney Cordenonsi, Linda Cornwall, Ari Datta, Abdeslem Djaoui, Laurence Field, Steve Fisher, Steve Hicks, Stuart Kenny, James Magowan, Werner Nutt, David O'Callaghan, Manfred Oevers, Norbert Podhorszki, John Ryan, Manish Soni, Paul Taylor, Antony Wilson Xiaomei Zhu. *The Canonical-Producer: an instrument monitoring component of the Relational Grid Monitoring Architecture* Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, July 2004.
6. Brian Coghlan, Abdeslem Djaoui, Steve Fisher, James Magowan, Manfred Oevers. *Time, Information Services and the Grid* 31st May 2001.
7. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, P. Vanderbilt. *Grid Service Specification* http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04_2002-10-04.pdf, 2003.
8. The DataGrid Project. http://www.eu-datagrid.org
9. DataGrid WP3. *DataGrid Information and Monitoring Final Evaluation Report* https://edms.cern.ch/document/410810/4/DataGrid-03-D3.6-410810-4-0.pdf, 2004.
10. Brian Coghlan, Stuart Kenny. *SANTA-G Software Design Document*
11. Brian Coghlan, Stuart Kenny. *SANTA-G First prototype Description* http://www-eu-crossgrid.org/Deliverables/M12pdf/CG3.3.2-TCD-D3.3-v1.1-SANTAG.pdf, 2004.
12. CrossGrid WP3. *Deliverable D3.5, Report on the Results of the 2nd and 3rd Prototype* http://www-eu-crossgrid.org/Deliverables/M24pdf/CG3.0-D3.5-v1.2-PSNC010-Proto2Status.pdf, 2004.
13. The CrossGrid Project http://www.eu-crossgrid.org, 2004.
14. DataGrid WP3 Information and Monitoring Services http://hepunx.rl.ac.uk/edg/wp3/, 2004.
15. Global grid forum. http://www.ggf.org, 2004.
16. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2: Computational Grids, pages 15–51. Morgan Kaufmann, 1999.
17. 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.
18. Globus Toolkit. http://www.globus.org, 2004.
19. Enabling Grids for E-science in Europe http://egee-intranet.web.cern.ch/egee-intranet/gateway.html, 2004.