# CrossGrid Developers Guide

—

**WP3.3 Grid Monitoring**

| | |
|---|---|
| Document Filename: | **developer.pdf** |
| Workpackage: | **WP3.3 Grid Monitoring** |
| Partner(s): | **TCD, CYFRONET, ICM** |
| Lead Partner: | **TCD** |
| Config ID: | **cg-santag-develop** |
| Document classification: | **PUBLIC** |

Abstract: This document explains what you need to know as a developer of the SANTA-G system.

## Delivery Slip

| | Name | Partner | Date | Signature |
|---|---|---|---|---|
| From | Stuart Kenny | TCD | February 2004 | |
| Verified By | | | | |
| Approved By | | | | |

## Document Log

| Version | Date | Summary of changes | Author |
|---|---|---|---|
| 1-0 | 18 Feb 2004 | First public version | Stuart Kenny |
| 1-1 | 28 June 2004 | Added fix for test build fail | Stuart Kenny |
| 1-2 | 02 Dec 2004 | Rewritten to follow new CrossGrid template | Stuart Kenny |

# Contents

# Copyright Notice

This research is partly funded by the European Commission IST-2001-32243 Project CrossGrid.

# 1 INTRODUCTION

SANTA-G services are a specialized non-invasive complement to other more intrusive monitoring services. One application of these services would be in validation and calibration of both intrusive monitoring systems and systemic models, and also for performance analysis. The objectives are to:

1. allow information captured by external monitoring instruments to be introduced into the Grid information system,
2. support analysis of performance using this information.

The prototype illustrates these concepts with a NetTracer, a demonstrator that allows a user to analyse the network traffic on a site. 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.

Installation and User guides for the SANTA-G system are provided in [SANTAGINSTALL] and [SANTAGUSER].

## 1.1 Abbreviations and Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| CA | Certificate Authority |
| DN | Distinguished Name |
| EDG | European DataGrid |
| GMA | Grid Monitoring Architecture |
| GOC | Grid Operations Center |
| GUI | Graphical User Interface |
| LCG | LHC Computing Grid Project |
| LCFGng | Local ConFiGuration System next generation |
| NFS | Network File System |
| RDBMS | Relational Database Management System |
| R-GMA | Relational Grid Monitoring Architecture |
| SAN | System Area Network |
| SANTA | System Area Networks Trace Analysis |
| SANTA-G | Grid-enabled System Area Networks Trace Analysis |
| SQL | Structured Query Language |
| SSL | Secure Sockets Layer |
| TCP | Transmission Control Protocol |
| VO | Virtual Organisation |

## 1.2 References and Source Code

The full source code for the SANTA-G system can be found in the FZK CVS repository. Please refer to http://gridportal.fzk.de/?group=cg-wp3.3 for details on how to obtain it.

The source code can also be browsed at the following address:
http://savannah.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp3/wp3_3-moninfr/wp3_3_2-santag/

References can be found in the bibliography.

# 2 IMPLEMENTATION STRUCTURE

## 2.1 Product Use Cases

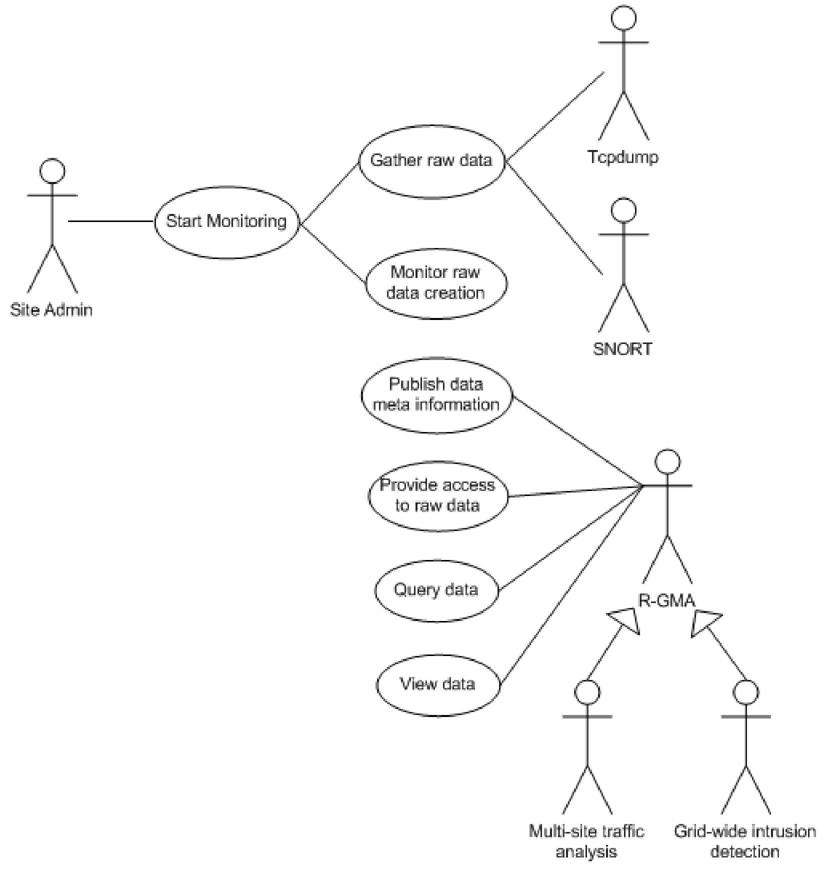
Site Admin
Start Monitoring
Gather raw data
Monitor raw data creation
Tcpdump
SNORT
Publish data meta information
Provide access to raw data
Query data
View data
R-GMA
Multi-site traffic analysis
Grid-wide intrusion detection

Figure 2.1: SANTA-G NetTracer Use Cases

Figure 2.1 shows the use cases defined for the SANTA-G NetTracer. The first use case involves the 'Site Administrator' actor, who is responsible for starting the monitoring at their site. This interaction involves two further use cases, 'gathering the raw data' and 'monitoring the raw data creation'. To start the monitoring the site administrator would configure and invoke one or more sensors on the nodes to be monitored. The system makes use of the external tools Tcpdump, or SNORT, to gather raw data, depending on the type of monitoring initiated. The purpose of the 'monitoring raw data creation' use case is to collect meta-information about the raw data being gathered by the external tool. This meta-information is stored by the system, and subsequently published to users so that they can see what data is available.

The remaining uses cases make use of the external R-GMA information system. This provides the interface to the system for external users. The NetTracer system allows for direct querying of the raw data, and also browsing of the data by way of the Viewer GUI.

The users, shown as generalisations of the R-GMA actor, would include those interested in network traffic analysis, for debugging or performance analysis in a grid-wide manner, and intrusion detection applications in the case of SNORT monitoring.

## 2.2 Product Component Model

This section describes the design and architecture of the SANTA-G NetTracer. The NetTracer can be broken down into two main modules: a publishing module, and a viewer module.

The purpose of the publishing module is to monitor the log files created by the external monitoring instrument, Tcpdump or SNORT, and to provide access to the data contained in these log files through the grid information system. Once the data is available through the system, users (including dependent tasks of the CrossGrid project) can access it for further analysis by using other R-GMA components such as Consumers and Archivers. The Viewer module is provided to allow users to visualise the data and to serve as an example of the use of the R-GMA Consumer API to access the monitoring data. The Viewer module consists of a Java Swing GUI that uses the Consumer API to collect subsets of the available data, which is then presented to users graphically or in a table.

The publishing module is itself composed of a further two components, the QueryEngine, and the Sensor, as can be seen in Figure 2.2. The reason the functionality is separated into two components is that there may be many sensors for each QueryEngine. The Sensor works in conjunction with the external instrument to monitor the log files of data created. The QueryEngine provides the remaining two elements of the framework: it implements the CanonicalProducer code that accesses the data gathered, and it also makes use of the CanonicalProducer API in order to register with the R-GMA system.

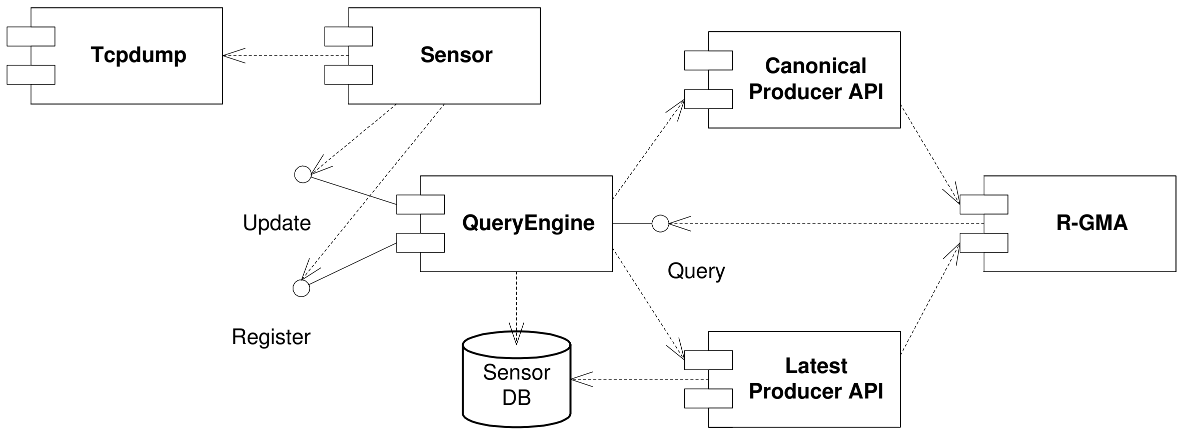

Figure 2.2: Publishing module implementation structure

It is intended in the design for a sensor to be deployed on each of the nodes to be monitored. The QueryEngine would then be hosted on a single machine to which the R-GMA host and each of the monitored nodes has access. It is possible for the QueryEngine host and the R-GMA host to be the same machine. The NetTracer could be deployed as shown in Figure 2.3.

The sensor, as it will run on the node to be monitored, must be as lightweight and as easy to install as possible. For this reason the majority of the processing and data storage should be handled by the QueryEngine. In order for a user to be able to query the NetTracer they must be able to obtain information about the available sensors, and the log files stored on the nodes hosting these sensors. The file information is stored by the QueryEngine and published to users through the R-GMA. This situation is analogous to the R-GMA, where the QueryEngine becomes a registry, storing information on the available sensors, and the sensor becomes a producer, publishing information on available log files to the QueryEngine.

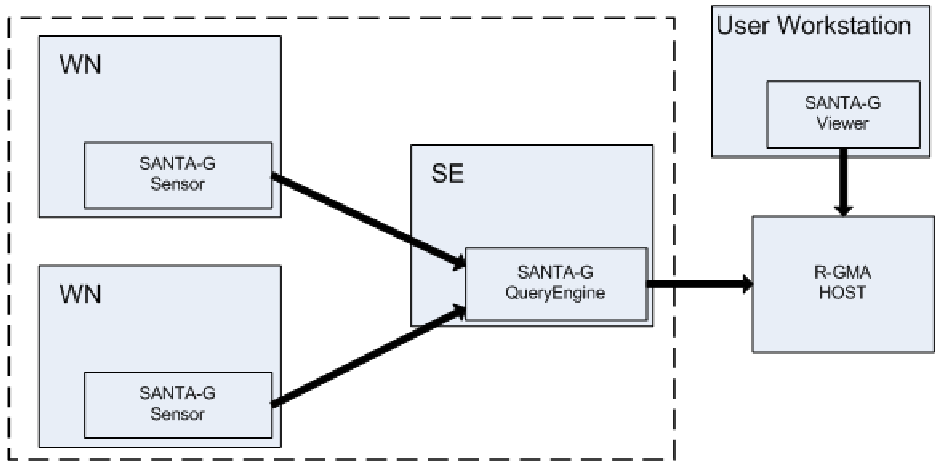

Figure 2.3: Example NetTracer deployment

The QueryEngine is the central component of the publishing module. It is the QueryEngine which provides the interface from the NetTracer to the R-GMA by using the CanonicalProducer API. It is the QueryEngine that receives queries submitted to the R-GMA. It then maps the query to a specific log file, or set of log files, maintained by a sensor, or set of sensors, accumulates the required data from the files to satisfy the query, and returns the resulting data set to the R-GMA.

The Viewer module provides a custom consumer and graphical user interface (GUI) to allow users to browse the data obtained by the NetTracer. The Viewer GUI allows users to view full packets from the log files of sensors or to submit an SQL query in order to obtain subsets of the available data. Certain other utilities are also provided, such as a query builder to construct complex queries.

The design of the Viewer Module is reasonably straightforward. A Java Swing GUI is provided which makes use of the Consumer API to submit SQL queries to the R-GMA and to receive ResultSets in response. The individual fields of these ResultSets are then extracted and displayed either graphically or in a tabular form.

Figure 2.4 shows the three possible ways of accessing the NetTracer monitoring data. The Viewer has been built, and is provided as part of the NetTracer. Its implementation is described in the next chapter. The other two methods involve making custom use of the R-GMA API's, the Consumer or the Archiver, to select subsets of the available data.

### 2.2.1 THE SCHEMA

The information available through the SANTA-G NetTracer system is specified by a schema. The design of the schema is conceptually divided into two: the tables of information related to the sensors, and the tables of packet data obtained from the log files. There are two main sensor information tables, `sensors`, and `sensorFiles`. `Sensors` stores the basic sensor information, such as the sensor's ID, sensor type, the sensor host, etc. The `sensorFiles` table holds the information on the log files stored on the sensor's node, the file ID, the file name, etc. A third table, `snortAlerts`, is used to store and publish the alerts detected by a SNORT sensor.

The remaining tables in the schema store the information in the network packets. Network packets can be divided into a series of network protocols. Each layer in the network protocol appends its own header to the packet. The schema was designed therefore so that each table relates to a particular network protocol, from the internet protocol down to the transport protocol. This means there has to be five tables; `Ethernet`, `IP4` (the four indicates IP version 4), `TCP`, `UDP`, and `ICMP`. The header information from each packet can then be taken and inserted into these tables. For example the header information of a

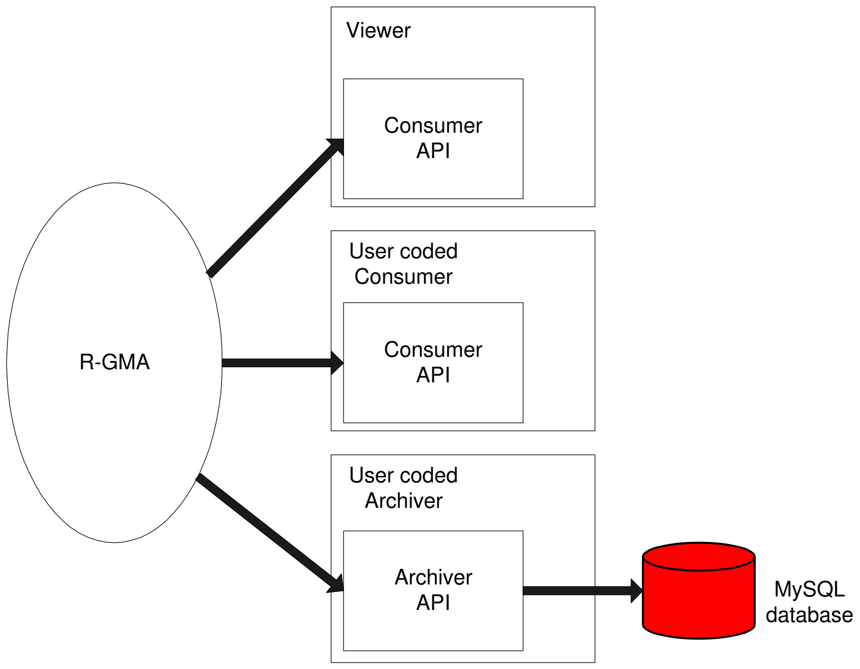

Figure 2.4: Consuming NetTracer data

TCP packet would occupy a row in three of the tables, `Ethernet`, `IP4`, and `TCP`. A further table, `Data` was added, in order to allow for the payload of the packet to be obtained. Each column in this table represents sixty four bytes of the packets payload.

Two additional tables are needed, `File` and `Packet`. Each log file created by Tcpdump has a file header appended to the start of the file. It contains information on the version of Tcpdump used to create the file and additional information on the log file, such as the type of link Tcpdump was capturing packets from, and the maximum number of bytes of a packet that was captured, and this is inserted into the `File` table. Tcpdump also appends a header to each packet captured. This header contains the size of the packet and a timestamp of when it was captured. This information is stored in the `Packet` table.

Every table contains four primary keys, `siteid`, which identifies the site on which the NetTracer system is running, `sensorId`, which is the identifier of the sensor that is running on a node within the site, the `fileId`, which identifies a particular log file on a sensor node, and the `packetId`, which identifies a packet within the log file. These keys can be used to uniquely identify a packet on a sensor node within a site.

Each table also contains two time fields, `MeasurementDate` and `MeasurementTime`. These fields are required by the R-GMA system to be present in every table published. The important point with these fields is that they do not correspond to the time the measurement was made, i.e. the time the packet was captured, rather the time that the measurement was entered into the R-GMA system. Because the CanonicalProducer only inserts the data into the R-GMA in response to a submitted query, these fields will always contain the current time and date. However, the time of measurement is recorded. In the initial schema design the timestamp of when the packet was actually captured was stored in only one table, the `Packet` table. The timestamp is represented as two fields, one containing seconds and one containing micro-second values. It was felt later that it would be advantageous to provide these fields in every table as this would significantly aid analysis of the data. Each table was therefore extended to include a `timestamp_Secs`, and `timestamp_uSecs` field.

The schema is defined as an XML document (see Appendix A), which is parsed by the QueryEngine on startup.

## 2.3 Detailed Implementation Model

### 2.3.1 The Sensor

In the SANTA-G NetTracer system three sensor types have been implemented, `static`, `dynamic`, and `snort`. The startup and shutdown sequence for each of the types is the same. At startup the sensor will attempt to register with the QueryEngine and if successful will receive an ID in response. This ID is used by the sensor in all future communications with the QueryEngine. When shutdown the sensor will send a message to the QueryEngine, in order that the QueryEngine can remove the sensors details from the table of currently running sensors. It is the functionality of the sensors whilst they are running that distinguishes the three types.

A static sensor is the simplest, and was the first implemented. It is used to publish details of a set of static, pre-acquired log files. This is particulary useful for testing. With this type of sensor a set of log files is obtained by running Tcpdump in advance. The set of files is then stored in a directory, referred to as the *trace directory*. As with all three sensor types, configuration is done by editing a configuration file. In this file the sensor type is defined, as well as the location of the directory which contains the Tcpdump log files. On startup the sensor registers with the QueryEngine, and then scans the trace directory to obtain information on the log files. This information is then sent to the QueryEngine where it is stored.

The dynamic sensor is used to monitor dynamically generated log files. When started the dynamic sensor will invoke Tcpdump using arguments specified in the sensor's configuration file. Tcpdump will create log files of network packet data in the specified directory. The sensor monitors the directory into which Tcpdump is writing. When it detects a new log file the sensor informs the QueryEngine, and the QueryEngine records this event. When the sensor is shutdown, Tcpdump will be stopped, and a message is sent to the QueryEngine to tell it to remove the sensors details from its information tables. The dynamic sensor maintains the created log files in a circular queue of configurable length. The maximum size of each file in the queue is also configurable. When a file reaches the maximum size a new file is created. If the creation of this file exceeds the maximum queue length then the oldest file is removed from the queue and the new file added. The oldest file can then be archived, i.e. compressed and moved to an archive directory, or it can be deleted. Figure 2.5 shows the start up sequence for a dynamic Sensor.

The snort Sensor type was added to allow users to query and view log files created by the SNORT Network Intrusion Detection System. The log files created by SNORT are in libpcap format, an open source packet capture library. This is the same format as Tcpdump, so the log files created by SNORT are compatible with the NetTracer system. SNORT works by monitoring network packets received on the host's network interface card. Any packet found which matches a rule from a set of defined security rules is logged to a log file, and an alert is generated, which is also stored in a separate alert file. The snort Sensor monitors the alerts file. 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. The startup sequence for a SNORT sensor is shown in Figure 2.6

As stated it was intended to keep the sensor as simple and lightweight as possible. The following class diagram, Figure 2.7, shows the structure of the sensor component. The following describes the classes and their function:

**Sensor:** this is the main class of the sensor component. It reads in the configuration variables stored in the sensor configuration file. In this file a variable, Sensor_Type, defines the type of sensor that is to be run. Depending on the type then certain other variables are set. In the case of a dynamic sensor for example, the arguments to be used when invoking Tcpdump are read.

**TCPdumpInvoker:** this class is used by the dynamic sensor to start a Tcpdump process. Tcpdump is started using the arguments read from the configuration file. The other important variables are the `trace directory`, the directory into which Tcpdump will create the log files, and the `maximum log file size`, the size beyond which Tcpdump will close the current log file and create a new

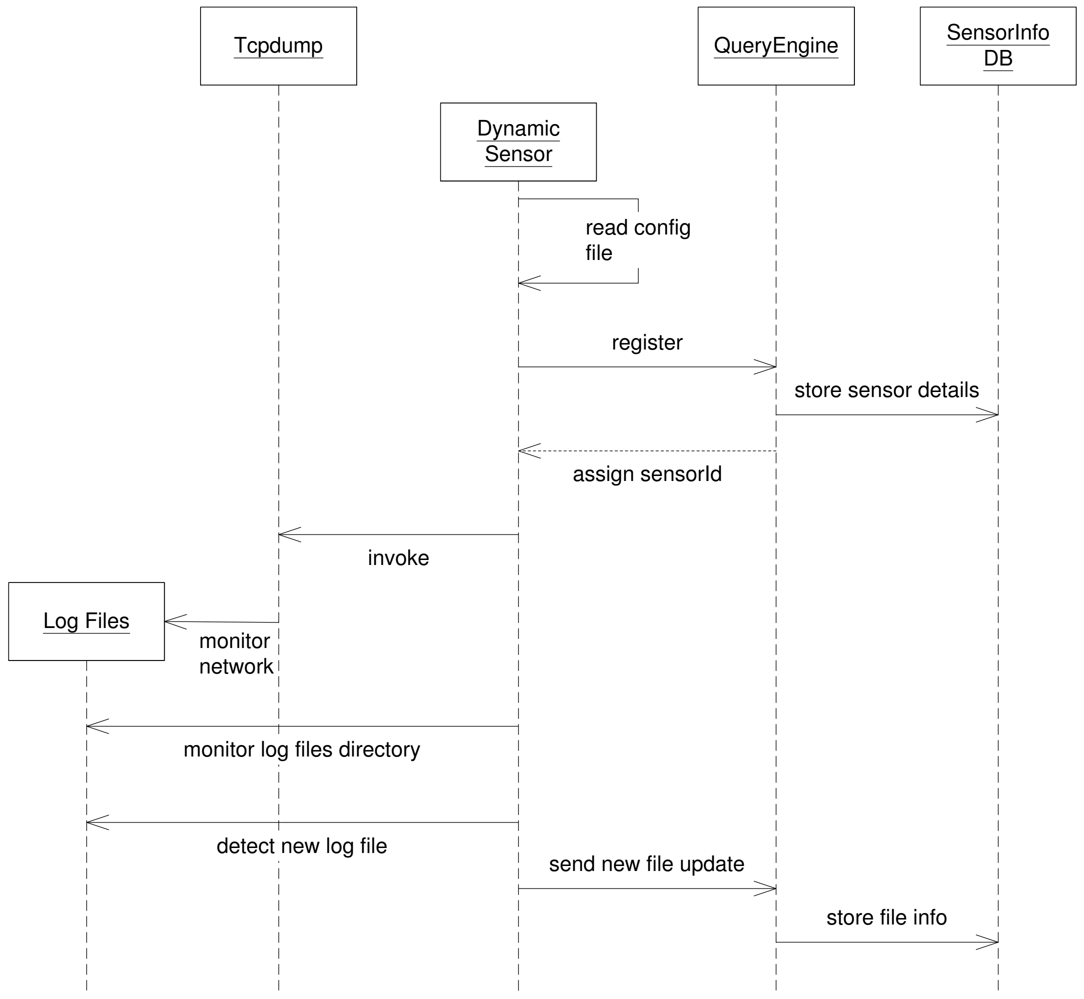

Figure 2.5: Dynamic Sensor startup sequence

file. The filename is composed from the hostname of the machine followed by a number, which increases with each file created, i.e. hostname, hostname2, hostname3.

**FileMonitor:** this is also used by the dynamic sensor. It periodically polls the trace directory to check if a new file has been created. The FileMonitor is also responsible for managing the queue of log files.

**ArchiveFile:** when the maximum number of log files are being maintained in the queue and a new log file is detected by the FileMonitor class then the oldest file in the queue is passed to the ArchiveFile class. If the archive variable is set to ARCHIVE in the configuration file then this class will compress the file and move it to the archive directory, otherwise the file is deleted.

**SnortMonitor:** this class is used by the SnortMonitor. It monitors the SNORT alerts log file. When a new alert is entered into the file the SNORT monitor reads the alert and sends an update message to the QueryEngine that encapsulates the alert.

**UpdateQueryEngineThread:** this class is used by the sensor to send update messages to the QueryEngine.

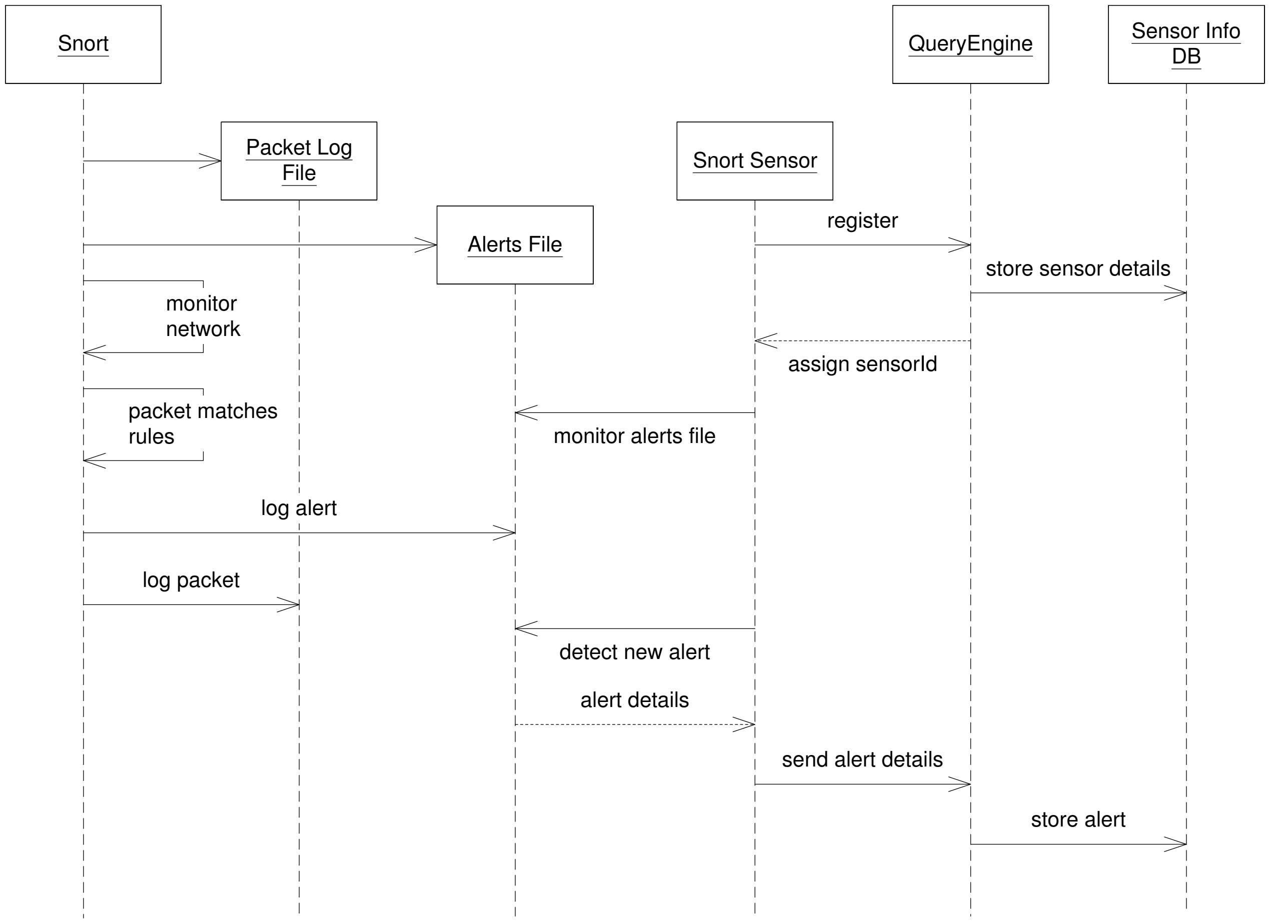

Figure 2.6: SNORT Sensor startup sequence

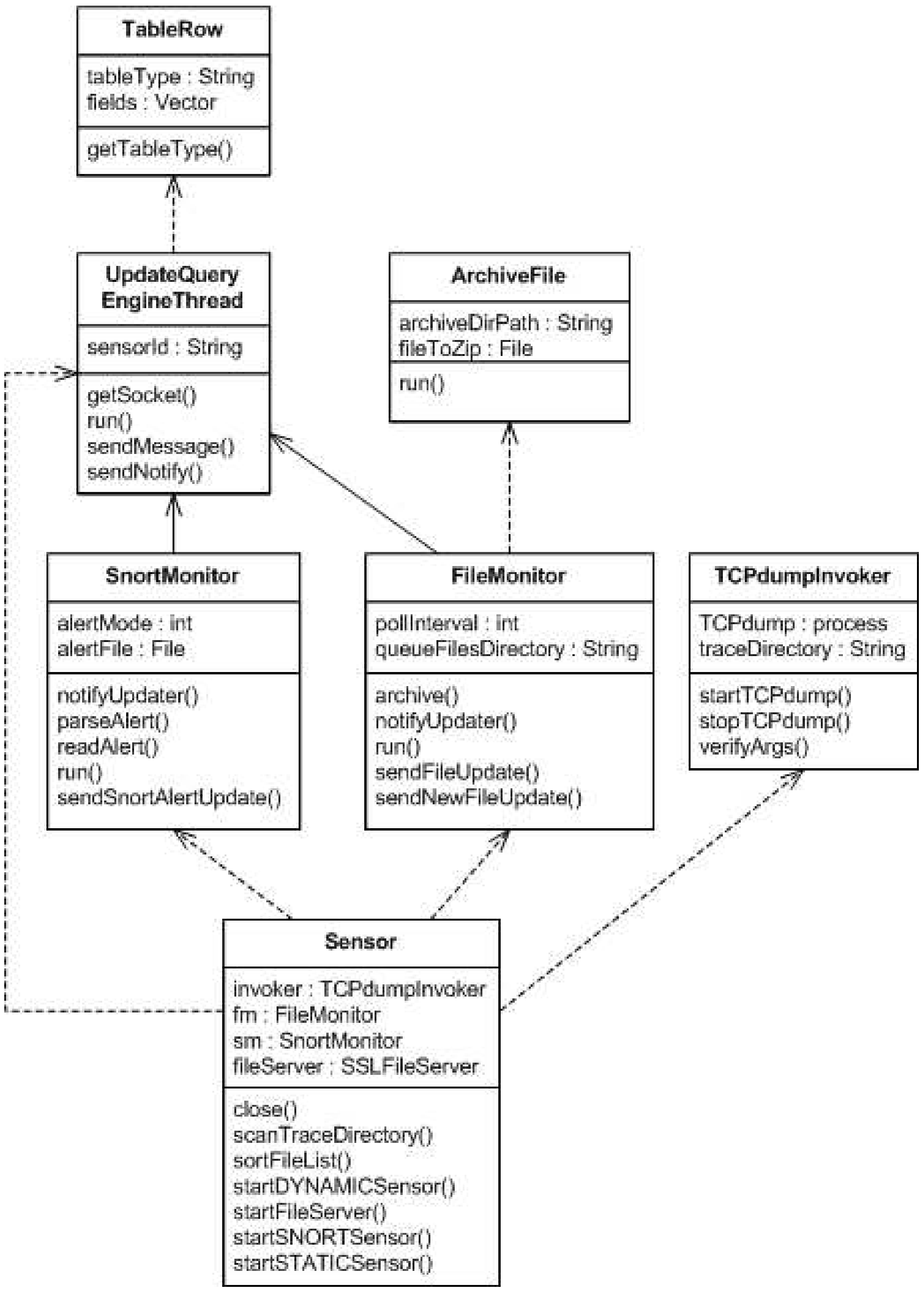

Figure 2.7: Sensor class diagram

### 2.3.2 The QueryEngine

The QueryEngine has three main functions: it provides the interface to the R-GMA through the CanonicalProducer API, it executes the queries received from the R-GMA and returns the result sets, and it publishes information on the currently connected sensors to the R-GMA.

To provide the interface the QueryEngine creates a new CanonicalProducer object. This object contacts the CanonicalProducer Servlet, which in turn registers the CanonicalProducer instance in the R-GMA registry. The QueryEngine then uses to the CanonicalProducer object to declare the tables of information that it provides. For the NetTracer these are the tables of Ethernet network packet data, each table relating to a layer in the Ethernet protocol, i.e. Ethernet, IP4, TCP, UDP and ICMP, and the sensor information tables, i.e. sensor, sensorFiles, and snortAlerts. When creating the CanonicalProducer the QueryEngine specifies a port number, and the QueryEngine then begins listening on this port. This sequence is shown in Figure 2.8. When the CanonicalProducer Servlet receives a query bound for this producer instance, it forwards the query to the QueryEngine by creating a socket connection to this port. The QueryEngine reads the query from the socket, executes the query and returns the results to the servlet through the same socket connection.

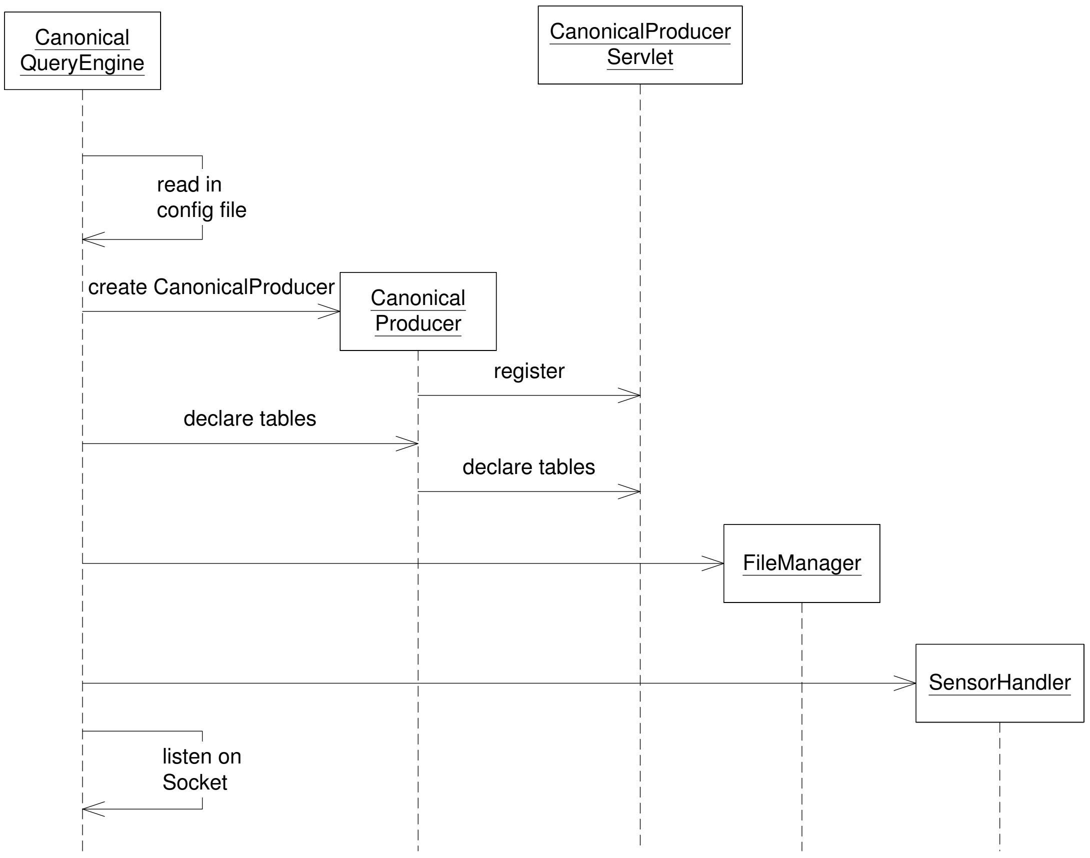

Figure 2.8: QueryEngine startup sequence

Figure 2.9 shows the sequence of events that occur when the QueryEngine receives a query. When the QueryEngine detects a socket connection being made by the servlet it creates a new EngineThread instance to handle the query, and then returns to listening for new connections. The EngineThread reads the SQL query from the socket and passes it to the SQLParser. The parser ensures the query is valid and then breaks the query into a form suitable for the Search class. It is the Search class that collects the data that satisfies the query from the raw Tcpdump log file by performing seek operations. At present there is no optimization of the search algorithm, which uses a simple linear scan. The data is collected into a ResultSet. Communication between components in the R-GMA is in XML, so the ResultSet is returned to the Servlet by way of the Responder class, which converts the ResultSet to an XML format.

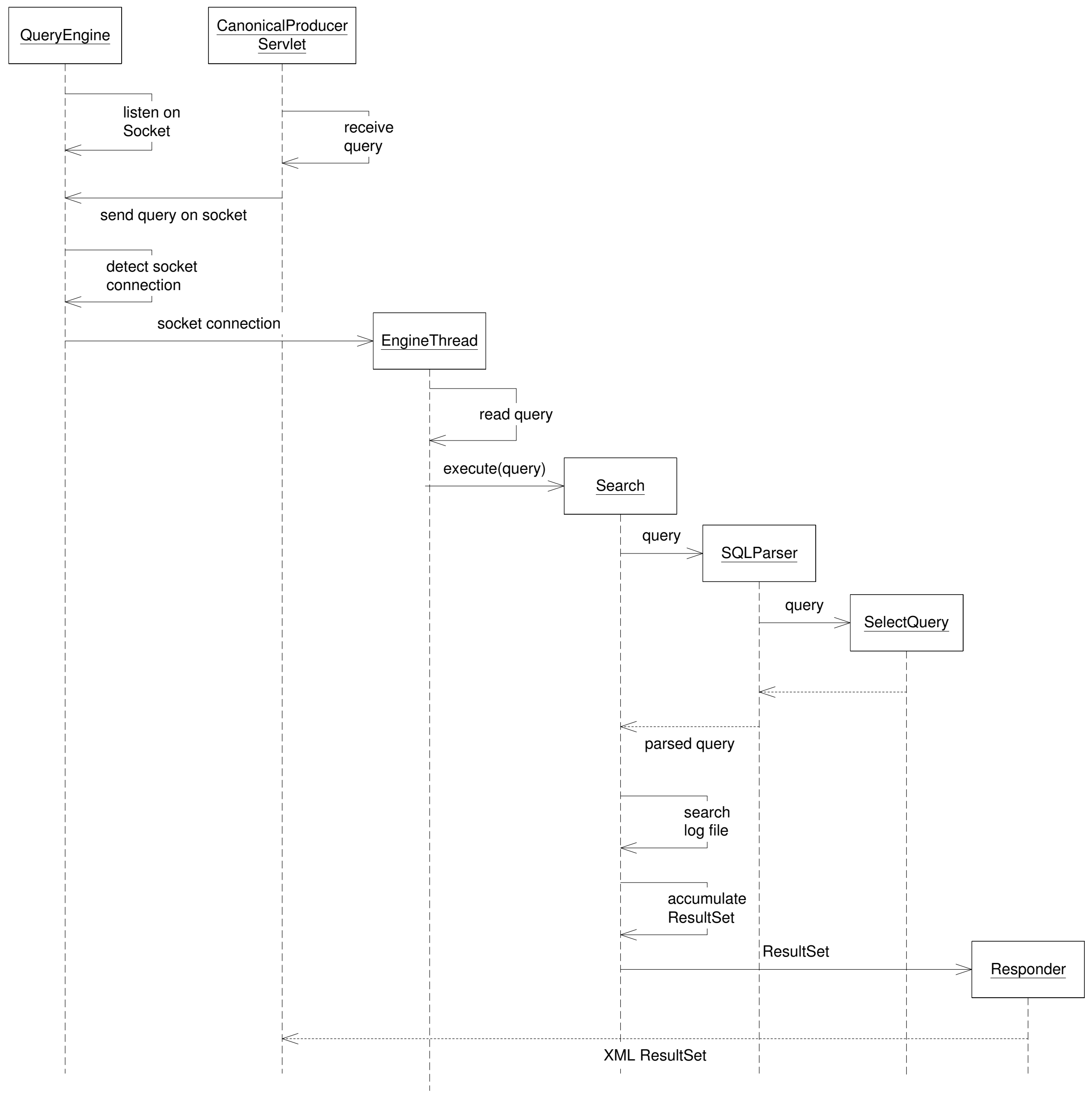

Figure 2.9: QueryEngine query processing sequence

The subset of SQL currently supported by the QueryEngine is as follows:

```
SELECT {* | [Table.]column_name [, [Table.]column_name...]} ]
FROM Table
[ WHERE [Table.]column_name { = | < | > } value
[ AND [Table.]column_name { = | < | > } value, ...] ]
```

The SQLParser class ensures that all received queries are of this form before it attempts to parse the query. Any malformed, or unsupported, queries received result in an exception being returned to the servlet. Otherwise the SQLParser breaks the query into a series of lists, a select list that contains the fields to be obtained from the log files, the table that the fields belong to, and the where predicates that must be matched. For example the following query:

```
SELECT destination_address, source_address, packet_type
FROM Ethernet
WHERE siteId = 'csTCDie'
AND sensorId = 'cagnode19.cs.tcd.ie:0'
AND fileId = 5
AND packetId < 100
```

would return the destination address, source address, and packet type of the first 100 packets contained in log file assigned ID 5, stored on the sensor with ID 0 that is hosted on cagnode19.cs.tcd.ie.

To do this the SQLParser parses the query as described above and then passes it to the Search class. The Search class checks to see if a sensor is currently connected with the ID 0 specified in the where clause. If so then the file ID 5 is mapped to the directory and filename of the log file on the sensor host machine cagnode19.cs.tcd.ie. The Search class can then access the log file required. If the sensor is running remote to the QueryEngine this is done over an SSL socket connection. The mechanim used for the remote file access is described in Section 2.3.3. An offset is calculated into the file for each of the fields needed. The size of each field in bytes is also known, so the Search class offsets into the file the required amount and reads the bytes from the file. In some cases the bytes that represent the field need to be converted. For example in the case above the destination and source addresses are converted from byte values into MAC addresses. This is done for each packet which matches the where clauses, in this case the first 100 packets in the file. The resulting data set is then accumulated. In order to return the data to the servlet it must first be converted into an XML resultset as described above. This is done by the Responder class. The XML result set that would be generated in response to the example query given above would be of the form:

```
<?xml version = '1.0' encoding='UTF-8' "standalone='no'?>
<edg:XMLResponse xmlns:edg='http://www.edg.org'>
<XMLResultSet>
<rowMetaData>
<colMetaData>destination_address</colMetaData>
<colMetaData>source_address</colMetaData>
<colMetaData>packet_type</colMetaData>
</rowMetaData>
<row><col>00:30:ax:40:19</col><col>00:30:b4:12:0f</col><col>0x800</col></row>
<row><col>00:30:ax:40:19</col><col>00:30:b4:12:0f</col><col>0x800</col></row>
.
.
.
</XMLResultSet>
</edg:XMLResponse>
```

The QueryEngine can also receive messages from a sensor. When the QueryEngine reads a message from a socket connection the message is parsed to see if it is an SQL query from the servlet or a message from a sensor. Messages sent from the sensors are of the form, `message_header;message`. The QueryEngine checks for the presence of one of the known message headers, and if found the message is passed to the SensorHandler class, otherwise it is sent to the SQLParser. For example when a sensor is first started it sends a new sensor message to the QueryEngine, `newsensor;sensorhost;sensortype`, where sensor host is the hostname of the machine hosting the sensor, and sensor type is the type of sensor being run. The SensorHandler class parses this message and uses the information to build a row of the sensor table, which it then inserts into the sensor information tables by using a R-GMA LatestProducer. The QueryEngine stores three tables of information relating to sensors. The tables are, `sensors`, which stores information on the sensor, such as sensor hostname, and the sensor's type, `sensorFiles`, which stores information on the log files stored on the Sensor host node, such as filename, and file ID, and `snortAlerts`, which is used if their is a SNORT Sensor running, to store the detected SNORT alerts. Through these tables users can then find out how many sensors are currently running, and the number of log files currently stored by them.

When a new sensor registers with the QueryEngine is details are stored in a recovery file. In the case of a QueryEngine restart the Recovery class uses the information in this file to try and recover the state of the sensor information tables. It attempts to re-register each of the sensors stored in the file. If successful the sensors information is re-entered into the sensor information tables. If the sensor is not contactable, because, for example, it was shutdown whilst the QueryEngine was stopped, then its details are removed from the recovery file and discarded.

The QueryEngine is composed of several classes, as shown in the class diagram, Figure 2.10.

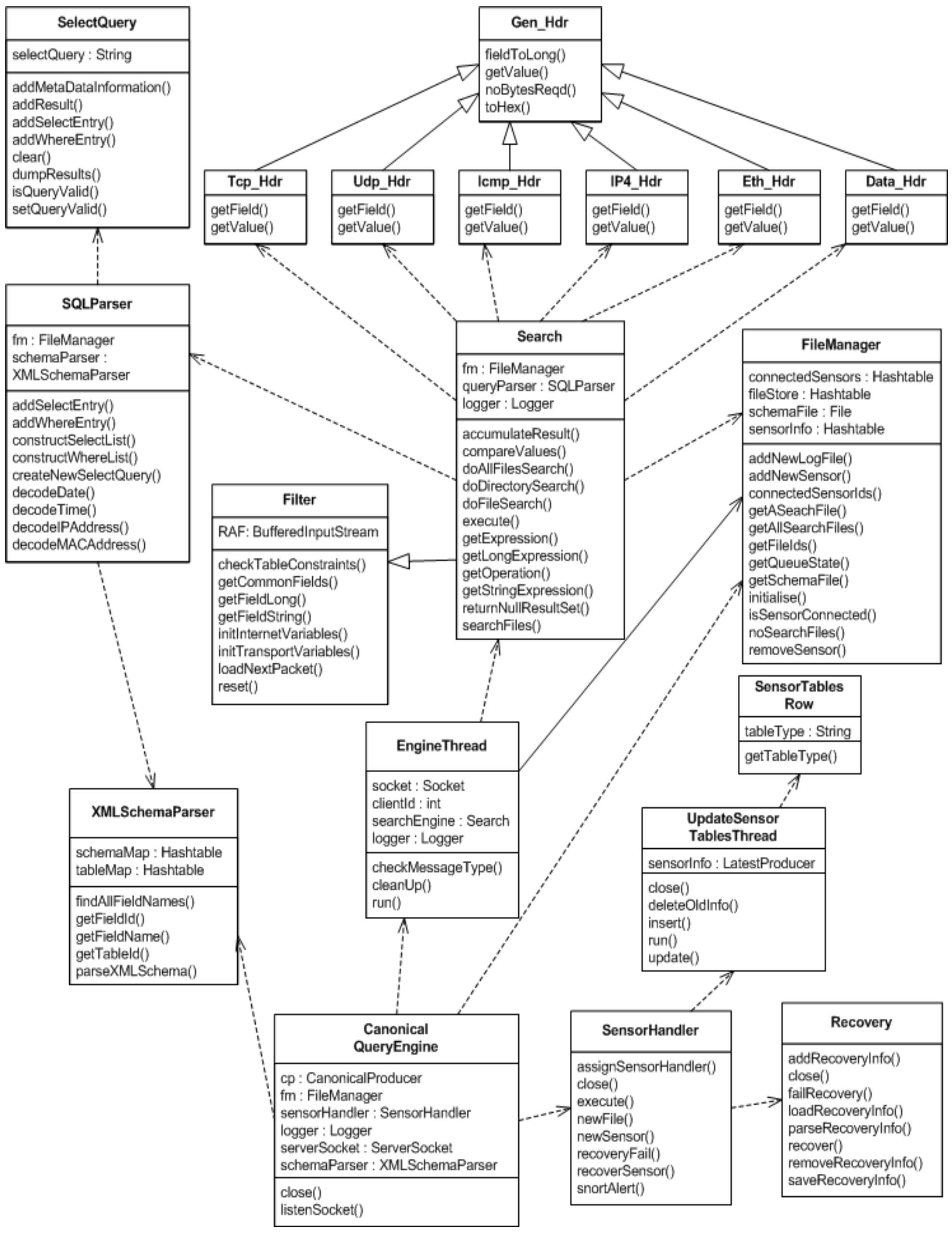

Figure 2.10: QueryEngine class diagram

### 2.3.3 Remote File Access

In order to provide access to the remote log files the sensor component starts a simple file server that listens for file access requests from the QueryEngine. A class diagram for the file server is shown in Figure 2.11.

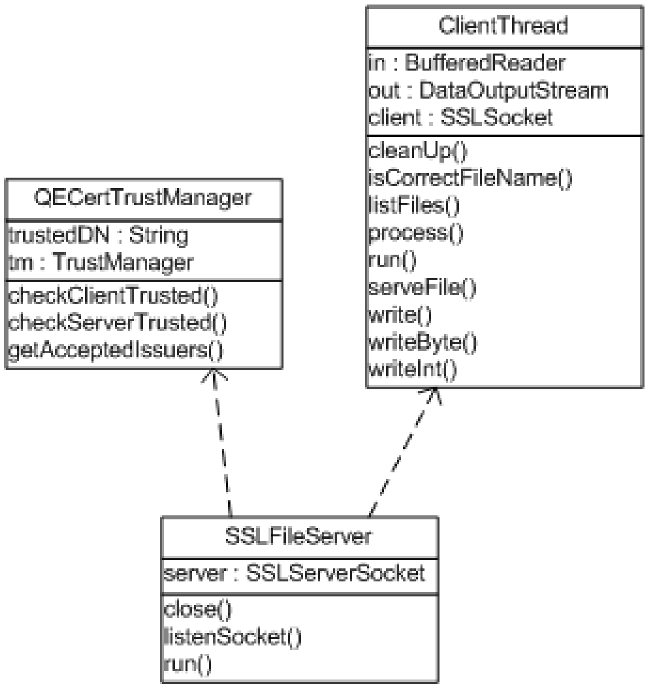

Figure 2.11: Remote File Access, Server Side

The SSLFileServer class instantiates a QECertTrustManager in order to create an SSLServerSocket, on which it listens for connections from the QueryEngine. The QECertTrustManager makes use of the DataGrid Java Security package [EDGSECURITY] in order to validate the certificates presented to it by the client during the SSL handshake. The DN of the trusted QueryEngine, entered in the Sensor configuration file, is passed to the SSLFileServer. If a certificate is validated by the default TrustManager (by matching against the trusted CA certificates) then the DN of the client certificate is checked against the trusted DN. If they match then the client is allowed to connect and the file request is passed to the ClientThread.

The ClientThread class is responsible for actually serving the requested file to the client. It reads the filename from the socket and checks it against the allowed filename pattern. This pattern is passed to it by the Sensor and will only match files that 'belong to this sensor', i.e. files being monitored by this sensor within the sensors trace directory. If the file access is allowed then bytes are read from the file by the ClientThread and sent back to the client over the SSL connection.

Figure 2.12 shows the client side of the remote file server. This only has two classes, RemoteFile and RemoteFileInputStream. These classes are intended to mimic the interface of the standard Java File and FileInputStream classes. When the QueryEngine tries to access a log file, instead of creating a File object it creates a RemoteFile object, which is then passed to the RemoteFileInputStream. The RemoteFileInputStream can then be used in exactly the same way as the normal FileInputStream to read the remote file. When instantiated it attempts to connect to the file server running on the remote host. It does this by creating an SSLSocket, again using the DataGrid Java Security packages. The host certificate of the machine is used in making this connection. If a connection cannot be made then an IOException is thrown, otherwise the number of available bytes that can be read from the remote

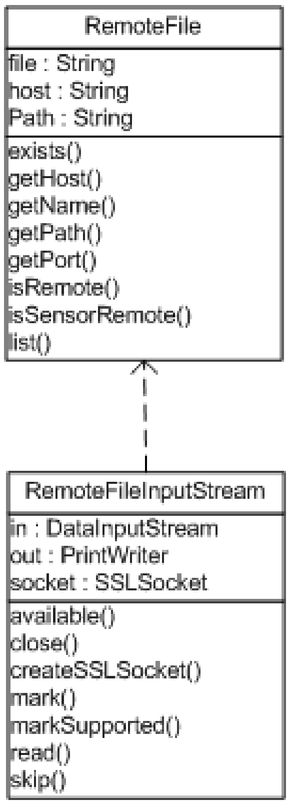

Figure 2.12: Remote File Access, Client Side

file is returned. Calls to read the file now read in from the socket connection. The RemoteFile class implements methods similar to the standard File class such as exists, and list. Exists checks to see if the file is accessible on the remote host by attempting to create a connection to the file server. List returns an array of the filenames of the files currently being served by the remote file server (i.e. files in the sensors trace directory that match the allowed filename pattern).

### 2.3.4 The Viewer module

The Viewer module provides a Java Swing GUI that allows users to collect and view the data published by the NetTracer. The Viewer GUI has two main panels, the packet view and the query panel. The packet view displays packets from the selected log file. The query panel allows a user to submit an SQL query to collect subsets of the available data.

The packet view panel provides a number of controls that allow the user to specify the packet to view. Two drop-down boxes and a textfield allow the user to choose the sensor, file and packet ID. These are used by the Viewer to construct a SQL query to collect the packet's data from the log file. The Viewer, by using the Consumer API, will contact a Consumer Servlet, which in turn contacts a R-GMA registry in order to locate the required producers of the information. The information is returned by the same mechanisms to the Viewer in the form of a ResultSet. The sequences of events that occur when a query is submitted from the Viewer are shown in Figure 2.13.

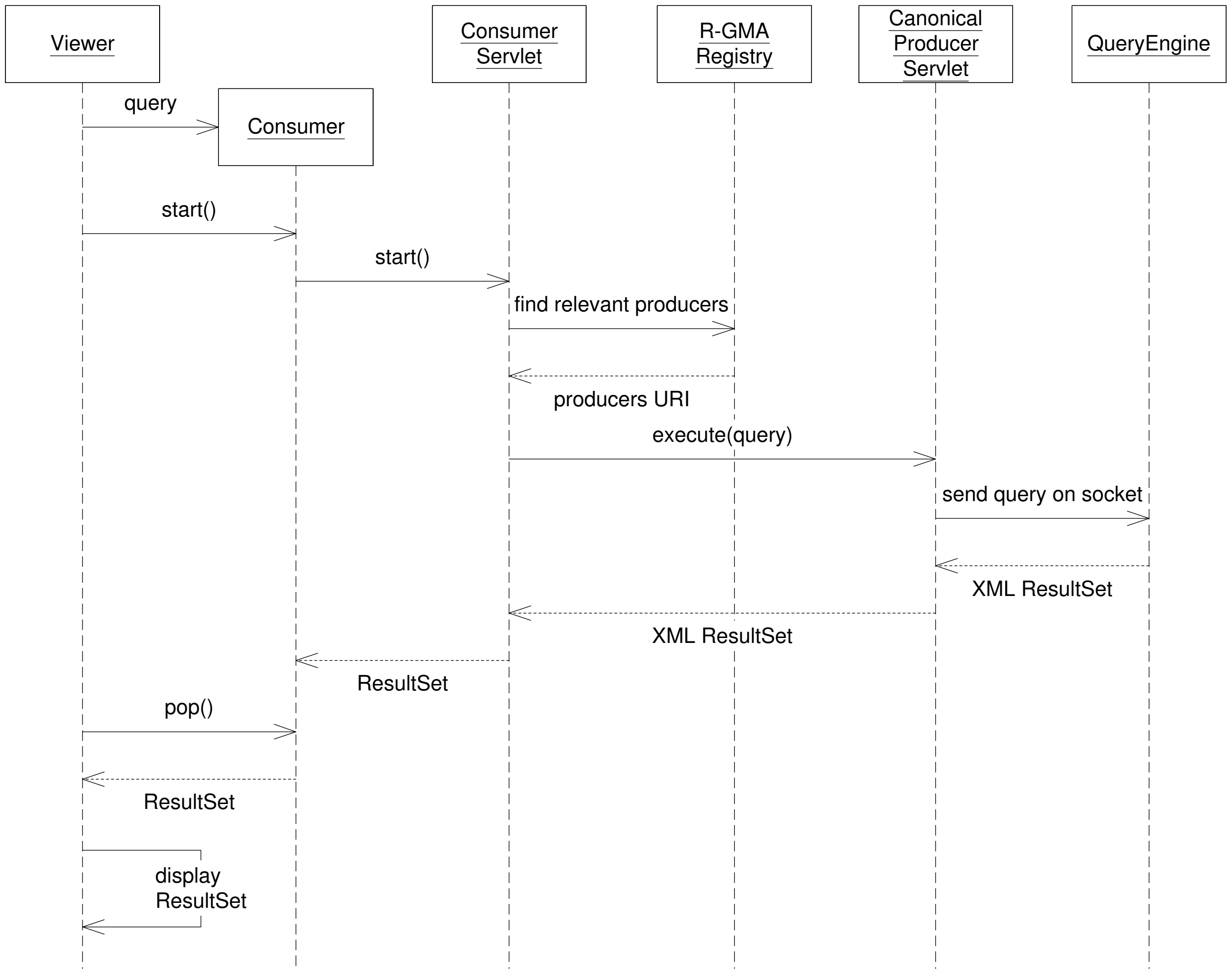

Figure 2.13: Viewer query submission sequence

# 3 PRODUCT TESTING

Testing of the NetTracer system was carried out continuously during the development phase. This took the form of functional testing, i.e. ensuring the basic functionality of the system was correct, and that any changes to the code did not result in a defined test failing. To do this the system was deployed on the CrossGrid testbed cluster located in TCD, and a series of predefined tests were performed. The following section describes these tests.

For the test deployment a sensor was installed on each worker node. The QueryEngine was installed on the storage element. The R-GMA services required were installed on a separate node. Figure 3.1 summarises the test deployment.

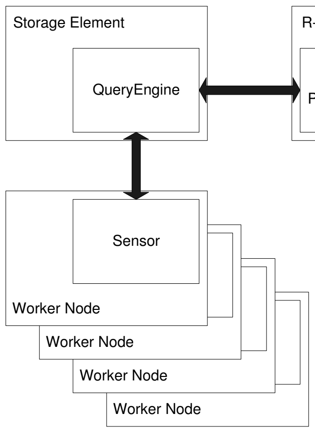

Figure 3.1: Test deployment

The tests were defined in order to ensure the correct operation of the system. They take two forms. The first is a series of test scenarios, defined and carried out after each major change in the code. They define a series of steps that should be performed, after which the results should be recorded, and a decision made on whether the test has passed or failed. For example a number of test scenarios were defined which start the QueryEngine, Sensor and Viewer in a series of different temporal orderings to ensure no exceptions or errors occur:

1. Start the QueryEngine
2. Start a Sensor
3. Start the Viewer

The above scenario describes the normal start-up sequence for the system, and as such no errors or exceptions would be expected. The following test, however, should result in an error:

1. Start the QueryEngine
2. Start the Viewer

When this test was initially defined and performed, start-up of the Viewer was failing. The reason for this was that the Viewer was hanging whilst attempting to retrieve information on the sensors that were currently running. This was resulting in a number of null pointer exceptions as no sensors were currently registered with the QueryEngine. This was corrected, and the test now results in an error message being displayed to the user informing them that no sensors are currently connected to the QueryEngine.

The second type of functional tests created were in the form of JUnit test cases. A suite of test cases was implemented to perform both system and unit tests of the code.

The system tests provide both completeness and correctness tests. The completeness tests are used to ensure that results sets returned to the consumer contain the expected number of tuples. The correctness tests check that the data contained in the returned tuples is correct. To perform these tests a pre-acquired set of log files was used. The files were viewed using Ethereal, an application for viewing Tcpdump format log files, and the data from a selected set of packets was recorded. A set of queries were then executed and the returned result sets were compared to the known data stored. Any deviations would result in a failed test. The following is an example of a Completeness test:

```
public void testLessThanRangeQuery() throws Exception {
        String query = "SELECT * FROM Packet "
                     +"WHERE sensorId = '" + sensorId + "' "
                     +"AND fileId = 0 "
                     +"AND packetId < 10";

        consumer = new Consumer(query, Consumer.HISTORY);

        consumer.start();

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

        if (!consumer.canPop()) {
            fail("No ResultSet returned, is a QueryEngine and Sensor running");
        }

        ResultSet rs = consumer.pop();

        assertEquals(10, rs.size());

    }
```

This test executes a range query, selecting the first ten packets in the log file. Obviously this query would be expected to return ten tuples, if not an error is reported and the test fails. Tests of this form were repeated for other types of query, such as queries across files.

Also defined were a number of QueryEngine system tests. These test cases test whether the QueryEngine responds correctly to the different types of messages that it can receive, i.e. a SQL query, a new sensor connecting, a new log file update, etc. For example, a new sensor message is sent to the QueryEngine, and the Sensor information database is then queried to test whether the sensor's details have been correctly recorded.

The unit tests defined bypass both the R-GMA and the QueryEngine to allow direct testing of the code.

On completion of these tests, the SANTA-G NetTracer could be said to be in an operational state.

# 4 EDG License Agreement

## A NetTracer SCHEMA

```
<?xml version="1.0" encoding="UTF-8"?>
<SANTAGSchema name="NetTracer" type="Ethernet">
        <table id="0" name="File">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">magic_number</field>
                <field id="1">version_major</field>
                <field id="2">version_minor</field>
                <field id="3">tzone_offset</field>
                <field id="4">time_accuracy</field>
                <field id="5">snapshot_length</field>
                <field id="6">link_type</field>
                <field id="7">timestamp_Secs</field>
                <field id="8">timestamp_uSecs</field>
                <field id="9">MeasurementDate</field>
                <field id="10">MeasurementTime</field>
        </table>
        <table id="1" name="Packet">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">timestamp_Date</field>
                <field id="1">timestamp_Time</field>
                <field id="2">timestamp_Secs</field>
                <field id="3">timestamp_uSecs</field>
                <field id="4">capture_length</field>
                <field id="5">actual_length</field>
                <field id="6">MeasurementDate</field>
                <field id="7">MeasurementTime</field>
        </table>
        <table id="2" name="Ethernet">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">destination_address</field>
                <field id="1">source_address</field>
                <field id="2">packet_type</field>
                <field id="3">timestamp_Secs</field>
                <field id="4">timestamp_uSecs</field>
                <field id="5">MeasurementDate</field>
                <field id="6">MeasurementTime</field>
        </table>
        <table id="3" name="IP4">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">header_length</field>
                <field id="1">ip_version</field>
                <field id="2">service_type</field>
                <field id="3">packet_length</field>
                <field id="4">datagram_id</field>
                <field id="5">control_flags</field>
                <field id="6">fragment_offset</field>
                <field id="7">time_to_live</field>
                <field id="8">data_protocol</field>
                <field id="9">header_checksum</field>
                <field id="10">source_ip</field>
                <field id="11">destination_ip</field>
                <field id="12">timestamp_Secs</field>
                <field id="13">timestamp_uSecs</field>
                <field id="14">MeasurementDate</field>
                <field id="15">MeasurementTime</field>
        </table>
        <table id="4" name="ICMP">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">icmp_type</field>
                <field id="1">icmp_code</field>
                <field id="2">checksum</field>
                <field id="3">identifier</field>
                <field id="4">sequence_number</field>
                <field id="5">timestamp_Secs</field>
                <field id="6">timestamp_uSecs</field>
                <field id="7">MeasurementDate</field>
                <field id="8">MeasurementTime</field>
        </table>
        <table id="5" name="TCP">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">source_port</field>
                <field id="1">destination_port</field>
                <field id="2">sequence_number</field>
                <field id="3">acknowledge_number</field>
                <field id="4">header_length</field>
                <field id="5">reserved</field>
                <field id="6">code</field>
                <field id="7">window</field>
                <field id="8">checksum</field>
                <field id="9">urgent_pointer</field>
                <field id="10">timestamp_Secs</field>
                <field id="11">timestamp_uSecs</field>
                <field id="12">MeasurementDate</field>
                <field id="13">MeasurementTime</field>
        </table>
        <table id="6" name="UDP">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">source_port</field>
                <field id="1">destination_port</field>
                <field id="2">message_length</field>
                <field id="3">checksum</field>
                <field id="4">timestamp_Secs</field>
                <field id="5">timestamp_uSecs</field>
                <field id="6">MeasurementDate</field>
                <field id="7">MeasurementTime</field>
        </table>
        <table id="7" name="Data">
                <field id="-4">siteId</field>
                <field id="-3">sensorId</field>
                <field id="-2">fileId</field>
                <field id="-1">packetId</field>
                <field id="0">byte000_063</field>
                <field id="1">byte064_127</field>
                <field id="2">byte128_191</field>
                <field id="3">byte192_255</field>
                <field id="4">byte256_319</field>
                <field id="5">byte320_384</field>
                <field id="6">byte385_447</field>
                <field id="7">byte448_511</field>
                <field id="8">byte512_575</field>
                <field id="9">byte576_639</field>
                <field id="10">byte640_703</field>
                <field id="11">byte704_767</field>
                <field id="12">byte768_831</field>
                <field id="13">byte832_895</field>
                <field id="14">byte896_959</field>
                <field id="15">byte960_1023</field>
                <field id="16">byte1024_1087</field>
                <field id="17">byte1088_1151</field>
                <field id="18">byte1152_1215</field>
                <field id="19">byte1216_1279</field>
                <field id="20">byte1280_1343</field>
                <field id="21">byte1344_1407</field>
                <field id="22">byte1408_1471</field>
                <field id="23">timestamp_Secs</field>
                <field id="24">timestamp_uSecs</field>
                <field id="25">MeasurementDate</field>
                <field id="26">MeasurementTime</field>
        </table>
</SANTAGSchema>
```

# Bibliography

[EDGSECURITY] EDG WP2 Security; **http://edg-wp2.web.cern.ch/edg-wp2/security/**

[TEST] Jorge Gomes, LIP; **Middleware Test Procedure**; May 2002

[QAP] WP5, CYRFRONET; **Quality Assurance Plan**; Evolving document

[RGMAINSTALL] DataGrid WP3; **R-GMA Installation Guide**;
http://hepunx.rl.ac.uk/edg/wp3/documentation/doc/installation.pdf

[RGMAUSER] DataGrid WP3; **R-GMA User Guide**;
http://hepunx.rl.ac.uk/edg/wp3/documentation/doc/user.pdf

[SANTAGINSTALL] Stuart Kenny; **SANTA-G Installation Guide**;
http://gridportal.fzk.de/autobuild/i386-rh7.3-gcc3.2.2/wp3_3_2-santag/userdoc/installation/installation.pdf

[SANTAGUSER] Stuart Kenny; **SANTA-G Users Guide**;
http://gridportal.fzk.de/autobuild/i386-rh7.3-gcc3.2.2/wp3_3_2-santag/userdoc/user/user.pdf

[SNORT] SNORT Network Intrusion Detection System; **SNORT Website**;
http://www.snort.org